Simulation Modelling of Distributed-Shared Memory Multiprocessors

Worawan Marurngsith

Doctor of Philosophy

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2006

Abstract

Distributed shared memory (DSM) systems have been recognised as a compelling platform for parallel computing due to the programming advantages and scalability. DSM systems allow applications to access data in a logically shared address space by abstracting away the distinction of physical memory location. As the location of data is transparent, the sources of overhead caused by accessing the distant memories are difficult to analyse. This *memory locality problem* has been identified as crucial to DSM performance. Many researchers have investigated the problem using simulation as a tool for conducting experiments resulting in the progressive evolution of DSM systems. Nevertheless, both the diversity of architectural configurations and the rapid advance of DSM implementations impose constraints on simulation model designs in two issues: the limitation of the simulation framework on *model extensibility* and the lack of *verification applicability* during a simulation run causing the delay in verification process.

This thesis studies simulation modelling techniques for memory locality analysis of various DSM systems implemented on top of a cluster of symmetric multiprocessors. The thesis presents a simulation technique to promote model extensibility and proposes a technique for verification applicability, called a Specification-based Parameter Model Interaction (SPMI). The proposed techniques have been implemented in a new interpretation-driven simulation called DSIMCLUSTER on top of a discrete-event simulation (DES) engine known as HASE. Experiments have been conducted to determine which factors are most influential on the degree of locality and to determine the possibility to maximise the stability of performance.

DSIMCLUSTER has been validated against a SunFire 15K server and has achieved similarity of cache miss results, an average of $\pm 6\%$ with the worst case less than 15% of difference. These results confirm that the techniques used in developing the DSIMCLUSTER can contribute ways to achieve both (a) a highly extensible simulation framework to keep up with the ongoing innovation of the DSM architecture, and (b) the verification applicability resulting in an efficient framework for memory analysis experiments on DSM architecture.

Acknowledgements

I would like to thank my supervisor, Professor Roland Ibbett, for the invaluable advice and supports. I am very grateful for his patience and understanding for my chaotic ways of working. I would also like to thank my second supervisor, Marcelo Cintra, and the panels who gave some suggestions during the year reports discussion.

Many colleagues in the HASE group provided constructive discussions. Juan Carlos provided the on-demand helps, bug fixes, and user supports in the HASE tools which made the statistic collection possible. Frederick has transferred the entity templates used in DSIMCLUSTER from the original Windows HASE, and implemented the clock model into the HASE framework, which ease the implementation.

I am very grateful for some valuable comments, encourangement, supports and distractions provided in large quantities from Khun Manfarang, without which I would never have many smiles and would never have finished this thesis.

I would like to thank Ajarn Wasutep Thaprasop and Ajarn Rachada Kongkachandra who have the fate in me, and giving me the great opportunity to work at Thammasat University. I would like to thank Asst. Prof. Prakorn Sermsuk, Asst. Prof. Prajot Thammakornbunjut, Asst. Prof. Sumalee Pisithakasem, Dr. Cholyeun Hongpaisanwiwat, P'Toi and P'Odd who helped me get through all unexpected difficulties. I am grateful for the supports from P'Pom who made these four years very enjoyable. I acknowledge the full sponsorship from Thammasat University and the support of EPCC for the Lomond machine (the SunFire 15K).

I am in debt to the Srikaranonda family for their helps; Sorraya, Wendy, Pailin, Auan and X for taking care of my parents and me; Chalita who is always be there for me; Neung Sittiporn for his friendship and helps on the statistical analysis.

I am thankful for many friends who have made everyday meaningful: Itsara, P'Jake, Kay, P'Kob, P'Ob, P'Ekk, Jialin, P'Bua, Steve, Paula, Jim, Mandisa, Teresa, Christiane, Yan and Sadaf. Thanks to P'Pom, Len, P'Rachadaporn for their friendships; and thanks to P'Kanit, P'Wan, P'Ben, P'Yao, P'Nai, P'Noi, P'Tung for making every weekend so entertain.

This thesis would never have finished without my beloved parents, for their ultimate love, understanding and for always be there for me.

Korb khun ka.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Parts of this work have been published as:

- Marurngsith, W. and Ibbett, R. N. (2004).“DSIMCLUSTER: A Simulation Model for Efficient Memory Analysis Experiments of DSM Clusters”. *in Proceedings of the 2004 Summer Computer Simulation Conference (SCSC 04)*, page 191. The Society for Modeling and Simulation International, SCS. Best Paper Award.

- Marurngsith, W. and Ibbett, R. N. (2005).“Specification-based parameter-model interaction: Towards a correct reflection of memory characteristics in a DSM cluster simulation”. *in Proceedings of the 2005 Summer Computer Simulation Conference (SCSC 05)*. The Society for Modeling and Simulation International, SCS.

(*Worawan Marurngsith*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the concept of a distributed shared memory (DSM) was first introduced in 1973, it has been recognised as a viable solution for scalable and programmable multiprocessors. Due to the inherent distinction between local and remote memories, the advantage of the DSM concept has been offset by the reduction of *memory locality*. During the last three decades, many optimisation techniques have been proposed to tackle the performance degradation caused by the loss of locality. Many solutions have been developed and successfully used to alleviate different types of overhead caused by the loss of locality on different legacy DSM implementations. The process of developing such a solution depends to a large extent on various pre-design studies using simulation modelling as a tool to search for a solution that permits sustained high performance. However, recent advances in DSM designs towards software hierarchy have forced a re-evaluation of current simulation techniques. The challenges lie in how to effectively extend the model and apply model verification to emulate design innovations in both software and hardware aspects of a DSM system.

This thesis studies simulation modelling techniques for memory locality analysis of a DSM system implemented on top of a cluster of symmetric multiprocessors. This chapter states the thesis motivation and introduces the context of the work undertaken. The chapter begins by showing in Section 1.1 that a DSM system is compelling as it provides programming advantages within a scalable and cost-effective hardware solution. This section also presents the main question about the architecture, *i.e.* , how to achieve a solution that permits sustained performance with minimum loss of memory locality. To answer this question, simulation modelling is the most widely used

methodology to assist the design space exploration process. In Section 1.2, challenges to simulation frameworks to carry out a performance evaluation of DSM architecture are discussed: the diversity of architectural components and the heterogeneity of the DSM implementation. Section 1.3 shows how both of these aspects have imposed constraints on simulation techniques: the limitations of the framework on *model extensibility* and the delay caused by the verification process (*verification applicability*). In Section 1.4, simulation techniques to overcome these two limitations are proposed and the main contributions of this thesis work are presented. Finally, the last section gives an outline of the content presented in the remainder of the thesis.

1.1 Distributed Shared Memory Systems

The demand for computation power in scientific research, within a limited expenditure on parallel software development, has motivated the innovation of *distributed shared memory* (DSM) systems. Central to a DSM architecture is its logical implementation of the shared-memory model on a physically distributed-memory system. A DSM system provides users with the means to access the shared memory image using conventional load and store operations. The underlying DSM infrastructure hides away from users both the communication mechanism and the layers of memory management. Consequently, users can benefit from both the scalability of distributed-memory machines and the programmability of the traditional shared-memory paradigm.

The evolution of address mapping techniques led to the progressive development of DSM systems. Approximately a decade after the first appearance of a multiprocessor system, the concept of single address space (SAS) on top of a distributed system was first introduced. A machine called Computer Module (CM*) developed at Carnegie Mellon University [Fuller et al., 1973] was the first DSM system [Bell, 1999] implemented solely in hardware. As shown in the block diagram of Figure 1.1 (a), the CM* was built by chaining together some Computer Modules using busses to let address information traverse from one module to another. This interconnection allows the physical memory of all modules to be shared by having inside each module an address mapping circuit that routes each access request to its target memory. CM* introduces the concept of a SAS on top of the physical memory address range

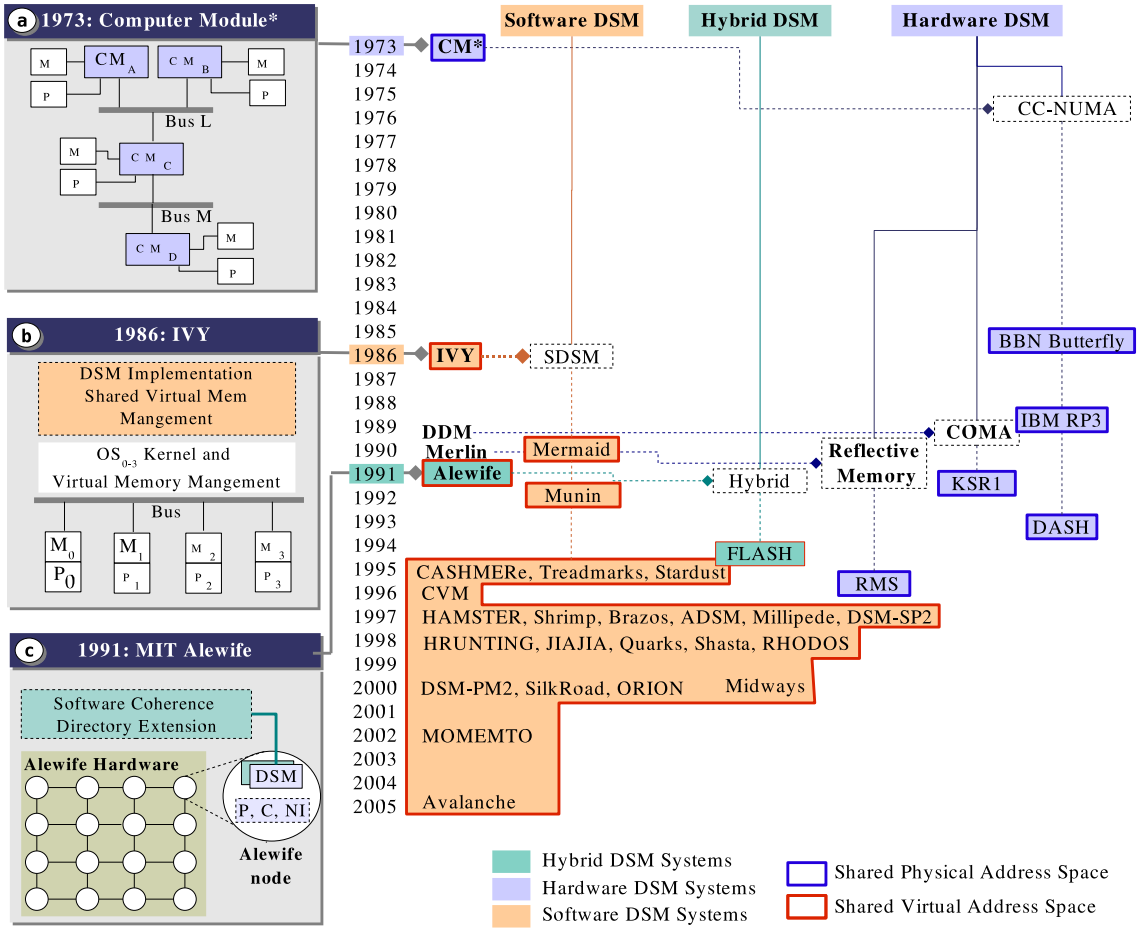


Figure 1.1: The time line of DSM architectural development.

permitting the processors to access the entire memory content of a distributed system. The concept was also developed in the later DSM machines of different architecture like IBM RP3 and BBN Butterfly [BBN Laboratories, 1986].

Later on, the early SAS concepts were extended to cover the range of virtual memory address space. This single virtual-address space concept introduced another branch of the DSM architecture referred to as a *Shared-Virtual Memory* (SVM) system. In 1986, IVY [Li, 1986] was the first system that introduced the SVM image using a software implementation on top of the operating system's kernel (see Figure 1.1 (b)). The IVY project demonstrated the portability of shared memory programs to distributed multiprocessors whilst causing less complexity to compilers in marshalling (*i.e.* interpreting data references into an address-space neutral format). However, the delay of software-controlled memory management has been a drawback issue. To alleviate this, in 1991, the MIT Alewife [Agarwal and et. al., 1991] introduced a hybrid implementation of the SVM image using both software and hardware support. In addition to the systems mentioned, 29 on-going DSM projects of different implementations established during 1984-1992 have been featured in two comprehensive surveys [Nitzberg and Lo, 1991, Raina, 1992]. In 1992, DSM entered the symmetric multiprocessors picture with the emergence of the KSR-1 [Frank et al., 1993]. This stimulated an interest in scalable multiprocessors based on multiprocessors as a component. Since 1998 several manufacturers have delivered DSM multiprocessors with up to 32 or up to 128 processors [Bell, 1999], and it is believed that moderate-scale DSM multiprocessors are likely to become one of the most important architectures for large-scale commercial computing [Hennessy et al., 1999].

Over the last 30 years of DSM architectural development, numerous research studies have been conducted into the feasibility of scalability within an effective parallel performance [Salehi et al., 1995, Bilas et al., 1999, Qin and Baer, 1997, Kudlur and Govindarajan, 2004], and have indicated the possibility of scaling up to at least 256 (and projected to 1024) processors [Bilas et al., 2003, Dongarra et al., 2005]. These well-designed research studies have clearly demonstrated the efficacy of DSM architecture based on different implementations. DSM systems have become one of the main computing platforms in the Grid[1]. A recent workshop on DSM [CCGrid,

[1]Grid or computational grid [Foster and Kesselman, 1998] is a model for solving massive computational problems by making use of the unused resources of large numbers of disparate computers treated as a virtual cluster embedded in a distributed telecommunications infrastructure. Central to

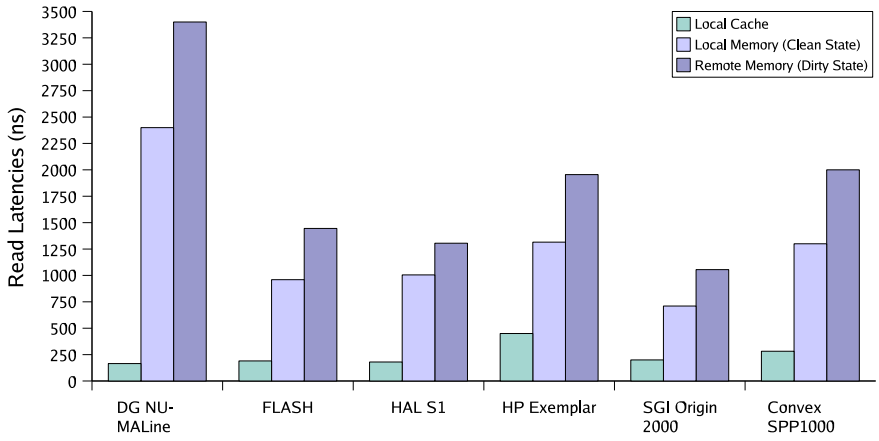


Figure 1.2: Read latencies of some DSM machines (ns).

2005] featured a number of papers addressing issues of shared-memory synchronisation, consistency management, and performance issues associated with a networking infrastructure. An emergence of DSM to support *Mobile Grid* [Marco Ballette, 2005] is of interest among researchers. This particular research demonstrated the feasibility of viewing an SVM image as an unstructured DSM. This concept was then used to connect handheld computers and similar mobile devices within a Grid. The area is large, reflected by the number of on-going studies.

Set against the cost-performance and programmability gained in a DSM architecture is the reduction in memory locality (*i.e.* the degree to which data is located close to the processor(s) that access it). This problem arose and was made more severe by the increasing gap between processing speed and the interconnection speed. Figure 1.2 shows the average time (in nanoseconds) that six DSM machines spent on reading a shared data at three degrees of locality: accessing local cache, local memory (home), and remote memory[2]. This figures highlight the fact that a DSM system may suffer a penalty caused by accessing data from a remote memory of up to 20 times the delay in accessing data in local cache.

Many DSM implementations have tackled individual components to reduce the overhead caused by loss of locality. For example, the JIAJIA [Eskicioglu et al., 1999] and the ORION [Ng and Wong, 1999] implementations use a home-based software

Grid computing is the ability to support computation across administrative domains that sets it apart from traditional computer clusters or traditional distributed computing.

[2]Source from [Heinrich et al., 1999, Abandah and Davidson, 1998]

consistency model to minimise the round trip time when an access fault occurs. A page-based software DSM, Mome [Jégou, 2003], uses a weak consistency, multiple-writers DSM to reduce the number of operations on shared pages via explicit consistency requests. Other implementations, like CAS-DSM [Manoj et al., 2004], use software DSM in collaboration with static compiler analysis to identify the possibility to optimise locality from the source program. Unfortunately, these efforts have turned out to increase either the system overhead or program requirements. Moreover, the benefits of optimisation are difficult to analyse by measurement as there are a large number of overhead factors residing across the system hierarchy. Therefore, an effective analysis methodology that can both quantify the impact of each source of overhead of a memory access, and also analyse the interaction of such overheads, is essential.

1.2 Performance Analysis of DSM System - Challenges

Performance analysis clearly has crucial implications for the way the design alternatives of DSM architectures are studied. In general, various experiments have been conducted to study the performance of DSM design alternatives to project a solution that permits sustained performance. The conclusions that can be drawn from experiments depend to a large extent on which design factors they were intended to analyse. In this section, the issue of memory locality in DSM architecture is considered. The architectural components and the aspects of system implementation are investigated. This investigation identifies the factors that might be the cause of reductions in locality, and the challenges to carry out the performance analysis.

1.2.1 Memory Locality - Problem Statement

The term *memory locality*[3] refers to the degree to which data is located close to the processors that access it. The degree of memory locality can be measured by *data access latency*, *i.e.*, the average of the time intervals a processor waits to access shared data. In a DSM architecture, there is a time penalty for accesses to local memory (the memory of the same node), and for accesses to memory of different nodes. For the

[3] In this thesis, the term *memory locality* is used to refer to the DSM architecture unless otherwise stated.

architectures shown in Figure 1.2, this penalty can introduce a delay of 10-20 times the access latency to local cache. Therefore, the DSM system applies many strategies to keep data in volatile memories close to the processors that use it.

Despite this, DSM applications still have portability limitations on different DSM configurations. Shan *et al.* found that porting applications to different DSM configurations may result in poor spatial locality of physically distributed shared data [Shan and Singh, 2000]. This problem is stated as *the loss of locality*, *i.e.* the situation when the required data is stored at a distant location from the requesting processor. From the DSM system perspective, the loss of locality can be caused by various factors that can be either controllable or uncontrollable. Examples of uncontrollable factors are activities at start-up time (*e.g.* cold-start miss), or the pattern of data access and data sharing of individual applications. Controllable factors are those that can be tuned or configured in a DSM system to minimise the effects. Examples of these factors are: actions that cause local data to be invalidated (*e.g.* inclusion misses, false sharing); the architectural capacity (*e.g.* capacity miss); the cost of context switching; *etc.*

This thesis studies the loss of memory locality in a DSM architecture based on the following two objectives:

1. to determine which factors are most influential on the degree of locality;

2. to determine what values to choose for the influential controllable factors so that the effects of the uncontrollable ones are minimised, therefore maximising stability of performance.

In order to design suitable experiments for these objectives, the choice of factors, levels, and range has been investigated. The next two subsections describe this investigation and summarise the observed characteristics and the potential design factors of a DSM system.

1.2.2 Diversity of Architectural Configurations

The configuration of DSM architecture is diverse and its impact on performance was revealed by a series of unrelated studies. The generic architecture of a DSM multiprocessor (Figure 1.3 (a)) can be observed at three levels: the node, memory and interconnection architecture. At the node architecture level, Figueiredo and Fortes

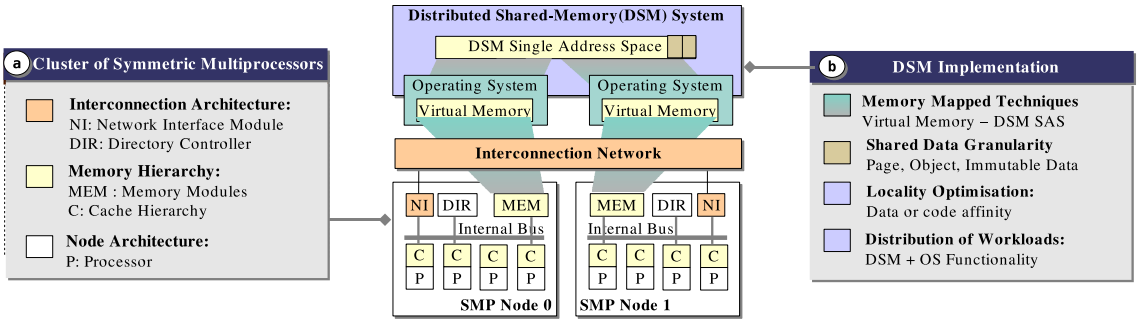


Figure 1.3: A basic architecture of a 2×2 DSM multiprocessor.

studied the performance of a heterogeneous DSM using a factorial design experiment based on simulation to carry out the sensitivity analysis. The study shows that processor performance has the greatest effect on the DSM performance (at 59.3%), followed by cache size, memory latency, and network latency [Figueiredo and Fortes, 2000]. Chaudhuri *et al*. showed that using a prefetching technique in DSM nodes also contributes to the performance [Chaudhuri et al., 2003].

At the memory architecture level, several studies have been carried out to analyse the impact of memory components on DSM performance. Foglia proposed an algorithm to classify the overhead sources caused by memory coherence activities [Foglia, 2001]. Performance of shared memory multiprocessors and DSM systems have been related to the overhead caused by coherence protocols [Grbic, 2003, Heinrich et al., 1999]. A detailed analysis of cache performance shows that the overhead of the coherence protocol is associated with the data access patterns and the parallel constructs used in the workload applications [Marathe et al., 2004].

At the interconnection architecture level, the topology, speed, reliability, latency and bandwidth are key contributors to DSM performance. Kodi presents the design of a high-speed optical interconnection for a scalable DSM [Kodi, 2005]. The study compares the performance of the proposed architecture with other topologies using CSIM simulator.

1.2.3 Heterogeneity of DSM Implementations

Many DSM implementations use different combinations of techniques to provide programmability while achieving performance and scalability of a distributed memory architecture. DSM implementations centre on the policies used to manage the ex-

tended memory hierarchy. Figure 1.3 (b) shows that these policies can be grouped into memory mapping techniques, management of shared data granularity, distribution of workloads, and locality optimisation techniques. A memory mapping technique identifies how to dynamically transfer data across the different memory levels. An impact of different memory mapping techniques on DSM performance has been studied on two DSM systems, TreadMarks and Cashmere, by using a measurement technique[4]. The study shows that different memory mapping techniques effect overall DSM performance differently based on data access patterns [Kontothanassis et al., 1997]. Management of shared data granularity is also an issue in terms of DSM performance. Niwa *et al*. compared the performance of page-based and segment-based software DSM using a compiler optimisation technique. The study found that a segment-based software DSM scheme reduces transmission of unnecessary data and automatically prevents the severe false sharing at fetch-on-write, which is the problem in the page-based scheme [Niwa et al., 2000].

The effects of workload distribution policies on DSM performance have also been addressed by a number of researchers. One issue has been the cost of paging due to the limited capacity of physical memory. Liu *et al*. proposed a workload distribution policy that takes into account the memory resources (*i.e.* the available physical memory space is enough to accommodate a new thread to avoid page replacement). The proposed policy has been successfully tested on a DSM on clusters of uniprocessors (*i.e.* a $1 \times n$ DSM system) [Liu et al., 2004].

Two approaches used in optimising memory locality have been shown to impact DSM performance: data affinity and code affinity. Moving data close to the requesting processor, *i.e.* a *data affinity approach*, (using prefetching, page-migration, or replication) can reduce the total number of non-local memory access. The cost of migration overhead can be minimised by maximising the use of remote memory for page replacement as demonstrated in the Cashmere-VLM remote memory paging [Dwarkadas et al., 1999]. The effectiveness of the data affinity technique also depends on the size/unit of a shared data. Lai *et al.* showed that memory caching with a small unit of shared data (*fine-grain*) can reduce traffic in DSM clusters more than migrating or replicating data at the page granularity [Lai and Falsafi, 2000]. A technique called dynamic (user-level) page migration based on information obtained

[4]In this study, both DSM systems were implemented on a 4×8 DEC AlphaServers

from hardware counters has been shown to be efficient in reducing traffic caused by load imbalance [Tikir and Hollingsworth, 2004].

The alternative to data migration, *replication*, is also widely used to obtain data affinity. Replicating data close to the requested processors requires mechanisms or protocols to maintain coherence among data replicas. Two fundamentals incorporated in the implementation of different coherence protocols have been shown to impact on DSM performance: *consistency model* and *home-related model*. A consistency model defines a set of rules specifying the desired order of memory operations on a shared page (*e.g.* Sequential Consistency (SC), Lazy Release Consistency (LRC)). The home-related model defines whether the actions of a coherence protocol are based on communicating to a particular node which is assigned as the *home* of a shared data (*i.e.* home-based *v* homeless models). Iosevich *et al.* compared the performance of DSM systems implementing a home-based LRC (HLRC) protocol and a multithreaded SC protocol. The study showed that the average speedup of the HLRC protocol is slightly better than that of the SC protocol (approximately 30%). In addition, this study highlighted that changing data granularity also has an effect on the parallel speedup obtained on the system [Iosevich and Schuster, 2004]. Yu *et al*. investigated the scalability of DSM in workstation clusters implementing both home-based and homeless LRC protocols. The study showed that the home-based DSM has better scalability than the homeless one. It is believed that poor scalability in the homeless protocol is a consequence of a hot spot and garbage collection [Yu et al., 2004].

Instead of transferring data, another common approach to obtain memory locality is to move a computation close to the node where data reside. This approach, called *code affinity* or *computation migration*, has been shown helpful for dynamic load distribution, fault resilience, eased system administration, and locality optimisation. Dynamic computation migration is useful for concurrent data structures with unpredictable read/write patterns [Hsieh et al., 1996]. Simultaneously considering thread memory access types and global sharing could reduce the communication during load balancing by half [Liang et al., 2002]. Despite these goals and ongoing research efforts, migration has not achieved widespread use on its own, yet rather put together with a data-affinity technique [Milojíčić et al., 2000].

So far, a number of researchers have reported the results from the performance

studies of different existing systems. Many successful techniques have been achieved on different legacy DSM implementations. These achievements have shifted the DSM picture towards a compelling, cost-effective platform for high performance computing. Maturity and variety of implementation techniques have been accomplished at different levels across the system hierarchy. Alongside, these different methods for pre-design performance analysis have been developed and successfully used for many comparison studies.

An interesting research has reported the indicators of performance, scalability, and long-term efficacy of generic design concepts used in four different hardware DSMs of the cache-coherent non-uniform memory access architecture [Grujić et al., 1996]. Using the generic concepts comparison, this study has made it feasible to identify the key factors for different performance aspects. Extending this method may make it possible to achieve the objectives set for this study as stated in Section 1.2.1. These objectives are to determine which factors are most influential on the degree of locality and to determine what combination of values in controllable factors might maximise performance stability. However, the comparison of generic DSM systems including software DSM mechanisms is still a challenge due to the large number of factors, levels and range that could impact the overhead of locality. A way to make this comparison study possible is to have an analysis tool that filters the essential features from some unnecessary details of the DSM implementations, so that the generic characteristics can be obtained and analysed through statistical methods.

1.3 Open Issues in Memory Analysis Tools

Simulation modelling is the most widely used methodology to evaluate new design alternatives of multiprocessor architectures. More than half of research works on the performance analysis of DSM systems presented in the previous sections reported results based on simulation[5]. A number of generic simulation frameworks have been developed to serve as a tool for performance studies of multiprocessor systems. Examples of these well-established works are Simics [Magnusson and et al., 2002], DSMSim [Thaker and Chaudhary, 2003], RSIM [Hughes et al., 2002], LIMES [Ikodinovic et al., 1999], Wisconsin Wind Tunnel II (WWT-II) [Mukherjee et al., 2000],

[5]Fifteen out of eighteen references presented their results based on simulations.

and HASE Shared Memory Multiprocessor Model [Coe, 2000]. These studies have shown increasing maturity in simulating the different levels of multiprocessor hardware architectures within a reasonable simulation performance. Nevertheless, recent advances in designing DSM systems have forced a re-evaluation of current multiprocessor simulation methodology in relation to two aspects: *model extensibility* and *verification applicability*.

As discussed earlier, machine architectures and DSM implementations differ usually in their organisation and operations. Therefore, the simulation structure would have to be flexible enough to be custom tailored to each individual DSM. The rapid development of software-based techniques in DSM implementations also puts a constraint on extensibility of a simulation model. Currently, DSM implementations aggressively exploit memory locality through software DSM techniques such as hidden pages, manager migration, double faulting, remote fetching, and prefetching [Pinto et al., 2003, Jégou, 2003, Baylor et al., 1997]. Most multiprocessor simulation studies, however, keep their focus on hardware activities and use a much simpler model of system software behaviour to alleviate accuracy against speed tradeoffs. Such simulations do not reflect the influence of software management features and consequently, may not only exhibit experimental errors [Desikan et al., 2001] but also limit the level of customisation available to explore new design alternatives at the software level.

Extending a model to cover new innovations requires careful attention to model verification as it normally takes more time to verify a model than to design it. Applying on-the-fly verification of a modelled component to a simulation requires the semantics of the component to be realised by the simulation program. So far, some techniques have been proposed to incorporate the specification-based verification to a simulation model such as, an automatic verification technique based on the discrete-event system specification formalism [Wainer et al., 2002], and an automatic generation of cache simulation models using the verifiable protocol specifications [Field et al., 1998]. However, integrating a verifiable specification with a component of multiprocessor simulation model, and using the specification semantic to direct the component behaviour is still a challenge.

1.4 Thesis Contributions

This thesis focuses on simulation modelling techniques to support the analysis of the overhead caused by memory locality in a variety of DSM systems. The research work capitalises on the challenges of developing a simulation skeleton with highly reusable components that are also provided with an automated behavioural verification. This approach permits the accuracy of simulation results to be maintained with the technological advances of the target architectures. An investigation of existing works is observed and these show that the exploitation of highly-reusable simulation components can be achieved by (a) decoupling construction of model components from a discrete-event simulation (DES) engine, and (b) using interpretation-driven techniques to provide an extensible framework for workloads of heterogeneous systems. An automatic technique to verify the simulated behaviour against the model semantics obtaining from the discrete-event system specification (DEVS) formalism has been studied [Wainer et al., 2002]. Following this approach, a similar concept is proposed to incorporate into a simulation model a specification-based verification module that allows behaviour verification to be carried out during a simulation run. Accordingly, four key operations have been incorporated into the proposed simulation model to recreate the characteristics of a DSM multiprocessor. These four key operations are listed below:

1. a mechanism to transfer control among the simulation kernel, the workload execution unit, and the objects implementing different model parameters;

2. an emulation of a multithreaded runtime environment to simulate parallel multithreaded workloads within the simulation address space;

3. a specification-based verification technique to simulate a vital component, the bus-based coherence protocol;

4. a hierarchical organisation of performance metrics.

This thesis demonstrates the potential of the proposed techniques via the implementation of a new simulation model for DSM systems called DSIMCLUSTER. The development of DSIMCLUSTER aims to achieve a simulated DSM platform that can

execute real programs. Each instruction of a workload program is taken from its object file and is translated by the emulation of the instruction execution steps inside the simulation. Therefore, the DSIMCLUSTERmodel is considered an interpretation-driven simulation. The model is based on the implementation of a discrete event simulation on top of a well-founded and broadly applicable DES engine developed at the University of Edinburgh, called HASE (Hierarchical computer Architecture design and Simulation Environment). In summary the thesis contributions are listed as follows:

1. presenting a simulation technique to promote *model extensibility* by

   (a) formulating the model using a well-formed specification,
   (b) separating the model implementation from the underlying DES engine, and
   (c) using the interpretation technique to drive the simulation for host machine independence;

2. proposing a solution that promotes *verification applicability* of the model, called a Specification-based Parameter Model Interaction (SPMI), to incorporate the behavioural verification into the simulation model in order to shorten the time spent on model verification that might delay the process to develop a new simulated environment;

3. conducting experiments and summarising the results from the study of the impacts of the architectural alternatives and DSM software management policies on DSM memory locality;

4. evaluating the proposed simulation techniques in terms of the obtained performance, flexibility and customisability.

Model Assumptions The research work presented in this thesis is based on the following assumptions.

1. This work studies the DSM memory locality exhibited from running the scientific workloads written in C and describing parallelism by using OpenMP, a standard shared memory specification. It is assumed that data required on each workload is available at local storage. Thus, data transmission prior to a workload execution is not considered.

2. This study focuses on the memory modules that are used for keeping data. Therefore, the effect of instruction caches, instruction translation lookaside buffers (TLBs), and the effect of accessing memory address holding code segments are not considered.

3. Accesses to instructions are always satisfied at the processors' instruction caches, *i.e.* 100% instruction hit is assumed.

4. Each SMP node of a DSM model uses a bus-based protocol to maintain intra-node cache coherence.

5. The modelled DSM system is dedicated. The interference from external jobs is not covered.

6. All SMP nodes of a DSM model are homogeneous. Therefore, for each DSM model, it is assumed that the configurations of processors, caches, buses, network interconnection, operating systems, and DSM policies are identical.

7. The interference of input/output modules, network congestion, and fault tolerant are not considered.

1.5 Thesis Overview

Following this introduction chapter, this thesis comprises four major sections. In the first section, the thesis work is introduced and placed in the context of related work on memory performance analysis and simulation modelling of multiprocessor architectures. This section includes two chapters. In Chapter 2 the background material of the thesis is presented. In Chapter 3 the related work and the solutions proposed in this work are discussed.

The second section of the thesis describes the modular design and implementation of a simulation model for DSM systems (DSIMCLUSTER) on top of HASE, a well-established discrete-event simulation engine. In particular, this section describes the decomposition of model components into two groups of *framework components* and *parameter objects* that promotes interoperability of simulation codes. The separation of DSM models from the DES engine is proposed to achieve the extensibility of the model framework. This section comprises two chapters. Chapter 4 explains

the model specifications by using the DEVS (Discrete EVent systems Specification) formalism prior to translating them into the HASE environment. Chapter 5 describes how DSIMCLUSTER models a distributed-memory multiprocessor and emulates the activities of DSM software in providing a single address space abstraction. It also presents the Specification-based Parameter-Model Interaction (SPMI) technique used in the DSIMCLUSTER to verify both the coherence protocol specifications and the emulation of coherence protocols during a simulation run.

The third section presents the potential of the DSIMCLUSTER as a framework to support experiments on memory locality analysis. This section is devoted to the analysis and discussion of simulation results obtained from various experiments over a subset of workloads from the NASA NPB 2.3 benchmarks. The section comprises two chapters. In Chapter 6, the process to obtain data access patterns from the workloads by using DSIMCLUSTER is presented. The chapter also presents experiments and results from the study on the impacts of different memory coherence protocols to memory locality overhead of each data access pattern. This chapter also demonstrates the relationship between locality overhead and the data ownership used in some coherence protocols. The DSIMCLUSTER model itself has been evaluated, based on the analysis of simulation runtime performance and the impact on experimental factors to the memory requirements. The scope of model extensibility and reconfiguration is also discussed.

Chapter 7 describes a set of experiments to identify the dominant factors of the DSM locality and to conduct a detailed analysis using these factors. The DSIMCLUSTER model has been extended to model sixteen different architectural alternatives to carry out a set of screening experiments. From these screening experiments, two factors that are most dominant to the DSM locality, the cluster architecture and the choice of coherence techniques used in the DSM manager, have been identified. The second set of experiments were carried out to analyse the impact of these two factors on memory locality of different DSM architectures. The results of the detailed analysis are summarised based on the experiments on DSM models of 128 processors.

The last section includes Chapter 8 which contains a summary of the thesis work, its implications, and future directions.

Chapter 2

Background

This chapter presents the background material of this thesis. Section 2.1 presents the concepts, mechanisms and architecture of a DSM system implemented on a cluster of symmetric multiprocessors, the target system of this research. In Section 2.2, the central concepts involved in a DSM to provide a single address abstraction and maintain a coherent view of memory across distributed replicas using different coherence policies are explained. Since actions undertaken by each of these policies can introduce system overhead and consequently cause locality problems, a number of optimisation techniques based on data- or computation relocation have been introduced to alleviate this limitation. These optimisation techniques are described in Section 2.3. Section 2.4 presents three different paradigms for programming a DSM system and their examples. In the last section of this chapter, three methodologies used in performance evaluation research on multiprocessor systems are presented, namely: analytical modelling, measurement and simulation. This section also presents a discussion based on the issues of complexity, viability, timing and accuracy that result in the potential of simulation modelling as a tool for the performance analysis of DSM systems.

2.1 Distributed Shared Memory System

A DSM system is one of the innovative architectures attempting to provide the key features of server systems, that is a combination of a high-performance, scalable multiprocessor platform and a programmable, easy-to-deploy computing environment.

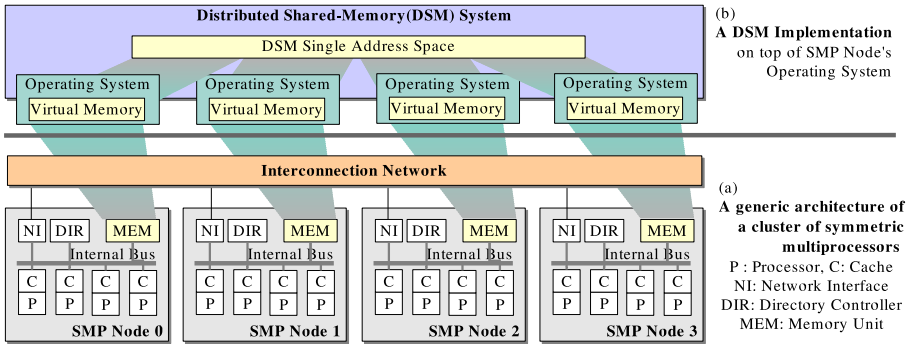


Figure 2.1: A basic architecture of a 4×4 DSM cluster.

As shown in Figure 2.1[1], the basic architecture of a DSM cluster is composed of two components, namely: (a) a distributed-memory multiprocessor machine (b) an implementation of a global shared-memory image on the top of the machine. This latter component, also called a DSM implementation, can be categorised into three styles: software, hardware and hybrid [Protić et al., 1995]. This thesis covers the generic characteristics of both components, and keeps the focus on the implementation of a software-DSM system on top of a cluster of symmetric multiprocessors. Note that from this point on, the term *DSM* is used to refer to the system in which the processors logically share the virtual memory address space (as depicted in Figure 2.1). In this section, the key characteristics of such a system are described and the scope of parameters of interest in this thesis work is also stated.

2.1.1 Cluster of Symmetric Multiprocessors

Commonly, under any DSM implementation, there is a distributed-memory multiprocessor machine. The model presented in this work is based on a particular type of distributed-memory architecture called a cluster of symmetric multiprocessors (so-called CLUMPs or a cluster of SMPs). Figure 2.2 depicts a basic architecture of a CLUMP in more detail, including the interconnection plane for input/output modules (I/O), directory (coherence) controllers and intra-node coherence bus. The cluster consists of individual SMP nodes and an interface to an interconnection network connecting all the nodes.

[1] An $m \times n$ DSM cluster describes an architecture that comprises m SMPs, and each of the SMPs has n processing nodes. It is also described as m clusters of n-way SMPs

In a cluster, each SMP shares a single memory unit via an internal bus. The implication of this is that the time to access memory is always uniform across local processors. This characteristic, called UMA (Uniform Memory Access architecture), is applied to intra-node communications in an SMP. However in a cluster configuration, a processor can also access alien memory (*i.e.* it can access memory belonging to any other SMP). In this case, the time to access remote memory is not uniform since access time strongly depends upon the physical distance from the processor and the SMP node where the required memory resides. This characteristic of inter-node communication categorises a cluster as a NUMA, Non-uniform Memory Access architecture.

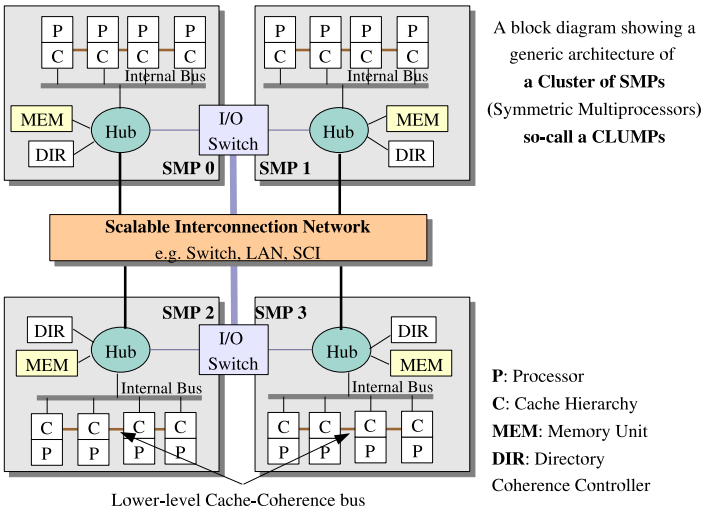


Figure 2.2: A generic architecture of the 4×4 CLUMPs.

The fact that all processors in a CLUMP can access the content of alien memories implies the logical image of a shared physical memory space on top of the system. In this case, any processor holding appropriate access rights can request data in any memory location. In the case that some required data is not present in local memory, such data will be *migrated* or *replicated* from its location into the requesters' local memory. During this process, multiple data replicas may reside in different SMP nodes, causing a coherence problem. Therefore, the design and implementation of protocols to maintain memory coherence is considered a key issue in terms of performance of a cluster implementation.

Coherence between data and its replicas residing in memory and cache is achieved

using a hardware controller based on two types of coherence protocol: a *bus-based* (or *snoopy*) protocol and a *directory-based* protocol. Generally, SMP nodes apply a bus-based protocol in a coherence controller to ensure coherent data among multiple, intra-node caches. For the inter-node caches and memories, a directory-based protocol is commonly used to maintain coherent content across the cluster. Basically, a *coherence protocol* ensures that multiple processors see a consistent view of memory by being aware of any memory-write commands requested from any of the participating processors. The coherence policy used in these two types of protocol is based on either an *update* or an *invalidate* technique:

- *write update*: In this case, the coherence protocol will update every shared copy by propagating the modified copy to replace any replica in the system. This technique is expensive as it can frequently be the case that one or more replica holders may no longer require to use this data.

- *write invalidate*: In this case, the strategy observed by the coherence protocol is based on communicating to all the replica holders that their copies are no longer valid. This technique can be expensive when there are multiple holders of the read-only replicas keep reading the data while one writer keeps updating it. In such case, once the replicas have been invalidated, the readers will request for data of the same location again.

Coherence protocols communicate among the replica holders by passing some *coherence messages*. This message passing can be an expensive approach if coherence messages have to be sent for each and every modification. To avoid this, coherence messages are only sent when a processor must see a value that has been modified by any other processor in the cluster. The policy of deciding when a coherence message has to be sent strictly depends on when the system should provide a consistent view of the shared memory. The policy to ensure this view is called *memory consistency policy*.

In table 2.1, each entry lists the summary of the architectural components of a CLUMP and states the parameters of interest for this thesis work. Note that the simulation model presented in this work does not observe I/O interference, thus the variety of I/O switches and their interconnection are not reflected in the table. This

Table 2.1: Parameters of interest in a cluster of Symmetric Multiprocessors.

Cluster of Symmetric Multiprocessors			
Symmetric Multiprocessors			
Component	*Parameters*	*Varieties/Options*	*Modelled*
Processor	Instruction Set	SPARC, IA32, MIPS, *etc.*	Yes
	Clock Rate	Varies	Yes
	Register Organisation	Register Files, Register Windows, *etc.*	Yes[a]
Cache	Organisation	Multiple Ways	Yes
	Associativity	Direct Mapped, Fully/Set Mapped	Yes
	Write Policy	Copy Back, Write Through	Yes
	Write Miss Action	Allocation, No Allocation	Yes
	Indexing (via TLBs)	VIPT, PIPT, VIVT[b]	Yes
Coherence	Cache Level	Multiple levels of Coherence	Yes
Protocol	Snoopy Protocols	Write Invalidate,	Yes
		Write Update	Yes
		Synapse, Illinois, Dragon	Yes
		MESI, MOESI, Berkeley, Firefly	Yes
	Directory Protocols	Central Invalidate	Yes
		Central Update	No
		Distributed Invalidate (SCI)	Yes
		Distributed Update	No
Memory	Unit of Addressing	Byte	Yes
	Byte Ordering	Big-, Little- Endian	No[c]
	Alignment	Word	Yes
Bus	Transaction or Split-transaction bus with Centralised Arbitration		Yes
Cluster Interconnection			
Component	*Category*	*Varieties/Options*	*Modelled*
Scalable	Static Network	Ring	Yes
Interconnection		Grid (2-D Mesh)	No
Network		K×K Grid (2-D Torus)	No
		(k,n) Cube	No
	Dynamic Network	Switch	No
		Crossbar switch	No
		Multistage IN (MIN)	No

[a] Only the register file has been modelled

[b] There are three indexing techniques using in both data caches and instruction caches: Virtually Indexed, Physically Tagged (VIPT), Physically Indexed, Physically Tagged (PIPT), and Virtually Indexed, Virtually Tagged (VIVT).

[c] The little-endian byte ordering is used.

table also serves as the route map to the remainder of the background material related to the CLUMPs architecture.

2.1.1.1 Symmetric Multiprocessor Architecture

In general, a single SMP is composed of a small or moderate number of identical processors referred to as *n*-way[2], which might be interconnected by a small bus or a different interconnection technology. In a configuration like this, each of the SMP nodes contains its own cache hierarchy, local memory, and I/O modules. Typically, processors in an SMP share the same copy of the operating system and have equal access to all peripheral devices (although they work independently on the jobs assigned). As mentioned earlier, an SMP has a UMA characteristic, *i.e.* all processors can access the local memory within the same access time. Therefore, in terms of function, all processors are identical or are *symmetric*[3]. The UMA characteristic of an SMP is sometimes referred to as *Cache Coherent UMA (CC-UMA)*. This is because the content of every cache in an SMP has to be kept *coherent* whenever they hold replicas of the same memory address.

In the following paragraphs four basic components of an SMP node (Figure 2.2) are further described including: processors, cache hierarchy, memory organisation and intra-node interconnection.

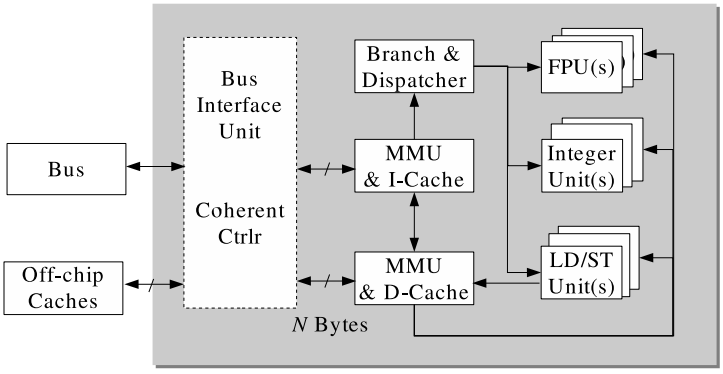


Figure 2.3: A block diagram of a generic processor.

[2]Recent commercial machines commonly offer the selection of 2-, 4-, or 8-way SMPs. However, larger scale models such as 16-, 32- to 64-way are also available in some models, *e.g.* in IBM POWER5

[3]In contrast, a shared memory multiprocessor can be *asymmetric* if there is one *master* processor in the system. In an asymmetric system, only the master processor executes the OS and has (exclusive) access to the I/O devices [Hwang, 1993].

Processor. The processors used in SMP machines are usually commodity processors available in both families known as complex-instruction-set computing (CISC), and reduced-instruction-set computing (RISC). The instruction set architecture includes a set of vector arithmetic and logical operations and/or a full set of scalar and memory reference instructions. Each processor contains a number of registers that normally support both 32- and 64-bit computations, and synchronisation features. Current processor speed in an SMP node ranges from 600 MHz to 1.7 GHz[4], with an average execution rate of 2 cycles per floating point instruction.

The block diagram shown in Figure 2.3 depicts generic components inside a processor core and their interconnection, including a group of arithmetic and logic units, load and store units, memory management unit, and interface unit. Arithmetic and logic units, *i.e.* floating point unit (FPU) and integer units, perform calculations on the corresponding data types. Load and store units (LD/ST) are used for transferring data between the processor and main memory. Contents of main memory (both data and instructions) are cached locally and normally managed by a separate memory management unit (MMU). Data cache (D-Cache) supplies computational data to the LD/ST units, while instruction cache (I-Cache) supplies the instructions to the dispatcher unit. The instruction dispatcher performs several routines to issue instructions to the appropriate units. These routines involve some basic operations (such as instruction decoding, solving address reference, and solving branch target locations) as well as some optimisation features such as speculative multi-threading, out-of-order execution, pipelining *etc*.

Operational units inside a processor core communicate to some off-chip units via the interface unit. This unit is normally composed of a bus interface logic to permit communication between processor core and main memory and other functional unit via a bus. This interface unit may also comprise a coherence controller for maintaining data coherence between on-chip and off-chip caches. The following paragraphs describe both cache organisation and bus interconnection further.

Cache Organisation. Most SMP systems use the memory caching technique to bridge the gap between processor speed and main-memory access time. As depicted

[4]For example, ranging from 600-MHz to 1.7GHz in IBM pSeries, 900-MHz used in SunFire 15K, and 1.13GHz in Cray X1 machine

in Figure 2.2, an SMP commonly uses private caches associated with different processors. Cache addressing models are categorised into physical address and virtual address caches [Hwang, 1993] depending on whether the cache content is accessed with a physical or virtual memory address. In case of a *physical address cache*, the cache is indexed and tagged with the physical address (*i.e.* physically indexed, physically tagged (PIPT)). Cache lookup must occur after address translation in the *Translation Lookaside Buffer* (TLB) residing in an MMU of the processor. Another addressing model, a *virtual address cache*, is a cache that is indexed with a virtual address and tagged with either a virtual or physical address (*i.e.* VIVT or VIPT). This model allows cache indexing to carry on in parallel with the memory address translation, thus giving enhanced efficiency and faster access. The major problem associated with a virtual address cache is *aliasing*. Since multiple processes may use the same range of virtual addresses, *i.e.* the same *alias*, different logically addressed data can have the same index/tag in the cache. One way to overcome this is to flush the entire cache when aliasing occurs.

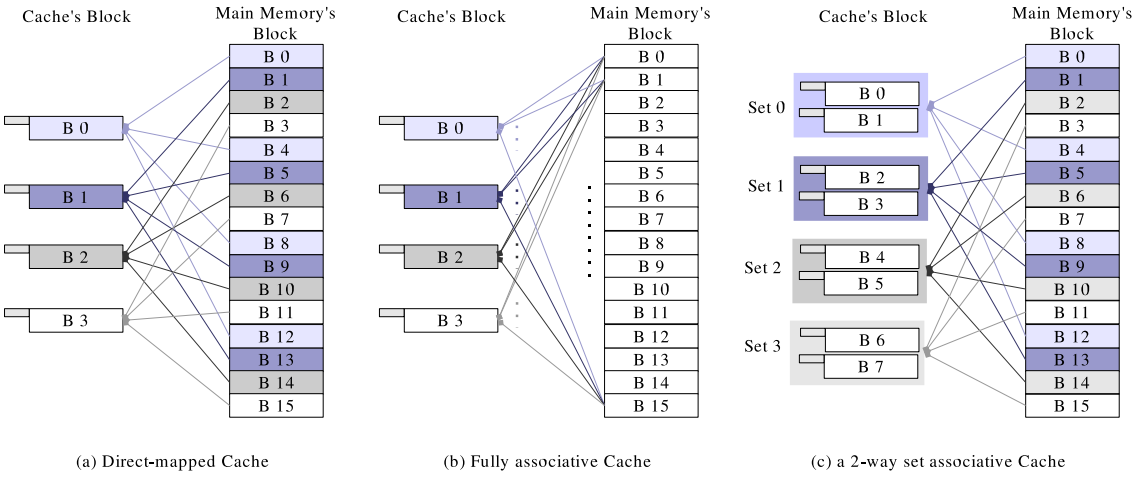


Figure 2.4: Cache-memory block mapping in three different organisations.

Cache content is divided into multiple blocks/lines of the same size. Each cache line has associated status flags identifying if the content is valid, invalid, or has been modified. Whether the requested address is found in the cache is referred to as cache *hit* or *miss*. When the machine is started up, the cache content is empty, so the first set of accesses are *compulsory* cache misses (so-called *start-up misses*). After a miss, the whole cache block is loaded with data from memory resulting in a hit on the

next access to the same address. If a cache miss occurs when there is no cache line available, *i.e.* a *capacity miss*, one of the cache lines will be evicted to accommodate the requested data. One of four possible replacement policies is normally used to choose a victim line. These policies are: choosing the one that has been loaded first (First-in First-out (FIFO)), choosing the one that has been least recently used (LRU), choosing the one that has been least frequently used (LFU), or choosing randomly. Some cache architectures also provide a *victim cache* for keeping these victim lines to alleviate the case of conflict misses (*i.e.* cache misses caused by two lines sharing the same cache location).

On a write hit, data can be written to cache and passed through to the main memory immediately in a *write-through* (WT) cache, or delayed until the block is replaced in a *write-back* or *copy-back* (CB) cache. When data of a CB cache has been modified, the cache line status will be marked as *dirty*. In case of a write miss, both WT and CB caches can use either allocation or non-allocation policy. *Allocation on write miss* means that once a write-missed line has been written to memory, the cache will read this content and allocate a cache line for it. In contrast, the no-allocation on write miss policy does not allocate a cache line for a write-missed line. The write command is passed directly through the main memory, thus a following read to this location will also cause a cache miss.

A CB cache normally has a *write buffer* to hold the modified data that will be transferred to main memory periodically or when the cache lines are *flushed*. Data transferred between cache and memory is conducted in units of cache blocks or cache lines. Blocks in caches are called block frames in order to distinguish them from the corresponding blocks in main memory. There are three different organisations to map cache block frames to memory block. Figure 2.4 illustrates each of these organisations including (a) direct mapping, (b) fully-associative mapping and (c) *n*-way set associative mapping.

Most commercial SMP nodes use two-level caching per processor. The primary or level-1 (L1) cache is normally separated into instruction and data caches and implemented on-chip. The level-2 (L2) cache is normally unified and implemented off the processor chip. Total size of caches per each processor range from 2 to 8MB.

Memory Organisation. Each SMP node globally shares the local main memory (as depicted in Figure 2.2) which can be observed as a cache domain. Recent commercial SMP modules provide main memory with sizes ranging from 16 to 96 GB[5]. In an SMP node, the main memory is normally built with multiple parallel RAM modules[6] allowing pipelined access in parallel, so-called *interleaving*. Two types of interleaved memories are used in an SMP node, namely: the low-order and high-order addressing, shown in Figure 2.5. *Low-order interleaved memories* distribute contiguous addresses across the memory modules, allowing a block of consecutive words to be accessed in parallel. On the other hand, *high-order interleaved memories* assign contiguous memory space to the same memory module, allowing accesses to memory addresses with a certain striped size to be done in parallel.

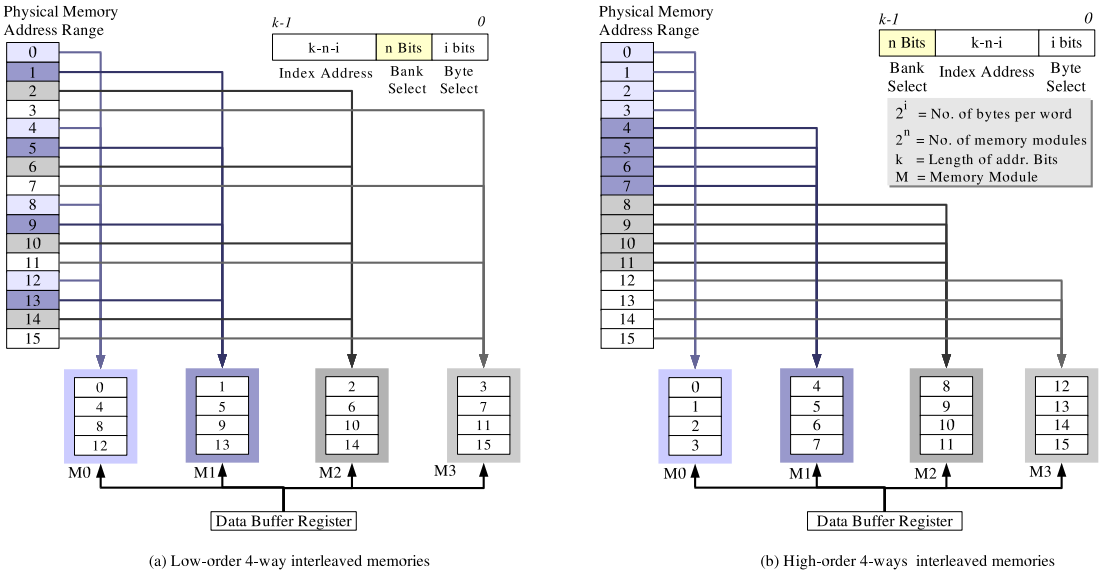


Figure 2.5: Mapping of physical address space on interleaved memories.

In a low-order interleaved memory system, access to the *m* memory modules can be overlapped in a pipelined fashion. This approach subdivides a memory (access) cycle (called the *major cycle*) into *m minor cycles*.

Data transferred from memory on each access is referred to as a *memory word*. Commonly each memory word comprises a set of data bits and error correction codes. Some SMP models support accesses to half memory words for operations on 32-bit

[5]Data from IBM pSeries, Cray X1, SunFire series, and SGI Origin
[6]Available in both RDRAM and SDRAM-based technology

registers. Local memory is accessible by all processors on the SMP node, processors on other nodes, and all I/O devices. Local memory latency is completely uniform for all processors within a single node. However, memory bandwidth also depends on network traffic and I/O transfer.

Some models that support redundancy (Cray X1, for example) allow local memory to be operated in degraded modes. These modes use only a quarter or a half of physical memory on a node to tolerate the loss of memory chip or the failure of memory cards.

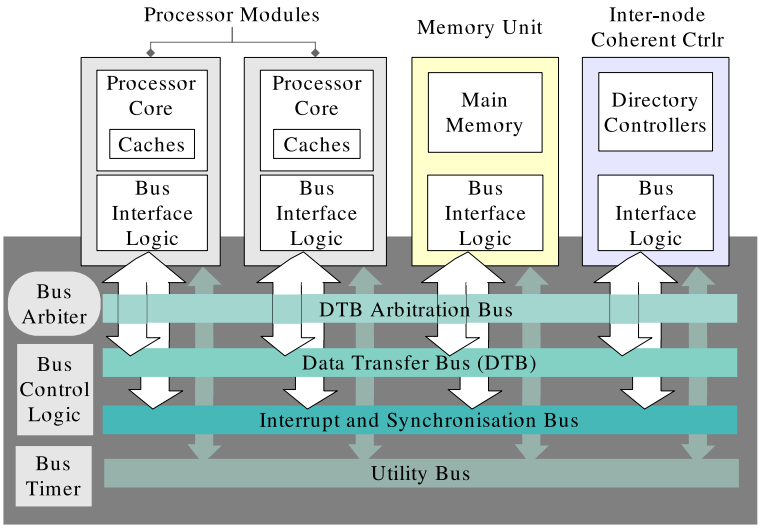


Figure 2.6: Typical system interface to a backplane system bus.

Intra-node Interconnection. Processors, caches, main memory and other functional modules of an SMP node are interconnected in a tightly coupled configuration using a backplane bus system. Figure 2.6 shows a typical interface interconnecting the SMP components (processors, caches and memories) together using a backplane bus system. Data are broadcast to the target components via a *data transfer bus* (DTB). DTB is a collection of signal lines comprising a number of address lines, data lines, and control lines carrying recipient address, data, and command respectively. Address modifier lines may also be used to define special addressing modes. A component can request to use the DTB by using a DTB arbitration bus. This process, called bus arbitration, is controlled by a bus arbiter, *i.e.* a hardware circuit that grants control of the DTB to one requester at a time. Figure 2.6 also depicts the interface of

SMP components to an interrupt and synchronisation bus, and to a utility bus. The interrupt bus is used to send interrupt signals, and some dedicated lines may be used to synchronise some parallel activities among the processors. The utility bus comprises dedicated lines sending periodic clock signals.

Fundamental to the design of the backplane bus system are timing protocols for bus allocation and operational rules to ensure orderly data transfer. The design goals are commonly to minimise the time of communication between components, and minimise the interference of bus operations to the attached components' internal activities. A bus can be categorised by its timing protocols as a *synchronous* or *asynchronous* bus. The operations that takes place when a component transfers data to a receiver on the bus (*bus transaction*) take place at fixed clock edges in the synchronous bus. The clock signals are broadcast to all components, and the slowest connected device determines the clock cycle time. In an asynchronous bus, all bus transactions are hand shaken between sender and receiver.

Using a backplane system bus, data is routed (by a bus control logic) to memory without going through the processors. The interconnection to I/O channel controllers is beyond the scope of this research.

2.1.1.2 Bus-based Cache Coherence Protocol

The fact that each node in an SMP possesses a local cache implies that all processors holding appropriate rights to access the same memory segment can hold data replicas. Possibly, one of these replicas may be locally modified by its holder, resulting in the most-updated (valid) content being different from the other replicas. To maintain coherence between these replicas, each of the SMP's processing nodes[7] contains a coherence controller. Mechanisms to maintain cache coherence are implemented in accord with the set of rules and agreements defined by a *cache coherence protocol*. As mentioned at the beginning of this section, a coherence protocol can be categorised into a *bus-based* (or *snoopy*) protocol or a *directory-based* protocol. This is regarding which approaches the protocol uses to distribute coherence messages (*i.e.* information about updates on replicas) to the other processors. Generally in an SMP's processing nodes, coherence controllers ensure coherent data among multiple, intra-

[7] A processing node refers to a physical module or processor core comprising one or more processor(s) and a cache hierarchy

node caches by applying a bus-based protocol, *i.e.* using the system bus to broadcast a coherence message and snooping on the bus to get updated information.

Normally in a cache hierarchy of an SMP node, caches at the outer-most level (*territory caches*) are kept coherent as shown in Figure 2.7 (a) and (b). However, in an SMP system with a shared cache[8], control of cache coherence can be applied either to the level before the shared cache, or to the level before the main memory. Figure 2.7 (c-d) depicts two possible architectural designs of a 2×2 SMP with one and two shared cache(s), respectively. The coherent caches are observed at the cache level before the shared cache in Figure 2.7 (c), while at both levels before the shared cache and before the main memory in Figure 2.7 (d).

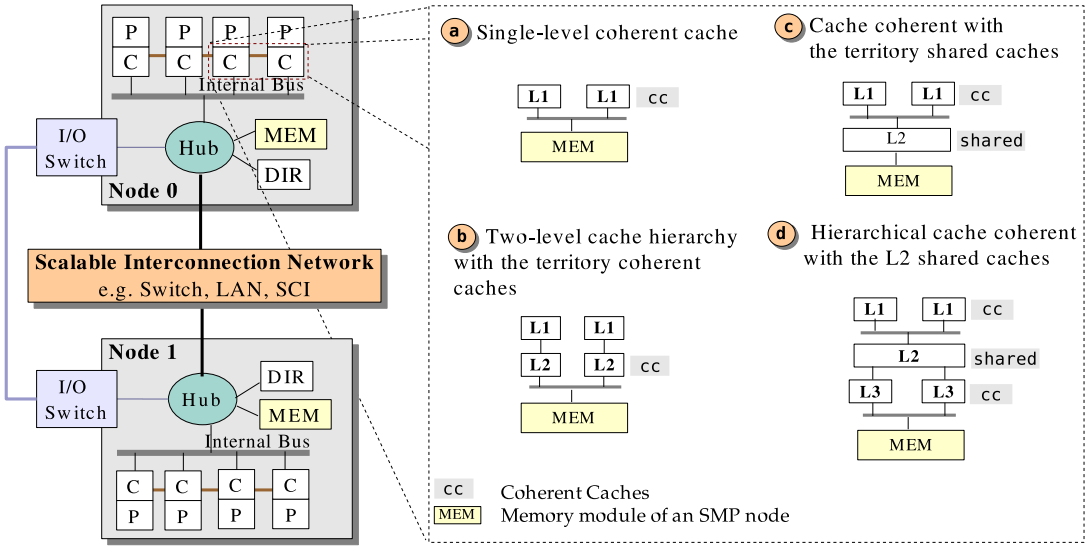


Figure 2.7: Examples of cache-hierarchy design on a 2×2 cluster of SMPs.

All example configurations of coherent caches in a 2×2 cluster of SMPs shown in Figure 2.7 use a snoopy protocol. In this case, each attempt to write data on a coherent cache is broadcast to all processors in the system via the bus. All of the processors on the bus must constantly monitor the bus for write transactions. Appropriate actions will be taken if a monitored write is to the same memory segment of a replica cached locally. Actions of a coherence protocol are normally based on two techniques, update or invalidate, as described in Section 2.1.1 (page 20). Snoopy protocols can be further categorised by the source of the new data for a cache line. The source of new data

[8]The term *shared cache* refers to a configuration of caches in one or more levels where their contents are united into single piece and shared among all processors.

can be either a remote cache or the main memory. Table 2.2 lists some examples of snoopy protocols categorised by update-or-invalidate techniques, source of new data, and the number of states. The description, state-transition diagram and detailed specification for each of these protocols are presented in Appendix A, Section B.2.

Table 2.2: Bus-based or snoopy cache coherence protocols (extended from [Flynn, 1995a]).

		Number of States			
		3	4	4	5
	Source of a new cache line	Invalid Shared-Clean Private-Dirty	Invalid Shared-Clean Private-Dirty Shared-Dirty	Invalid Private-Clean Shared-Clean Private-Dirty	Invalid Private-Clean Shared-Clean Private-Dirty Shared-Dirty
Invalidation	Memory	Write-Inval.		MESI[a]	
Invalidation	Cache-to-cache data movement	Synapse Berkeley	Illinois		MOESI
Update	Cache-to-cache data movement			Firefly	Dragon

[a]Different varieties of MESI and MOESI protocols have also been used and custom defined the commercial machines.

2.1.1.3 Memory Consistency Model

Because coherence protocols can cause significant bus traffic, a number of different memory consistency models (so call *consistency policies*) have been developed to specify when it is necessary for coherence actions to take place. A *memory consistency model* determines the time at which a processor will see the latest modified value of a shared datum by defining the desired order of memory operations to be performed on a shared address space. Similar to the scenario of multiple threads working together by interleaving on a uniprocessor, multiple threads working in parallel on a multiprocessor machine should also maintain the desired ordering of accesses to a shared memory. Different consistency models define the desirable order of memory operations differently based on assertions required for each memory access operation.

The intuitive model describing the formal specification of the *desired ordering* is defined as *sequential consistency* (SC) by Lamport [Lamport, 1979] as follows:

> "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program."

Essentially, SC requires that writes to shared memory are visible 'immediately' at all processors. Therefore, whenever a shared datum is written based on the SC model, coherence actions have to be taken immediately to maintain the coherent value across data replicas.

The fact that the strict order defined by the SC model requires a number of unnecessary communications has motivated the development of a number of more relaxed models. These relaxed models add some extra *synchronisation* operations to a shared memory access. Instead of conventional loads and stores, each access to shared memory is further divided into *acquire* and *release* accesses. Before each memory access, a process makes a request to acquire the memory content. Once the acquisition is granted, the process performs its memory operation, before submitting a release signal to the memory controller.

Adve and Gharachorloo [Adve and Gharachorloo, 1996] discussed a number of consistency models and their effectiveness in practice. The paper presents a way to categorise memory consistency models based on two key characteristics. Firstly, how the models relax the program order requirement when multiple processors attempt to access different memory addresses. Secondly, how the models relax the write atomicity requirement when multiple processors want to access the same memory address. These two key characteristics have been further refined into five relaxation cases, as shown in Figure 2.8.

Figure 2.8 (1-3), presents the relaxation cases of the first characteristic, *i.e.* when two processors attempt to access different memory addresses in a shared memory. These three figures describe the order of *read-after-write* (RAW), *write-after-write* (WAW), and *read-or-write after read* (RWAR), respectively. Figure 2.8 (4-5), presents the relaxation cases of the second characteristic regarding the write atomicity of accesses to the same memory address. Figure 2.8 (4) shows an example of a *Read Other's after Write-Early* (RdOth-WrEr), when a consistency policy allows a read

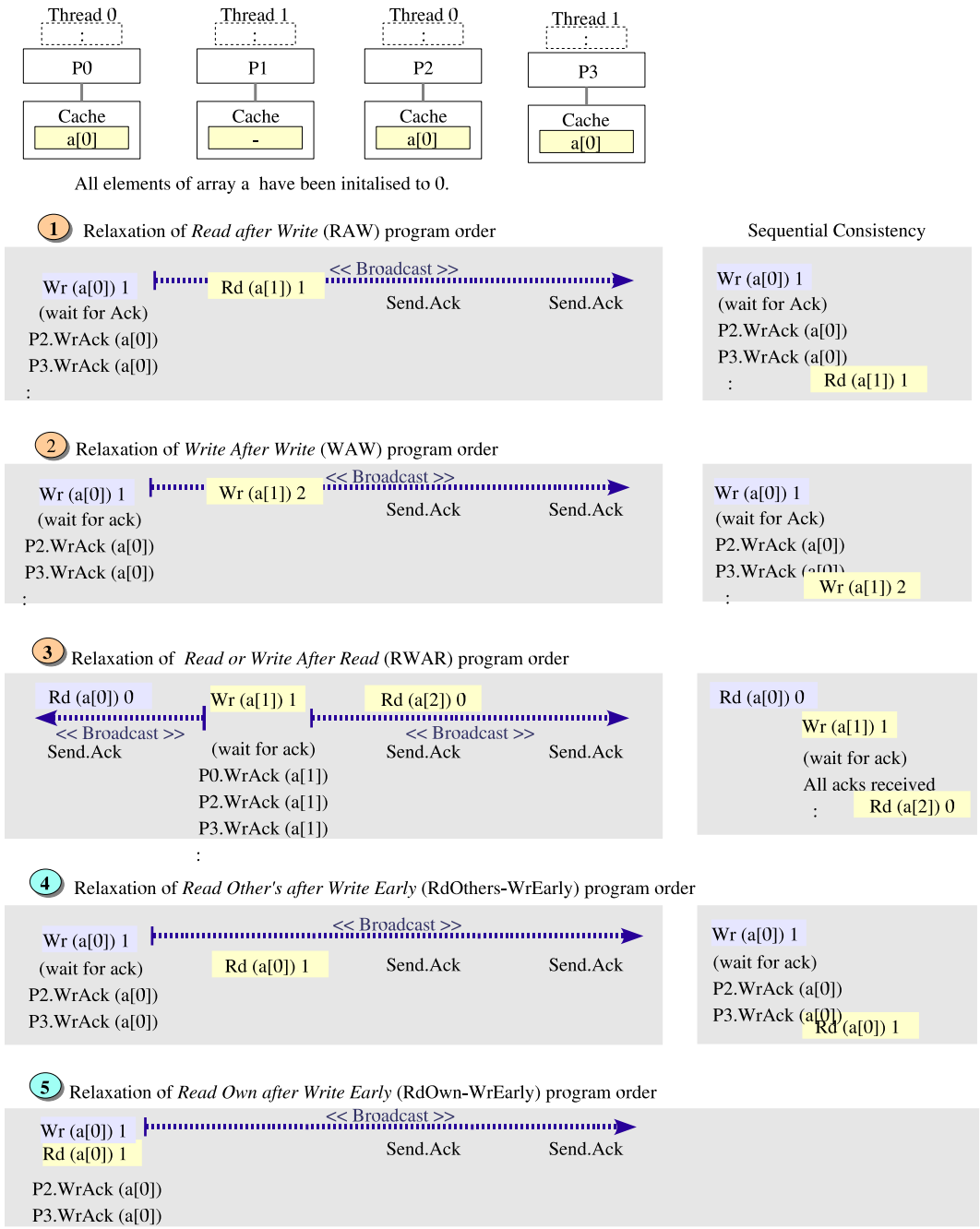


Figure 2.8: Scenarios of five relaxations allowed by memory models.

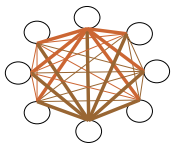
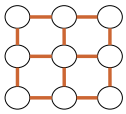
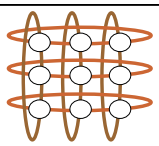
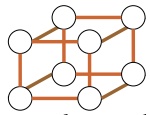
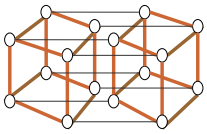
Table 2.3: Categorisation of memory consistency models (from [Adve and Gharachorloo, 1996]).

Consistency Policy	Accessing different location			Accessing the same location	
	RAW	WAW	RWAR	RdOth-WrEr	RdOwn-WrEr
Sequential Consistency					√
Total Ordering	√				√
Processor Consistency	√			√	√
Partial Store Ordering	√	√			√
Weak Ordering	√	√	√		√
Release Consistency(sc)	√	√	√		√
Release Consistency(pc)	√	√	√	√	√

to return the value of another processor's write before the write is made visible to all processors. Figure 2.8 (5) shows an example of a *Read-Own after Write-Early* (RdOwn-WrEr) relaxation case, when a consistency policy allows a processor to read the value of its own previous write before the write is made visible to other processors. In the figure, all relaxation cases are demonstrated against the SC policy that assumes no relaxation orders except RdOwn-WrEr.

So far in this subsection, concepts and functionalities of basic architectural components of an SMP node have been described. Recently, commercial SMPs have been packaged in modules of three types: single board, backplane and single chip[9]. The most cost-effective package is a single board SMP [Bell, 1999] which comprises a number of processors and memory mounted on one printed circuit board. A single chip SMP has now been delivered with the size of 2- and 4- processors on a die. In 2002, research has projected that by 2008, a $400mm^2$ chip would be able to accommodate 16 processors running at 6 GHz or higher [Nair, 2002]. Recently, an SMP with dual-processor-cache modules interconnected with a memory and other functional units via a scalable backplane switch is a common solution. The following subsections present the architecture of a system which uses such a backplane interconnection to form a cluster of SMPs, and the issue of data coherence across the cluster using directory protocols.

Table 2.4: Static network topologies and characteristics.

	Connection	Topology	Characteristic[a]
(a)		Fully Connected Network	$n = 1$
(b)		Linear Array	$n = N - 1$
(c)		Ring (Linear Array with closure)	$n = N - 1$
(d)		Grid (2-D Mesh)	$N = k \times k$ $n = k + 1$
(e)		2-D Torus or $k \times k$ grid with closure	$N = k \times k$ $n = k - 1$
(f)	 *binary three-cube*  *binary 4-cube*	(k,n) cube or k-ary n-cube with closure and bidirectional channels	$N = k^n$ or $n = log_k N$

[a]The network characteristic is described by the total number of nodes (N), the number of node in one dimension (k), and the network diameter (n).

2.1.1.4 Cluster Interconnection

Generally, the term *cluster* is identified as a group of interconnected computers and the coordinated use of them. In CLUMPs, SMP nodes are typically clustered on a scalable high-speed interconnection to achieve scalability. Connections between nodes are structured in some orderly way in order to provide efficient inter-node communication. Many different types of structures, or *topologies*, can be categorised by the relationship between nodes in the network for both static and dynamic networks. In a *static network*, the topology or the interconnection structure between nodes is fixed. Two nodes in a static network can simply communicate via the fixed path(s) or channel(s) resulting in predetermined bandwidth. However, fixed connectivity in a static network can limit the scalability and the number of concurrent communications among different node pairs. Alternatively, a *dynamic network* topology offers both scalability and adaptability by allowing the path(s) between nodes to be altered to establish connectivity. A dynamic network provides configurable routing that can facilitate concurrent communications. Thus an improvement in network bandwidth can be achieved.

Static Networks. Static networks are usually direct networks that can be subdivided into ones with or without preferred nodes. Static networks with preferred nodes are those in which communication cost and accessibility to all nodes in the network are not equal. A linear (array) network is one example of this type (Table 2.4 (b)). A linear network offers some advantages to communication between neighbouring nodes, thus having node preference. In contrast, a linear network with closure (forming a ring topology (Table 2.4 (c))) removes the node preference as the connection enhances the case of communications between the two-end tiers. A configuration like this, where every node has equal right to access all other nodes in the networks, is referred to as a static network without node preference. A cluster of SMPs is normally networked using one of these no-node-preference topologies such as, ring, 2-D mesh, or cube (see table 2.4 for the interconnection structure).

Message transmission can be done using either the successive or relative method. In case that a message must be transmitted via any intermediate nodes as shown in Figure 2.9, the successive method assumes that each intermediate node must buffer

[9]Different packaging is due to the issues of electrical signalling and shared bus bandwidth.

the whole message before it starts forwarding it to the next node. This method is known as *store-and-forward* routing. An alternative to buffering the whole message is to use the relative method, as in the *wormhole routing*. This relative method allows an intermediate node to buffer only a minimal amount of information. The buffered amount is normally only long enough to decode the message header to determine its destination. Figure 2.9 depicts a scenario where a source node is sending a message subdivided into four portions to a destination via two intermediates nodes. Figure 2.9 (a) shows the sequences of transmission and the buffer length at the two intermediate nodes if using store-and-forward routing. Figure 2.9 (b) shows message transmission using wormhole routing; the transmission time and the buffer size are noticeably smaller than that of store-and-forward routing.

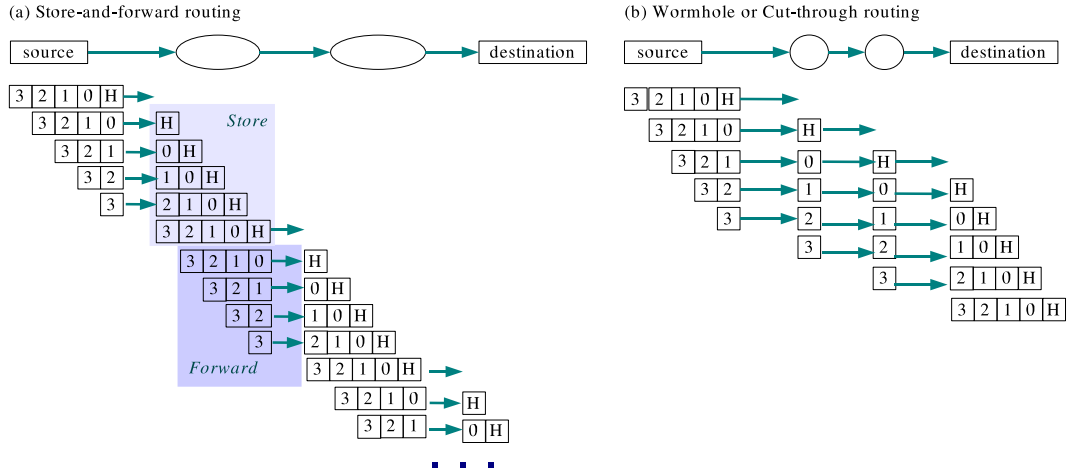


Figure 2.9: Message transmission in store-and-forward and wormhole routing.

For each network, the *distance* (*i.e.* a number of hops used to establish a connection between two nodes) may be fixed or variable. In either case, the largest distance in the network, the *diameter*, is normally used to describe the communication in the worst case. In general, the number of nodes (N) and the diameter (n) are determined by the topology used (as summarised in Table 2.4).

Similar to the diameter, the characteristics of the links in a network play an important role in terms of cost and performance. Links are characterised in three ways: the cycle time of the link (T_{ch}), the width of the channel (w), and the directionality of the channel (*i.e.* uni- or bi- directional). Consider a message (having l-bit data and h-bit header length) travelling between two adjacent nodes at distance d apart. The

total delay time taken to accomplish the communication with different routing policy can be described by the following models [Flynn, 1995b]:

$$T_{store-and-forward} = T_{ch}[d \times \frac{l}{w} + d \times h] \tag{2.1}$$

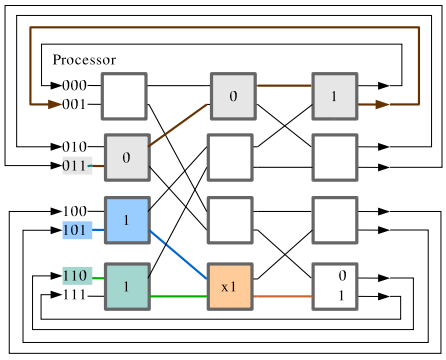
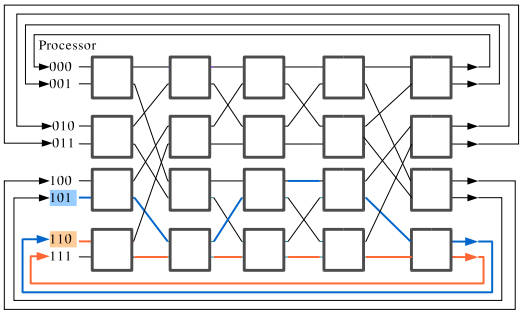
$$T_{wormhole} = T_{ch}[d \times h + \frac{l}{w}] \tag{2.2}$$

Consequently, the delay of package transmission between two tiers can be derived from the product of the total delay time between two adjacent nodes (presented above) and the number of hops required to establish the connection. Integrating these equations into the simulation model, the predetermined network bandwidth without contention can be derived. Thus, in a simulation model of a large-scale multiprocessor system, the detail of physical interconnection could be abstracted in a study where the interconnection is considered as a nuisance factor. Static network interconnections provide a fixed connectivity and thus restrict the choices of routing for transmitting packages between two tiers. As mentioned earlier, the fixed connectivity in a static network limits the scalability and number of concurrent communications. Therefore, for a highly scalable system, the dynamic network topology is an alternative for implementing a cluster interconnection plane.

Dynamic Networks. Dynamic networks use a switching mechanism to dynamically establish connection between two nodes. The basic element in a dynamic network is a crossbar switch connecting one of the k sources to any of another k destinations. A crossbar switch can connect multiple sources to multiple destinations concurrently, as long as these sources do not refer to the same destination. Normally, a dynamic network is composed of multiple crossbar switches connecting between nodes. This configuration is referred to as a *Multistage Interconnection Network* (MIN) as it reflects that a message will traverse multiple steps before reaching its destination. Example of two topologies of dynamic networks, shown in Table 2.5, are the baseline and beneš network.

Many researches have analysed various network configurations (as discussed in [Flynn, 1995b]). Two important issues in different developments are (a) a preference for direct (static) networks over dynamic networks and (b) a preference for small dimension over higher-order cubic. The first preference is due to cost and connection affinity of

Table 2.5: Example of two dynamic networks and their characteristics.

	Characteristics
	*topology*: baseline network *examples*: Delta, Omega, Banyan, *etc.* *diameter*: $\lceil log_k N \rceil$ *characteristic*:Simple switch encoding (1=upper link, 0=lower link) Network blocking is possible. *note:* As shown in the figure, if P_{101} wants to send a message package to P_{110} (blue line) while P_{110} is sending message to P_{111} (green line), network blocking occurs at the *x* switch.
	*topology*: beneš network *diameter*: $2\lceil log_k N \rceil - 1$ *characteristic*: *Non blocking network if reconfigure *note*: As shown in the figure, if P_{101} wants to send a message package to P_{110} while P_{110} is sending a message to P_{111}, different routes can be taken. (as shown in blue and red lines)

the interconnection, while the second preference is based on the maximum wire cross section and the delay on wiring transit[10] when the network is mapped onto many dimensions. Following these preferences, in this thesis work, an abstraction of a static network with ring topology has been modelled. The performance of this interconnection network is measured in terms of network transmission time derived by using the equations 2.1 and 2.2.

2.1.1.5 Data Coherence across a Cluster

Many SMP clusters now provide a shared physical address space across the SMP nodes. This implies that data replicas of the same memory location can be cached at different SMP nodes causing the coherence problem among inter-node caches. In Section 2.1.1.2, the cache coherence issues among intra-node caches of an SMP have been described. These intra-node cache-coherence protocols rely on the broadcasting of all transactions on the bus, yet inherently restrict the system scalability due to the limitation of bus capacity. Consequently, in a large-scale clustered system like CLUMPs, data coherence across the cluster is normally kept by using a *directory-based protocol*.

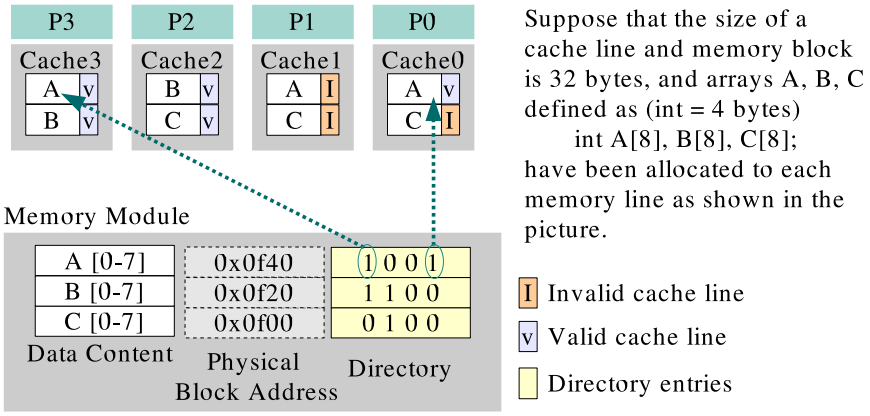


Figure 2.10: Centralised directory structure.

The coherence directory is a storage space holding a list of caches that have replicas of a particular line. Two types of structures have been used for implementing

[10]The wiring transit refers to the activities and routing mechanisms to allow a connection to transfer from one dimension to another in a multi-dimensional static network.

coherence directories: the central directory and distributed directory. In *central directory*, the directory content keeps information for all physical memory lines about which cache(s) are holding their replicas. In fully-mapped central directories (Figure 2.10), each entry in the directory (representing each memory line) contains a series of *presence/absence* bits, one mapping to each cache. The directory content is kept up to date in association with accesses to the physical memory address.

In a *distributed directory*, the directory is kept at both memory and at each processing element module (processor-caches module). Entries in a distributed directory are kept based on cache line ownership. As shown in Figure 2.11, the directory entries in main memory will point to the cache that last modified the data line (*i.e.* the owner of this line). Following the link, the owner cache also holds a directory pointer to the next cache that has a replica of the same line. This link continues until the last cache which holds a null pointer (*i.e.* the terminator of the link). Figure 2.11 shows a distributed directory implemented using a single linked list. An alternative approach to this is a double-linked distributed directory structure that is used in the SCI IEEE standard specification.

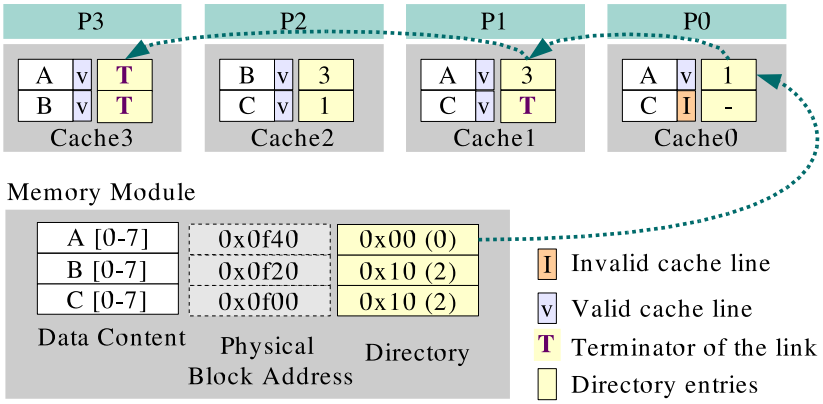


Figure 2.11: Singly-linked distributed directory structure.

There are four general approaches to maintaining data consistency using directory-based protocols. These approaches are distinguished by what action a protocol uses to deal with data replicas in main memory and remote cache(s), once a memory write is observed. Table 2.6 shows four directory-based protocols categorised by (a) whether the protocols update the main memory, once a memory write is observed, and (b) whether the protocols invalidate or update all replicas in the remote caches.

Table 2.6: Four approaches in directory-based coherence protocols (adapted from [Flynn, 1995b]).

Memory	**Remote Caches**	
	Invalidated	*Updated*
Not Updated	Scalable Coherence Interface (SCI) Scalable Distributed Directory (SDD)	Distributed-Directory Update Protocol (DD-UP)
Update	Central-Directory Invalidate Protocol (CD-INV)	Central-Directory Update Protocol (CD-UP)

Scalable Coherence Interface (SCI). The Scalable Coherence Interface is an IEEE interface standard for high performance multiprocessor systems that supports a coherent shared-memory model scalable to systems with up to 64K nodes [IEEE-SA Standards Board, 1993]. The SCI standard encompasses two levels of interface, the physical level and the logical level[11]. The physical level is beyond the scope of this thesis and its related information can be found in [IEEE-SA Standards Board, 1993]. One of the issues defined at the SCI logical level is the cache coherence mechanisms.

SCI uses a double-linked list, distributed-directory coherence protocol for any DSM configurations (structure shown in Figure 2.12). Each shared line of memory is associated with a distributed list of processors sharing that line. All nodes with cached copies participate in the updating of this list. Every memory line that supports coherent caching has an associated directory entry that includes a pointer to the processor at the head of the list. Each processor cache-line tag includes pointers to the next and previous nodes in the sharing list for that cache line. Thus, all nodes with cached copies of the same memory line are linked together by these pointers.

SCI communicates among replica holders so as to *invalidate* their copies if any of the holders has requested a write access. The write requester has to wait for all acknowledgements to arrive first before performing the write. The time waiting for the acknowledgement is also used to update the memory directory. Further information can be found in [IEEE-SA Standards Board, 1993].

Note that in a multiprocessor system, data can be replicated in two ways (1) horizontally across multiple instances of the same memory level and (2) vertically across the hierarchy depth of multiple levels of memories. Different memory management units are responsible for maintaining coherence across memory levels, and different

[11] Both levels define operations over distances less than 10 metres.

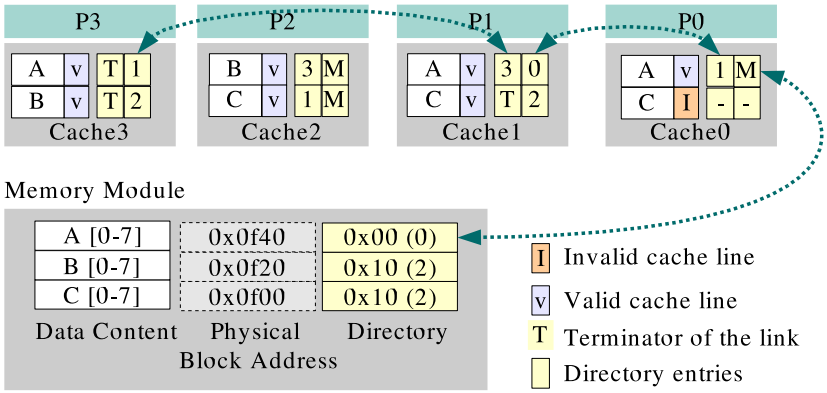


Figure 2.12: Doubled-linked distributed directory structure.

coherence controllers work in collaboration with the MMUs to maintain coherence across multiple replicas at the same memory level.

2.1.2 Applications and Operating System

In the previous subsections, the concepts and architecture of a cluster of SMPs have been described. This machine architecture serves as a computing platform for applications or workloads. Typically, an application workload running on a CLUMP has been a mix of development tools, engineering and scientific applications [Bell, 1999]. Each workload is an independent program having different memory demands and exhibiting different data access patterns. Instances of multiple workloads may be running concurrently on different processors in a DSM system (multi-programming). Alternatively, instances of a parallel workload, in the form of either threads or processes, may be running on several processors of the same nodes or across different nodes (SPMD). Applications require different memory resource bandwidth based on their memory access patterns. Workload characterisation is derived from studies of applications that are typical for the architectures. The representative characteristics for each type of application are used to generate different benchmarks for the purpose of performance studies.

This subsection presents the background material related to how application workloads interface to a CLUMP machine and how multiprocessor operating systems support this process. The key features, concepts and parameters of interest regarding this

are summarised in Table 2.7. The follow subsections present the explanation of the content listed in this table.

Table 2.7: Parameters of interest in the application interface and operating system.

Application Workload and Features of Multiprocessors OS			
Component	*Category*	*Benchmarks*	*Modelled*
Application Workload	scientific applications	Splash2, NPB	Yes
	development tools		No
	commercial applications	OTLP	No
Component	*Category*	*Varieties/Options*	*Modelled*
Workload Execution Model	Parallel SPMD	Multiple threads	Yes
		Multiple processes	No
Binary File Format	ELF		No[a]
	XCOFF		No
Process/Thread Management	Creation and termination		Yes
	Dynamic Context Switching		Yes
	Scheduling	Static	Yes
Memory Management	Segmentation		Yes
	Paging		Yes
	Virtual Memory		Yes
File System Management			No
I/O Management			No

[a]Custom defined format is used in the proposed model by obtaining and manually editing the data and code segments of ELF object dump.

2.1.2.1 Application Binary Interface

In many multiprocessor systems, applications are executable binary files written in the standard Executable and Linking Format (ELF). Generally, legacy source programs are compiled into object files before they are linked into one ELF executable file (as the steps shown in Figure 2.13). The ELF was originally made available by Unix System Laboratories as part of the application binary interface for 32-bit applications, and has become the standard in file formats [TIS Committee, 1995]. It is used as the default binary format on many operating systems[12] such as Linux, NetBSD, Solaris 2.x, Unix System V Release 4 (SVR4), Cray UNICOS/mp, and SGI IRIX 6.5[13].

[12]Except in AIX (IBM), in which the XCOFF (eXtended common object file format) is used.
[13]The Cray UNICOS/mp uses the kernel based on IRIX 6.5 that is known to be an SVR4.

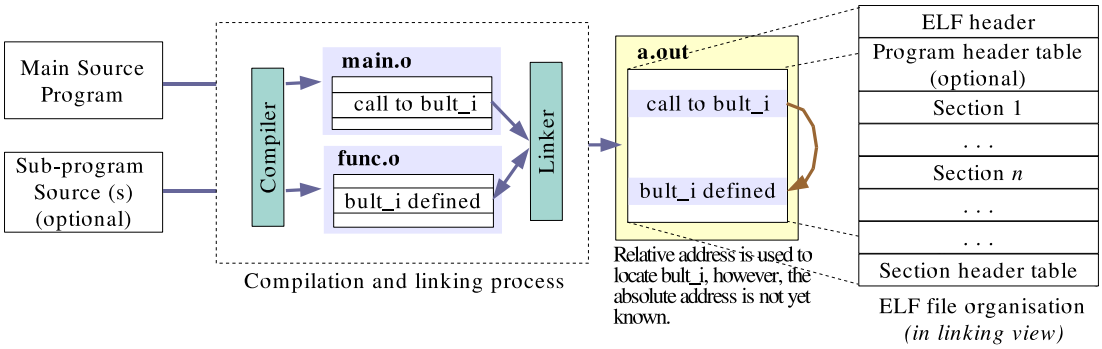


Figure 2.13: Steps in compilation and linking process and the `a.out` file organisation.

The main advantage of ELF over the earlier binary file formats (such as COFF, for example) is that control data in an object file is represented in platform independent format, allowing compatibility across multiple platforms and architectures of different size. As shown in Figure 2.14, the ELF representation comprises ELF header, program header table, section header table, and ELF sections. The ELF header identifies the type of ELF file, whether it is an executable, relocatable or a shared object file. Each of these file types hold information about the program that tells the operating system to perform appropriate actions on it. The three types of ELF files are summarised as follows:

- an *executable file* holds information that is used by the operating system to create a process image. This information includes executable code, data section and accessing policy.
- a *relocatable file* contains codes and data that have to be linked with other object files in order to create an executable file or a shared library
- a *shared object file* holds codes and data that are required for linking with more than one context (program) in both static and dynamic linking fashion.

Some of the capabilities of ELF such as dynamic linking, dynamic loading, imposing runtime control on a program, and an improved method for creating shared libraries, can be found in [TIS Committee, 1995].

Figure 2.14 depicts the steps of transforming an executable file into a process ready for execution. These steps are done by a loader and linker module in the op-

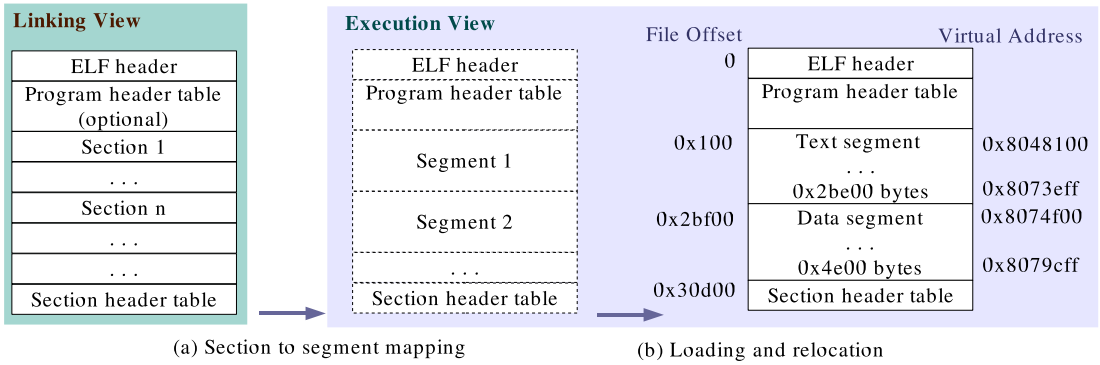
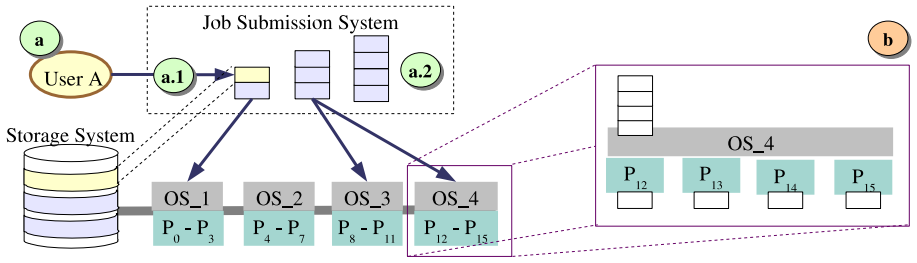


Figure 2.14: Organisation of the Executable and Linking Format (ELF) and its transformation from linking to executable process.

erating system. When a program executes, the loader loads an object program and resolves relocation, *i.e.* allocating the process a virtual memory address space. This loading activity involves assigning physical memory space to accommodate the process and creating an entry in a job or process table (PT) holding information about this process. Once the process entry is created, it will be queued in a 'ready' list waiting to be scheduled for execution.



(a) A user submits a job (a program) to the job submission system
(a.1) The user's job is mapped to the corresponding ELF executable storing in the user's disk space.
(a.2) A job submission system can have multiple job queues, each of which allows different numbers of processors or memory requirements for the job.
(b) In an SMP node, all processors share the same copy of OS. However each of them individually executes the job assigned. In case of a multithreaded job, context switching is handled by the node's OS.

Figure 2.15: Job submission.

Interface to Job Submission System. Practically, a user submits a *job* which includes a workload executable file and its resource requirement (*e.g.* the requested memory space, number of processors, limited run time *etc.*) to a CLUMP via a job

submission system. Figure 2.15 depicts the interface of the job submission system to the node's OS. As shown in Figure 2.15 (a.1 and a.2), a job is associated with the storage space of the user who submitted it. It is common to have several job queues categorising the jobs by their resource requirements (common criteria are the time for executing the job and number of processors requested). The job submission system dispatches a job to the OS of an SMP node. As shown in Figure 2.15 (b), the node's OS creates an execution process for the job and maintains a run queue. Once a job is scheduled to one of the SMP node's OS, the OS is then responsible for process/thread management, scheduling, memory management and parallel context switching. When execution finishes, the output and an execution report are stored to the disk of the user's space. In the next section, the OS functionality regarding the ways to create the execution context of a job and to assign the memory address to the process is described.

2.1.2.2 Operating System (OS)

An operating system is software that manages hardware resources. A conventional OS consists of four distinct abstractions: process management, memory management, file system, and input/output [Tanenbaum, 1995]. An OS comprises multiple service layers, central to these layers is the kernel. An OS kernel includes low-level codes providing the means for communication among multiple resources. Typically, a kernel works based on the message-passing paradigm. On top of message-passing communication, a kernel comprises a minimal set of services providing support for process and thread managers, inter-process communication (IPC), exception handlers, interrupt service routines, and memory manager.

Process Management. Once a process is assigned for execution, the OS creates a Process Control Block (PCB) to keep the information about the process. The PCB is kept in a Process Table (PT) that is used to map the active processes to the corresponding storage space, the registers, and the required resources. The operating system also defines the memory abstraction or address space seen by the process. Address spaces are protected containers of virtual memory in which any number of processes can execute. The memory model is created and taken care of by the memory management module.

Memory Management. The memory management ensures an effective, full-extent parallelism usage of the physical memory resources while providing the memory abstraction to support program execution. Typically in a DSM system, the memory management comprises data structures and operations to support the abstraction of (and allow valid accesses to) virtual memory. A unit of memory resource that a program can directly access is referred to as an *address space*. The memory management allocates and de-allocates address spaces (memory regions) for a program's execution unit. It also sets various attributes of the address spaces, *e.g.* access permissions, cacheable flag. Operations on virtual memory resources include moving data between address spaces and sharing data within and across address spaces. When data are shared, memory management ensures that the protection semantics are not violated.

Virtual Memory. The physical main memory is mapped to the virtual memory's address space using paging and segmentation. Physical memory is divided into frames, each of which is a small unit of contiguous space. Virtual memory is also divided into units of the same size as the memory frame, referred to as *page*s. A page is the lowest unit maintained by the memory management and is the smallest unit to which attributes and mapping can be applied. Operations at the physical level apply per page and are initiated by the hardware demand (*i.e.* by issuing signals related to memory faults, for example, page faults or page access violations). If cache or memory coherence is not supported by the hardware, the memory manager must also ensure the consistency view of data replicas across the memory hierarchy.

Recent memory management systems use one of these allocation schemes: paged memory allocation, demand paging, segmented memory allocation, or segmented/demand page allocation. The scheme used in memory management is associated with how a process refers to the content of the memory. In this project, the segmented/demand page allocation scheme has been emulated. In this scheme, the pages that comprise a process are grouped into several segments. Each process has at least four of these segments:

- *text segment* consists of the actual binary machine instructions mapped from the ELF executable file (read-only)
- *data segment* contains variables initialised and presented in the executable file

data section(s). Data segment is also mapped to the ELF executable on-disk (as shown in Figure 2.14), but has access permissions set to read/write/private. The private access permission means that modifications on any variables declared in these segments cannot be seen by any other processes sharing the same executable.

- *heap space* consists of the area containing pointers to dynamic memory space
- *user stack*

These groups of segments are allocated to a process while an executable is loading. The information regarding these allocated memory segments is kept in the PCB. The OS loads the PCB entry to a special register and also loads some information about segment references into the processor's Segment Descriptor Table during the process scheduling.

So far, the generic concepts of the architectural and system software components of a CLUMP have been introduced. How a CLUMP works in a distributed system fashion with a central, batch job submission system has been described. In the next section, the implementation of a DSM image on top of such a system is presented in four aspects: how it creates a single address abstraction, how the shared data are structured, what responsibility is required to manage a DSM system and how the multiple replicas of shared data are maintained consistent at the DSM level.

2.1.3 Distributed-Shared Memory Implementation

Existing DSMs have been classified into three types by the level of implementation used: Hardware DSM, Software DSM, and hybrid [Protić et al., 1995,Coulouris et al., 2001]. The concepts and mechanisms of the machine architecture and OS presented in previous subsections were extended to support the implementation of a Hardware DSM or a Software DSM at OS kernel. Although the APIs of different DSM systems vary greatly, they employ similar key concepts. In this section, these key concepts are presented in general and the levels of implementation (*i.e.* software, hardware, or hybrid) are abstracted.

Practically speaking, an end-user can access a DSM environment using facilities provided by a DSM implementation. The key mechanism of these facilities is how

to support the execution of a shared-memory, multithreaded parallel application on distributed memory multiprocessors.

Since DSM systems run applications in a shared-memory multithreading style, a DSM implementation commonly centres on a set of techniques to schedule threads and manage shared memory. Scheduling involves assigning jobs (*i.e.* the executable units of a running program in the form of threads or processes) to a set of processors in the system, and also allocating memory pages in the hosting node. Once a job has been scheduled, memory management plays an important role in providing access to memory pages. Figure 2.1, which depicts the block diagram of a cluster of four symmetric multiprocessors (SMPs) with a DSM implementation as an execution environment, also shows an abstraction of a single address space (SAS) providing by a DSM system. From the figure, an abstracted SAS is obtained by an interaction of the DSM system with the virtual memory management of each and every node's operating system.

DSM memory management provides the illusion of a global shared memory on top of the distributed-memory multiprocessor machine by using each local memory in the nodes as local cache for the shared data space. This implies that any processor holding appropriate access rights can request data in any memory location. In the case that some required data is not present in local memory, such data will be migrated or replicated from its location into the requesters' local memory. During this process, multiple data replicas may reside in different nodes, causing a coherence problem. Therefore, the design and implementation of protocols to maintain memory coherence is considered a key issue in terms of performance of a DSM implementation.

In summary, the key components of a DSM implementation are listed in Table 2.8. This list also includes those components of interest to this research.

2.2 Single Address Space Abstraction in DSM

A DSM system creates an abstraction of a range of contiguous memory locations, the so-called *DSM address space*, that all processes or processors in the system can access via conventional load and store operations. Depending on the DSM implementation, the DSM address space could refer to the logical assembly of either physical or virtual memory of every node. This address space is the set of numbers that a process can

Table 2.8: Parameters of interest in the DSM implementation.

Distributed-Shared Memory Implementation			
Component	*Category*	*Varieties/Options*	*Modelled*
DSM Management Node	Centralised		Yes
	Distributed	Static Allocation	No
		Dynamic Allocation	No
Structure of Shared Data	Linear (non-structured)		Yes
	Object-based		No
	Language-type based		No
	Associative		No
Granularity of Shared Data	Byte		Yes
	Word		Yes
	Page/Segment		Yes
	Complex data structure (*e.g.* object)		No
Consistency Model	Homeless Model	RC Model[a]	No
		LRC Model[b]	Yes
	Home-based Model	RC Model	No
		LRC Model	Yes
Memory Coherence Policy	Write Invalidate		Yes
	Write Update		Yes
Locality Optimisation	Data Affinity Approach	Data Migration	No
		Data Replication	Yes
	Code Affinity Approach	Thread/Computation Migration	No
	Hybrid Approach		No

[a]Release Consistency Model

[b]Lazy-Release Consistency Model

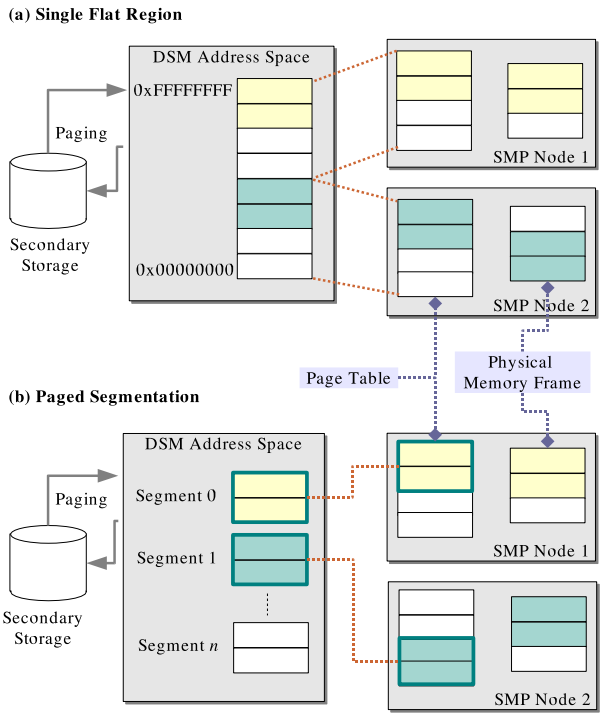


Figure 2.16: Structure of the DSM Address Space.

validly use as memory addresses. In practice, a DSM system comprises a software module or a *DSM manager* that facilitates any legal accesses to the content of such an address space. In the following subsections, the structure of a DSM address space is presented prior to the description of the responsibilities and facilities of a DSM manager.

2.2.1 Structure and granularity of Shared Data

A DSM address space is a shared region that is normally structured in two ways: (a) using a single flat region or (b) using paged segmentation. Figure 2.16 (a) depicts a generic structure of a single flat region space used in some systems, for example, IVY and Mether. In the figure, the DSM region can be observed as one continuous range of the 64-bit virtual addresses that is usually sub-divided into pages. The latter way to structure a DSM address space, using paged segmentation, is illustrated in Figure 2.16 (b). This structure, used in most systems, represents the shared region as a composition of disjoint pieces (referred to as *segments*, *memory objects*, or *windows*) each of which is managed separately and is dynamically mapped to a process address

space.

The way a shared region is represented is also associated with the unit of the shared data provided by a DSM system. A unit of the shared data, also known as *data granularity*, can be categorised into the following three groups [Coulouris et al., 2001], based on how data is associated with the storage unit:

Byte-oriented. *A byte-oriented DSM system* provides accesses to the DSM address space as a contiguous array of bytes. Accesses to a shared data item are done by conventional read and write (or loads and stores) operations. In a byte-oriented DSM, the granularity of shared data to maintain memory coherence is normally a *page*. Therefore, a DSM system of this type is also referred to as a *page-based DSM* system.

Object-oriented. In an *object-oriented DSM system*, a shared data region is bound to a language-level object. A data access is done by invoking methods upon the object, instead of using simple read and write operations. Using this technique, data access ordering enforced by a consistency model is done on an object basis.

Immutable data. In an *immutable-data DSM system*, a shared data region is arranged by both data types and the logical association among data fields, defined at the high-level language level. An example of a shared region of an immutable-data DSM is a *tuple space* provided by the Linda DSM system and its derivatives. A tuple space is a shared data region that comprises many *tuple*s each of which consists of associated data fields of one or more data types. Processes share data by accessing the same tuple space using the write (extract) and read (take) operations. Data consistency is kept at the unit of tuple.

In practice, the SAS concept of a DSM has been successfully used to provide high-performance communication in some hardware implementations such as in the Cray T3E parallel computer and Cray X1. Subsequently, the concept has been adopted for the standard language defining message passing interface (MPI-2) under the name of one-sided communication. The concept is attractive as a target model for compiler-generated code, and has motivated a number of researches on a single address space operating system.

2.2.2 Responsibility of the DSM Manager

In this thesis, the term *DSM manager* is used to refer to a collection of hardware and software modules that maintain a DSM address space. A DSM manager provides the facilities to allow accesses to the content of such an address space. As mentioned earlier, the application interface of the DSM manager varies greatly from implementation to implementation. Nevertheless, the facilities that are common among different DSM managers are listed below:

- thread/process creation and termination
- thread/process scheduling
- address allocation and deallocation
- access right control
- address mapping (in collaboration with the virtual memory management of a node's operating system)
- access faults handlers (*e.g.* supplying data when a processor sends a *segmentation fault* signal notifying that a process is referring to some part of the memory that is not present or an access violation signal notifying that a process is trying to write to a read-only address)
- keeping information about the shared region (*e.g.* which nodes have a copy or which node is a home node)
- maintaining data consistency (if using data replication, a DSM manager also maintains data coherence across multiple replicas).

Apart from the facility listed above, a DSM manager also provides ways to allow concurrent accesses to the content of the shared address space. Two issues are essential for managing concurrent accesses: ensuring an atomic operation and controlling accesses to the shared resources. The following paragraphs give brief descriptions of these two issues.

Atomic page update. In a page-based Software DSM, all accesses are controlled at the unit of page. In order to guarantee that each update to a shared page is atomic, other threads should be prevented from accessing the shared page while a thread is waiting for a valid page. There are many possible ways to implement this in a DSM manger. Some example techniques are: to suspend all the application threads until the system finishes updating the invalid page; to modify the OS scheduler not to schedule threads that may access the invalid page; to implement a new thread package; to map a file to two virtual addresses and to assign different access permissions to them. In addition, Kee *et al*. proposed four approaches based on different Unix system calls to ensure the atomicity in the update operations [Kee et al., 2004].

Mutual exclusion. To control accesses to shared resources, mutual exclusion algorithms are used to resolve conflicting accesses to shared resources by concurrent processes [Anderson et al., 2003]. Many DSM machines use a local-spin algorithm to maintain mutual exclusion. Generally in a local-spin algorithm, each process must have its own dedicated spin variables stored in its local memory module. In the local spin algorithms for cache-coherent DSM machines, spin variables are shared among processes, since each process can read a different cached copy. Different local-spin algorithms employ different techniques to control concurrent access to shared resources by test and set operations on the spin variables. The test command is used to check the value of the spin variable to see whether it is possible to access a shared resource. If the resource is available, the set operation is used to modify the value of the spin variable in order to prevent accesses from other processes. Normally one spin variable is assigned per shared resource. Thus, concurrent accesses to a shared resource are controlled by interleaving the test-and-set operation on the spin variable. Detailed discussion on various local-spin algorithms can be found in [Anderson et al., 2003].

Responsibilities for the DSM management can be centralised on one SMP node or distributed across multiple nodes [Protić et al., 1995]. The distribution of DSM manager determines which node has to handle actions described above and which node has to maintain data consistency in the system. Distribution of responsibility for DSM management is closely related to the distribution of directory information, which can be organised in the form of linked lists or trees. The next section presents the issues and concepts related to data consistency maintained by a DSM manager.

2.2.3 Consistency of Shared Data

In most DSM systems, shared data is replicated in the memory space of multiple nodes for speed of access. This raises the coherence problem similar to caching replicas in cache lines that is central to an SMP architecture (as discussed in Subsection 2.1.1.2 and 2.1.1.5). The original memory consistency models designed for shared-memory multiprocessors have been adopted and extended for supporting a higher abstraction of memory in a DSM system. Similar complications arise to maintaining memory consistency in the DSM system, *i.e.* the more strict the ordering defined by the model, the longer the access latency and the more bandwidth is required in order to accomplish a memory access. Apparently, more relaxed models result in better performance (in terms of access latency and memory bandwidth) as they allow more data accesses to be overlapped (*i.e.* exhibiting better concurrency). However, the cost of relaxing models is twofold. Firstly, it requires higher involvement of the programmer in synchronising the accesses to shared data. Secondly, it requires more complicated activities to maintain data coherence.

The issue of maintaining consistency across memory replicas is one of synchronisation of write accesses to shared data (*write synchronisation*). Similar to the hardware coherence policies, two main approaches are variously known as write-broadcast and write-invalidate.

2.2.3.1 Relaxed Consistency Models

Relaxed consistency (RC) models allow actions of maintaining data coherence to be delayed until certain synchronisation accesses occur [Gharachorloo et al., 1998]. In the RC model, synchronisation primitives are divided into *releases* and *acquires*, and consistency actions must be taken before an acquire can be successful. The RC model can be further divided into eager release (ERC) and lazy release consistency (LRC) depending on the delay in making the effects of shared memory known to the other processes.

Figure 2.17 shows an example scenario of two systems each having four processors, one using an ERC model and the other using an LRC model. The two systems run the same program which uses two synchronisation variables $s1$ and $s2$. In ERC policy, the information about all modifications is broadcast to all other processors in

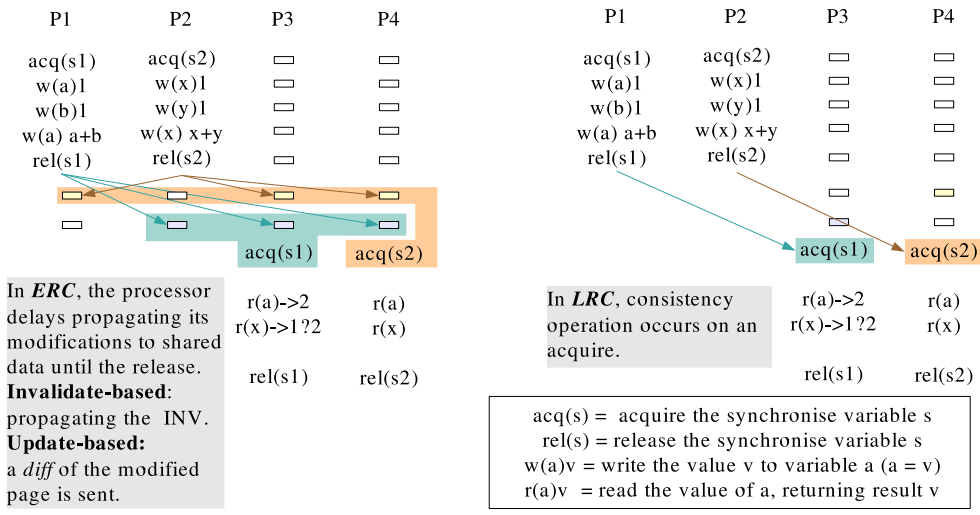


Figure 2.17: Comparison of eager release and lazy release consistency models.

the system at the time of the release operation. Using this technique, the shared data are kept coherent prior to the next acquire. Thus, the following acquire will be successful without any further consistency actions being required. In contrast, the LRC policy delays the notification of the modifications until the next acquire of the same synchronisation variable. Besides, LRC only sends the modifications to the processor that issues the acquire. The other processors that might cache the replicas of this location will receive a write notification to invalidate the replicas.

In this thesis, the LRC model is addressed as it is common in many DSM implementations[14].

The Page-based LRC model. In LRC, whenever a node writes something in pages of DSM, it must record information about the write. The interval data structure is used to contain all the information about the writes between two synchronisations, which is the vector time of its creation, a list of dirty pages and difference from the earlier copy (*diffs*) in order to update other nodes [Keleher et al., 1992]. According to the LRC model, this update can be delayed until the next 'acquire' time. At the acquire, the requested node finds out whether or not the memory accessed is the most up-to-date by using the page fault mechanism of the virtual memory system. If a node accesses a stale copy of the page, a page fault happens. The faulting node must get

[14]Information on other models can be found at [Steinke and Nutt, 2004, Adve et al., 1999, Shi et al., 1997].

the most recent copy of the page from another node or nodes which have the most up-to-date copy.

Home-based *v* homeless LRC. At the time of the page fault, the homeless and home-based protocols behave differently. In the *homeless* protocol, the faulting node requests time-ordered diffs of the page from the last writer or writers. The most up-to-date page is represented as time-ordered differences of the page which may be distributed over the nodes. In the *home-based* protocol, the faulting node requests the page from the home of the page which always has the most up-to-date copy.

Apart from different versions of the RC models, there are several innovative models proposed for dealing with different aspects of a DSM system. An example of these models is a Reconfigurable Object Consistency Model (ROCoM) [Pousa et al., 2005]. ROCoM is a consistency model for asynchronous architectures that execute an object-based software DSM and do not have atomic broadcasting implemented in hardware. This consistency model manages the state of a set of shared objects and allows the policy to be reconfigured by considering the workload's and architecture's characteristics. In this case, the model permits more flexibility and increased performance over the static model.

In a memory consistency policy, synchronisation mechanisms are used to maintain the desired ordering of accesses. It is important to note that programmers are responsible for stating the synchronisation points to ensure the correct execution. However, DSM systems guarantee the order by the consistency policy used. Therefore, in DSM systems, a consistency policy can be understood as a mutual agreement between programmers and the system as it defines the execution orders that can be expected.

2.3 Optimisation Techniques in DSM implementations

Using the mechanisms described in the previous section, a DSM system offers users programmability by providing a single address space and ensuring a consistent view among data replicas. In this case, location of data is transparent to users. An example scenario is shown in Figure 2.18. From a user's point of view, there is no difference among the three accesses to a shared data region shown in Figure 2.18 (a, b, and c)

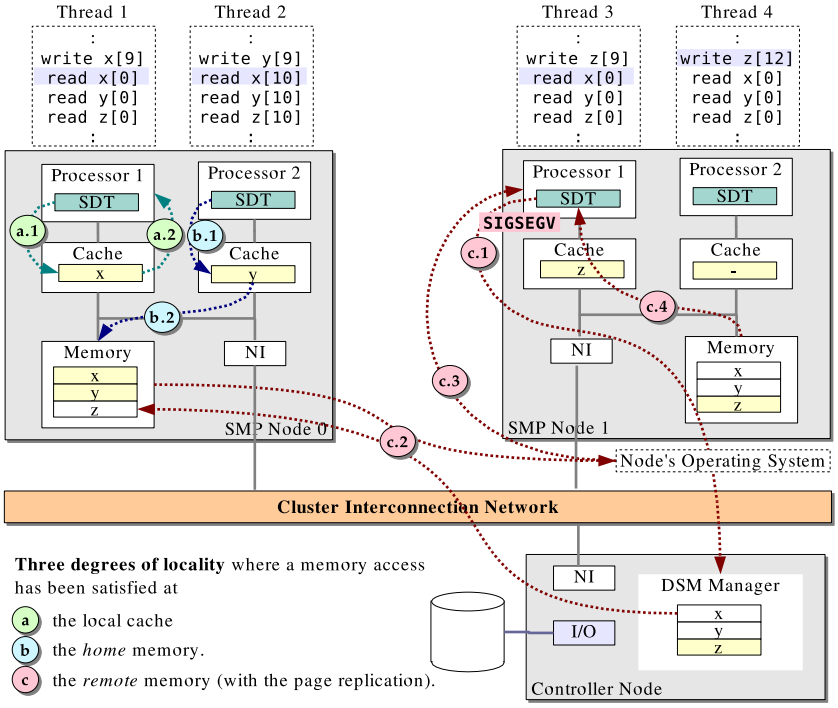


Figure 2.18: DSM Memory Locality.

(from the `read x[-]` command from P_1, P_2 and P_3, respectively)[15].

However, in practice, these three accesses to variable `x` in Figure 2.18 involve different latencies as each of the accesses is satisfied at a different locality: (a) local cache, (b) local memory and (c) remote memory. Remote accesses are considerably more expensive than local memory accesses as they involve interaction with the DSM manager via the node's operating system. Figure 2.18, steps (c.1 - c.4) illustrate the complication of a remote access. When a processor attempts an access to a memory page that is not present locally, a segmentation fault signal is sent to the node's OS. The DSM manager traps this access fault and performs operations to supply data. One of the operations the DSM manager will do is to search in the memory map for the fault address in order to locate the required page. In the figure, the required page kept at $Node_0$'s memory is then replicated to the requester. As discussed earlier in Chapter 1, in six architectures presented, accessing data from a remote memory can cost up to 20 times the delay in accessing data in local cache depending on the DSM

[15] Some DSM implementations divide the address space into private, shared local and shared remote to make threads aware of which regions of the shared address space is mapped to local, or to remote. This specific feature is not observed in this study.

implementation.

The delay of data access caused by the distant location, or memory locality problem (Section 1.2.1 page 6), is regarded as one of the critical factors that limits the performance of a DSM system. Different data access patterns show an impact on memory locality. There are four cases caused by data access patterns that are commonly targeted by locality optimisations: false sharing, double faulting, access hot spot, and thrashing.

- *False sharing* is a situation when two or more processors attempt to write different variables that are kept in the same memory page (or in the same coherence unit, *e.g.* a cache line). False sharing causes the shared page to be invalidated after a write because another processor wants to access it. Thus, a subsequent access will be a 'miss', resulting in the overhead to fetch the same page back again. This situation is referred to as the *ping-pong* effects.

- *Double faulting* is a situation when a process first reads (*i.e.* obtaining a read-shared copy) and then writes (*i.e.* requesting for an exclusive copy) a page causing a memory page to be transmitted twice over a network.

- *Access hot spot* refers to a shared page that is accessed frequently by processors of both remote nodes and the local node.

- *Thrashing* refers to a situation that occurs when a DSM manager spends an inordinate amount of time invalidating and transferring shared pages compared to the time spent on processing applications.

A number of optimisation techniques have been developed to tackle the memory locality problem, and have become common features found in DSM implementations. Approaches to optimising memory locality can be categorised into *data affinity* and *code affinity*. Moving data close to the requester in the data affinity approach can be done by either migration or replication. In contrast, moving code close to where data is kept in the code affinity approach can be done by the thread or computation migration technique. Alternatively, combinations of optimisation approaches have been developed to alleviate different cases that may cause a reduction in memory locality. In the following subsections, each of these techniques is described.

2.3.1 Data Migration

In data migration, the shared data (normally in a unit of a *page*) that tends to be obsolete locally is shipped to the potential users using the interrupt-based mechanism. Page migration is a tool to optimise the locality of memory accesses, as once the data has been shipped, the subsequent accesses are local, so that the overall number of remote memory accesses in the program is reduced.

Competitive page migrations. Competitive page migrations [Verghese et al., 1996] are on-line algorithms, based on the number of accesses to individual pages. Competitive page migration algorithms retrieve snapshots of page access counters at random points of execution and migrate pages using access traces obtained from these snapshots. Software installed on each node, known as a *migration engine*, uses an interrupt based-mechanism which periodically checks the access counters (hardware counters) of each page and assesses whether or not the page should be migrated. The counters are decayed progressively so that the page migration engine is not biased by obsolete page access history.

Dynamic page migration. Dynamic page migration [Nikolopoulos, 2003] is a tool for balancing remote memory accesses on a per-node basis rather than on a per-page basis. It migrates pages based on the accumulated latency of accesses to each node. The algorithm identifies the nodes in which the accumulated latency of remote memory shows that data are 'frequently' accessed remotely. The algorithm searches for access hot spots at page-level granularity and migrates pages so that hot spots are distributed, rather than concentrated in a few nodes.

Page migration allows data to be shipped to where it is most needed so that subsequent accesses are local. However, in migration, only processes on one node could access a shared data at one moment. Consequently, some drawbacks of using page migration are thrashing, false sharing, and the overhead of multiple readers. To allow concurrent read accesses by multiple nodes, data blocks are replicated across multiple nodes. The optimisation technique known as data replication is also commonly used in a DSM system.

2.3.2 Data Replication

There are two types of data replication, read-replication or full-replication [Singhal and Shivaratri, 1994]. In the read replication algorithm, the DSM system must keep track of the location of all the copies of data blocks. One way to do this is to have the owner node of a data block keep track of all the nodes that have a copy of the data block. Alternatively, a distributed linked list may be used to keep track of all the nodes that have a copy of the data block.

Read-replication Algorithm. In read-replication, when a remote node requests to read a page that is not present in its local memory, a page is replicated and sent to the requester's node. All replicas of the page are marked as multiple readers. When there is a write access to any of the replicas, the writer node acquires the page and invalidates or updates any other replicas. In this case, the read-replication algorithm allows multiple nodes to share the read-shared replicas (thus, allowing concurrent reads across multiple nodes). However, read-replication only allows one node to update data at a time. The technique is called *Multiple Readers/Single Writer* (MRSW).

Full-replication Algorithm. Full replication allows data blocks to be replicated even while being written to, therefore it adheres to a *Multiple Readers/Multiple Writers* (MRMW) protocol [Stumm and Zhou, 1990]. The full-replication algorithm works in collaboration with a consistency model or a sequencing strategy to ensure the order of the write accesses to a shared data and to keep data replicas up to date. An example of a sequencing strategy is a *single global gap-free sequencer* [Stumm and Zhou, 1990]. This strategy uses a sequence number generated from a central software called the *sequencer* to notify the order of a modification. The number is generated on each write and then broadcast. Every processor keeps the last sequence number received and requests the sequencer for the updated packages when a next sequence number received is not adjacent to the local one.

Data replication permits concurrent read operations locally, thus reducing the average cost of delay on read operations due to the communication overhead. However, in both read- and full- replication algorithms, prior to each write operation, all replicas have to be invalidated or updated to maintain data consistency. Therefore the overhead of broadcasting consistency messages can be expensive. Nevertheless, in

an application with a regular access pattern and having a large ratio of reads over writes, the overhead on writes can be offset by the lower average cost of the read operations. Despite this, in applications with irregular access patterns, data replication may cause thrashing. Therefore, different optimisation techniques have been proposed to deal with applications that have irregular access pattern. One of these techniques is computation migration.

2.3.3 Thread/Computation Migration

Application workloads with irregular patterns of data access can make the cost of data affinity approach become expensive. Therefore, moving threads closer to data, *computation migration*, is an alternative to reduce the distance of data accesses. Computation migration is the act of transferring a process or a thread from one node to another node. The aims of migrating threads are not only to achieve data locality but also for the purpose of load distribution, fault resilience, and sometimes easing of system administration. Steps of computation migration include pausing activity of the process/thread, recording it states (*e.g.* register values), packing the state record and codes, shipping the package to the target destination, unpacking the package, restoring thread/process states and resuming the computation.

A number of algorithms have been proposed in the context of process migration [Milojíčić et al., 2000]. A research of a computation migration using decision at runtime shows that dynamic migration is useful for concurrent data structures with unpredictable read/write patterns [Hsieh et al., 1996]. Another study has pointed out that when threads are migrated from heavily loaded nodes to lightly loaded nodes for load balancing in software DSM systems, the communication cost of maintaining data consistency is increased if migration threads are carelessly selected [Liang et al., 2002]. This study introduces an algorithm which includes a selection policy, called reduction of inter-node sharing costs, to decide if the migration should take place. The study concluded that by simultaneously considering thread memory access types and global sharing, this policy can reduce the communication of benchmark applications by 50% during load balancing [Liang et al., 2002].

2.3.4 Combination of Optimisation Approaches

There is a significant body of work on combinations of optimisation approaches to balance the benefit of different approaches described above. In many software DSM implementations, data migration and replication were used both as a mechanism to provide data consistency and for optimisation[16]. The combination of data replication and prefetching has been introduced as a new latency hiding technique in DSM machines [Roh et al., 2000]. This study proposed a technique which uses an adaptive granularity for bulk transfer and adds prefetching to obtain the replicated data to the cache at the right time. A combination of compiler analysis and thread migration is also introduced in a project called MigThread [Jiang and Chaudhary, 2004]. At compile-time, a preprocessor scans C programs to build thread state, detects possible thread adaptation points, and transforms the source code accordingly. At runtime, the MigThread moves DSM threads around to utilise idle cycles on remote machines.

So far, the main features of DSM implementations at both software and architectural levels have been presented. The area is large and rapidly developing. Some other interesting features that are not directly related to the locality problems have not been mentioned. Examples of these features and further references are the recovery and check-pointing issues in DSMs (discussed in [Morin and Puaut, 1997]), heterogeneity DSM systems, Grid-enabled environments, and DSM supports for mobile devices [CCGrid, 2005].

2.4 Programming Paradigms for DSM Systems

From the architectural point of view, a DSM system is a distributed system comprising multiple processing nodes that share information using message passing. Over such an architecture, multiple features of different DSM implementations have been developed to permit a single address space abstraction for programmability as described in the previous sections. Alongside architectural advances, the development of an effective programming paradigm (including both the language and compiler supports) for various DSM implementations has gained attention from both academics and vendors. Recent developments have resulted in the languages that are supported on many

[16] *E.g.* IVY, Munin, Treadmarks, Midway, Jade and Sam

vendor systems and by several compilers. These programming paradigms offer different trade-offs between programmability and performance.

Parallel programming involves dividing a problem into parts in which separate processors perform the computation of the parts [Wilkinson and Allen, 1999]. Parallel programming paradigms can be categorised by the provided programming model (*i.e.* a view of data and execution dynamics that programmers can expect when running applications on a particular architecture) into three traditional approaches: *message passing*, *data parallel* and *shared memory* programming paradigms, as shown in Figure 2.19 (a), (b) and (c) respectively.

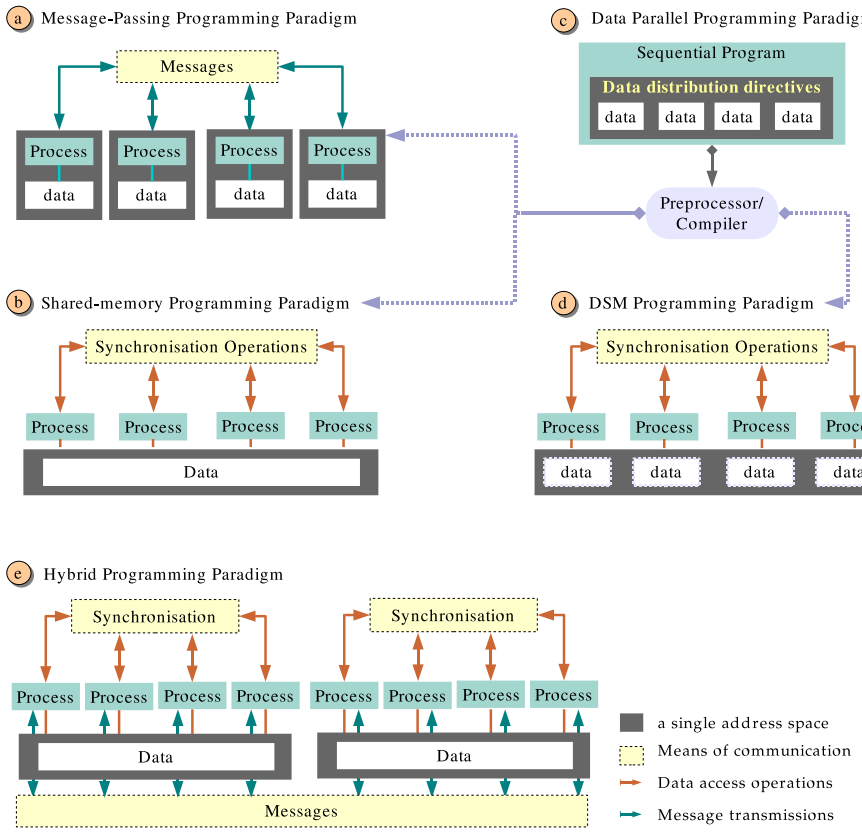


Figure 2.19: Parallel programming paradigms (adapted from [Agerwala et al., 1995]).

In DSM systems, the latter two approaches are normally adopted. Figure 2.19 (d) shows the DSM programming paradigm, an extension to the traditional shared-memory paradigm, which includes support for data locality (*e.g.* workload distribution, memory consistency, local/remote data scope). In addition, some innovative

hybrid paradigms have been proposed to explicitly state the message transmission in the system of a cluster-like architecture (Figure 2.19 (e)). In this section, the fundamental concepts associated with the data parallel, the (distributed) shared memory and the hybrid programming paradigms are presented. Programmability and performance trade-offs of each of these paradigms on DSM architectures are also discussed.

2.4.1 Data-Parallel Programming Paradigm

The simplest parallel programming paradigm is data parallel as parallelism is implicitly implied of data operations. For example, if there are two one-dimensional arrays, `A` and `B`, of equal size and the elements of both arrays are defined as N double precision floating point numbers, the traditional way to add such arrays in C++ would be:

```
1    for (int i=0; i<N; i++)
         c(i) = a(i) + b(i);
```

However, in data parallel language like High Performance Fortran (HPF), parallel operation on each array element is assumed. Therefore, the equivalent operation can be expressed as:

```
     !HPF PROCESSORS procs(4)
2    !HPF DISTRIBUTE (BLOCK) ONTO procs :: A,B,C
     C = A + B
```

The data parallelism paradigm implies parallelism on data elements of particular data types such as an array or object. As shown in Figure 2.19 (c), programmers use some directives to describe data distribution. Two example directives are highlighted in the HPF code fragment presented above. The `PROCESSORS` directive (line 1) identifies the number of processors to run the job. In addition, the `DISTRIBUTE-ONTO` directive tells the compiler how the array elements should be partitioned and assigned to each of the four processors. These descriptions are then taken by a preprocessor or a compiler to generate the code into either the message-passing, shared-memory or DSM models based on the machine architecture. Two examples of data parallel programming languages are HPF [HPF Forum, 1997] and DOSMOS [Brunie and Lefevre, 1996]. Furthermore, some object-based DSM systems also provide the

programming environments that employ the concept of data-parallel programming paradigm.

2.4.2 Shared-Memory and DSM Programming Paradigms

The shared memory programming paradigm does not imply parallelism of the operations of a particular object or data type like in data-parallel programming. Rather, in the shared-memory model, programmers can assume that the address space is all shared. An advantage of the shared-memory programming paradigm is the absence of marshalling[17]. Thus, existing multithreaded shared-memory programs can easily be ported to a DSM system.

In the DSM programming paradigm, programmers can exploit data locality using some language features to specify the location of data. Subsequently, compilers can distinguish between the remote and local accesses allowing more opportunities for locality optimisation. Programmability of the shared memory and DSM programming paradigms is well known as shown by the number of language consortium which have emerged[18]. Consequently, many shared-memory parallel languages have been proposed for different DSM systems. Four well established languages that are currently supported by major vendors are the Unified Parallel C (UPC), Co-Array Fortran, Titanium, and OpenMP. The first three languages are based on the DSM programming paradigm, while OpenMP is the standard specification for the shared memory programming paradigm.

Unified Parallel C (UPC). Unified Parallel C [Carlson et al., 1999] is an explicit parallel extension of ANSI C using the Single Program Multiple Data (SPMD) execution model and explicit parallelisation. The executable of a UPC program runs from the beginning and in its entirety on each processor independently. The implementations of UPC are supported on the Cray T3D/E, Compaq AlphaServer and the SGI Origin 2000[19].

[17] *i.e.* interpreting data references into an address-space neutral format, mostly for the transmission of program over a network

[18] *e.g.* The OpenMP community (http://www.compunity.org/), the UPC working group (http://www.gwu.edu/ upc/workgrps.html),the Titanium team (http://titanium.cs.berkeley.edu/)

[19] Ongoing and future implementations for: HP Sun multiprocessors Cray SV-2, IBM Beowulf Clusters

Co-Array Fortran (CAF). Co-Array Fortran [Numrich and Reid, 1998] is a simple parallel extension to Fortran 90/95 to incorporate the SPMD Model into Fortran. The CAF marks variables with co-dimensions that behave like the normal dimensions but can, however, be decomposed. CAF supports data exchange between co-arrays by using the compiler to manage remote addresses, shapes and sizes. The location of an object is explicitly specified, *i.e.* normal rounded brackets ( ) to point to data in local memory while the square brackets [ ] are used to point to data in remote memory. Although syntactic and semantic rules apply to these specifications separately, programs can access both local and remote objects equally.

Titanium. Titanium [Yelick et al., 1998] is a Java dialect with some features added to support the SPMD execution model on a DSM system. Titanium has the same execution model as UPC and CAF. The features added to traditional Java are support for the global address space such as, synchronisation, zone-based memory management, operations on sub-arrays without copying and operations for small objects. The small object (*e.g.* the complex number) is called an *immutable* or a *value class* in Titanium.

OpenMP. OpenMP [OpenMP, 2002][20] is a set of APIs that may be used to explicitly direct multi-threaded, shared memory parallelism. The set of OpenMP APIs comprises three primary components: the compiler directives, runtime library routines and some environment variables. OpenMP standard specifications have been defined for indicating shared-memory parallelism in C, C++ and Fortran programs. The specifications have been implemented on multiple platforms, including most Unix platforms and Windows NT. Despite the fact that OpenMP is not meant for distributed memory parallel systems by itself, and its features are not necessarily implemented identically by all vendors, OpenMP is increasingly used on NUMA architectures [Tao et al., 2005].

Figure 2.20 shows the OpenMP execution model in fork-join style. An example of a OpenMP-based C program shown in Figure 2.20 (a) illustrates the use of the directive `#pragma omp parallel` to specify a *parallel region*, *i.e.* the scope of a multi-threading parallel job. This scope is specified by the lexical unit of the C program (*i.e.*

[20]The name OpenMP stands for the Open specifications for Multi Processing via collaborative work between interested parties from the hardware and software industry, government and academia [OpenMP, 2002].

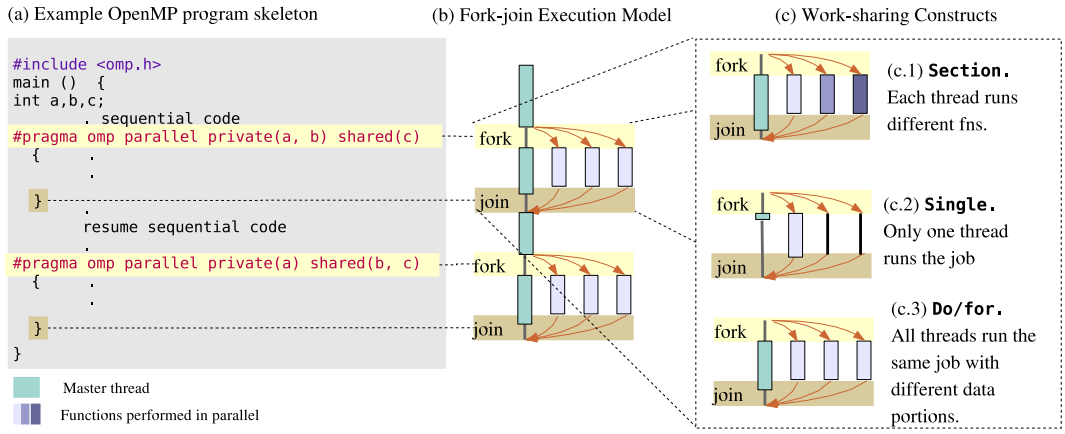


Figure 2.20: The OpenMP execution model and work-sharing directives.

in C/C++ it is identified by the curly brackets, {}). At the end of each parallel region, results obtained from all parallel threads are joined to the master thread as shown in Figure 2.20 (b)[21]. For each parallel region, three different work sharing directives can be used to decompose a task to be performed in parallel. Figure 2.20 (c.1-3) shows the execution model for each of these constructs with a brief description.

In terms of performance, both the shared-memory and DSM programming models raise the same issues of concurrency control of accesses to the shared region. This complication affects performance differently based on each DSM implementation. The impact of the programming model on performance is more noticeable in a DSM system that implements the synchronisation primitives in a distributed fashion and maintains data consistency by software. The absence of any marshalling in both programming paradigms also implies that the individual nodes need to be homogeneous. Otherwise, some other mechanisms are required to deal with heterogeneity, which could also impact the performance. For example, sharing memory pages between a 32-bit and a 64-bit architecture requires some extra features in the DSM manager. While the performance implications cannot be ignored, the shared-memory and DSM programming paradigm seems to be preferable due to the programmability, the possibility for optimisations and the feasibility for auto-parallelising compilers.

[21] The nested parallel region is supported in the OpenMP C/C++ specification.

2.4.3 Hybrid Approach

SMP clusters are considered to be a mixed configuration of shared memory and distributed memory. Therefore, several hybrid programming models have been proposed to match the program behaviour to the architectural constraints. A combination of OpenMP with Message Passing Interface (MPI) is an example model of a hybrid programming paradigm. In this model, multiple OpenMP threads are defined under each MPI process. Therefore, the OpenMP threads can be used within each SMP node while message passing can also be used for inter-node communications (Figure 2.19 (e)).

Several DSM implementations provide the programming interface to obtain information about shared data[22]. Some hybrid programming models have been proposed for example, Global Arrays [Nieplocha et al., 2005] and Mome with OpenMP. Among many alternatives, OpenMP is the most widely used in different scientific benchmarks for example, in Splash-2 and NPB parallel benchmark. Therefore, in this thesis, the OpenMP-based workloads are selected from different benchmarks to represent the workload characteristics of scientific codes running on the simulated DSM systems.

2.5 Performance Analysis of DSM Systems

Three approaches are generally used in an exploration and evaluation of multiprocessor design trade-offs: analytical modelling, measurement, and simulation.

2.5.1 Analytical Modelling

Analytical modelling refers to the process of using mathematical methods to create a conceptual representation, or a *model*, of a system of interest. A model is usually described by a simplified mathematical form and used as an attempt to approximate the behaviour of a system and to predict its performance. The design options of the system under study are defined as the model's parameters. A series of interesting actions (mostly memory accesses) extracted from a benchmark are input to the model prior to the calculation of the approximate responses of the target architecture. Two examples

[22] *e.g.* in Brazos, CASHMERe, Millipede, DSM-Threads, Quarks, and Object View

of analytical models representing the generic execution model of multiprocessors are the BSP and LogP models.

Both BSP and LogP models describe a generic parallel computation across a wide range of architectures, and can be viewed as closely related variants within the bandwidth-latency framework for modelling parallel computation [Bilardi et al., 1996]. In both models, a machine is described by a group of n serial processors of the same clock speed $(P_0, P_1, ...P_{n-1})$ communicating by message passing via an interconnection medium.

BSP Model. In the BSP model [Valiant, 1990], the machine operates by executing a sequence of *supersteps* each consisting of three phases: a local computation, a global communication and a barrier synchronisation phase. The execution time of a superstep, $T_{superstep}$ is expressed by the summation of three terms: (a) the maximum number of local operations, (b) the message transmission time, (c) the time required for global barrier synchronisation. The overall time of a BSP computation is the summation of the time $T_{superstep}$ of its constituents [Bilardi et al., 1996].

LogP Model. Culler *et al*. [Culler et al., 1993] extended the BSP model to cover the cost of communications and termed the model *LogP* after four main parameters representing the characteristics of a system. The first parameter, L, is an upper bound on the latency incurred in communicating a message. The second parameter, o, is the time interval that a processor is engaged in a message transmission, thus cannot perform other operations. The last two parameters, g and P, refer to the minimum time interval between consecutive message transmission and the number of processor/memory modules, respectively.

Many analytical-based studies are for performance predictions and contention analysis. Nikolopoulos studied memory contention using an analytical model and measurement [Nikolopoulos, 2003]. In this paper, several works that extended the BSP or LogP models have been discussed. Two examples are an extension of the BSP model for modelling memory bank contention and an extension of the LogP model to study the overall contention on both distributed and shared memory multiprocessors.

An analytical-based performance analysis that studied the cache-coherent cluster architecture, which is similar to the target architecture of this research, is the work

of Qin and Baer [Qin and Baer, 1997]. In this work, various aspects of the cluster architectures were analysed using an analytical model based on the mean value analysis. The model takes two groups of parameters, the architectural and application-dependent parameters. The application-dependent parameters were obtained from a legacy trace-driven simulation called MINT. Experiments were carried out using three workloads of different data access patterns, running on various cluster configurations of sixteen processors[23]. The main response variable used is the cache-miss ratio characterised into intra-node and inter-node misses of six different types. Among several findings, the study concluded that the contention of shared resources is determined mainly by three factors: the rate at which misses are issued, the cluster size, and the presence of remote cache. In addition, the contention on data bus and on the protocol processor are the prime factors that influence the overall performance of the cluster-based architecture.

To summarise, mathematical models can give more insight into the behaviour of target components than other experimental techniques. In addition, models can be used to suggest useful ways of tuning and optimisation after studying the interaction of every target component. The analytical modelling approach is fast and viable, however it has a limitation in capturing the complexity of computer system components of the parallel computer architecture [Shi and Tang, 1998]. The parameters of an analytical model are static. Subsequently, an additional technique to capture the dynamic behaviours is essential, *e.g.* using simulation to obtain the application-dependent parameters [Qin and Baer, 1997]. Besides, the abstraction defined by the model assumptions can make it difficult to verify the results produced against measurement values. Therefore, when the dynamic responses and precise information are crucial, like the study of a new architectural component at the design phase, the other two approaches (measurement and simulation) are preferable.

2.5.2 Measurement

While it is difficult to reflect the detailed and realistic responses via an analytical modelling, the measurement approach ensures the correctness and completion of results by executing the benchmarks on an actual system. The term *measurement*[24] refers to

[23] *i.e.* 16×1, 8×2, 4×4, 2×8, 1×16 CLUMPs

[24] so-called the *empirical analysis*

the procedures to gather a quantitative description and empirical data to determine the size or magnitude of a system under study. A number of system facilities (*e.g.* compilers, operating system services, hardware counters, utility programs *etc.*) are used to set up the framework of an experiment to observe the execution profiles of various benchmarks. Two key components used in the measurement-based experiments are benchmarks and performance analyser software. The following paragraphs give brief descriptions of each of them.

Benchmarks. A benchmark is an existing program or a set of programs that is coded in a specific language and executed on the machine being evaluated [Lucas, Jr., 1971]. The benchmark is used to obtain performance data as a baseline of a comparison study or as an interim measurement of progress for performance optimisation. Benchmarks represent the patterns of accesses to the demanded system resources. A benchmark suite normally represents a set of real-world applications in which users can reconfigure the application's size to match the power of the system under study. Various benchmarks for parallel computer systems have been reviewed in [Gustafson and Todi, 1998]. An example of a benchmark suite that is commonly used in a performance analysis of the SMPs or DSM architectures is the NAS NPB.

An alternative to a complete application benchmark is a *microbenchmark*, *i.e.* a small computational kernel used for some quantitative measurements of a particular component of a computer system. Several microbenchmarks have been proposed. Two examples are a microbenchmark suite for measuring memory hierarchy performance in both uniprocessor optimisations and the contention and coherence effects of multiprocessors [Hristea et al., 1997], and a microbenchmark for analysis of multiprocessor system performance using the selected features of OpenMP specifications [Bull and O'Neill, 2001].

Performance Analysers. Most multiprocessor systems provide software suites used for collecting program execution profiles in the course of performance tuning or optimisation. A performance analyser comprises several mechanisms to obtain the values of hardware counters and the information about the OS activities during a program execution. Some performance analysers also display raw data in a graphical format and generate the plots of different performance metrics as a function of time. Per-

formance metrics collected from a performance analyser can be classified into four groups, including: the timing metric, the memory usage metric, language-specific or operations-specific metric (*e.g.* MPI tracing, synchronisation delay) and the run profile obtained from hardware counters (*e.g.* cache misses, number of memory accesses). An example of a performance analyser tool is Collect&Analyzer [Sun Microsystems, 2002].

A measurement-based research that is most closely related to this research is the work of Bilas *et al*. that examined the scalability of the Shared Virtual Memory (SVM) clusters in comparison with the hardware cache-coherent DSM machine [Bilas et al., 2003]. In this study, the SGI Origin 2000 was used as the experiment platform representing the hardware cache-coherence system. The represented SVM system was a software called GeNIMA, implemented on a cluster of Intel Pentium Pro SMP with a shared snoopy coherence bus, interconnected via a Myrinet system network interface. The results measured from running some applications of the Splash-2 benchmark on the GeNIMA system were compared against those measured on the SGI Origin of the size of 16, 32 and 64 processors. Timing metrics were measured, including: the total execution time and the normalised execution time breakdown per processor. In this study, scalability was reflected by the application speedup obtained from executing different problem sizes on different processor scales. The study found that (a) applications require a reasonable amount of restructuring for the SVM system in order to obtain good performance at a similar level to that obtained from the hardware DSM system, and (b) the SVM system could achieve at least half the parallel efficiency of a high-end DSM system of the same scale.

Unlike using the analytical modelling approach, by using measurements, concrete and realistic results can be obtained from running benchmarks on the system under study. However, since the experiment customisation relies heavily on the implementation of the desired frameworks on a specific machine, the measurement approach limits the experimental flexibility. In situations where the components under study are not available, *e.g.* when the performance of a new hardware design is of interest, a measurement-based experiment seems barely possible unless the prototypes are available. The fact that, in a measurement-based study, some architectural characteristics of the target system are difficult to vary (*e.g.* cache size, memory organisation) leads to the preference for simulation modelling as an experiment platform.

2.5.3 Simulation Modelling

Simulation modelling is a technique of representing the real system by implementing its approximate behaviours in a computer program. Several ways to classify and describe simulation models have been reviewed by [Miller et al., 2004]. Generally, prior to implementing a simulation program, four characteristics of the system under study are gathered in order to formulate a model. These characteristics are the system's timing, states, functionality and randomness. The first characteristic describes how the system deals with time for example, whether the system gives a response based on time (*time-varying v time invariant*) or whether the time when an event occurs is definite (*static v dynamic*). The second characteristic, the system's states, describes whether the system's conditions are carried forward without a pause or interruption, or having a clear independent and separate state (*i.e. continuous state v discrete state*). The third characteristic, the system's functionality, is observed based on how each function is initiated (*e.g. time-driven v event-driven*), and how the function can be represented (*e.g.* descriptive *v* prescriptive, analytic *v* numeric). The last characteristic, randomness, describes whether the system's attributes can be specified without uncertainty once the relevant conditions are known, or the attributes are associated with random probability (*i.e.* deterministic *v* stochastic).

Many simulation models of computer systems represent the model as a discrete-event system. Cassandras and Lafortune defined a discrete-event system model as the discrete-state, event-driven, time-invariant, dynamic model [Cassandras and Lafortune, 1999]. After a model is defined, simulations of the model can be further described by the way a simulation is implemented, *i.e.* execution-driven, trace-driven, interpretation-driven, or application-driven.

Simulation-based research has become a mainstream activity in computer architecture development. This trend is confirmed by statistics showing an increased number of technical papers presenting results based on simulation. An investigation into the publication archive of a premier conference, the International Symposium on Computer Architecture (ISCA), shows that there is a dramatic shift toward analysis using simulation from 28% of simulation-based papers in 1985 up to 90% of such analysis in 2001 [Skadron et al., 2003]. This direction still continues in recent years[25]

[25] In 2002, more than half of technical papers from three system-based conferences sponsored by ACM Special Interest Group on Computer Architecture report simulation results (there are at least 64

A number of simulation models have been developed and some recent approaches have been introduced to make existing simulations more effective. Two examples of the later developments are, a staged simulation [Walsh and Sirer, 2004] that uses function caching, event restructuring and time shifting to improve runtime performance and scale of discrete-event simulators. SimSnap [Szwed and et al., 2004] used a fast forwarding technique to alleviate the simulation performance of the detailed cycle accurate simulation.

From the thirty-two different simulation models referred to in this thesis, the author believes that the availability of standard programming languages (*e.g.* C++ and JAVA), libraries, and computing platforms (*e.g.* SunSparc station and Linux-based platform) plays an important role in the development of simulation models and simulation tools. All of the simulation models published during the year 1991 to 1996 were implemented in C. The majority of the more recent models (published during 1997-2005) were implemented using the mixture between C and C++. A Java-based simulation tool (SimJava) was published in 1998 when the Java language was only a few year old. This tool has been used by a number of researchers to develop many recent large-scale simulation models. Ten recent models which extend a legacy implementation have covered the study on the Intel-based instruction set using a Linux platform. Therefore, the features of the programming languages and programming platforms also impact both the direction and the development of the features which have been included in a simulation.

2.5.4 Discussion

There were several interesting developments in evaluation techniques pinpointed from the works referred to above. Simulation was the most popular method for the evaluation of DSM performance in terms of either using it as a platform for conducting experiments or as a tool to obtain results for verification against another analysis approach. A number of papers present analytical results verified by data obtained from a simulation. Many researchers have presented the measurement-based results compared against simulation results. Several works focused on techniques and methods to

simulation-based research papers out of 116 technical papers which were presented at ISCA, MICRO, and HPCA conferences [Wunderlich et al., 2003]). Moreover, [Vachharajani et al., 2004] remarks from an investigation of simulation techniques that at least 23 out of 38 papers presented at ISCA'2003 used C or C++ as programming languages.

make simulation more efficient in terms of simulation time and scope. The four most popular metrics used in various performance analysis studies are the execution time breakdown, speedup, memory latency and different type of cache miss latencies. Exploration of design trade-offs in various multiprocessors architectures is of interest as much as the impact of programming paradigms on the overall performance. Therefore mechanisms for performance investigation should take into account the flexibility of benchmarks which drive the study.

2.6 Summary

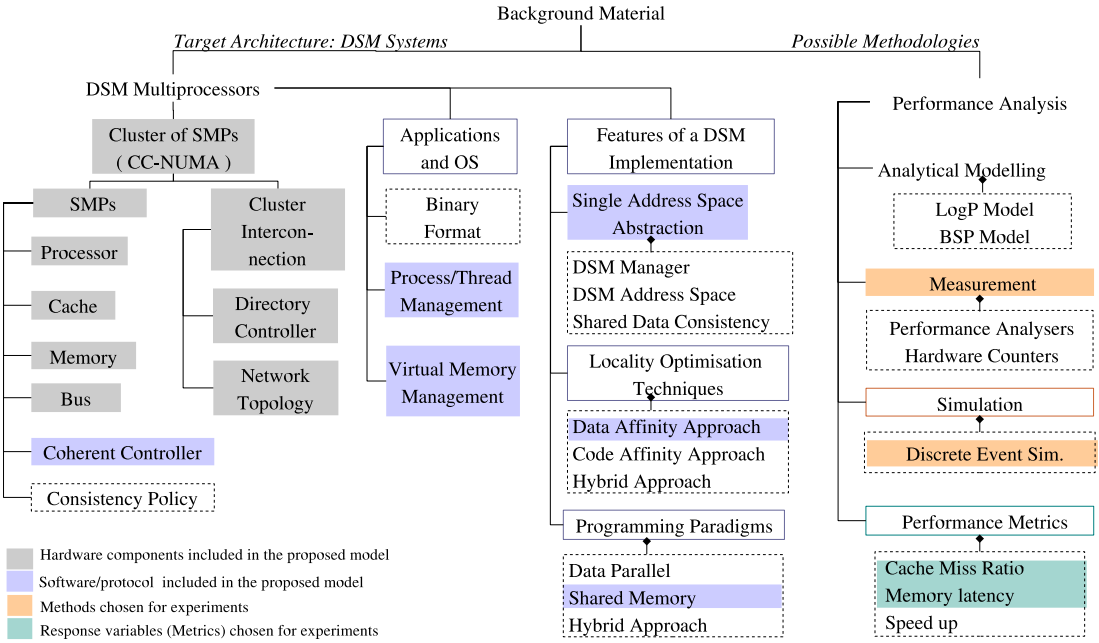


Figure 2.21: Chart summarising the background material and components included in the proposed model.

In this chapter the background material related to the target architecture and the possible methodologies in performance analysis have been described. Figure 2.21 shows a chart summarising the contents presented. The highlighted boxes show the components, software or protocols, methods and response variables included in the research described in this thesis.

The concepts, mechanisms and architecture of a DSM system implemented on a cluster of symmetric multiprocessors have been described at three levels, including:

the machine architecture, the applications and operating system, and the generic features of the DSM implementations. In the latter level, the central concepts involved in a DSM system to provide a single address abstraction and maintain data coherence using different coherence policies are presented. Alongside, three techniques of the locality optimisation based on two approaches (*i.e.* data affinity and code affinity) have been discussed. Three different programming paradigms (*i.e.* data-parallel, shared-memory and hybrid paradigms) used in current DSM systems have been presented.

Another branch of the background material presented is to the summary of three methodologies used in a performance evaluation research of multiprocessor systems: analytical modelling, measurement and simulation. In addition, the issues of complexity, viability, timing and accuracy have been discussed. It has been observed that simulation modelling is a potential tool for the performance analysis of DSM systems. In the next chapter, the related work on performance evaluation of DSM systems using simulation modelling is further discussed.

Chapter 3

Related Work

This chapter presents a survey of related work in simulation modelling of multiprocessor systems published during the years 1990–2005. The perspective or *observation environment* provided in the existing simulation models is used to categorise existing models into the following groups: *workload-oriented*, *system-oriented* and *transaction-oriented* simulation. The first section gives a summary of the key characteristics that allow the simulations of these three groups to reflect the target system response based on the observed workloads, system services, or transaction volume respectively. In Section 3.2, seven of the legacy simulation models used in some performance studies of DSM systems are reviewed. The features of the existing works presented in the first two sections are characterised using five aspects (Section 3.3): (a) how the workload is entered into the simulations; (b) how the behavioural emulation is progressed; (c) how the different architectures are represented; (d) what results have been obtained from the simulations; and (e) what verification features have been provided. This analysis indicates two challenges in simulation techniques to support the exploration space of a DSM system: model extensibility and verification applicability (Section 3.4). In the last section, concepts and challenges of the related work are summarised.

3.1 Simulation Modelling of Computer Systems

In performance evaluation studies of computer systems, simulation modelling is commonly used as a tool to capture a set of behaviours of a target system, in particular,

how such a system would respond to a workload of interest under a variety of possible operational conditions. Once this set of behaviours is captured, it is analysed through repeated manipulation of the simulation model to derive predictions. These predictions are inferred from the statistical analysis of the experimental conditions observed. Computer system simulations used different approaches to set up the observation environments in order to represent the status of a target system with reference to the duration of the target system's uptime. Figure 3.1 illustrates three simulation approaches and the provided observation environments that are commonly used in modelling computer systems. The highlighted area in the figure shows the observation environments created for a simulation run using each of simulation approaches.

Using this observation, the mechanism that a simulation uses to accomplish a representation of a target system can be categorised into the following approaches: a *workload-oriented* simulation, a *system-oriented* simulation, and a *transaction-oriented* simulation. The techniques and characteristics achieved using each of these three approaches are summarised by using the information obtained from the survey of existing simulation models published during 1990-2005. In the next subsections, the characteristics of each approach are presented along with the description of the main issues obtained from the survey. Recent state-of-the-art techniques in modelling simulation of computer systems are also highlighted.

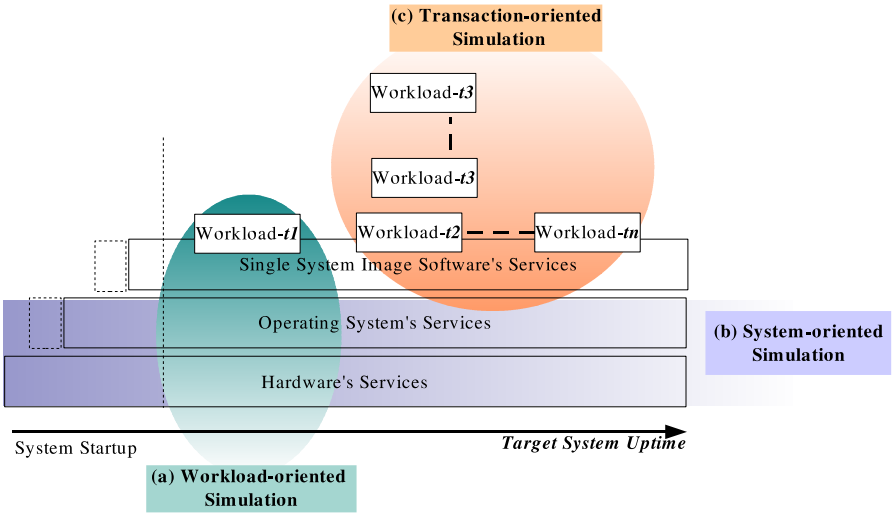


Figure 3.1: Three simulation approaches used for creating different observation environments.

3.1.1 Workload-oriented Simulation

Workload-oriented simulations are those that primarily target performance of a target system in response to the execution of a single application, per each simulation run. During a simulation run, the simulation emulates the behaviour of a target system by creating an environment in which only this job is running on the target machine. Subsequently, in this observation environment, the simulation can be configured to reflect either the intrinsic characteristics or the practical characteristics of the workload under both startup and steady state conditions (*i.e.* the green shaded area in Figure 3.1 (a) can be slid along the x-axis to observe the system responses at the startup or the steady state).

A large number of existing simulation models provide a simulated environment using this approach, for example, TANGO and TangoLite [Davis and Goldschmidt, 1990, Herrod, 1993], PROTEUS [Brewer et al., 1991], MINT and Augmint [Veenstra and Fowler, 1994, Nguyen et al., 1996], LIMES [Ikodinovic et al., 1999], ALITE [Talbot, 1999] *etc.* These models have some common characteristics as listed below.

- Generally, a simulation executable represents an executable of a workload on a target architecture.
- Parallelism in a workload source program (generally in C or FORTRAN) is described by macros[1]. Some of these macros are specific to a simulation, and are used for configuring the observation environment of the target machine to accommodate this workload.
- Workload compilation is managed by simulation tools in the following three steps: macro expansion, code instrumentation and linking with the simulation kernel. Macro expansion configures the observation environment and identifies the place to insert a trap into the simulation code. Code instrumentation inserts a trap into either the assembly code or the object code to transfer control to the simulation engine. This trap is commonly a system interrupt[2] invoking a service handler (a routine of the simulation engine) when a memory access miss occurs.

[1] The majority of existing models use the m4 PARMAC, a set of shared-memory parallel macros defined by the Argonne National Laboratory

[2] Some of the models, WWT for example, used bad error correction code (ECC) instead of an interrupt trap for a fine-grained extension of shared virtual memory [Reinhardt et al., 1993]

- Workload scheduling commonly assigns one simulated thread/process to one target processor. Consequently, the number of threads that a workload creates is assumed to be less than or equal to the number of available target processors, so that a thread can always be accommodated in one of these processors.

- During the simulation run, simulation time is controlled by global, logical timestamps, and is normally synchronised when the simulated thread performs a memory access.

Most of the existing models drive a simulation using an execution-driven technique either on its own, or in a hybrid fashion (*i.e.* used in combination with an interpretation-driven, or a trace-driven technique) to alleviate the speed-accuracy trade-off. This means that a parallel workload is directly executed on a host machine in the form of multiple processes or light-weight threads running interleaved. The execution-driven technique comes with some extra cost affecting the efficiency of a simulation, namely, the cost of synchronisation and context switching, the cost of collecting the memory profile, and the cost of rebuilding the workload for each reconfiguration.

Regarding the cost of synchronisation and context switching, a number of techniques have been developed for different systems. For example, PROTEUS [Brewer et al., 1991] uses a light-weight thread to significantly reduce the cost of synchronisation of a Unix process semaphore. PROTEUS also provides a *quanta* mode to limit the period of simulation time granted on each thread to achieve time balancing and to facilitate synchronisation. MINT [Veenstra and Fowler, 1994] and later models [Nguyen et al., 1996, Thaker and Chaudhary, 2003], however, use a simulation-controlled synchronisation to reduce the cost of context switching by recording only selected registers.

Efficiency in reducing the cost of collecting the memory profile is achieved in the WWT project [Reinhardt et al., 1993] by introducing a memory mapped technique between host- and target- virtual address space, which also shows an improvement in simulation speed. MINT also shows a contribution in this respect by accommodating any simulated threads in a single address space within the simulation process. In addition, MINT also shows that using a form of *code synthesis* to encapsulate each straight-line block of code can reduce the number of unnecessarily generated internal

events.

Customisation and reconfiguration in execution-driven simulation is one of the main issues on which recent workload-oriented simulation models have focused. MINT, for example, shows that by using software interpretation, some customisation can be done by re-linking the workload object file with the parameterised kernel without having to re-compile and re-instrument the original workload. Finally, some of the recent models, like DSM*Sim* [Thaker and Chaudhary, 2003], MICA [Hsiao and King, 2000], COMPASS [Nanda et al., 1998] and SIMT [Tao et al., 2003], achieve customisation and reconfiguration without re-building the simulation executable by implementing an extension on top of the legacy models.

In short, the highlight in workload-oriented simulation is that it provides an insightful understanding of how a particular workload would affect the behaviour of a system. Some existing models, *e.g.* ccSIM [Marathe et al., 2004] and SIGMA [DeRose et al., 2002], indicate the cause of the system response to the high-level language construct of the workload. In this respect, SIGMA uses the information gathered by debugging-mode compilation to relate performance results to the structure imposed by a source program without perturbation of the code sequence. On the other hand, ccSIM uses dynamic binary rewriting to instrument the workload with traps to emit address reference information, including the relation of a reference to its source line. Finally, another interesting mechanism in a workload-oriented simulation is the incorporation of statistical analysis. Different groups of statistics and performance metrics based on the type of architectural component are used in RSIM [Hughes et al., 2002]. This project demonstrated the advantage of carrying out statistical analysis as part of the simulation.

3.1.2 System-oriented Simulation

Despite the potential that a workload-oriented simulation may offer, there are still some limitations in studying the performance impacts of I/O devices' activities, and in the amount of workload to be input into a target system to capture realistic responses. This sort of limitation has motivated the development of system-oriented simulation. The system-oriented approach focuses on overcoming these limitations by providing a representation of a complete system and the full range of its services

during system uptime. Figure 3.1.b shows that with this technique, simulations can set up an observation environment for users which emulates the complete functionality of a target system, *i.e.* similar to the concept of a virtual machine. In general, different types of workload and different patterns of workload arrival are accommodated by an emulation of the application program interface (API). Consequently, this API emulation offers the opportunity to investigate the impact on the behaviour of a target system of job scheduling, load balancing techniques, I/O interface techniques, and the cost of context switching.

In system-oriented simulation, there are two techniques used to implement the representation of a target system (a) *instruction-set simulation* and (b) *full-system simulation*[3]. Instruction-set simulation emulates the operations of every instruction in a target architecture's instruction set, as well as mimicking I/O device routines. Central to instruction-set simulation is an interpretation technique to reflect the operational sequence exhibited during the instruction execution of the target machine. Examples of existing models belonging to this category are SimpleScalar [Austin et al., 2002] and derivative models that extend its core [Huang, 2000, Sandri et al., 2004]. Full-system simulation models all of the hardware components of the target architecture including PROMs. The main difference from other types of simulation technique is that full-system simulation can bootstrap the operating system of the target machine during the emulation. Examples of existing models belonging to this category are SimOS [Herrod, 1998,Magnusson and et al., 2002], SimICs [Magnusson and et al., 2002], and SparcSultima-based models [Clarke et al., 2002, Clarke, 2004]. In summary, these models have some common characteristics as listed below.

- A simulation executable generally represents a target architecture/machine.
- Simulation provides an application program interface (API) to receive a workload executable without manipulating the original workload.
- Simulation emulates the functionalities of the operating system and basic hardware components to provide the runtime environment for the workload.
- Mostly the address translation mechanism is simulated in detail.
- Simulation time is advanced using a fixed-increment approach to represent the time scale of the target system.

[3] So called, *complete machine simulation*

- A full-system simulation also simulates complete functionalities of device drivers, the trap architecture and PROMs to bootstrap the operating system.

Existing system-oriented simulation models drive a simulation using interpretation-driven technique similar to those used in a virtual machine. The simulation provides a runtime environment where the executable file of a target architecture can be run with no workload manipulation required. When a workload is submitted, its execution environment is emulated, and the sequence of code is interpreted using host machine instructions. Key issues in system-oriented simulation include address translation/mapping, instruction interpretation and level of detail of simulated components.

In short, the highlight of system-oriented simulation is that it facilitates a detailed analysis of the target architecture characteristics with respect to operating system interaction. For example, SimICs [Magnusson and Werner, 1995] proposes user-mode address translation and two-phase clock synchronisation to minimise the simulation slowdown. SimpleScalar [Austin et al., 2002] presents a highly extensible skeleton to simulate a new instruction set that permits a new implementation to be plugged in by using a map of mnemonics to callback functions. A new approach to achieve both functional fidelity and performance fidelity of a full-system simulation based on execution-driven technique called *Timing-First Simulation* is introduced and demonstrated in TFSim [Mauer et al., 2002]. Two key issues are proposed including: (a) decoupling the simulation into timing and functional routines and (b) running the timing simulator ahead of the functional simulator in order to get the time-information first, followed by a detailed analysis from the functional simulator.

3.1.3 Transaction-oriented Simulation

The main challenge of both workload-oriented simulation and system-oriented simulation is how to reflect the response of a system to a large-scale transaction-based environment, such as resource management and scheduling on a large-scale, heterogeneous system. This challenge has motivated the development of a transaction-oriented simulation technique where the response of a target system is captured against a variable pattern of incoming workloads. Simulations based on this approach, GridSim [Buyya and Murshed, 2002], Bricks [Takefusa et al., 1999], SimGrid [Casanova, 2001], generally have a set of characteristics listed below.

- A simulation executable represents a set of services provided by a large-scale heterogeneous target system.
- The simulated features mostly centre on system's services rather than on detailed behaviour of architectural components.
- Simulations receive a number of workloads with a variety of incoming patterns to observe the response of the system's services and fault tolerance.

Generally, transaction-oriented simulation models are derived from an application-driven technique using an analysis based on queueing theory. The issues which play an important role in the performance of this type of simulation are: (1) the interface definition required to represent a heterogeneous system, (2) scalability and coverage, (3) timing routines. In short, the highlight of transaction-oriented simulation is that it allows interactions between the workload and a broad range of the services of the target architecture to be captured.

An example of a transaction-oriented simulation is the GridSim [Buyya and Murshed, 2002] simulator. GridSim achieves high scalability by separating the discrete event simulation (DES) engine from the model structure. In GridSim, the simulation of different Grid engines have been implemented by using SimJava [Howell and McNab, 1998], a Java-based DES engine developed at the University of Edinburgh. In SimJava, each simulation component is represented by a SimJava entity running as a thread. Entities are connected together by ports, and communicate via sending and receiving event objects. The SimJava central system manages entity threads, the simulation time, and events delivery. The SimJava framework also allows the progress of the simulation to be recorded through a trace file. The GridSim toolkit provides features for composing workloads, discovery of requested resources, and assigning workloads to resources. It also models heterogeneous computational resources of varied configurations. The model of a large Grid resource broker, Nimrod-G, and the extension of various network topologies [Sulistio et al., 2005] has shown that the design of GridSim on top of the SimJava package is highly modular and can promote model extensibility.

3.2 Simulation of DSM systems

In the previous section, three approaches in which observation environments are provided ranging from workload-oriented, system-oriented, and transaction-oriented approaches are surveyed. Among a large number of legacy models, those that have been used in some performance studies of DSM systems mostly use the workload-oriented approach. In this section, seven such legacy models are presented in chronological order along with their techniques used to accomplish the emulation of a target system. Besides, the models' strengths and limitations are discussed.

3.2.1 TangoLite

TangoLite [Herrod, 1993], developed at Standford University, is an execution-driven simulator that provides a multiprocessor environment through the interleaving of threads on to a MIPS-based uniprocessor machine. The simulator supports target programs implemented in C/C++ or FORTRAN using a shared-memory, multithreading paradigm. TangoLite provides some essential APIs used to define parallelism of a workload program. These APIs comprise (a) some primitives to declare variables in a shared-memory space, (b) some routines to initialise the parallel threads[4], (c) the synchronisation routines and (d) the routines to control accesses to the variables defined in the shared-memory area.

A workload program is built by TangoLite into a simulator (*i.e.* a MIPS executable file) using five steps. Firstly, users insert into the workload program some of TangoLite's APIs to express parallelism of the workload. Secondly, users use the `m4` macro preprocessor to preprocess the modified code. During this step the TangoLite routines are expanded. Thirdly, users compile the expanded code using native C/C++ or FORTRAN compilers into the assembly format. During this compilation process, the expanded macros are translated into the routines of TangoLite's simulation kernel that are instrumented into the assembly code. Fourthly, users use the native `MIPS assembler` to compile the instrumented assembly code into an object program. Lastly, users use the native MIPS loader to link the object program with the required runtime libraries, including the memory simulator library of TangoLite.

[4]Most *routines* in TangoLite are predefined functions and procedures implemented in MIPS assembly code.

After this step, the workload program is processed into a MIPS executable file instrumented with some simulation routines that emulate the architecture of a generic shared-memory multiprocessor.

Once a simulator of a workload is built, users can customise the target architecture by defining a *configuration file*. This file is read by the workload simulator during its execution. The mechanisms of TangoLite can support workloads that extensively use external library calls and provide facilities to explore different workloads reasonably easily. Driving a simulation by using direct execution, TangoLite significantly outperformed the older trace-driven simulation called Tango. However, the simulator generated by TangoLite suffers from lack of portability since it depends on facilities that are specific to MIPS.

Nevertheless, using this scheme, TangoLite offers the possibility to observe the characteristics of a workload in detail as demonstrated by the work of [Connelly and Ellis, 1995]. Observation of various workload characteristics could be used to analyse the architectural bottlenecks of a large scale DSM machines as presented in [Holt et al., 1996]. This work also demonstrated the potential of TangoLite as an experiment platform to simulate a DSM system.

3.2.2 Augmint

Augmint [Nguyen et al., 1996], developed at University of Illinois at Urbana-Champaign, is a multiprocessor simulation toolkit that provides infrastructure to construct both a trace-driven and execution-driven simulation. Augmint translates a target workload into the Intel x86 executable. Similarly to TangoLite, Augmint consists of (a) the compile-time components for workload instrumentation, (b) a run-time environment and (c) a set of predefined routines provided in the Augmint library to emulate the environment of a target machine. The process of building a simulator using Augmint is similar to that used in TangoLite, however the programming interface is different. Augmint supports workloads written in C/C++ expressing parallelism by ANL macros[5]. To simulate a workload, the workload program is preprocessed, compiled, instrumented and linked using Augmint's utilities. The workload program is linked with *(a)* the simulator of a target architecture and *(b)* the Augmint kernel codes. The

[5] Augmint uses the same macro definition as that used in the SPLASH [Singh et al., 1992] and SPLASH-2 [Woo et al., 1995] parallel benchmark suites.

major difference between Augmint and TangoLite is that, in Augmint, the executable file of a workload is *translated* by the simulator of a target-architecture. By doing so, Augmint promotes the flexibility by permitting different instruction sets to be simulated. However, the overhead of the interpretation process is a tradeoff. Therefore in workloads with a large size, performance of Augmint can be noticeably degraded.

Augmint provides facilities to define, create and schedule some arbitrary operations for modelling some features of a target architecture, referred to as *tasks*. Each task has a timestamp value, a priority flag and a function pointer associated with it. The tasks are executed in priority order at their pre-specified timestamps. Creating a task allows some simulated features to be instrumented inside a workload. By this mechanism, the control of the workload execution can be transferred to these simulated features. When execution of the workload hits a task definition, the callback function associated with it is invoked. This function pointer points to the simulator callback function corresponding to the type of event. The value returned from this function controls the thread execution.

A number of simulation models have been developed based on Augmint for example, ABS, Prism, SIMT (Section 3.2.6) and DSM*Sim* (Section 3.2.7). ABS [Sunada et al., 1998] is an implementation of the same engine as Augmint, however, using the SPARC instruction set. Prism [Acquaviva and Jalby, 2000] is an extension of Augmint used to simulate the hardware prediction scheme for data coherency of scientific codes on a DSM system. This work demonstrated the potential to simulate a DSM system by extending the legacy Augmint.

3.2.3 Wisconsin Wind Tunnel II

Wisconsin Wind Tunnel II (WWT-II) [Mukherjee et al., 1997], developed at the University of Wisconsin-Madison, is an effort to develop a portable, direct-execution simulator by expanding a legacy model called WWT-I (*i.e.* a parallel, discrete-event, direct-execution simulator) with four interface operations. These operations are (a) the calculation of target execution time, (b) the simulation of features of interest, (c) the communication of target messages and (d) the synchronisation routines. WWT-II provides these operations in two tools, `Elsie` (*i.e.* an executable editor[6]) and `SAM`–

[6]The program which instruments the target executable file.

Synchronised Active Messages (*i.e.* the neutral programming model involving synchronisation and communication operations among simulation threads).

In contrast to Augmint and TangoLite, WWT-II does not compile a workload program. Instead, it uses the `Elsie` tool to directly instrument the workload executable file. This instrumentation process involves inserting the *call* instructions to redirect the control from workload execution program to the `SAM` library routines. The routines in `SAM` manage communication among the simulation threads, and emulate the execution environment of the target architecture. Different target architectures can be emulated by parameterising the design options of the `SAM` routines. Therefore, in WWT-II, the emulation of target architecture is independent of the workload executable file.

The `SAM` library interface is flexible enough to model different multiprocessor architectures (*e.g.* massively parallel processors, network of workstations, and shared-memory multiprocessors). Two studies from the same research group have shown the potential of the WWW-II to model the hardware features of DSM clusters. Lai and Falsafi proposed a model to predict the memory sharing patterns in order to allow speculative execution on a cache-coherent DSM system [Lai and Falsafi, 1999]. The researchers also compared the effectiveness of using fine-grain memory caching against the hybrid data affinity technique (*i.e.* page migration/replication) in reducing traffic in different DSM clusters [Lai and Falsafi, 2000]. In both works, WWT-II was used as the experiment platform to simulate different DSM architectures on a parallel host machine.

3.2.4 RSIM

The Rice SIMulator for ILP multiprocessors (*RSIM*) [Pai et al., 1997], developed at Rice University, is an attempt to simulate the complete parallelism in processors by modelling the detail of Instruction Level Parallelism (ILP) used in processors. RSIM uses the *interpretation* technique, similar to that introduced in Augmint, to drive a simulation. In contrast to Augmint, RSIM does not produce a workload executable nor translate the workload's machine codes into the host machine's instructions. Instead, RSIM parses a workload program (written in C/C++) into some pre-defined

syntactic tokens and directly interprets them[7]. During the interpretation, the tokens are translated into *SPARC V9* instructions before being passed to the ILP processor simulation. This ILP processor models the detail of processor features allowing the study of processor-level mechanisms to be investigated. An example of the performance study based on RSIM was presented in [Acacio et al., 2002]. In this work, RSIM was used as an experiment platform to study the prediction of cache ownership aiming for accelerating cache-to-cache transfer misses in a cc-NUMA architecture.

However, the more detailed the simulation, the slower the response time. RSIM drives the simulation by translating every instruction using a single cycle to activate processors, caches and memory threads as well as changing simulation state at every clock cycle. This clock accuracy feature provided in RSIM has resulted in a very slow response time.

3.2.5 HASE Shared Memory Multiprocessor Model

The HASE shared memory multiprocessor (SMM) model [Coe, 2000], developed at the University of Edinburgh, is an application-driven simulator modelling the execution of an LU decomposition algorithm on various configurations of the shared memory multiprocessor. The model was built on top of a generic discrete-event simulation (DES) engine called HASE++. Central to this model was a well-formed definition of the model components and their implementations that were then integrated into the DES routines by using a simulation tool, HASE. Unlike any other models mentioned above, the HASE SMM model integrates the C++ program of the LU decomposition algorithm into the simulation kernel. This allows simulation routines to interact with the workload at source code level.

One of the distinctive features of this model is the graphical representation of the target architecture's components that can be used as the means to verify the model interconnection and to reconfigure the component's attributes. Using the HASE tool, the HASE SMM model also provides post-mortem animation to facilitate the model verification. Therefore, the model could be reconfigured to represent different sizes of the shared-memory multiprocessor relatively easily. However, it is difficult to observe different benchmarks, since the source code of a particular workload was inte-

[7] using the `YACSIM` library

grated into the simulation kernel. The HASE SMM model was used to conduct some experiments to quantify the impact of coherence protocols, cache organisations and data granularity on the performance of shared memory multiprocessors of different size. This study demonstrated the highly modular structure of the simulation model achieved by using templates to define the model structure and separating the model implementation to the DES engine.

3.2.6 SIMT

The SIMulation Tool (SIMT) [Tao et al., 2003] is an Augmint-based execution-driven simulator which models various multiprocessor systems with a global memory abstraction. SIMT implements the detail of memory hierarchy including the facilities for designing and evaluating the memory system. These facilities support the design of cache coherence schemes and data allocation policies. In addition, SIMT models DSM features including, a single address space, a DSM manager, and a spectrum of data distribution policies (*e.g.* round-robin, centralise allocation, first-touched) at different data granularity.

An extension of the SIMT model called SIMT/OMP [Tao et al., 2005] is used as an experimental platform to study the impact of a NUMA multiprocessor on the performance of OpenMP applications. The extension of SIMT/OMP from its SIMT original includes modifying the backend of the OMNI OpenMP compiler and providing a new OpenMP runtime library to map the OpenMP programming model to the simulation platform. This modification allows the simulation model to capture the relationship between the execution profile and the OpenMP constructs. Tao *et al*. studied the performance improvement of the benchmarks by varying the memory layout and using a visualisation tool to facilitate the experimental process [Tao et al., 2005]. This study demonstrates the potential of using a simulation model to capture the impact of programming language constructs to the system performance by extending a legacy model. In addition, the extension of Augmint and SIMT allows SIMT/OMP to cover the configurations of a large system based on using the Intel x86 instruction set.

3.2.7 DSM*Sim*

DSM*Sim* [Thaker and Chaudhary, 2005] is an execution-driven simulator that is also based on Augmint. DSM*Sim* is an attempt to replicate the behaviour of a page-based software-DSM system called "Strings", which implements the DSM facilities inside the UNIX kernel. The process to build a simulation using DSM*Sim* follows those used in Augmint. The features of the Strings DSM system were emulated by the routines of the DSM*Sim* library which is linked to the workload executable at the last step of the build process.

At the beginning of a workload execution, a shared-memory region is allocated for this workload (using paging). The simulator uses some hash tables to keep track of the pages residing on each node. Every simulated processing node has a hash table in which its entries keep the information about all shared pages residing on the node. The base address of the page is used to calculate a *hash key* when inserting the new entry into the tables. Every page is owned by one node. Page ownership is migratory at first fault. The simulator also models the atomic memory accesses, page fault handlers, and some other features of the "Strings" DSM system.

Thaker and Chaudhary evaluated DSM*Sim* by comparing the number of page faults obtained from the simulation with the results measured from Strings [Thaker and Chaudhary, 2005]. The study showed that the simulation results of five workloads match the behaviour of Strings across five DSM configurations[8]. The difference in number of page faults[9] confirmed the accuracy of the results obtained from the DSM*Sim* model.

DSM*Sim* is portable across both SPARC and Intel platforms by using the ABS or Augmint as the front end. Some features related to memory consistency models, coherency protocols and data visualisation are planned to be added to the model in future work.

[8] 2×2, 2×4, 2×8, 4×2 and 4×4 DSM models have been tested.

[9] The average difference between the number of simulated page faults to the number of page faults measured is 1.2 percentage point, with the worst case difference of 12 percentage points.

Table 3.1: Five features of multiprocessor simulations.

Feature	Characteristics		Simulation Models						
	Technique	**Ways to integrate to a simulation**	*TgLt*	*Agmt*	*WWT-2*	*RSIM*	*H-SMM*	*SIMT*	*DSMSim*
Workload Injection	HLL program	preprocess, instrument and compile	√	√	-	√	√[a]	√	√
	Assembly code	instrument, assemble	-	-	-	-	-	-	-
	Executable file	instrument by binary code editing	-	-	√	-	-	-	-
	Trace	obtain by a tracer program	-	√	-	-	-	√	√
Driving Technique	Direct Exec.	run a workload executable on the host	√	√	√	-	√	√	√
	Interpretation	translate or use application-driven	-	√	-	√	√	√	√
	Trace-driven	response to commands in trace	-	√	-	-	-[b]	√	√
Supports for Reconfi g-uration	Static Config.	have to modify the simulator code	√	√	√	√	√	√	√
	Recompilation	modify the model desc. &recompile	√	√	√	√	√	√	√
	Library link	link with a new custom library	√	√	√	√	√	√	√
	Config File	set a config file before running a sim.	√	√	√	√	√	√	√
	GUI menu	configure via the graphical interface	-	-	-	-	√	-	-
Profi ling and Performance Statistics	Perf. Counters	provide users the performance counters	N/A[c]	N/A	√	√	√	√	√
	Perf.Metrics	calculate performance metrics	N/A	-	√	√	-	√	√
	Run Profiles	get workload's execution profile	N/A	-	√	√	-	-	-
	Graphs or Plots	create graphs or plots	-	-	-	√	-	-	-
Verifi cation	Trace/Debug	record logs of the simulated behaviour	-	-	√	√	√	√	√
	Specification	use specification-based verification	-	-	-	-	-	-	-
	Visualisation	verify module connections on screen	-	-	-	-	√	-	-
	Animation	verify sim. behaviour via animation	-	-	-	-	√	-	-

[a] integrated to the simulation kernel

[b] Traces are used for verification purposes.

[c] information not available

3.3 Analysis of Related Works

From the survey of related works presented in the previous two sections, five features of the legacy multiprocessor simulations are summarised, including: the workload injection, the driving technique, the support for reconfiguration, the support profiling and performance statistics and the support for model verification. Table 3.1 shows the techniques used in each of these features, the ways the features have been integrated into a simulation model, and the summary of features supported by the seven legacy models listed in chronologically order. Firstly, the *workload injection* describes the way a workload is entered to the simulations. Four types of workload file have been used: (1) the source program written in a high-level language (HLL), (2) an assembly program, (3) an executable file and (4) a trace file. It is observed and shown in the table that the majority of simulation models take a workload in the form of a HLL program and manipulate it to connect to the simulator.

The second feature, the *driving technique*, describes the way the behavioural emulation is progressing (*i.e.* using the direct execution-, interpretation- or trace- driven). Direct execution and interpretation driven are most popular among the legacy models, while trace-driven is rarely used in the newer models. The reason that trace-driven simulation is less preferred in modelling complete multiprocessor systems is twofold. Firstly, performance of a trace-driven simulation is dominated by the size of trace files that are accessed extensively during the simulation. Secondly, the accuracy of results obtained from a trace-driven model depends on the completeness of a trace file obtained by a tracer program. Therefore, the newer models tend to use either execution-driven for simulation speed or interpretation-driven for independence from the host machine instruction sets. The interpretation technique is widely used to obtain flexibility of the workload instruction set.

The third feature, the simulation *support for reconfiguration*, describes the techniques used in simulations to customise the model to represent different target architectures. The structure of a model[10] is normally the most difficult to customise as the customisation normally requires a reasonable amount of modification to the simulation code (*i.e.* the configuration is static). Quantitative parameters like cache sizes, data size or bus size are normally easy to configure by using command line param-

[10] *i.e.* the definition of how the basic components of a model are interconnected both physically and functionally

eters or a configuration file. Some parameters of a model are normally provided in the form of libraries or functions that can be reconfigured through re-compiling or re-linking with the models. Not so many models support reconfigurations via a GUI menu, despite the fact that it is the most user friendly way for reconfigurations.

The fourth feature, the simulation *support for profiling and performance statistics*, describes the types of information about a workload execution that can be obtained from the simulations. Techniques for profiling and statistical summarisation used in existing simulations vary. Most simulation models provide a set of counters associated with each architectural component. These counters may be automatically generated to support dynamic hierarchical caches or come in a fixed but selectable set. Including a set of performance metrics into the simulation model has been introduced in RSIM.

The last feature, the simulation *support for model verification*, describes the mechanisms provided in the simulation models to allow the model structure and the emulated behaviour to be verified. Generally, verification can be done at three phases, before running a simulation, during the simulation run and after the simulation run. Before running a simulation, the model interconnections and specifications of components can be verified by using a graphical representation of the model's components or using specification checking, respectively. During a simulation run, the emulated behaviour can be verified on-the-fly against the component specifications or by animation, as well as being recorded into a log or a trace file for using in a post-mortem verification. Moreover, after running a simulation, the record of the model's behaviour can be reviewed by using post-mortem animation.

3.4 Discussion

Despite the pervasiveness and importance of existing techniques, several concerns have been discussed [Skadron et al., 2003]. In this article, Skadron *et al*. stated the issues about interoperability among simulator codes, representation of a multiprocessor target on a uniprocessor host, and the relationship among three factors including: (a) level of detail, (b) simulation performance, and (c) accuracy of projection inferred from simulation results. Some recommendations have been listed, however, no concrete solution has yet been proposed. Besides, another two issues still remain open,

these issues are related to how to minimise the restrictions of a framework on the exploration space, and how to verify the emulation of new designs.

In a recent simulation symposium, Ammar [Ammar, 2005] pinpointed the limitation of simulation methodology in the context of the credibility of simulation results in emulating large scale networks. Among the four arguments, one reason that may also reflect the limitation of simulating a large scale multiprocessor system is that the acceptable standards for validity and repeatability of simulation experiments are still not yet available.

The analysis of related works has confirmed that, among thirty different simulation projects published during the year 1990-2005 (mentioned in this chapter), the issues regarding the support for profiling, performance statistics and verification of the simulation model are seldom mentioned. However, it has become the new issue in some recent simulation models.

3.4.1 DSM Simulation techniques

To analyse the memory locality problem stated in Chapter 1, the simulation framework has to reflect the essential functionality of the DSM management software to decide where the workload will be allocated in a cluster. A simulation has to incorporate this functionality on top of the emulation of the operating system. In this respect, to both support the exploration space of the memory locality problem and allow the study on a variety DSM configurations, model reuse and inter-operatibility of simulator code to promote model extensibility are essential.

Robinson *et al*. suggest that there is a spectrum of reuse, from using small portions of code, through larger components, to complete models [Robinson et al., 2004]. Regardless of the abstraction model to which reuse is applied, three obstacles to model reuse had been pointed out. First, the motivation for model developers to adopt procedures that would enable model reuse. Second, there is an issue with the confidence that can be placed in code obtained from another context. Third, there is a learning curve of someone else's code that might outweigh the time and cost benefits of reuse. From this discussion, and the analysis of the techniques achieved in the legacy models, model extensibility for the simulation of a DSM system could be applied at three levels as follow:

Extensibility of Simulation Input. To support the workload of different instruction sets, the interpretation technique has been successfully used to allow the flexibility of a simulation input within an acceptable performance.

Extensibility of the Model framework. The structure of a model, the behavioural parameters, and the quantitative parameters have been shown to impact model extensibility differently. The latter two components have been shown to be extensible by using routines implemented in a custom library (as in the Augmint project). However, in terms of extensibility to the model structure, it is noticed from the HASE SMM model that modularity can be achieved by (a) separating the implementation of the model from the simulation engine and (b) using the well-formed specification to define a model structure, so that it can be reconfigurable relatively easily.

Extensibility of Simulation Output. Information obtained from a simulation should also be able to adapt to the custom model and support reconfiguration. The RSIM project has demonstrated the potential of integrating performance metrics to the simulation facility. The collection of profiling data and statistics results has to follow up with the complexity of the target architecture in order to obtain the maximum information from the simulation.

3.4.2 Support for Verification of New Designs

Recent advances in DSM systems have shifted towards performance optimisation using innovative coherence protocols, for example in MOME [Jégou, 2003]. This phenomenon epitomises the complexity of the architectural alternatives. It also imposes the constraint on simulation methodologies not only to cover the new designs but also to ensure the correctness of emulating them. This is because in order to have confidence in the simulation results, it is often more time consuming to verify the correctness of behavioural emulations than to design them.

Over the years, a number of techniques have been proposed to verify that the coherence protocols will behave in accord with their specifications [Bennett et al., 1996, Field et al., 1998], and will maintain two crucial properties, safety and liveness [Pong and Dubois, 1997]. The *safety* (or *soundness*) property means that the protocols always guarantee data consistency, while the *liveness* property assures that there is no deadlock or livelock during the protocol's state transition. In this case,

deadlock is a situation in which two or more caches are indefinitely blocked while each of them waits for resources or acknowledgements to be released from another. Similarly, *livelock* is a situation in which one or more caches is prevented from proceeding further, yet stays indefinitely in a state with no exit. A survey of coherence-protocol verification techniques based on three widely accepted approaches[11] has shown the maturity in methodology to verify that a protocol specification has inherent safety and liveness [Pong and Dubois, 1997]. Moreover, some recent approaches [Pong and Dubois, 2000, Stoy et al., 2001, Tasiran et al., 2003, Delzanno, 2003] have extended one of these methodologies to verify more complex protocols, *e.g.* the adaptive or hierarchical protocol. In effect, this work allows the proof of correctness of a protocol specification by focusing on the protocol semantics without any consideration of the architectural behaviour factors.

Nevertheless, architectural characteristics, like the inclusion property, also play an important role in the correctness of execution results [Chame and Dubois, 1993]. In a multiprocessor system, a multiple-level cache hierarchy has an *inclusion* or *MultiLevel Inclusion (MLI)* property if "the contents of a cache at level $i+1$, C_{i+1}, is a superset of the contents of all its child caches, C_i, at level i" [Baer and Wang, 1988]. Therefore, when a coherence protocol *invalidates* a content of C_{i+1}, the corresponding content in C_i should also be invalidated. Subsequently, this content should not be seen by the processor.

A recent simulation-based research in coherence protocol characterisation [Marathe et al., 2004] compared simulation results against measurements on a real system in order to confirm that its simulation result was correct. Although there are techniques to include, in a simulation model, an automatic verification technique (in [Wainer et al., 2002] for example), a technique to verify all three properties, (safety, liveness, and MLI), in a protocol specification and also to direct a simulation based on the specification semantic, is not yet available.

The analysis of related works has shown the feasibility to develop a mechanism to verify a simulated component during a simulation run. This on-the-fly verification requires two components: (a) the well-form specification of a simulated component and (b) the mechanisms to obtain its semantic from the specification. In this thesis, a component which emulates different bus-based coherence protocols is the target com-

[11] state enumeration, (symbolic) model checking, and symbolic state models

ponent to develop the verification technique. From the related works discussed above, a part of the safety and liveness properties can be verified by checking the specification. However, the full extent of the emulated behaviour has to be verified during a simulation run by testing the simulated results against some verification rules. To allow this runtime testing, the functionalities of a protocol and the assertions of the safety, liveness, and inclusion properties have to be explicitly defined.

3.5 Summary

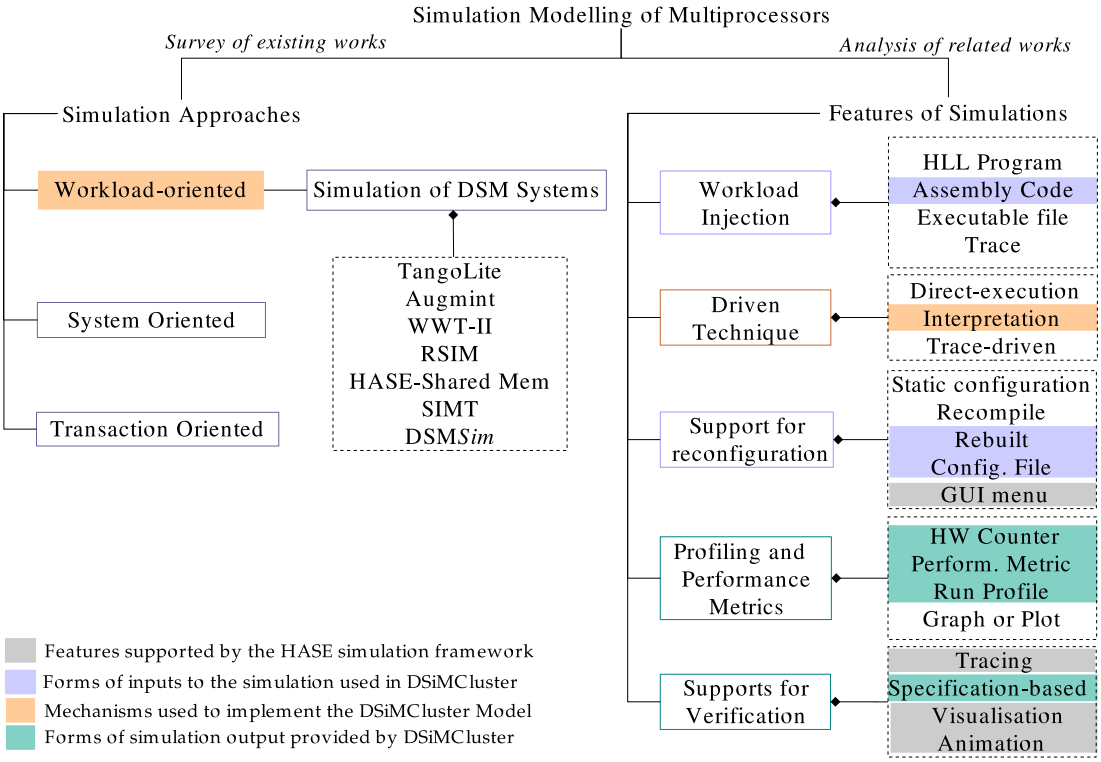


Figure 3.2: Chart summarising the survey and analysis of related works.

In this chapter, a survey and analysis of related works in simulation modelling of multiprocessor systems published during the years 1990–2005 have been presented. A vast number of legacy simulation models have been categorised by the simulation approach used to provide an observation environment of a simulation run with reference to the duration of a target system's uptime. Following this category, existing models have been divided into the workload-oriented, system-oriented and

transaction-oriented simulation. A summary of the key characteristics of each of these approaches has been described with respect to how they reflect the target system response based on the observed workloads, system services, or transaction volume respectively.

Seven of the legacy simulation models used in some performance studies of DSM systems have been further described. These show that workload-oriented is a common approach. Five features of the existing works have been summarised, including: the workload injection, driving technique, supports for reconfiguration, profiling and performance statistics and verification. This analysis has indicated the challenges in simulation techniques to support the exploration space of a DSM system in terms of model extensibility and support for model verification. The development of simulation techniques has shown the feasibility to cover these two issues. A discussion of the feasibility of model extensibility at simulation input, model framework and simulation output has been presented.

The possible solutions summarised from the analysis of existing simulations have been used to develop the model representing a generic DSM system. In the next chapter, the model formulation is presented.

Chapter 4

Model Formulation

Advances in DSM features have forced re-evaluation of simulation modelling techniques in two aspects, namely: *model extensibility* and *verification applicability* (as discussed in the previous chapters). While verification is still a challenge, the study of existing techniques has shown the feasibility of achieving an extensible and highly modular simulation model by distinguishing the model formulation from the details of the DES engine. This thesis considers both aspects as a basis to define the specification of a DSM multiprocessor model prior to constructing a simulation, named DSIMCLUSTER, based on this specification. In this chapter, the specification of the model is presented. The chapter begins by giving an overview of HASE, the DES construction framework used in this work (Section 4.1). This overview aims to clarify the functionality of HASE and its DES engine which serve as the foundation for the construction of the DSM multiprocessor model. After this overview, Section 4.2 presents the model specifications and outlines the DEVS formalism used to formulate these specifications for recreating the essential architectural components of DSM systems described in Chapter 2. In Section 4.3, the verification applicability in the DSIMCLUSTER model is presented. The proposed verification technique is applied to modelling a bus-based coherence protocol, a component that is vital to the accuracy of program execution. This section presents a proposed language called *Protocol State-transition Description* (PSD) to describe protocol semantics. In this section, the results demonstrating the potential to detect three possible errors that may be hidden in a PSD definition are also presented. In the last section, the model formulation is summarised.

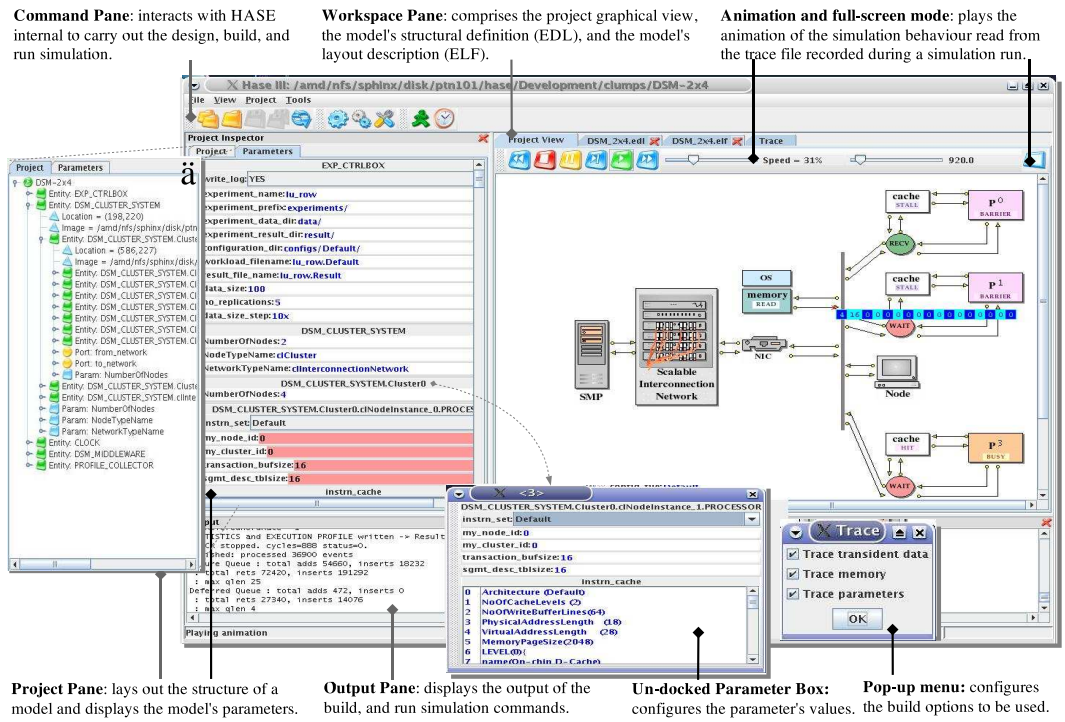


Figure 4.1: HASE application windows.

4.1 HASE Simulation Construction Framework

A simulation construction framework or *simulation tool* [Sulistio et al., 2004] is an underlying system on top of which developers establish a simulation model and its implementation. Primarily, a construction framework includes a language to describe a model structure, a simulation engine to execute (or *drive*) the model, and the mechanism to integrate them. In this respect, a *simulation engine* typically provides an application program interface that abstracts over either language constructs or function calls to allow developers to create the representation of model components. In DES, a simulation engine is generally a library providing support for a set of operations including: creating a set of execution units (*i.e.* threads or processes each of which represents a model component), scheduling these execution units, and managing the exchange of timestamped messages between them.

Among a number of construction frameworks for development of discrete-event simulations, those that provide the mechanisms closest to the target questions of this thesis work are *Parasol* [Mascarenhas et al., 1995], *SimJava* [Howell and McNab, 1998], and *HASE* [Coe et al., 1998]. In this work, HASE is used to construct the

model of DSM multiprocessors on top of the C++-based DES engine called HASE++. The reason HASE has been chosen is fourfold. Firstly, it offers a full framework to support modelling and simulation of a computer system. Secondly, it is based on the C++ language that allows both low-level programming as well as object-oriented design. Thirdly, unlike Parasol, HASE offers more flexibility and portability since it runs on a single processor machine. Lastly, recent publications and ongoing projects on performance evaluation of various computer systems have confirmed the capability of the DES engine in modelling systems at different levels of detail (*e.g.* the Storelite project [Courtney and Chevalier, 2004], UK QCDOC simulation [Alam, 2004]). Besides, the HASE project group at the University of Edinburgh also provides software support and maintenance of the HASE DES engine.

4.1.1 HASE Overview

HASE is an integrated development environment that facilitates the processes of creating, debugging, and running discrete-event simulation models. Figure 4.1 depicts a screenshot of the HASE application window captured on Linux RedHat 9. HASE provides some mechanisms to support the following four steps in modelling a simulation: (1) model design, (2) construction of a simulation executable, (3) experimental control to set parameters and run a simulation and (4) tracing and post-mortem animation. Figure 4.2 shows the software architecture of HASE and the process flow to create and run a simulation model[1].

Figure 4.2 from (a) to (b) shows the process flow of model design. To create a simulation model in HASE, developers first have to provide a *structural definition* of a model using a file written in EDL[2]. The structural definition of a model includes the declaration of a model's entities, their attributes (*e.g.* ports, parameters) and their interconnection. Developers may also provide a layout description to create a graphical representation of a model[3]. They can then use HASE to load the structural definition prior to running a simulation. During this loading process, HASE translates the struc-

[1]Note that the software architecture of the HASE tool presented in this section is based on the HASE version 2.x and 3.x. Later versions of HASE may use the different process flow from that described here.

[2]The EDL (*Entity Description Language*) is the description language used by HASE. This language has a set of rules and conventions used for describing the structure of simulation models.

[3]In HASE, a layout description is implemented via a text file written in ELF (Entity Layout Format).

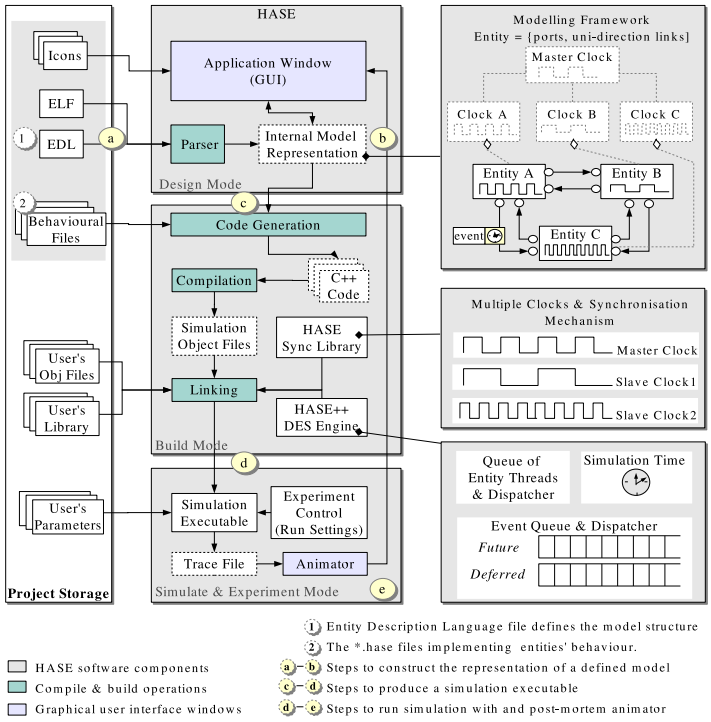


Figure 4.2: HASE software architecture.

tural definition of the model into an intermediate representation and also displays on screen a graphical view of the model.

Developers construct a simulation executable via *behavioural files* wherein the functionality of model entities is implemented[4]. Behavioural files are basically C++ compliant files with added meta tags identifying some HASE intrinsic sections. These sections are used for particular purposes, for example, the `$phase0` section (also called a *clock-phase routine*) is used to identify the semantics to be applied once a clock signal is fired. Figure 4.2 from (c) to (d) shows the flow of the process of building a simulation executable from both the internal representation of a model and model's behavioural files.

A simulation executable acts as a platform for running simulation experiments as its input is reconfigurable and parameterisable. Figure 4.2.(e) shows the process of running and controlling simulation experiments. Thus, HASE developers can customise their models by updating parameters and other configuration aspects on screen

[4]A behavioural file, a text file with `.hase` extension, is recognised in HASE by using a name matching pattern; that is, HASE matches the name of behavioural files against the type names of the model's entities.

before running a simulation. HASE always runs a simulation executable in an independent address space, which records a run profile into a trace file. Finally, HASE also provides a mechanism to animate models using the trace files as input. As mentioned in Chapter 3, animation is useful for model verification. The post-mortem animation allows users to observe the behaviour of a simulation run and the transition of component's states as a response to different types of incoming package.

The following section presents the detailed information about the model framework and mechanisms to handle the simulation time provided by HASE.

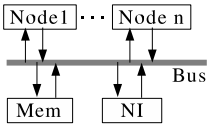
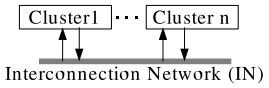
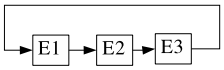
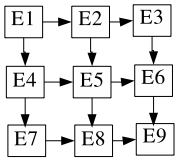
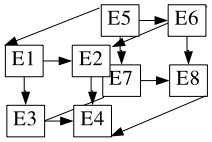
4.1.2 Modelling Framework

The way developers can represent the components of a target architecture in a simulation tool is referred as the *modelling framework* [Sulistio et al., 2004]. HASE is considered both an *entity-based* and *port-based* modelling framework. In a framework like this, architectural components are atomic units called *Entities* that communicate by passing events via *Ports*. In HASE, each entity is composed of states, parameters, ports, and uni-directional links. The states and parameters of an entity define its properties at a particular point in time along with its internal attributes, respectively. The other two components, ports and links, are used as the means of communication between entities.

HASE allows developers to define the structure of a model by grouping together sets of entities that are somehow logically related. A set of entities can be created in two ways: by aggregation into a single entity known as *compound entity* or alternatively using a *design template*. Compound entities describe hierarchical relationships among the entities of a set in *child-parent* fashion. Grouping entities in this way allows developers to describe the vertical composition of a more complex entity from its basic subunits up to its higher abstract-level units. Alternatively, a set of entities that collaborate as different functional units can be grouped together in a design template. A design template describes functional relationships among entities and also provides the means of connection between them. Table 4.1 shows the four architectural structures connecting the entities using the HASE built-in design templates, and those that have been used in the DSIMCLUSTER model. These templates are the `BUSENTITY` describing an SMP structure and the `NETWORKENTITY` describing a

cluster interconnection connected via a user-defined communication network. Alternatively, entities that are interconnected by a static network using one of the ring, 2-D MESH or the *n*-ary *d*-cube topologies (see Chapter 2, Table 2.4) can be structured using the multi-dimensional mesh template (*i.e.* the `MESHnD` templates). A research project has shown that the `MESHnD` templates offer flexibility and scalability for modelling large-scale multiprocessors like the UK QCDOC with up to 20,736 processing nodes [Alam, 2004].

Table 4.1: Architectural structure of HASE templates.

HASE Design Template			
Template Declaration and Architectural Structure			DSIMCLUSTER
BUSENTITY	BUSENTITY Name (NODE (Node) MEMORY (Mem) BUS (Bus) NETWORKINTERFACE(NI) NUMBERNODES(2));		Used for structuring an SMP model
NETWORKENTITY	NETWORKENTITY name (NODE (Cluster) NETWORK(IN) NUMBERNODES(2));		Used for structuring a cluster model
MESH1D	MESH1D name (ENTITY_TYPE (E) SIZE1(3) NO_LINKS(1) WRAP(1))		Not used in DSIMCLUSTER
MESHnD	MESH2D name (ENTITY_TYPE (E) SIZE1(size1) SIZE2 (size2) ... NO_LINKS(1 | 2) WRAP(0 | 1)) MESH3D name (ENTITY_TYPE (E) SIZE1(size1) SIZE2 (size2) ... NO_LINKS(1 | 2) WRAP(0 | 1))	(a) MESH2D  (b) MESH3D 	Not used in DSIMCLUSTER

4.1.3 HASE++ DES Engine

Central to the HASE environment is the HASE++ DES engine [Howell, 1997]. Practically, HASE++ is a multithreaded C++ library that provides core components for

implementing the simulation logic of a model. All parts of the HASE++ library are written as classes and their facilities are presented as a set of headers. The HASE++ library is highly-modular, and independent of the HASE application. The facilities of the HASE++ library enable developers to write DES code without relying on any components of the HASE environment (*i.e.* by implementing the main program, including the headers and linking the executable with the HASE++ library). Alternatively, using HASE, developers gain advantage from the rapid development of a simulation model using the build mechanisms that allow the DES facilities to be integrated into the model automatically.

The HASE++ library can be understood as an approach to converting the model components into executable units along with running them. This approach is carried out via two central classes[5] namely, the `sim_system` and `mgr` classes, that represent the DES system and the DES manager respectively. Figure 4.3 shows the structure of the HASE++ DES Engine as the group of classes composed in the DES system, and DES manager. The following subsection describes the mechanism of each of them in more detail.

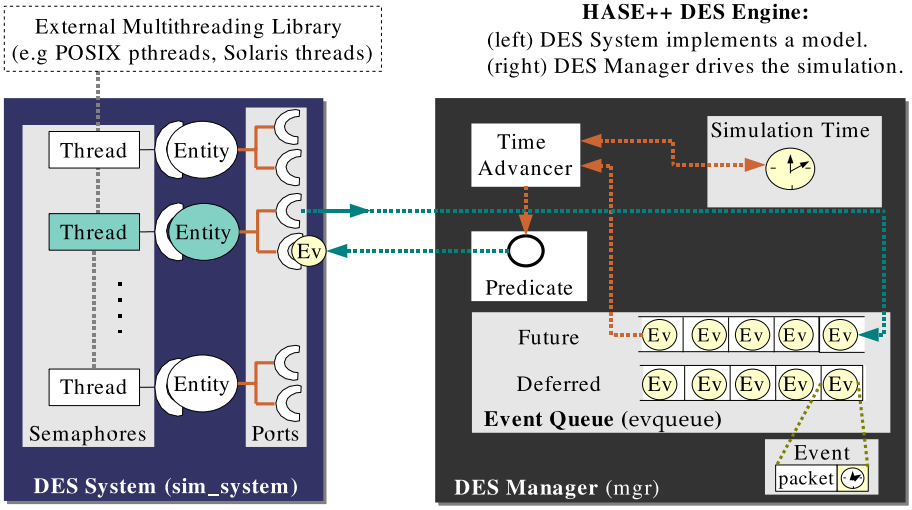


Figure 4.3: Structure of the HASE++ DES Engine.

[5]HASE++ library includes nine basic classes representing: entities, threads, event queue, simulation event, event predicate, semaphores, ports, DES system, and DES Manager.

4.1.3.1 DES System

The DES system is composed of multiple threads, each of which implements a model entity. In HASE++, a thread is a lightweight thread inherited from the thread class of an external multithreading library (*e.g.* POSIX threads, Solaris threads, or the Cray's threading library (REX)). Each HASE++ thread serves as a container holding a model entity. In general, developers can create a thread and plug an entity into the thread by calling the register method of the DES system (the `sim_system` object). In HASE, the code of this registering process is produced by the HASE application during the code generation step (Figure 4.2 (c), Page 105). Consequently, developers do not have to deal with this. Once the entity threads have been created, the DES system continues to manage the list of threads, scheduling the threads, and maintaining semaphores in case of accesses to a shared resource.

During the simulation run, an entity passes events to its connected entities via ports. The green highlighted thread in Figure 4.3 shows that the active thread can pass events to other entities by submitting them to the DES manager[6]. Similarly, the active entity thread can receive events from other entities by checking out events from the DES manager[7]. Both send and receive actions are performed through ports by calling the methods of the DES system. These methods then interact with the DES manager for delivering the requested events.

4.1.3.2 DES Manager

The DES Manager maintains simulation time and manages event queues. As shown in Figure 4.3, a DES manager comprises two event queues, namely a *future* queue and a *deferred* queue. The future queue is used to store events that contain timestamps referring to the time in the future when these events are expected to occur. On the other hand, the deferred queue is used to store the events that cannot be dispatched at the specified timestamps because the receiving entities are not ready to get

[6]To submit an event, the sender entity calls the method `send_<port_name>` of the DES System and passes the event with or without a predicate as a parameter.

[7]To check out an event, the receiver entity calls the method `sim_wait` of the DES system to search for the most imminent event from the future queue that matches the required predicate. For a non-blocking receive, the receiver entity calls the method `sim_waiting` of the DES system to search for an event matching the required predicate from the deferred queue. The matched event is then checked out using the `sim_get` method.

any events[8]. Normally, developers can decide to check out an event from either a future or deferred queue by invoking different methods (`sim_wait` for future queue, and `sim_waiting` for deferred queue). In a handshake situation, when a blocking send/receive is implied, events are normally dispatched from the future queue. However, in a non-blocking situation, when events can be snooped at any point in time, the deferred queue is normally used. Note that events are sorted in both queues in ascending order of the timestamps. Event timestamps also play an important role in advancing the simulation time. The next section describes the issue of simulation time and time advance in detail.

4.1.4 Simulation Time and Time Advance

In HASE, *simulation time* refers to a value representing an absolute or relative moment in time when the simulated events occurred. The HASE++ DES engine maintains the order of simulated events by keeping track of simulation time. It maintains the global *simulation time* by using the *next-event time advance* [Law and Kelton, 1991] mechanism.

The simulation time is an attribute in the HASE++ DES engine, which is initialised to zero at the start of a simulation. During a simulation run, entities create events, and also generate timestamps to set the times of occurrence of such events prior to submitting them into the future queue. Therefore the simulation time does not progress based on a fixed time unit[9], it is instead advanced to the time specified by the timestamp of the most imminent event in the future queue. After this process, the selected event is dispatched, and the state of the system is updated to account for the fact that an event has occurred. This process continues until either a set of prespecified stopping conditions is satisfied, or a terminate signal is scheduled from one of the participating entities. Using the next-event time advance approach, the HASE++ DES engine efficiently supports communications between entities using a handshake protocol, as it effectively saves computational resources to simulate periods of inactivity by jumping the simulation time from event time to event time.

[8] An entity can be made inactive by invoking the `sim_hold(delay_period)` method of the DES system. In this case the entity will be halted and will not receive any events.

[9] To advance a simulation time using a fixed unit of time related to the host wall-clock time is called *fixed-increment time advance* [Law and Kelton, 1991] approach.

4.1.4.1 Library of Synchronisation Routines

In addition to the HASE++ timing routine, HASE also provide a set of abstract entities and a synchronisation mechanism. These facilities for entity synchronisation are presented in the *Sync* library [Mallet et al., 2002][10], as an infrastructure for rapid development of a clock-based simulation model, such as a computer system simulation. The library is composed of a hierarchy of abstract classes (*e.g.* `Clocked` and `Biclocked` for the one-phase and two-phase clocked model respectively) whose implementation provides both the routines to maintain the simulation clock cycle and the interface to invoke behavioural functions of a model's entities according to each clock tick. Figure 4.5 (Section 4.2.2 on page 115) illustrates how the abstract entity `Biclocked` is plugged into an entity, and how clock-phase behavioural functions are defined.

This synchronisation mechanism is implemented on top of HASE++ DES simulation time by having the abstract class continuously submitting *clock-tick events* into the future queue. This clock-tick event will trap the control of simulation execution into the main `Clock` entity. Once control is transferred, the main `Clock` entity sequentially invokes the corresponding clock-phase routines of every entity registered with it, before submitting a new clock tick event to trap the next simulation time. This process continues until the end of a simulation. Details of the reusable and extensible design, and implementation of a multiple clock mechanism can be found in [Mallet et al., 2002].

4.1.4.2 Clock-enabled Entities

In the case of a synchronised model, an entity can be clock-enabled by the developer creating it as an extension of a `Clocked` abstract class. This abstract class, provided in the `Sync` library, maintains the timing information and transfers this information to the registered entities by calling the clock-phase routines. A group of entities registering to the same `Clocked` instance work under the same precision of clock tick, referred to as a *clock period* (*i.e.* a unit of time relative to the global simulation time). In HASE, developers can also define a synchronised model using multiple clocks. To do so, developers have to first create a *master clock* and set the precision of the *master*

[10] Library of Synchronisation Routines

clock period as a reference. Once a master clock is created, developers can create multiple `Clocked` instances, each of which uses a different clock period proportional to the master clock period. Using this mechanism, multiple periods of clock are accurately controlled by the master clock instance which maps the proportional time of its children to its timescale. Once the clock period of one of the children is due, the master clock issues a clock signal to the child clock, thus invoking the corresponding clock phase routine. Figure 4.2.(b) shows an example model composed of three clock-enabled entities in which each is set to work at a different clock period.

4.2 Model Specification

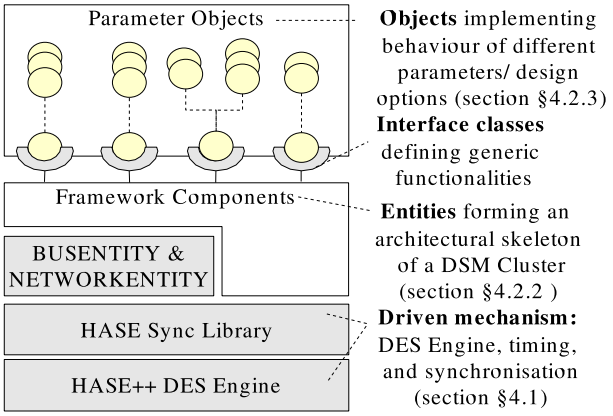


Figure 4.4: Block diagram of DSIMCLUSTER architecture.

In the previous section, the functionality and features of HASE and the included DES engine have been described. HASE and its tools provide the facilities essential to the construction of a simulation model of DSM systems. As mentioned in Section 2.5.3 (Page 74), prior to implementing a simulation program, four characteristics of the system under study should be stated in order to formulate a model. These characteristics are the system's timing, states, functionality and randomness. Many simulation models of computer systems represent the model as a discrete-event system. Cassandras and Lafortune defined a discrete-event system model as a discrete-state, event-driven, time-invariant, dynamic model [Cassandras and Lafortune, 1999]. Following this definition, a formalism known as the Discrete EVent systems Specification (DEVS) is commonly used to describe the model specification. Therefore, in

this work, a model of a generic DSM system is first formulated by using a DEVS-based language [Zeigler et al., 2000]. After this model is defined, simulations of the model (the DSIMCLUSTER simulation) can be developed in accord with the model specification.

This section presents the specification of the DSM simulation model by first describing the DEVS formalism used to define the model specification. The aim of the model specification is to recreate the essential architectural components of DSM systems by abstracting away the unnecessary detail. To achieve model extensibility, the model components have been classified into two groups: *framework components* and *parameter objects* (see Figure 4.4).

As shown in Figure 4.4, framework components are those that form a generic skeleton of a DSM cluster architecture, and also interface directly with the HASE++ engine. The behaviour of these components can be mapped into a variety of parameters via a generic interface. In this section, after the DEVS formalism is explained, an example is used to show how the DEVS description is used to define a framework component and how the definition is developed to the HASE definition. This section also shows an example of the DEVS formalism describing a *cache organisation* parameter. A generic interface for such a parameter is also described.

4.2.1 The DEVS formalism

DEVS (Discrete EVent systems Specification) is a well-known formalism which serves as an abstract basis for model specification. A DEVS atomic model describes that a model M represents a component of the target system by mimicking the states, events, and functions related to the component. A DEVS atomic model is formally described by [Zeigler et al., 2000]:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau >$$

where

$X \equiv$ set of input events,
$S \equiv$ set of sequential states,
$Y \equiv$ set of output events,
$\delta_{int} : S \rightarrow S \equiv$ internal transition function,

$\delta_{ext} : X \times Q \rightarrow S \equiv$ external transition function, where

$Q = \{(s,e) | s \in S, 0 \leq e \leq \tau(s)\} \equiv$ total state set,

$e \equiv$ time elapsed since last transition,

$\lambda : S \rightarrow Y \quad \equiv$ output function,

$\tau : S \rightarrow \mathbf{R}_0^+ \equiv$ time advance function.

The atomic DEVS formalism describes the attributes of each atomic unit of a target system based on its states, the state transition, and the duration of time allocated for each state. A system always stays in one of the states, s ($s \in S$), and remains in this state for the period of the resting time τ if no external event occurs. A system with the time τ as a real number will stay in state s, until the resting time expires (*i.e.* elapsed time $e = \tau(s)$)[11]. The system then outputs the value y ($y \in Y$) as a result of performing the output function $\lambda(s)$. After that the system changes its state to that signified by the internal transition function $\delta_{int}(s)$.

If an external event $x \in X$ occurs when the system is in a state s at elapsed time e ($e \leq \tau$), the system performs the external transition function $\delta_{ext}(s)$. The transition function identifies the system's new state s' according to the values of the corresponding input x, current state s, and elapsed time e. After the transition, the system is in some new state s' with some new resting time $\tau(s')$. This process of state transition continues until the end of the simulation.

4.2.2 Specification of Framework Components

The interpretation of the DEVS formalism described above is used to define the specification of the DSIMCLUSTER model in both the framework component and parameter object groups. As mentioned earlier, in the DSIMCLUSTER, framework components are entities that constitute the architectural foundation of a DSM cluster. These (nine) basic entities are: a processor, a cache hierarchy, a bus, a memory unit, an operating system, an interconnection network, a (coherence) directory controller, a DSM manager and profiler, and an experiment controller. A specification of each of these entities is gathered from a survey of existing machines taking into account lists of

[11] Note that a state s can have $\tau = 0$ or ∞ if it is a transitory state or a passive state respectively. A *transitory* state refers to the state where the time τ is too short for an external event to intervene. In contrast, a *passive* state refers to the state where the time τ is ∞. The system with a passive state will stay in the state indefinitely unless an interrupt from an external event occurs.

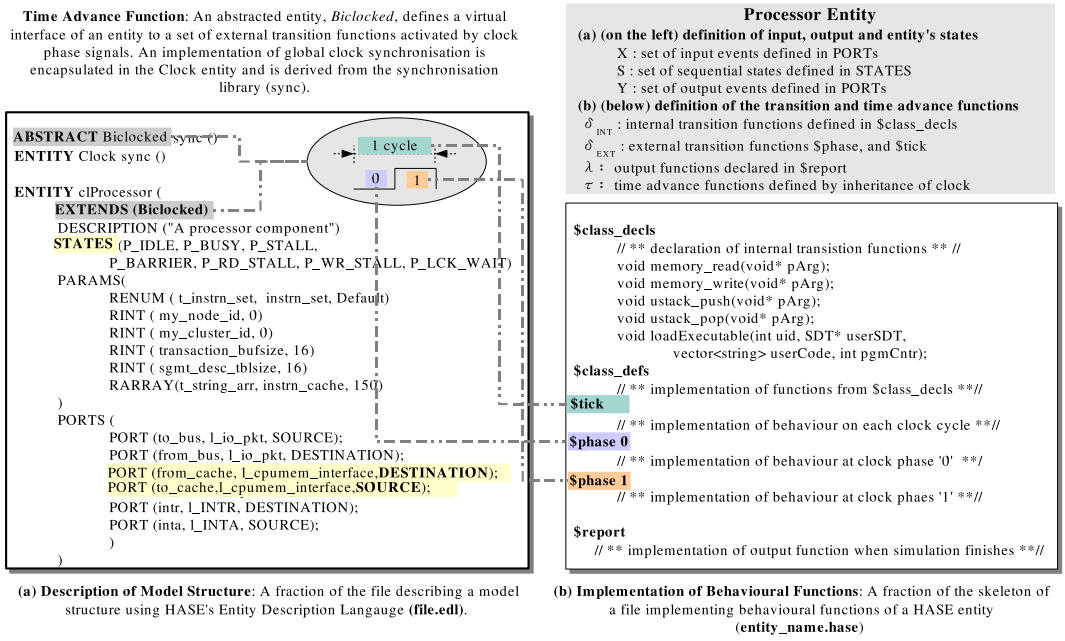


Figure 4.5: A specification describing the structure of a processor in HASE.

specific features, forecast contents, possible organisation schemes, and a description of common functionalities (presented in Chapter 2 Section 2.1 - 2.4). An example of this is shown in Figure 4.5, where the specification of a processor entity is outlined (the full specification is presented in Appendix A).

Translating an entity specification into HASE requires the entity to be defined in the EDL file of the model, and the entity's semantics to be described in a `.hase` file. This process is illustrated in Figure 4.5 using a DEVS model describing a processor entity. Firstly, the lists of specific features, states or functioning conditions, input and output events of the entity are defined in the model EDL file. In this case, as all of the framework components operate under a two-phase clock signal, the time advance function is defined by deriving the processor entity from the built-in type `Biclocked` that is also declared in the EDL file. Secondly, the rest of the specification is described in the *$class_decls* section of the entity's `.hase` file. This part of the specification, referred to as the functional definition of an entity, includes any behavioural routines that will be activated on each clock-tick, and the internal and external transition functions.

The architectural skeleton of a DSM cluster is structured into a set of compound entities that group together the nine basic entities. These compound entities are

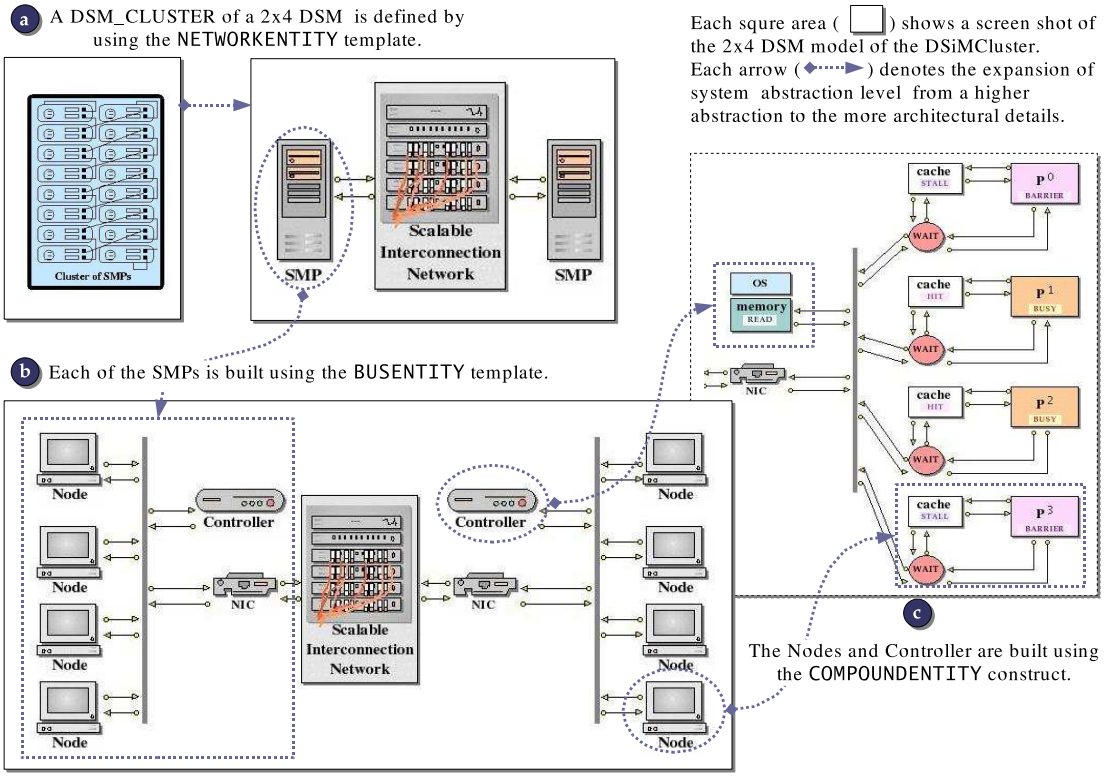


Figure 4.6: Structure of a 2×4 DSM defined in DSiMCluster.

plugged into HASE built-in design templates according to their role (Figure 4.6). Three of the entities, namely: the processor, cache hierarchy, and bus interface, are aggregated into a compound entity called *Node* (*i.e.* a processing element node). A variable number of Node entities, a memory entity, a bus entity, a directory controller entity and an operating system entity are plugged into a single SMP structure using the `BUSENTITY` template. Finally, to form a cluster, the set of SMP entities, an interconnection network entity and a DSM manager entity are encapsulated into a single entity called `DSM_CLUSTER` using the `NETWORKENTITY` template.

4.2.3 Specification of Parameter Objects

The unique characteristics of each DSM implementation arise from two factors: (a) the selection of operational elements included in each entity, and (b) the configuration of tunable parameters in the system. DSIMCLUSTER refers to these two factors as parameter objects whose combination determines the operational sequence of a simulated architecture. The specification of a parameter object is derived from the variety of existing implementations by filtering out the generic features, the description of common functionalities, and the operational sequence. A number of parameter objects are covered in the DSIMCLUSTER model as shown in the third column in Table 4.2.

Figure 4.7 shows an example of the specification of a cache organisation object. In the figure, a generic specification (Figure 4.7 (a)) describes the common attributes of this type of parameter and its interaction with the framework component, `Cache Entity`. The specification of a particular implementation of a cache hierarchy can then be obtained based on these common attributes. Figure 4.7 (c) shows an example of a configuration file describing a two-level cache hierarchy with a coherent territory cache. The specification consists of some header information and a description of the cache hierarchy organisation.

The header information identifies the name, the address length (of both the physical and virtual address), the size of a memory page, the size of a write buffer and the total number of cache levels. Each of these levels is further defined in a description section; each section includes a set of structural organisation and behavioural configurations. The structural organisation defines the architecture of the cache (*i.e.* line

Table 4.2: Summary of the components modelled in the DSiMCluster.

DSiMCluster **Specification**			
Architectural Components			
Framework Components	*Parameters*	*Parameter Object*	*Varieties/Options*
Processor	Instruction Set SDT Size	Instruction Set SDT[a]	DEFAULT, SPARC Direct Mapping
Cache	Configuration File Bus-based Coherence	Cache Organisation Coherence Protocol PSD Parser	As shown in Table 2.1 WI, Synapse, *etc.*
Bus Interface &Bus	Block Size Delay		
Memory	RAM configuration	RAM	As shown in Table 2.1
Interconnection Network	Routing techniques	Routers	Wormhole Store-&-forward
Directory Controller	Protocol	Distributed Directory Central Directory	SCI CD-INV
Emulation of System Software			
OS	Process Management Memory Management Segment Paging Resource Allocation Linking Loader	Process Table Memory Mapped Table SMT, PMT[b] Allocation Linking Loader	Thread/Process Scheduling Policies Virtual, Physical Allocation Policies Address length Frame size FIFO data distribution
DSM Management	Consistency Model Consistency Technique Optimisation	Consistency Object Consistency Technique Optimisation Object	Homeless LRC Home-based LRC Update, Invalidate Data Replication
Profiler &Experiment Controller	Counters Perf.Metrics	Counters Metrics Functions Metrics Machines	Processors User defined Fns Map to Fns

[a] Segment Descriptor Table

[b] Segment-Mapped Table, Page-Mapped Table

size, number of cache ways, number of sets, and cache associativity) and its indexing method (*i.e.* virtually or physically indexed, virtually or physically tagged). Note that the indexing method also denotes whether the translation lookaside buffer (TLB) is to be used to translate an effective address into the corresponding tagged address. If a cache is virtually-indexed physically-tagged, a TLB will be used for address translation and the size of TLB is also taken from the model configuration. The behavioural configurations of a cache level include specifying the policies for cache replacement, write access, and write miss action, as well as a protocol used for coherence control if required.

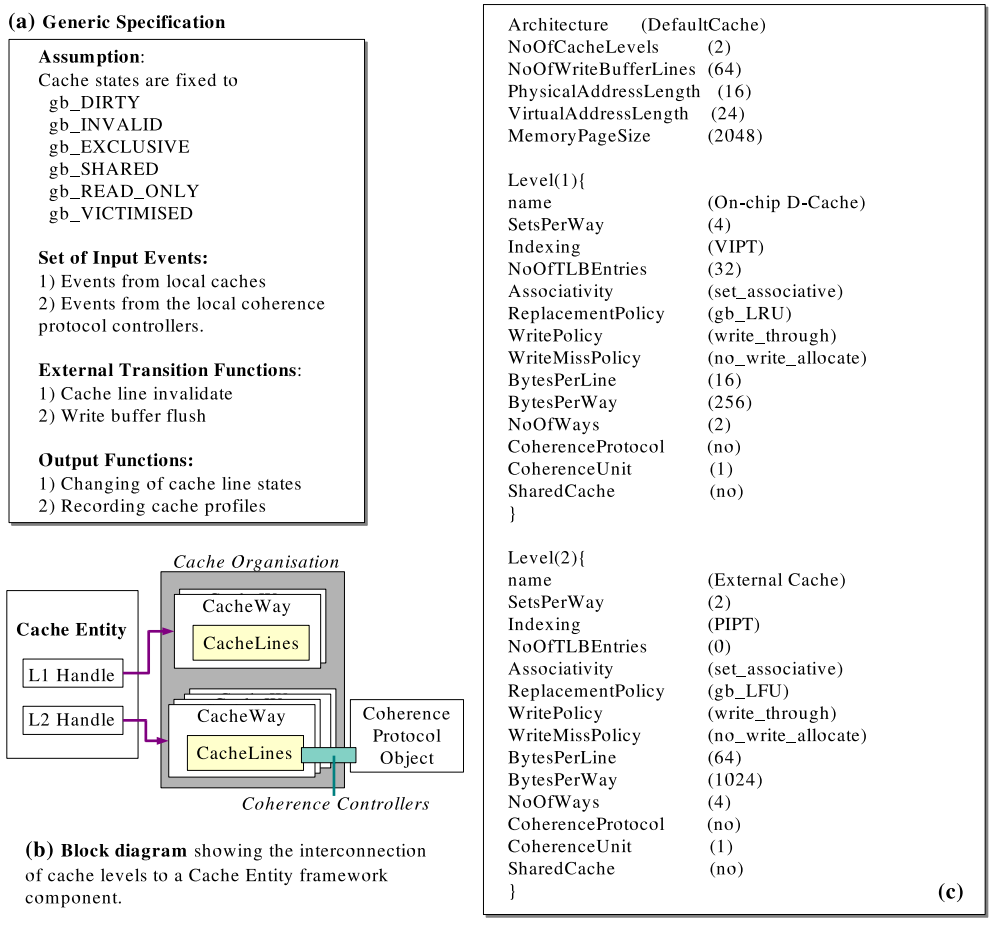


Figure 4.7: Specification of a `CacheOrganisation` object.

The configurations of parameter objects are described in text files. Some of these files are directly read by a framework entity in order to re-configure the entity's features, for example, a specification of cache configuration. The parameter

objects described and taken into the DSIMCLUSTER model in this fashion are called *configuration-based parameter objects*. It is important to note that a parameter object has no timing component. All states of a parameter object are *passive*. This means that a parameter object will stay in a state indefinitely unless there is an interrupt request invoked by its connected framework entity. In this case, the framework entity *drives* its parameter objects during its active period (*i.e.* the time allocated for the entity clock-phase routine). Therefore, the simulation time taken to emulate a model behaviour by a parameter object is shared with the framework entity to which the object is connected.

4.3 Specification-based Verification

Verification applicability is a challenge for computer system simulation. As described in Chapter 3, one of the limitations of existing simulation techniques is the time delay caused by the model verification process. The related works described in Chapter 3 have shown that coherence protocols and memory consistency models play an important role in terms of both performance and accuracy of a DSM system. This research aims to find a simulation technique that can both a) verify the specification of coherence protocols and b) apply the verified specification to drive a simulation. Research on coherence protocol verification has shown the possibility of ensuring two properties, *safety* and *liveness* (see Section 3.4.2 page 98), that lead to correctness of coherence protocols using the specification-based analysis. Based on observations drawn from existing research, a text-based description language to identify a coherence protocol specification is proposed. Alongside this, early in the design process, ways to detect possible errors that contravene both properties in the protocol specification have been developed. The following subsections describe each of these issues in detail.

4.3.1 Protocol Specification

Fundamental to protocol verification is a symbolic or formal representation to describe a protocol specification which can be used for semantic comparison against the actual behaviour. Recent research on coherence protocol verification has em-

ployed various specification languages relating to the techniques exploited, for example, Murψ [Dill et al., 1992], *Spade* formalism [Field et al., 1998], SSM [Pong and Dubois, 2000], TLA^+ [Tasiran et al., 2003], a table-based specification [Sorin et al., 2002], and EFSMs [Delzanno, 2003]. In general, a specification describes a coherence protocol as a composition of three components: a finite set of states, a finite set of actions or events, and a transition relation. In EFSMs, the global conditions and a description of global conditional action have been included. The conditional action is used to express the actions in some protocols in which different sources of a new cache line[12] cause the transition to different outcome states. Moreover, the global condition is used to describe the permissible *global states* that facilitate the protocol verification. A protocol global state, *i.e.* the collection of individual cache states [Delzanno, 2003], denotes the reachable state of all cache replicas as the outcome of a state transition. Thus, if a record of global states can identify the possible sources of data inconsistency which negates the safety property.

Existing representations are sufficient to represent the state machine of a protocol, thus allow the possible state transitions to be verified. However, as this work aims to find the way to use the verified specification as a script (or a mapping function) to directly drive a simulation, some practical aspects of the protocol should also be included. The components that have not been included in existing representations, yet are essential to describe an implementation of a coherence protocol in a simulation model are proposed. Three proposed features of a protocol representation include the ways to describe the following attributes:

- the coherence protocol actions that are based on a priority test or cache line ownership;

- the state when a protocol transition is halted, waiting for bus arbitration or held while waiting for an acknowledgement;

- the mapping functions needed to map the specification term to the implementation term and direct a simulation using these functions.

[12] *i.e.* from main memory or from a remote cache

4.3.1.1 Description Language.

To bridge this gap, a *Protocol State-transition Description* (PSD), a text-based description language designed for describing both the pragmatic and semantic attributes of a coherence protocol has been developed. In semantic terms, the PSD language describes a state machine of a coherence protocol as a composition of a finite set of states, a finite set of actions or events, and a transition relation, similar to the existing work described above. The global conditions and conditional actions as presented in EFSMs are also included in PSD. In pragmatic terms, the PSD language can express the actions between a coherence protocol and the interconnected components. Figure 4.8 shows the interconnection between two coherence protocols and the framework components in a 2-node SMP. Each coherence protocol is connected to a cache and the shared bus. To maintain cache coherence, a protocol snoops for the read/write events from the shared bus and also receives the read/write events from local caches.

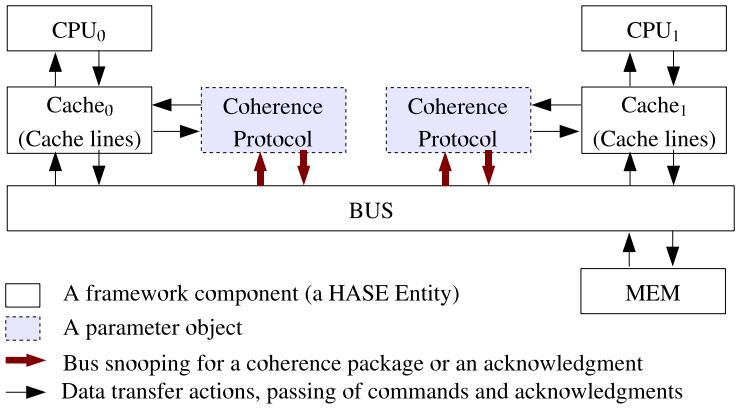


Figure 4.8: Organisation of coherence protocols and components in a 2-node SMP.

In PSD, there are four groups of reserved words describing: the (architectural) components (*e.g.* `BUS`, `CPU`, and `MEM`); the predefined states of a cache line; the accepted events; the transition-function identifiers. The component reserved words are used to specify the source or destination of an event, *e.g.* `CPU` identities that the events are initiated from the CPU that is connected to the local cache. The remaining three groups of PSD reserved words are listed in Table 4.3. The first column in the table gives an abbreviation code used to refer to each reserved word in the following sections. Five predefined states (CS1-CS5) of a cache line are used as the generic states

to map with any user-defined protocol states. Read and write accesses to a cache line cause eight possible events (EV1-EV8) to be received by a coherence protocol. These events identify the location of the access (*i.e.* CPU for local caches, and BUS for remote caches). The transition-function identifiers (F1-F13) are the predefined internal transition functions. Each of these function identifiers is used to describe a non-atomic action (or a partially executed action) to be performed in order to transit from one protocol state to another. These functions are then mapped to the implementation of a coherence protocol in a simulation (see Section 4.3.2.1).

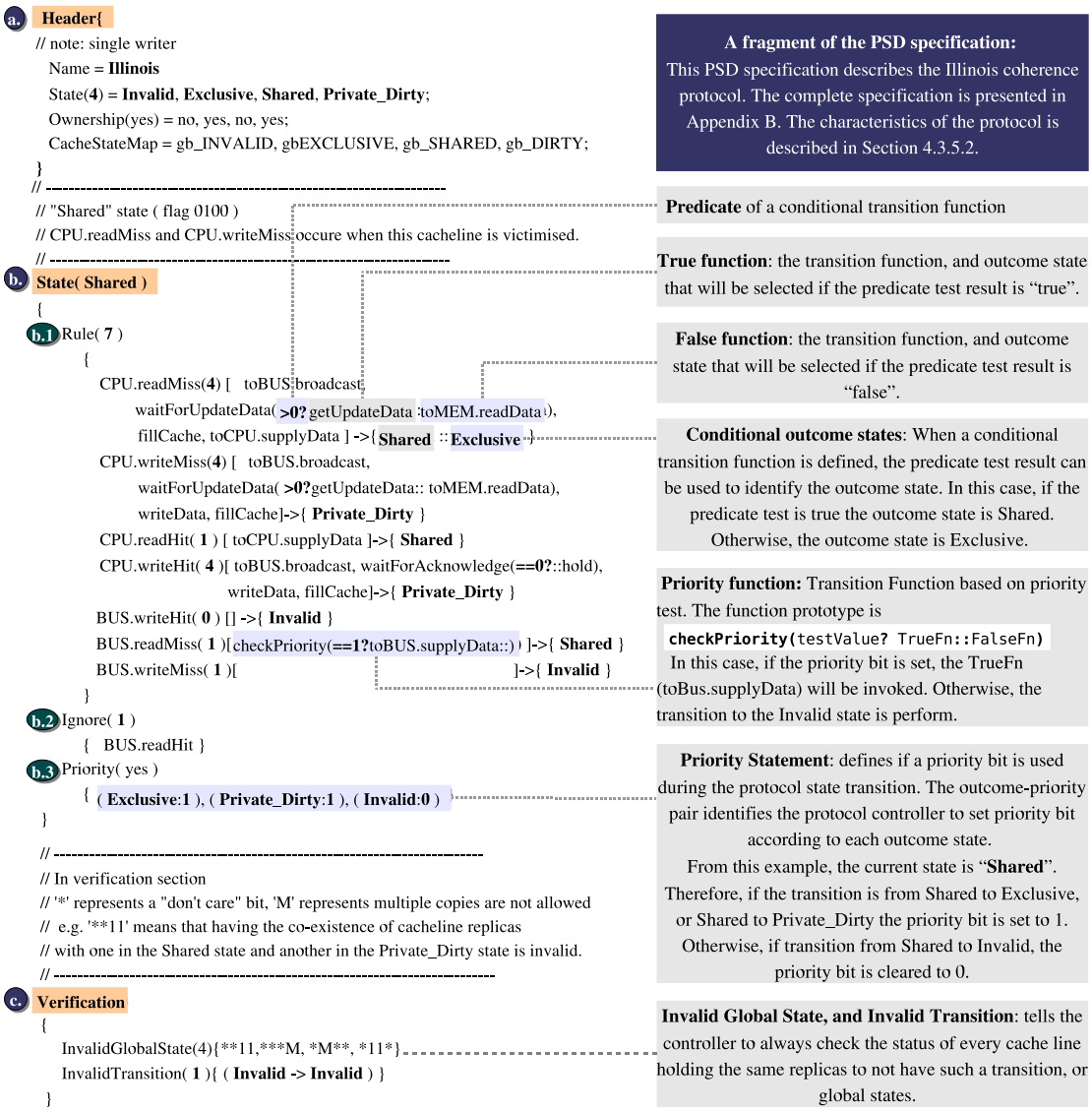


Figure 4.9: Specification of the Illinois coherence protocol.

Table 4.3: The PSD reserved words.

Predefined states of cache lines

	Reserved Words	Cache Actions
CS1	`gb_INVALID`[a]	cache content cannot be used
CS2	`gb_EXCLUSIVE`	reads&writes allowed with no delay
CS3	`gb_SHARED`	reads allowed with no delay, writes after coherence actions
CS4	`gb_READ_ONLY`	reads allowed with no delay, writes after coherence actions
CS5	`gb_DIRTY`	reads&writes after coherence actions

Eight accepted events

	Reserved Words	Description
EV1	`CPU.readMiss`	A read miss at a cache line of the local cache[b]
EV2	`CPU.readHit`	A read hit at a cache line of the local cache
EV3	`CPU.writeMiss`	A write miss at a cache line of the local cache[c]
EV4	`CPU.writeHit`	A write hit at a cache line of the local cache
EV5	`BUS.readMiss`	A read miss at a cache line of a remote cache
EV6	`BUS.readHit`	A read hit at a cache line of a remote cache
EV7	`BUS.writeMiss`	A write miss at a cache line of a remote cache
EV8	`BUS.writeHit`	A write hit at a cache line of a remote cache

Predefined internal transition functions

	Reserved Words	Description
F1	`writeData`	write the specified data words to the cache line
F2	`hold`	coherence action is held until a predicate test is success
F3	`waitForUpdateData`	wait for a data-updated package from bus
F4	`getUpdateData`	get the updated data from the received package
F5	`fillCache`	fill the entire cache line with the received data
F6	`sendAcknowledgement`	send an acknowledgement (to a remote cache)
F7	`checkPriority`	test the cache priority flag
F8	`waitForAcknowledgement`	wait for an acknowledgement
F9	`toBUS.broadcast`	broadcast an action/event to bus
F10	`toBUS.supplyData`	put a data-updated package to bus
F11	`toCPU.supplyData`	allow cache to send data to CPU
F12	`toMEM.readData`	send a memory-read request to the local cache
F13	`toMEM.flushCacheline`	send a memory-flush request to the local cache

[a] Initial State

[b] This happens because a) the cache line is in the gb_INVALID state or b) the cache line is to be replaced (is a victim line).

[c] Similar to a read miss, a write miss happens either because the cache line is in the gb_INVALID state or the cache line is victimised.

A unit of the PSD called a *protocol definition* is a text file describing one particular protocol (see Figure 4.9). Structurally speaking, a protocol definition comprises three sections, namely: a header; a list of protocol states; a verification definition. As highlighted in Figure 4.9, each of these sections begins with a corresponding tag, followed by its body, indicated by curly braces[13]. The complete lexical and grammar rules of the PSD language are presented in Appendix B Section B.1.2.

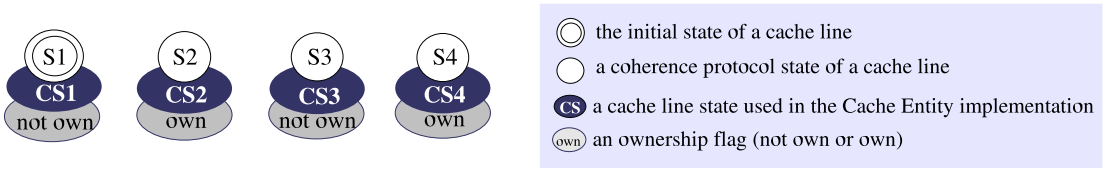


Figure 4.10: A conceptual view of the PSD header section.

The first section, *Header*, is a declaration part that introduces names into the protocol definition. It comprises four statements declaring the protocol name, state names, ownership definition, and cache-to-protocol state-mapping definition. Conceptually, the components of a protocol obtained from a Header section are presented in Figure 4.10. The figure shows that a Header Section introduces the protocol states (S1-S4). It also maps each protocol state to the corresponding cache state (CS1-CS4) and defines if the state implies ownership of a cache line. A cache state, `gb_INVALID`, (the CS1 state in the figure) identifies the initial state of all cache lines which is also used as the initial state of a protocol. The Header section of the Illinois protocol shown in Figure 4.9 (a) describes that the protocol has four states namely, Invalid, Exclusive, Shared, and Private Dirty. The ownership flag (`yes/no`) is mapped in the order defined in the protocol state list. In the Illinois protocol, the ownership technique is used. The cache lines in both Exclusive and Private Dirty states are the owners of data, *i.e.* holding the most recent values of cache lines. The owners will supply data to the other caches when a replica is requested.

The second section, *State*, describes the attributes of each coherence state using Rule, Ignored, and Priority statements. Conceptually, the State section is used to construct the state machine of a protocol, as shown in Figure 4.11. In the figure, the initial state *S1* can transit to states *S2*, *S3* or *S4* when *Rule1* or *Rule2* happens. For

[13] A double slash, //, begins a comment that extends to the end of the line

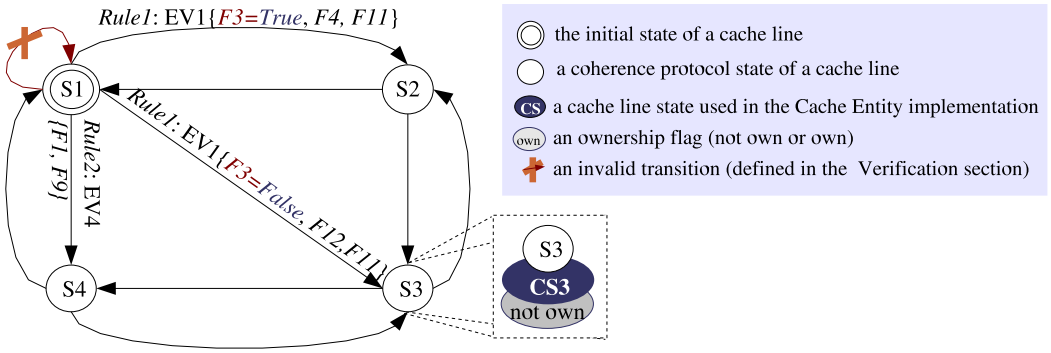


Figure 4.11: A conceptual view of the PSD State and Verification sections.

example, when an event *EV1* arrives at a cache line of state *S1*, according to *Rule1*, the transition function *F3* must be performed first. The following transition functions and the outcome state depend on whether the predicate test of the conditional function *F3* is true or false. If the predicate test is true, the protocol will perform functions *F4* and *F11* prior to transit the state of the cache line from *S1* to *S2*. Otherwise, the protocol will perform functions *F12* and *F11* before changing the cache line state from *S1* to *S3*.

Figure 4.9 (b.1-3) shows a State section describing the 'Shared' state of the Illinois protocol. This State section describes the state machine shown conceptually in Figure 4.11. A Rule statement defines a selection of transition rules based on an incoming event from a particular component. The body of a Rule indicates the names and number of transition functions a protocol has to perform before reaching an outcome state. Transition function names are reserved words that can be declared using either a conditional or unconditional form. A conditional transition function must be followed by round braces embracing a predicate declaration, the name of functions to be chosen according to the predicate test result. An unconditional function, on the other hand, can omit this part. The short descriptions of a conditional transition function (the Predicate statement and the True/False functions) and a priority declaration (the Priority function and Priority statement) are also given.

The last section (Figure 4.9 (c)), *Verification*, is used as a reference for the verification of the coherence protocol both during the parsing steps and also during a simulation run. This part comprises two statements declaring a set of invalid global

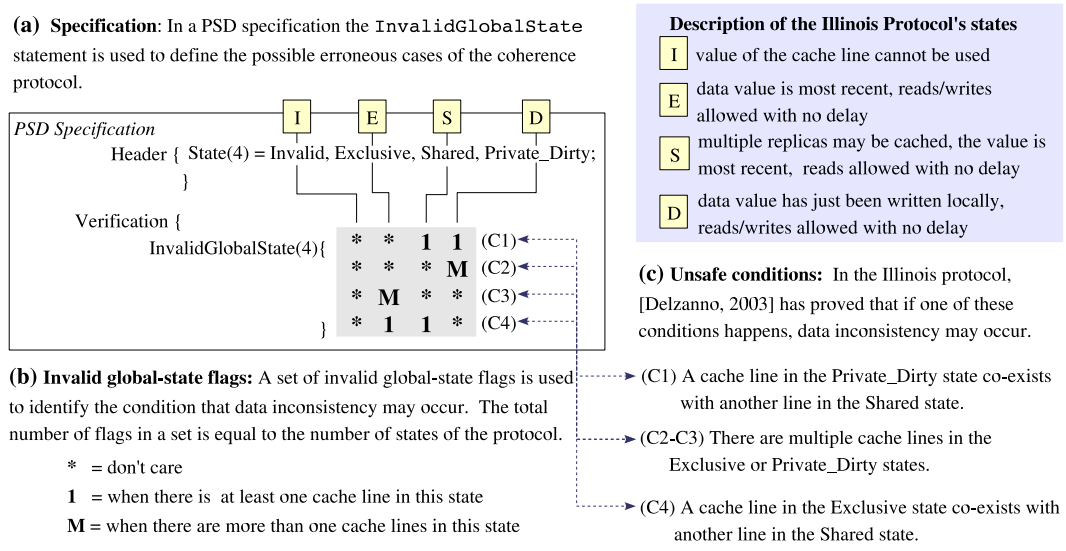


Figure 4.12: Invalid global states of the Illinois protocol.

states, and a set of invalid transitions. The invalid global state has been shown to be useful to tell the conditions where data inconsistency may occur [Delzanno, 2003]. In PSD, each condition, so-called an *unsafe condition*, is represented by a set of invalid global-state flags. The total number of flags in a set is equal to the total number of states of the protocol. Each flag position is matched to the order of the states of the State list defined in the Header section.

Figure 4.12 shows how the invalid global-state flags are defined in the Illinois protocol. Figure 4.12 (a) depicts the matching of the flag position to each protocol state defined in the Header section. Figure 4.12 (b) shows the description of the invalid global-state flag (*i.e.* '`*`' means that this state is not considered, '`1`' refers to when there is at least one cache line in this state, and '`M`' refers to when there are more than one cache lines in this state). Figure 4.12 (c) describes the four unsafe conditions of the Illinois protocol, each of which is mapped to a set of invalid global-state flags of the PSD specification. The set of invalid global-state flags is used in the PSD specification for two purposes: 1) for the checking of soundness property during a simulation run (Section 4.3.2.2) and 2) for the test of liveness property in the PSD parser (Section 4.3.3).

The last statement of the Verification section is the `InvalidTransition` statement. The invalid transition states the transitions that are not permissible in the pro-

tocol. As presented in an example protocol, the Illinois protocol performs transition functions corresponding to the accepted events. However after the transition functions are performed, the transition from the Invalid state to the Invalid state is not possible. The definition of this invalid transition is shown in Figure 4.9 (c).

4.3.1.2 The PSD Parser

The semantic values of a protocol specification are recognised by a PSD parser. The central role of the PSD parser is to produce a verified state machine from a PSD specification. Figure 4.13 (a) shows the components involved during the parsing steps. The PSD parser processes a specification in four steps as shown in Figure 4.13 (b). The first two steps comprise the lexical and the syntactic analysis of a PSD specification. If there are no syntax errors found, the result of these steps is a well-formed specification, *i.e.* the state machine of the coherence protocol. The well-formed specification must satisfy all of the PSD test conditions (PC) listed below.

PC1 Every state has been mapped to a cache line state and at least one protocol state is mapped to a `gb_INVALID` cache state.

PC2 The ownership flag must be defined.

PC3 When the ownership action is used, at least one of the protocol states is set as the owner of a cache line.

PC4 Every state must have an associated 'State' section.

PC5 A PSD specification must comprise a Header section, one or more State section, and a Verification section.

Once a protocol specification has passed the syntactic analysis step and satisfied the five test conditions listed above, the specification is considered as *well formed*. The last two steps of the PSD parsing process aim to verify the correctness of the well-formed specification. Firstly, the specification is checked to ensure the soundness or safety property. In summary, the test of soundness is to check for any unsafe conditions that can cause the inconsistent view of memory value. If the specification has successfully passed the test of soundness, the last step is to test its liveness property. The liveness test is to ensure that the specification does not cause a deadlock or

livelock. Thus, the state transition can proceed and eventually will produce a result. If there is a syntax error or if any erroneous definitions are detected, the parser will report errors and stop the simulation. In the following sections, the last two steps of the PSD parsing process are described in detail.

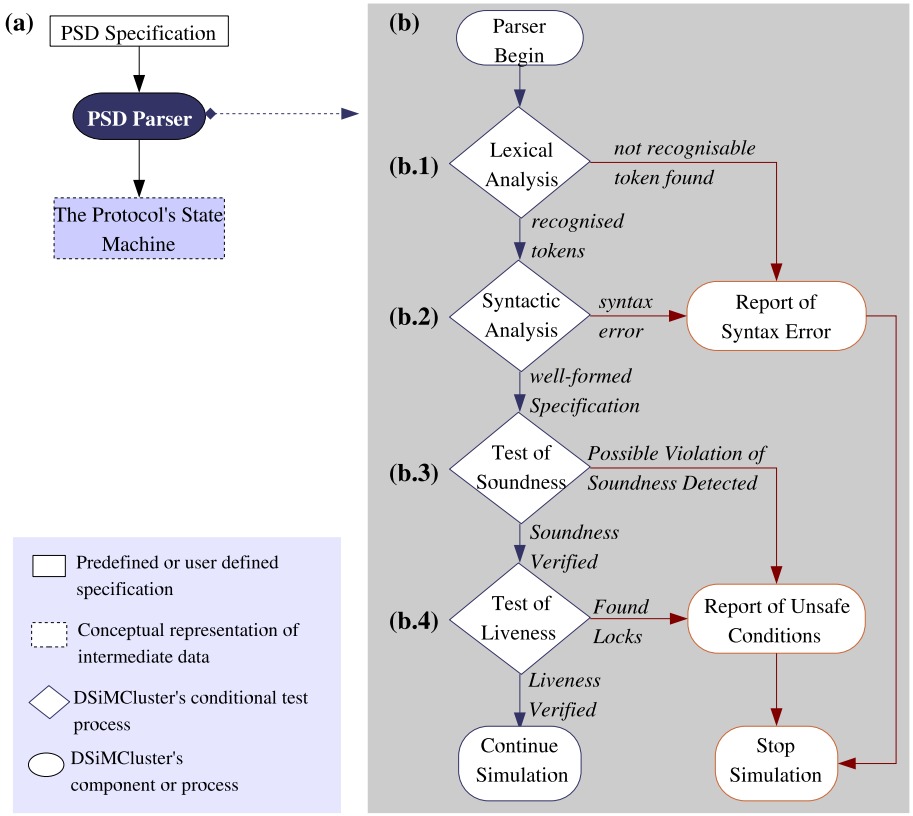


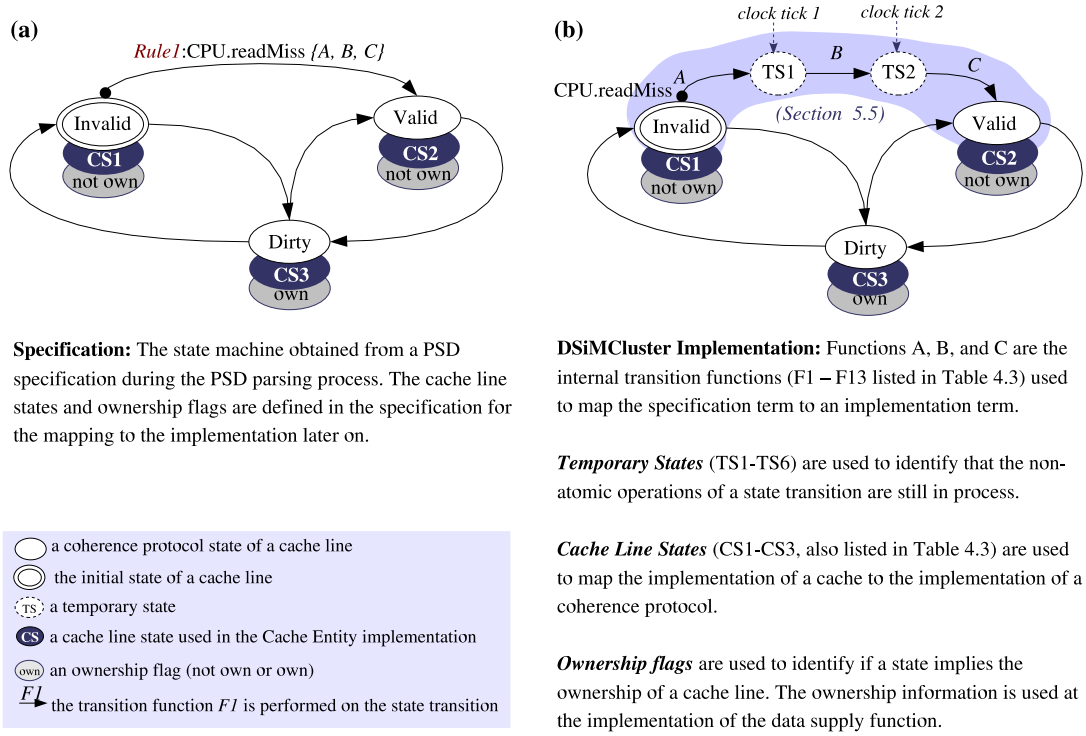
Figure 4.13: The steps of parsing a PSD specification.

4.3.2 Verification of Soundness

During the PSD parsing steps, the conditions that might lead to some errors in a simulation run is checked in two steps, the test of soundness and the test of liveness properties. The *soundness* (or *safety*) property means that the protocols always guarantee data consistency. A soundness property ensures that the implementation cannot take an action that is inconsistent with the specification [Shen, 2000]. The verification of soundness has been done in two aspects. The first aspect is to ensure that the the implementation of the simulation works in accordance with the state-transition

specification. The second aspect is to prevent the known conditions that might cause inconsistent values of data to be seen by processors.

4.3.2.1 State Machine Mapping



Specification: The state machine obtained from a PSD specification during the PSD parsing process. The cache line states and ownership flags are defined in the specification for the mapping to the implementation later on.

DSiMCluster Implementation: Functions A, B, and C are the internal transition functions (F1 – F13 listed in Table 4.3) used to map the specification term to an implementation term.

Temporary States (TS1-TS6) are used to identify that the non-atomic operations of a state transition are still in process.

Cache Line States (CS1-CS3, also listed in Table 4.3) are used to map the implementation of a cache to the implementation of a coherence protocol.

Ownership flags are used to identify if a state implies the ownership of a cache line. The ownership information is used at the implementation of the data supply function.

Figure 4.14: Mapping of specification term to implementation term.

The state-machine mapping technique has been used to ensure that the implementation of the simulation works in accordance with the state-transition specification. As described in the previous section, the state machine obtained from a PSD specification includes the set of states and rules which describe the transition functions to be performed during a state transition. Figure 4.14 (a) shows an example of a state machine obtained from a three-state coherence protocol. When a `CPU.readMiss` event arrives at an Invalid cache line, according to *Rule 1* of the Invalid state, the protocol must perform functions *A*, *B*, and *C* to maintain data coherence before transit to the Valid state.

To ensure that the DSiMCluster simulation will works in accordance with this *Rule*, functions *A*, *B*, and *C* are used to map the specification term into the imple-

mentation term. Note that these functions are the internal transition functions defined as the PSD reserved words shown in Table 4.3. As shown in Figure 4.14 (b), after the DSIMCLUSTER simulation executes function *A*, the coherence protocol stays in a temporary state *TS1*. A temporary state is used to provide channels through which a transition operation can be carried out using multiple steps. The protocol stays in a temporary state until all of the partially executed operations have been finished. The implementation detail of a coherence-protocol state transition is described in Section 5.5 (page 169).

In a PSD specification, three PSD test conditions are checked to ensure the correctness of the state machine. The first condition (PC6) is to ensure that each protocol state has been defined to respond to all possible events received (EV1-EV8 in Table 4.3). The second condition (PC7) is to ensure that the requested data will be provided for every read access. Moreover, the third condition (PC8) is to check that the updated data will be written to the cache line for every write access.

PC6 At each State section, all eight protocol events must be defined in either the Rule or the Ignore statements.

PC7 The Rules of both the `CPU.readHit` and the `CPU.readMiss` events must have the `toCPU.supplyData` function defined.

PC8 The Rules of both `CPU.writeHit` and `CPU.writeMiss` events must have the `toBus.broadcast` and `writeData` functions defined.

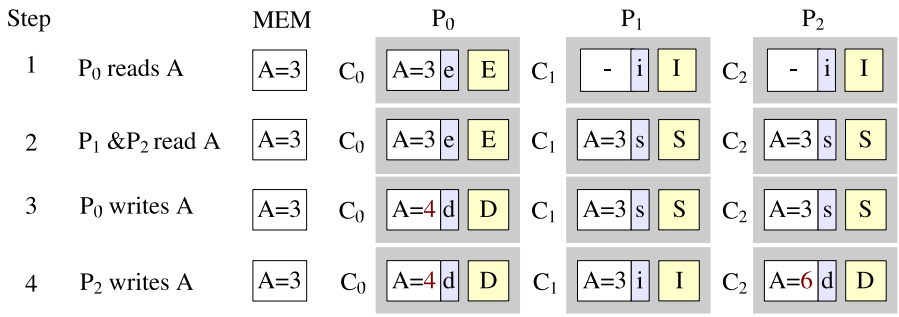
4.3.2.2 Prevention of Unsafe Conditions

The second aspect of the verification of soundness is to prevent known conditions that might cause inconsistent values of data to be seen by processors. Two possible unsafe conditions described in [Pong and Dubois, 2000] and [Delzanno, 2003] have been checked in the PSD parser. Firstly, the protocol should not perform a state transition when it receives any unexpected events. Secondly, the global state of a coherence protocol must be permissible.

Prevention of unexpected events. Eight possible events to be received by a coherence protocol are defined as reserved words in PSD. A PSD test condition, PC6, is

checked to ensure that each State section recognises all eight events either through a Rule or an Ignored statement. Following this, another PSD test condition is checked (PC9) in order to ensure that the events defined in the Rule statement (*i.e.* events causing a state transition) are different from the events defined in the Ignore statement (*i.e.* events causing no transition).

PC9 Corresponding to PC6, for each State section, each event must be defined only once either by a Rule or an Ignore statement.



An example scenario when a simulation implements the Illinois protocol wrongly.
(a) If in step 2 the cache line state at Processor P_0 has not been updated, in step 3 the processor P_0 can write to the cache line with no delay (as it holds an Exclusive cache line). Thus, the inconsistent values of A will be seen by P0, P1, and P2.
(b) In step 4, if P2 can write to a Shared cache line without any updates for the most recent value first, the inconsistency of data are seen from both P0 and P2 as they hold the Private_Dirty cache lines with different value.
If these errors are not detected by the simulation, the simulation results cannot represent the correct characteristics of memory accesses.

Figure 4.15: An example of a simulation error.

Declaration of invalid global states. When there are multiple replicas of data, processors may see different data values. Therefore, a coherence protocol places a coherence state for each replica to tell the processors whether their data is the most up-to-date one. However if a simulation has implemented a protocol wrongly, or if the mapping functions of the PSD specification has been defined wrongly, the results obtained from such a simulation cannot be used to represent the real characteristics of memory accesses. Figure 4.15 shows an example of such a case. Global states that are not permissible by the protocol definition are normally classified as *erroneous*

states [Pong and Dubois, 2000]. There are four erroneous states in the Illinois protocols [Delzanno, 2003] as listed in Figure 4.12. The definition of Illinois states and the invalid global states used in Figure 4.15 and 4.16 are referred to the description given in Figure 4.12.

To ensure that the DSIMCLUSTER will detect the situations where data inconsistency may be seen by processors, the set of invalid global states is used. As mentioned at the end of Section 4.3.1.1, a purpose of defining the unsafe conditions (using the set of invalid global-state flags in the PSD) is for the checking of soundness property during a simulation run. Figure 4.16 shows the process of soundness checking during a simulation run in the DSIMCLUSTER. The checking is performed in three steps. Firstly, the current global state of each cache line is recorded during a memory access (Figure 4.16 (a)). Secondly, when the memory access has been completed, a set of test flags is produced using the recorded global state (Figure 4.16 (b)). Finally, before the simulation can continue, the set of produced test flags must not match with the invalid global state (*i.e.* it is not invalid). If the test flags match with the invalid global state, the simulation is stopped as the unsafe condition has been detected. Figure 4.16 (c) illustrates this testing step. Note that if the errors shown in this figure happen, the DSIMCLUSTER will stop the simulation since the first unsafe condition is detected (after step 2).

Three flags are used when recording the global state of a cache line: `0`, `1` and `M`. The flag '`0`' means that there are no replicas in this state. Flags '`1`' and '`M`' show that there is only one replica or there are multiple replicas of the cache line in this state, respectively. The order of the states defined in the protocol state list (in the PSD Header section) is mapped to the position of a flag in a global state. At step 1 in Figure 4.16 (a), the global state of variable *A* is `0100`. The recorded global state means that there are no replicas of *A* cached in the Invalid, Shared and Private Dirty states, and there is one replica of *A* cached in the Exclusive state.

In the PSD parser, three PSD test conditions are included to check the definition of the invalid global states. The first condition, PC10, is to ensure that at least one invalid global state has been defined in a PSD specification. The second condition, PC11, is to test that when a local cache line enters a state that allows only one replica, all remote cache lines must exit the state. The third condition, PC12, is to ensure that when two states must not co-exist, then if a local cache line enters one of these states,

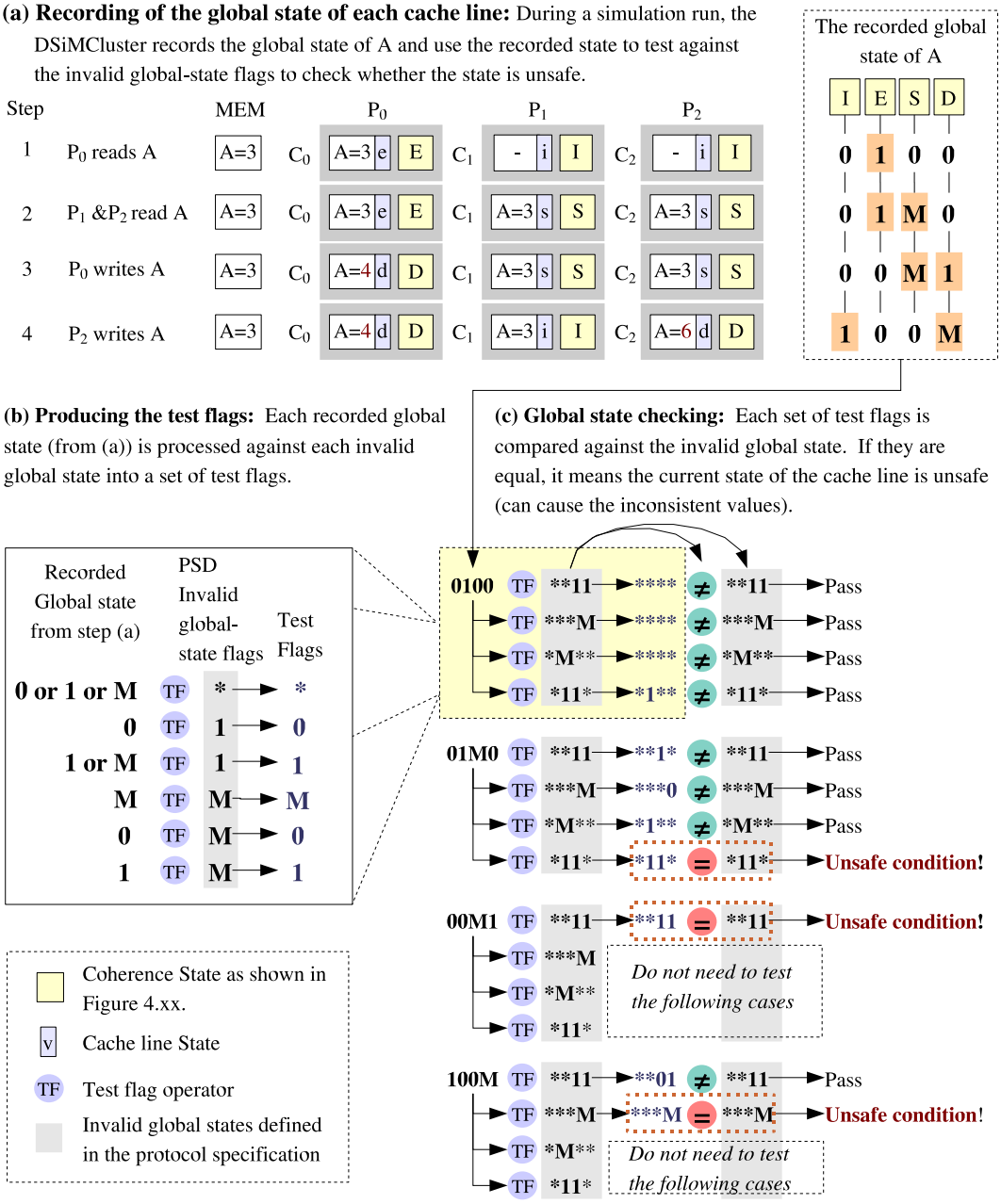


Figure 4.16: Process of unsafe condition checking during a simulation run.

remote cache lines must not enter the prohibited state.

PC10 At least one invalid global state must be defined in a PSD specification.

PC11 When a state is defined to have only one replica, all `CPU` events which cause the transition to the state must have the corresponding `BUS` events that exit the state.

PC12 When two states must not co-exist, all `CPU` events which enter one of the two states must not have corresponding `BUS` events which enter the other state.

If a well-formed specification satisfies the seven PSD test conditions (PC6-PC12), the specification has passed the test of soundness. In the last step of the PSD parser, the specification is then verified for its liveness property.

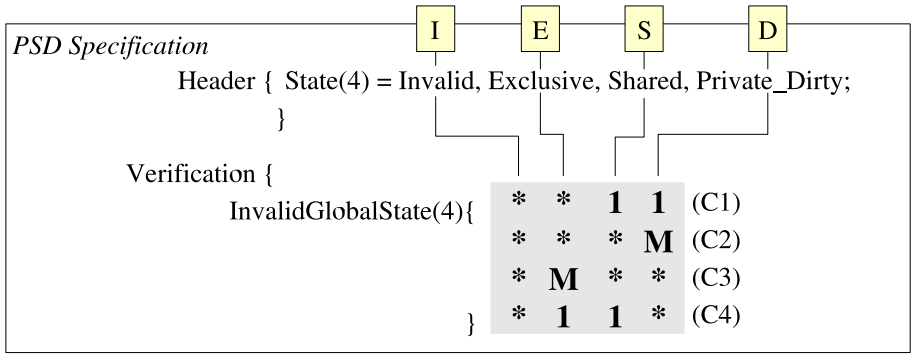
4.3.3 Verification of Liveness

The *liveness* property assures that there is no deadlock or livelock during the protocol's state transitions. In this case, *deadlock* is a situation in which two or more caches are indefinitely blocked while each of them waits for resources or acknowledgements to be released from another. Similarly, *livelock* is a situation in which one or more caches is prevented from proceeding further, yet stays indefinitely in a state with no exit.

The previous section shows the usefulness of the invalid global-state definitions for the test of soundness property. As mentioned at the end of Section 4.3.1.1, another purpose of defining the invalid global states is to obtain all co-existing state pairs that are valid, so that the test of liveness can be done on these pairs. Figure 4.17 shows the list of eight valid co-existence state pairs of the Illinois protocol. In the figure, a pair of *State1* and *State2* identifies that if a local cache line is in *State1*, another remote cache can have its replica in *State2*. The PSD parser uses each of these pairs to check for the liveness properties in both the deadlock and livelock testing steps.

4.3.3.1 Deadlock Prevention

A deadlock may occur if during a transition function (involving partially executed operations), at least two caches are waiting for resources or acknowledgements to



Valid co-existence state pairs: Each pair, *(State1, State2),* shows that if the local cache line is in *State1*, then a remote cache can have the replica of this line in *State2*. The valid co-existence state pairs of the Illinois protocol are listed below.

(Invalid, Invalid), (Invalid, Exclusive), (Invalid, Shared), (Invalid, Private_Dirty)

(Exclusive, Invalid)

(Shared, Invalid), (Shared, Shared)

(Private_Dirty, Invalid)

Figure 4.17: The valid co-existence state pairs of the Illinois protocol.

be released from one of the others. Once the specification has passed the test of soundness, it is guaranteed that these caches must stay in the valid co-existence states. Therefore, to prevent a deadlock, all possible co-existence states are checked against two PSD test conditions. Firstly, for all valid co-existence state pairs, the waits and supplies of resources must be matched (PC13). Secondly, for all valid co-existence state pairs, every broadcast operation must have the package acknowledgement sent from the cache in the co-existence state (PC14).

PC13 For all co-existence states, every `waitForUpdateData` function has a matched function, `toBUS.supplyData`.

PC14 For all co-existence states, every `toBUS.broadcast` and `waitForAcknowledgement` functions has a matched `sendAcknowledgement`.

4.3.3.2 Livelock Prevention

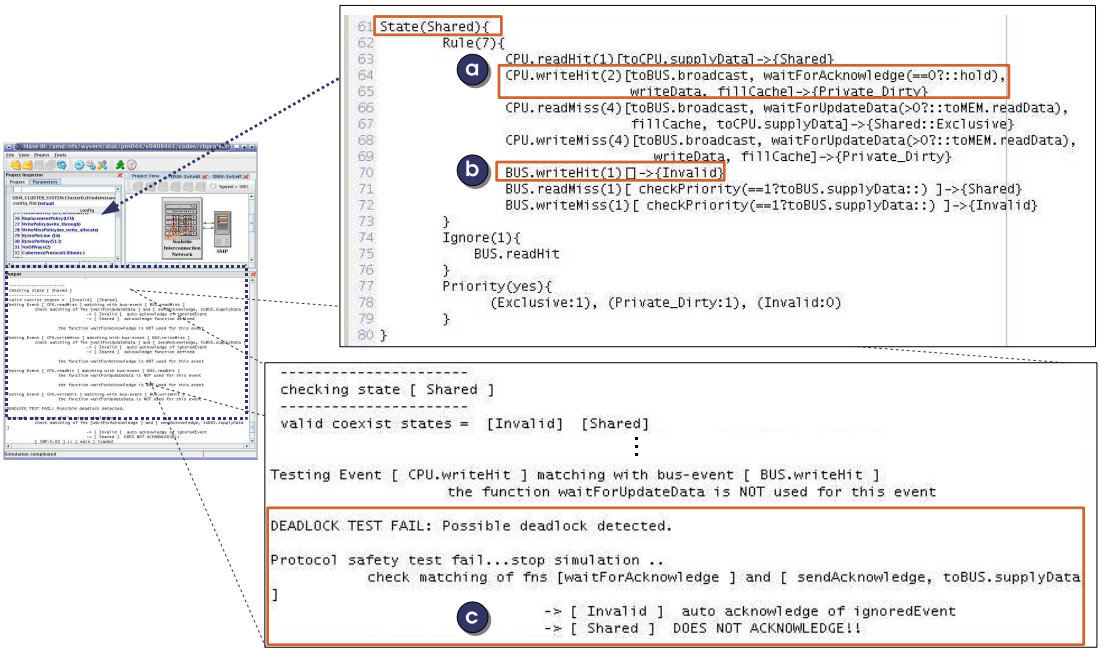
A livelock occurs when one or more cache stays indefinitely in a state with no exit after performing a transition function in response to a valid event. To prevent a live-

lock, two PSD test conditions are checked. Firstly, at least one event of the local cache accesses will cause a transition which exits the current state (PC15). This condition is to guarantee that there is no trapped state in a protocol specification. Secondly, to prevent a livelock, all transitions must be valid (PC16).

PC15 For each State section, there is a Rule defining one of the four local-access events (`CPU.readHit`, `CPU.readMiss`, `CPU.writeHit` or `CPU.writeMiss`) in which the outcome state must exit to another state.

PC16 For each Rule defined in a State section, the state transition must not violate the `invalidStateTransition` defined in the Verification section.

4.3.3.3 Examples of the livelock test



An example of a deadlock test fail by inserting a specification of the Illinois protocol with a contrived error into the DSiMCluster.

(a) In the Shared state definition of the Illinois protocol, the Rule of `CPU.writeHit` event defines that if there is a write hit at the local cache, the access must be broadcast and then the cache will be waiting for the acknowledgments from all of the other caches.

(b) A contrived error has been put in the Rule of the `BUS.writeHit` event. This contrived error cause a Shared cache line to skip the acknowledgment of a Bus message, yet directly change its state to be Invalid.

(c) The error report shows that if multiple cache lines are 'Shared' and one has got a `CPU.writeHit`, the others (receiving the `BUS.writeHit` event) fail to acknowledge this event, causing possible deadlock.

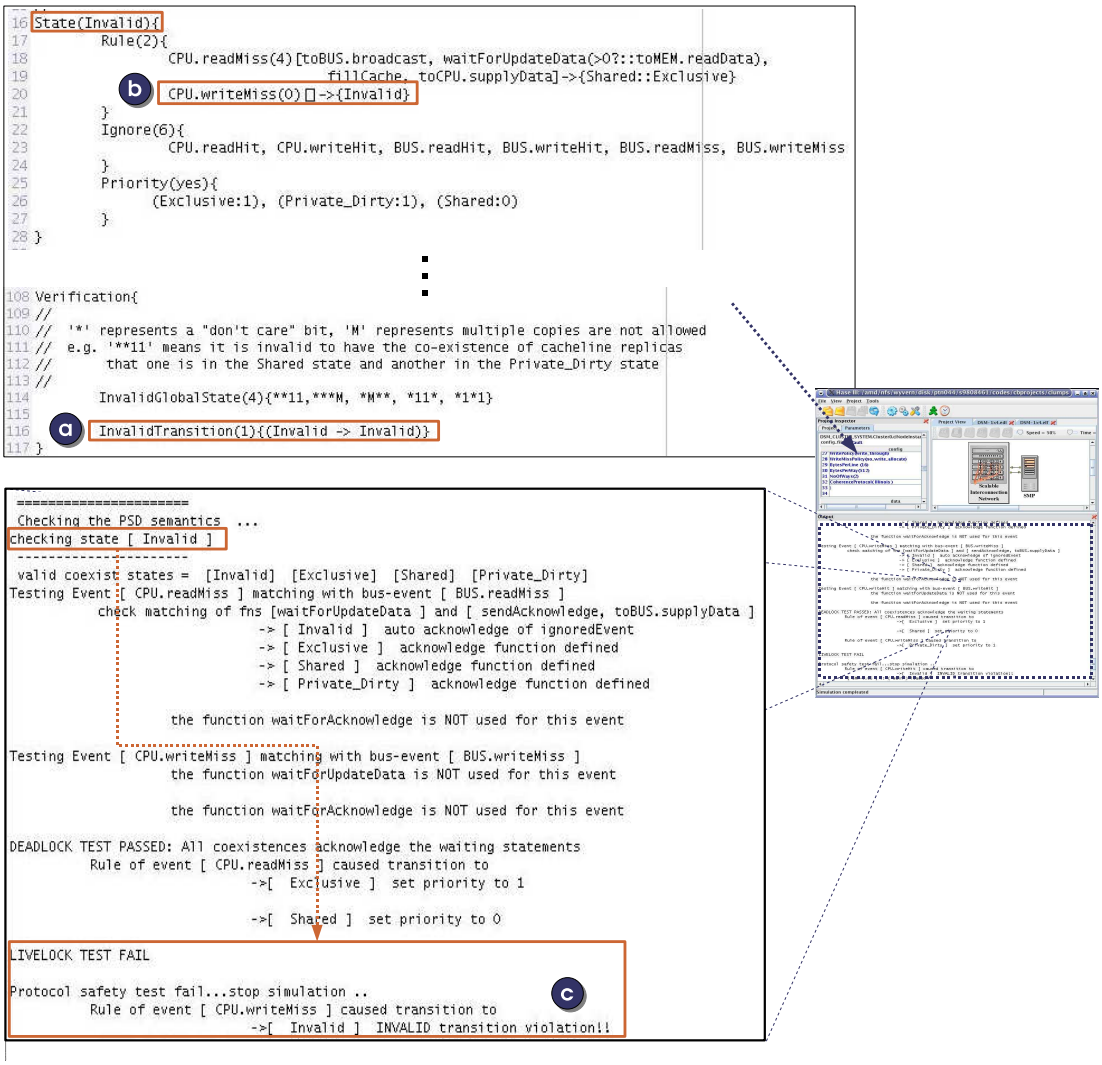
Figure 4.18: An example of a contrived error which causes a deadlock test fail.

Two specifications of the Illinois protocol with contrived errors have been used to show that some liveness violations can be detected by the PSD parser. In the first erroneous specification, the Shared state of the Illinois protocol was modified. As shown in Figure 4.18 (a), the Rule of the `CPU.writeHit` event identifies that on a local-cache write hit at the cache line in the Shared state, the cache first broadcasts this access and then waits for the other caches to acknowledge. The contrived error has been put at the Rule of the `BUS.writeHit` event (Figure 4.18 (b)). This erroneous Rule identifies that when a cache line in the Shared state has received a write-hit event from bus (the write request from a remote cache), the cache omits to acknowledge the package, yet directly changes the cache line state to Invalid. In this case, the requesting cache will stay indefinitely waiting for the acknowledgements. Figure 4.18 (c) shows that after entering this erroneous specification to the DSIMCLUSTER, the PSD parser can detect the possible deadlock in the Shared state specification.

The second erroneous specification demonstrates the liveness test based on the invalid transition definition. Firstly, in the Illinois protocol the transition from an Invalid state to Invalid state has been defined not permissible Figure 4.19 (a). A contrived error has been put at the Invalid state definition Figure 4.19 (b). This erroneous Rule identifies that when a cache line in the Invalid state has received a write-miss event from local cache, the cache performs no transition functions, yet changes its state back to Invalid. In this case, when a write miss happens to a cache line in the Invalid state, the cache will be trapped in the Invalid state. Using this specification, the protocol will pass the PC15 test condition but fail to pass the PC16 condition. As shown in Figure 4.19 (c) that the livelock is detected because the definition of the erroneous Rule violates the `invalidStateTransition` of the Verification section.

4.3.4 Summary of the PSD Parser

The previous sections have shown the vaule of using the PSD specification. It is useful in two aspects: a) it can be used to verify both the soundness and liveness properties of a protocol and b) the verified state machine and its transition functions can be mapped to the implementation of a simulation. To state the unsafe conditions as the invalid global states in a PSD specification allows the correction of the protocol states to be checked both prior to the beginning of a simulation and during a simulation run.



An example of a deadlock test fail by inserting a specification of the Illinois protocol with a contrived error into the DSiMCluster.

(a) Verification section defining the transition that is not valid if some transition functions have been taken.

(b) An erroneous specification has been inserted for the `CPU.writeMiss` event, at the definition of the Invalid state.

(c) The error report shows that when a cache line is Invalid, the definition of event `CPU.writeMiss` cause the transition to Invalid state, *i.e.* this transition violates the Verification definition (the transition from Invalid to Invalid state is not permissible).

Figure 4.19: An example of a contrived error which causes a livelock test fail.

In summary, to obtain a verified specification prior to running a simulation, the PSD parser checks for well-formedness during the lexical and syntactic analysis steps, and checks the semantics during the soundness and liveness tests. A verified specification must pass sixteen of the PSD test conditions (PC) summarised in Table 4.4.

4.3.5 Specification-based Verification of Eight Protocols

In this work, eight bus-based coherence protocols have been defined using the PSD specifications including, the Write Invalidate, Synapse, Illinois, Berkeley, DEC Firefly, Dragon, MESI, and MOESI protocols. The PSD specifications of these protocols have been parsed and verified according to the PSD parsing steps presented in the previous section. To ensure the correctness of the protocols, the unsafe conditions presented in [Delzanno, 2003] have been adopted and included into the PSD specifications. In this section, the conditions used to verify three protocols (Synapse, Illinois and MOESI) are described in detail. The information of the other five protocols are presented in Appendix B Section B.3.

At the beginning of a simulation, all cache states are set to invalid (using the gb_INVALID cache state). Consequently, the state of a coherence protocol which is mapped to the gb_INVALID state is used as an initial state of the coherence protocol state machine.

4.3.5.1 Synapse

Synapse is a write-allocation protocol with three states, namely, Invalid, Valid and Dirty. When there is a read miss, the cache line state is changed to Valid. In the following read hit, there are no coherency actions required. On a write hit at local cache, the action is broadcast and the cache is halted waiting for the other caches to acknowledge the write (and to change their states to Invalid). Once the requested cache has received all acknowledgements, the cache line is written and marked as Dirty.

The state machine shown in Figure 4.20 is obtained from parsing the PSD specification of the Synapse protocol. Each state in the state diagram refers to the state of a physical cache line (or cache entry) implemented in the Cache Entity. All cache lines are initially marked as Invalid. A read miss causes the data to be transferred from

Table 4.4: Summary of the test conditions carried out by the PSD parser.

Syntax Test

Condition	*Test Description*
PC1	Every state has been mapped to a cache line state and at least one protocol state is mapped to a `gb_INVALID` cache state.
PC2	The ownership flag must be defined.
PC3	When the ownership action is used, at least one of the protocol states is set as the owner of a cache line.
PC4	Every state must have an associated 'State' section.
PC5	A PSD specification must compose of a) a Header section, b) one or more State section, and c) a Verification section.

Test of Soundness

Condition	*Test Description*
PC6	At each State section, all eight protocol events must be defined in either the Rule or the Ignore statements.
PC7	The Rules of both the `CPU.readHit` and the `CPU.readMiss` events must have the `toCPU.supplyData` function defined.
PC8	The Rules of both `CPU.writeHit` and `CPU.writeMiss` events must have the `toBus.broadcast` and `writeData` functions defined.
PC9	Corresponding to PC6, for each State section, each event must be defined only once either by a Rule or an Ignore statement.
PC10	At least one invalid global state must be defined in a PSD specification.
PC11	When a state is defined to have only one replica, all `CPU` events which cause the transition to the state must have the corresponding `BUS` events that exit the state.
PC12	When two states must not co-exist, all `CPU` events which enter one of the two states must not have corresponding `BUS` events which enter the other state.

Test of Liveness

Condition	*Test Description*
PC13	For all valid co-existence states, every `waitForUpdateData` function has a matched function, `toBUS.supplyData`.
PC14	For all valid co-existence states, every `toBUS.broadcast` function and `waitForAcknowledgement` function has a matched function, `sendAcknowledgement`.
PC15	For each State section, there is a Rule defining one of the four events (`CPU.readHit`,`CPU.readMiss`, `CPU.writeHit` or `CPU.writeMiss`) in which the outcome state must exit to another state.
PC16	For each Rule defined in a State section, the state transition must not violate the `invalidStateTransition` defined in the Verification section.

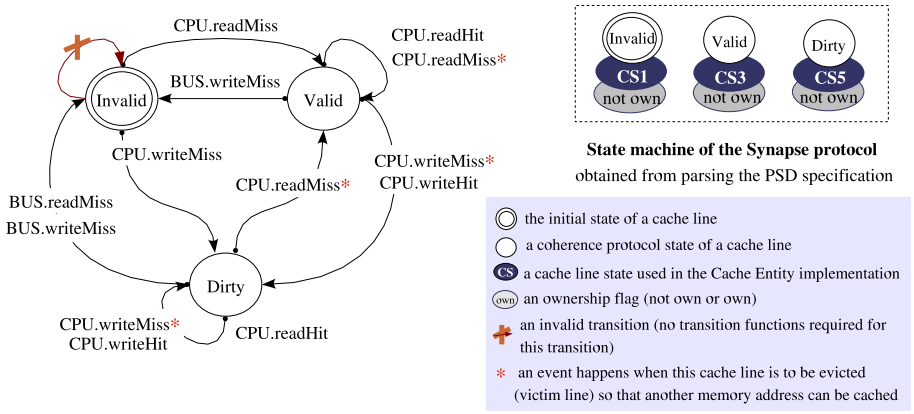


Figure 4.20: State machine of the Synapse protocol.

main memory to this cache line and its state is then marked as Valid. This activity is also used when there is a subsequent read miss to a Valid cache line by conflicting memory line addresses. Thus, a read miss to a Valid cache line causes the transition to the Valid state.

When a CPU writes to a cache line (causing the `CPU.writeHit` or `CPU.writeMiss` events), the line is marked as Dirty. The write event is broadcast to bus (causing the `BUS.writeHit` or `BUS.writeMiss` events in respect of the CPU event) and causes the other caches to mark their lines Invalid. This is repeated in any subsequent write to the Dirty line. When a remote cache tries to read a Dirty cache line (causing the `BUS.readMiss` event), data of this line is updated to main memory. This data is transferred from memory to the requested cache. Both caches holding the data mark the cache line to Valid. A read miss to a Dirty cache line by conflicting memory line addresses causes the previous data to be flushed to (and the new data is transferred from) the main memory. The line is marked as Valid after the coherence actions.

From the definition of the Synapse protocol, the possible sources of data inconsistency and the corresponding invalid-global states defined in the PSD specification are as follows.

Unsafe Condition	PSD Global State (Invalid, Valid, Dirty)
A Dirty cache co-exists with caches in state Valid.	*11
There are more than one data replicas in Dirty state.	**M

4.3.5.2 Illinois protocol

The Illinois protocol is a write-invalid protocol with four states namely, Invalid, Exclusive[14], Private Dirty[15] and Shared[16]. The ownership technique is used in the Illinois protocol. A cache line in either the Exclusive or Private Dirty state owns the data. Cache lines in the Shared state do not have a specific owner. In an ownership-based protocol, the owner forwards its data to a requester. This reduces the time to access data in comparison to transferring it from the main memory.

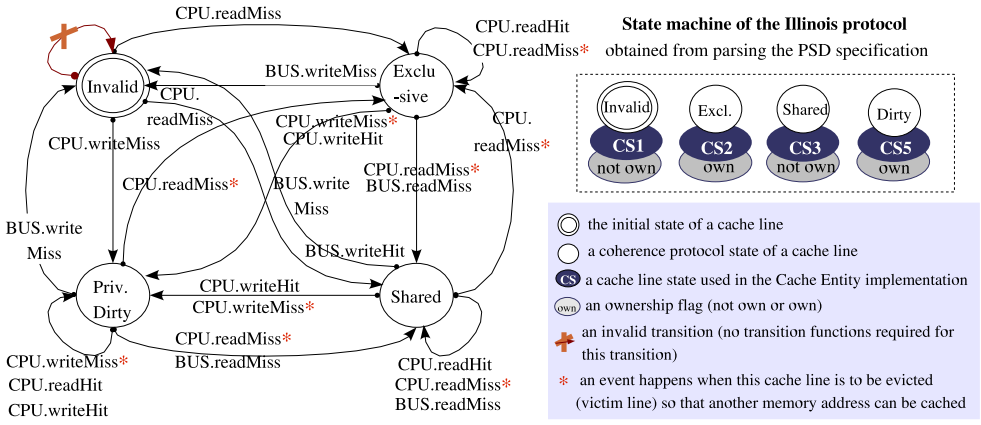


Figure 4.21: State machine of the Illinois protocol.

The state machine obtained from parsing the PSD specification of the Illinois protocol is shown in Figure 4.21. Similar to the Synapse protocol, all cache lines are initially marked as Invalid. When an access miss occurs to an Invalid line (either the `CPU.readMiss` or `CPU.writeMiss` event), the access is broadcast (causing the `BUS.readMiss` or `BUS.writeMiss` event). Data can be supplied either from a remote cache or the main memory. If the data is from a remote cache (*i.e.* there is an owner who receives the matched `BUS` event), the read access causes the cache line to be marked as Shared, while the write causes the line to be marked as Private Dirty. In either case, the previous owner changes its state to Shared or Invalid, respectively. If the previous owner was in the Private Dirty state, the data is also written back to memory at the same time as the data is supplied to the requester. If data is supplied

[14] also called *Valid Exclusive* or *Read Private*
[15] also called *Modifi ed*
[16] also called *Read Shared*

from the main memory, the requested cache line becomes the owner, while the state is marked as Exclusive for the read and as Private Dirty for the write.

For a write hit to a cache line in one of the owner states (either in the Exclusive or Private Dirty state), the update proceeds without delay. However, if the line is shared (in Shared state), the update must be performed after the other caches have marked Invalidate on their lines (*i.e.* waiting for the positive acknowledgements). The state of the updated line is Private Dirty and becomes the owner.

From the definition of the Illinois protocol, the possible sources of data inconsistency and the corresponding invalid-global states defined in the PSD specification are as follows.

Unsafe Condition	PSD Global State (I, E, S, D)
A Private Dirty cache line co-exists with a Shared line.	**11
There are multiple cache lines in the Exclusive states.	*M**
There are multiple cache lines in the Private Dirty states.	***M
An Exclusive cache line co-exists with a Shared line.	*11*

4.3.5.3 MOESI

MOESI is a five-state write-allocation protocol with ownership-based technique. The protocol states are Modified, Owned, Exclusive, Shared, Invalid states. The acronym MOESI denotes the name of its states. Similar to the Illinois protocol, the ownership technique is used in MOESI. A cache line in either the Exclusive, Modify or Owned state owns the data. Data kept at the owner may not be consistent with data in the main memory. The state *Shared* identifies cache lines which contain a copy of data owned by other caches. In contrast to the Illinois protocol, in MOESI, the Shared cache lines can have the copies of data that have not yet been updated in the memory. The state Exclusive has a different meaning from the Illinois protocol. In MOESI, the *Exclusive* state identifies a cache line which has been written to once but has not yet been accessed by other caches (only one replica exists). A subsequent local write to an Exclusive line causes the line to be marked as Modified. The *Modified* state means that the line has been written to more than one time and has not yet been accessed by other caches. A subsequent remote read to an Exclusive line causes the line to be marked as Owned, and its remote copy to be marked as Shared. The *Owned* state

means that the cache line has been written to more than once and its content is shared with other caches.

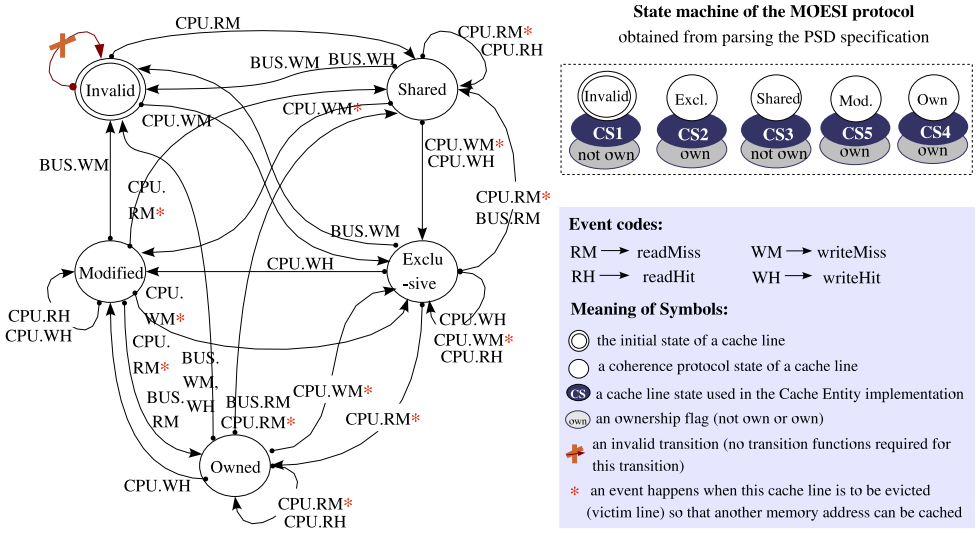


Figure 4.22: State machine of the MOESI protocol.

The state machine obtained from parsing the PSD specification of the MOESI protocol is shown in Figure 4.22. Similar to the Synapse and Illinois protocol, all cache lines are initially marked as Invalid. When a read miss occurs to an Invalid line (`CPU.readMiss`), the access is broadcast (causing the `BUS.readMiss` event) and the outcome state is Shared. If the data is supplied from other caches in the Modified state, the owner changes its state to Owned to recognised that there are multiple replicas of data. However, if the data is supplied from other caches in the Exclusive state, the owner changes its state to Shared. When a write miss occurs to a line of any state (`CPU.writeMiss`), the access is broadcast (causing the `BUS.writeMiss` event) and the outcome state is Exclusive. The other caches marked their copies as Invalid.

When a write hit occurs to an Exclusive or a Modified cache line, the line is written to with no delay and its state is changed to Modified. This action requires no broadcast message since both the Exclusive and Modified state identify that only one replica exists. When a write hit occurs to a Shared or Owned cache line, the line is written with no delay and the outcome state is Exclusive. This access is broadcast (causing the `BUS.writeHit` event to either the other Shared, or Owned cache lines). The other caches invalidate their copies. When a cache line is changed to the Exclusive state,

the content in the main memory is also updated to the current value.

From the definition of the MOESI protocol, the possible sources of data inconsistency and the corresponding invalid-global states defined in the PSD specification are as follows.

Unsafe Condition	PSD Global State (M,O,E,S,I)
A Modified cache line co-exists with a Shared line	***11
A Modified cache line co-exists with an Owned line	**1*1
A Modified cache line co-exists with an Exclusive line	*1**1
A Owned cache line co-exists with an Exclusive line	*11**
A Exclusive line co-exists with a Shared line	*1*1*
There are more than one Modified replicas	*M***
There are more than one Exclusive replica	****M

4.3.5.4 Verification Results

The specifications of eight coherence protocols have been parsed and tested for their soundness and liveness properties. These specifications are well formed, and have passed all of the sixteen PSD test conditions listed in Table 4.4. The characteristics of the protocols, the time to parse, and the obtained results are summarised in the Table 4.5.

Table 4.5 summarises the obtained results corresponding to the assessment used in the verification process for the safety and liveness properties. The protocol specifications have been examined to *prevent* the possible erroneous cases as described earlier. The time to parse the protocol specifications (column 3) shows that a very small fraction of time is spent on verifying the model specifications for error prevention. The characteristics obtained include the number of invalid global states (INV Gbl States), the critical states, and the matching of waiting and acknowledging pairs. The critical states are those that may cause a *deadlock* during the simulation as they perform more transition steps in comparison to the other states, and the transition steps include at least one function that has the waiting-for-acknowledge action. These characteristics are used to locate the monitoring states during the simulation run to ensure the absence of deadlock.

The number of possible transitions shows the complication of the state machine

Table 4.5: Results of PSD specification checking of eight coherence protocols.

Protocol	Characteristic		Safety		Liveness	
	Category	Parsing Time (sec:msec)	No. of INV Gbl State	Critical State	Possible Transtns	Wait-Ack Matching Pairs
Write-Inv.	3-State, Inv.	1:410s	2	Dirty	14	12
Synapse	3-State, Inv.	1:474s	2	Dirty	14	14
Illinois	4-State, Inv.	1:341s	4	Dirty,VldExcl	25	18
Firefly	4-State, Upd.	1:540s	4	VldExcl	30	23
Berkeley	3-State, Inv.	1:480s	3	Own.Excl.	21	28
Dragon	5-State, Upd.	1:653s	7	VldExcl,ShDrt	41	32
MESI	4-State, Inv.	1:552s	2	Modify,Excl.	20	11
MOESI	5-State, Upd.	1:590s	7	Modify,Excl.	40	29

(column 6). As seen in the table and the summary of the MOESI protocol described earlier, the PSD specification allows the state machine and the transition functions of a complicated protocol like MOESI to be defined, tested and used directly to the simulation. Section 5.7 in the next Chapter shows the results from further experiments on model validation against measurement. The verified specification of the MOESI protocol is used to model a SunFire 15K machine in order to compare the results against the results from a real machine. Once a valid result is obtained, the results of the other protocols were verified against the result of the MOESI, so that all specifications are verified on their soundness property.

4.4 Summary

In this chapter, the model specification of DSIMCLUSTER has been formulated as a basis to provide *model extensibility* and *verification applicability* in the simulation of DSM multiprocessors. Based on the analysis of related work presented in Chapter 3, model extensibility has been achieved by distinguishing the model formulation from the details of the DES engine. In doing so, HASE has been chosen as the simulation construction framework. In this chapter, the functionality and features of the DES engine included in HASE have been described, together with the DEVS formalism used to define the model specification outlined.

The DSIMCLUSTER specification is divided into two groups of components,

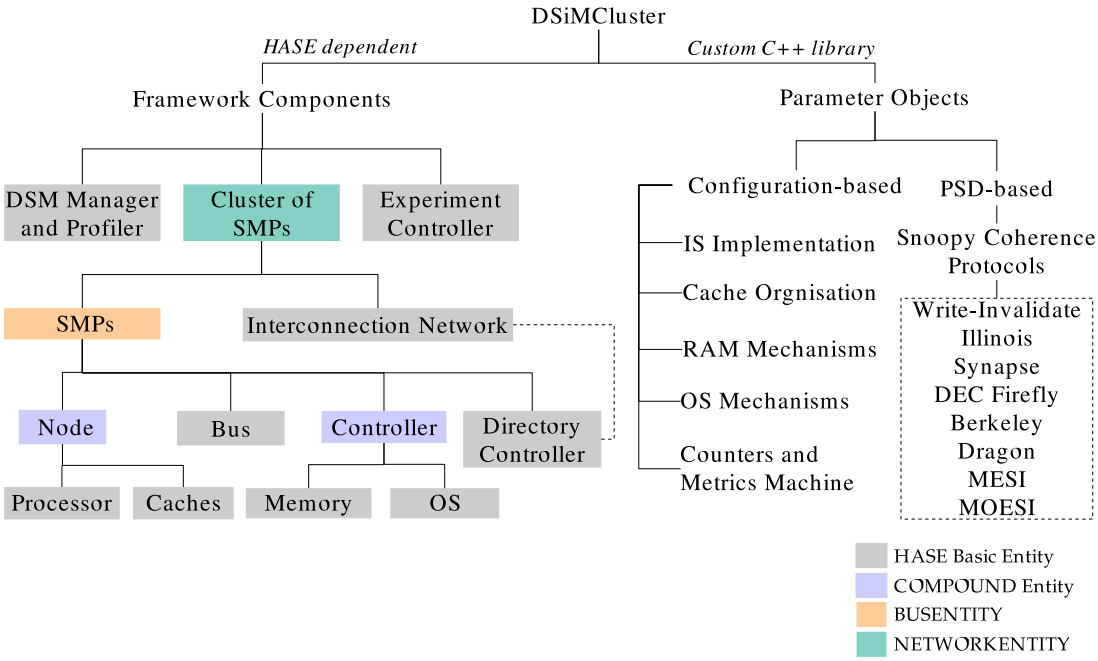


Figure 4.23: Chart summarising the structure of the DSIMCLUSTER model.

namely the *framework components* and the *parameter objects* summarised in the chart in Figure 4.23. The first group, framework components, refers to the specifications of HASE entities constructing the generic architecture of a DSM multiprocessor. The specification includes nine basic entities, each of which has been defined using the DEVS formalism[17]. These basic entities have been grouped according to their functional relationship using HASE compound entity and built-in design templates. Formulating the structure of the model in this way allows each entity of the model to be defined in a highly modular manner, yet promotes model extensibility. The hierarchical structure of the framework components also provides guidance on entities of interest for the analysis of each level of detail.

The second group of components in the DSIMCLUSTER specification, parameter objects, refers to classes that represent (a) the operational elements included in each entity, and (b) the configuration of tunable parameters in a DSM system. The combination of selected parameter objects in collaborating with the framework components simulates the unique characteristics of each DSM cluster implementation. Two groups of parameter objects have been defined, a *configuration-based* and a

[17]The specifications of five of these entities are presented in Appendix A

PSD-based parameter object. Configuration-based parameter objects represent the varieties of a model's parameters that are inserted into the DSIMCLUSTER model using configuration files. The PSD (Protocol Specification Description) based objects represent different bus-based cache coherence protocols. This latter group has been designed to apply automatic verification into the DSIMCLUSTER model through a well-formed specification. The specification of each of these objects has been defined using the DEVS *passive* states. These objects stay indefinitely in a state until they receive an external request from the attached entity (*i.e.* the framework component to which they are attached). Once a request has arrived, the objects perform the corresponding state transition functions and change their state. Consequently, behaviour emulation occurring during the state-transition of these objects shares the timing information with the attached entity.

At the end of this chapter, preliminary test case results have been presented. These tests have demonstrated how the possible errors in protocol specifications that may impact the *soundness* and *liveness* properties of the coherence protocols can be detected early while parsing the PSD specifications of eight coherence protocols. In the next chapter, some experiment results on model verification against data obtained from measurement will be presented to demonstrate the potential of applying the specification-based verification to the DSIMCLUSTER model.

Chapter 5

Simulation of DSM Multiprocessors

Central to a simulation are the model specifications defining the structure and semantics of the components derived from the target system. In the previous chapter, *model specifications* that describe the building blocks of a DSM system were presented. This chapter describes how the DSIMCLUSTER simulation implements these specifications. The chapter begins with Section 5.1 giving an overview of the working of the framework, with particular reference to the inputs and the outputs produced. After this overview, the remaining content of this chapter progresses by describing the modules that implement four key operations of the DSIMCLUSTER. The first operation, a mechanism to transfer control among the simulation components, is integrated into a workload file definition described in Section 5.2. In Section 5.3, the steps to accomplish the second key operation, *i.e.* the emulation of a multithreaded runtime environment to simulate parallel multithreaded workloads are presented. Section 5.4 demonstrates how the architectural components of a DSM system are simulated. The third operation, an implementation of a specification-based verification technique to simulate different bus-based coherence protocols, is demonstrated in Section 5.5. In Section 5.6, the fourth operation that involves the process to obtain statistical results from a simulation run using a hierarchy of performance metrics is described. The DSIMCLUSTER simulation model has been verified against a SunFire 15K machine using a workload of a particular function obtained from the LU decomposition program of the NPB 2.3 benchmark. This verification experiment is described in Section 5.7. In the last section, the issues and techniques used in implementing the DSIMCLUSTER simulation are summarised.

5.1 DSiMCluster Overview

DSiMCluster is an interpretation-driven, discrete-event simulation implementing the specifications of a generic DSM system described in Chapter 4. The simulation model was implemented using the DES engine and facilities provided by the HASE environment. Figure 5.1 shows an overview of the DSiMCluster framework. The workload file written in a format specific to the DSiMCluster model is taken from the execution program of the target workload (Figure 5.1 (a)). The DSiMCluster executes this workload in the simulated environment and produces five different types of output (Figure 5.1 (c)).

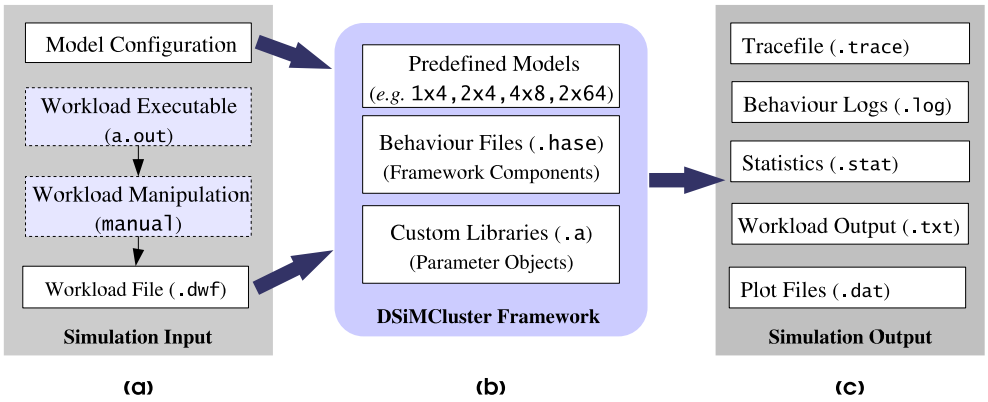


Figure 5.1: Overview of the DSiMCluster framework.

The first simulation output is a tracefile created automatically by HASE functionality. A tracefile is a record of the simulated behaviours (*e.g.* the states of entities, and the sending and receiving of packages) sorted in the logical time-stamp order. In addition to the trace file, the other four outputs are implemented specifically for the DSiMCluster model. The second output, *log files*, are text files recording the step-by-step actions performed by each simulated component. The third output, a *statistic file*, is a text file keeping the performance counters and metrics obtained from the simulation. The fourth output, *plot files*, are tab-separated data files recording different counter values against the simulation time[1]. Moreover, the outputs obtained from executing the workload file in the simulation model are dumped into a text file.

Figure 5.1 (b) shows the structure of the DSiMCluster implementation. The

[1]The list of the available plot files is shown in Appendix C Section C.4.2.

DSIMCLUSTER comprises a group of predefined models[2], the behaviour files, and six custom libraries implementing parameter objects. The six custom libraries used in the DSIMCLUSTER model are listed below:

- cache parameters and the PSD parsing routines (`libclc.a`)
- processor parameters and instruction set routines (`libclp.a`)
- operating system parameters and DSM routines (`libclos.a`)
- memory organisation and facilities (`libclm.a`)
- simulation profiler and statistical routines (`libpc.a`)
- generic utilities (`libgc.a`).

To use the DSIMCLUSTER, the first step is to build the executable file of a DSM cluster (DSIMCLUSTER Kernel). This step is referred to as *customisation* as it produces a customised model of a DSM cluster from the available framework components (Section 5.1.1). The second step is to re-configure the framework by selecting parameter values to create an experimental platform representing a specific target architecture, and to run experiments on it (Section 5.1.2). This step is sometimes referred as *reconfiguration*. The following subsections describe the process of building and running the DSIMCLUSTER.

5.1.1 Building the DSIMCLUSTER Kernel

Figure 5.2 shows the process of building the DSIMCLUSTER executable file from the package which includes customised library source files and HASE behavioural files. To build this, users have to follow three steps. Firstly, users invoke the Unix `Make` utility to build the customised library from the source files (Figure 5.2 (1)). These customised libraries implement parameter objects that are used for customising DSIMCLUSTER to emulate the behaviour of the *target* parameter, which is set to represent different target architectures.

The second step is to build the simulation executable. In this step, users define the desired target architecture by configuring the number of SMP nodes and the number of processing elements for each SMP in the `NETWORKENTITY` definition in the DSIMCLUSTER structural definition file (`.edl` file). Once the target architecture is

[2]including the models of different DSM architectures of 4, 8, 32, 64 and 128 processors

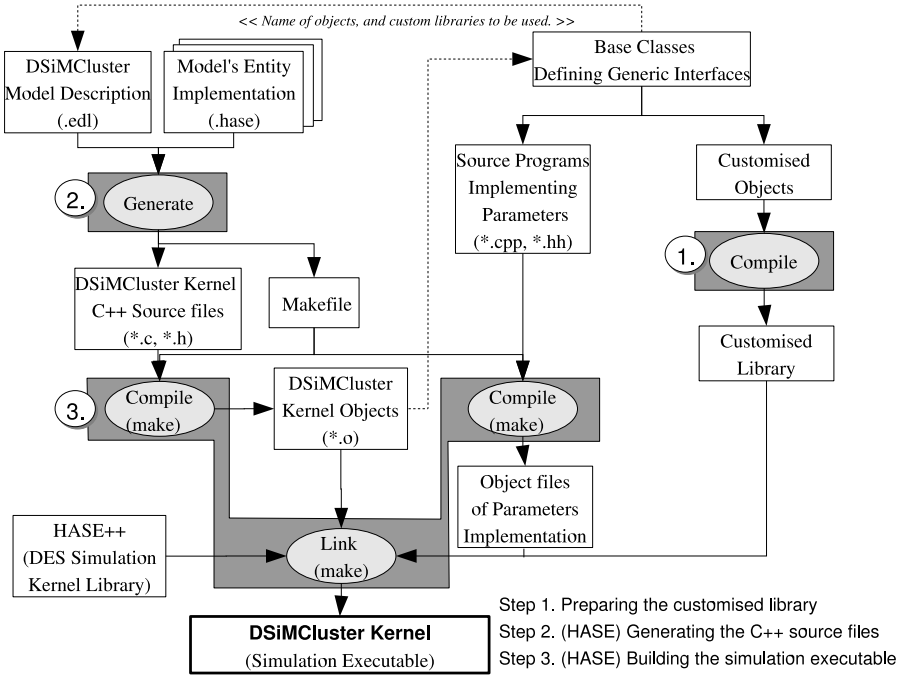


Figure 5.2: Process to build DSIMCLUSTER kernel.

defined, users load the project via the HASE GUI and use the `Build Project` command to create the simulation executable (`<model name>_hase`). As described in Section 4.1.1, this step will command HASE to generate the simulation source files from the pre-defined behavioural files and link them with the customised library (Figure 5.2 (2 and 3)).

5.1.2 Running Simulation Experiments

Once the DSIMCLUSTER kernel has been built, users can use this kernel as an experimental platform. To run an experiment, users configure parameters according to the experimental factors of interest, and invoke a `simulation run` command via the HASE GUI. Figure 5.3 shows the steps involved in running a simulation experiment on the DSIMCLUSTER. During a simulation run, the model kernel works in four phases: a configuration phase, a loading phase, a simulation phase, and a report phase. In the first phase, DSIMCLUSTER kernel reads the configuration files, and creates objects reflecting the design parameters. It is important to note that users can configure the model by both selecting the parameters on screen (Figure 5.4 (a)) and

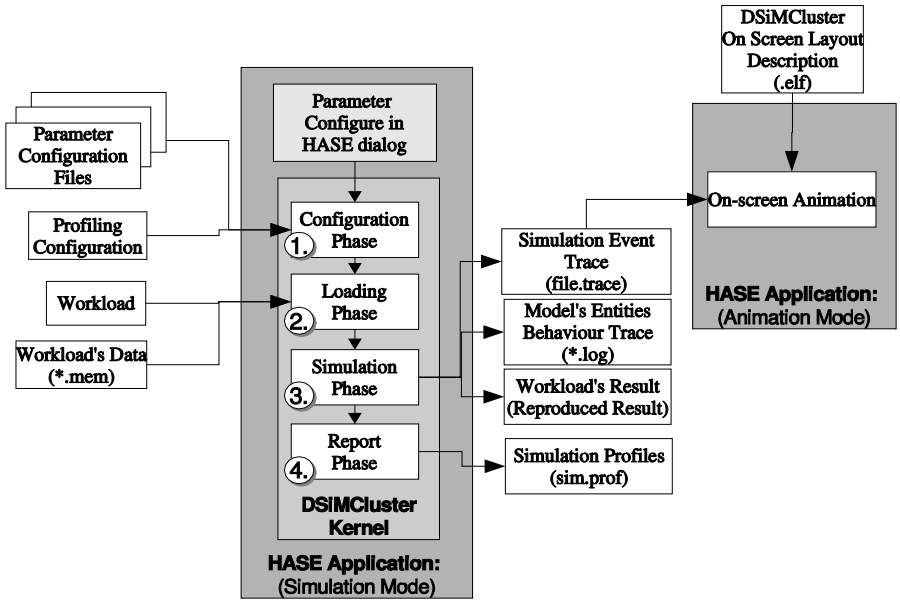


Figure 5.3: Steps to run a simulation experiment.

using a set of configuration files (Figure 5.4 (b and c)). Examples of the performance metric configuration shown in Figure 5.4 (c) demonstrate that users can not only control the calculation at each hierarchical layer of the architecture, but can also define a custom metric function to calculate a different metric from the set of simulation counters.

In the second phase of a simulation run, the DSIMCLUSTER kernel reads the workload file and any input data before emulating the linking, loading and process scheduling tasks of the operating system. Once this second phase has finished, the processors' instruction caches are loaded and the workload is ready to be simulated.

During the third phase of the run, DSIMCLUSTER emulates the behaviour of the DSM cluster components that run the workload. During this process, DSIMCLUSTER will record the simulation events and the components' behaviour traces into text files. In addition, if users set DSIMCLUSTER to record the plot files (Figure 5.4 (a)), several plot files will be generated and stored in the result directory of the experiment.

The last phase of a simulation run is the report phase. Once behavioural simulation has finished, DSIMCLUSTER calculates the metrics from the counters collected during the workload simulation as described in the previous section. Once the calculation is finished, the result of the metric calculation is dumped into a text file. This

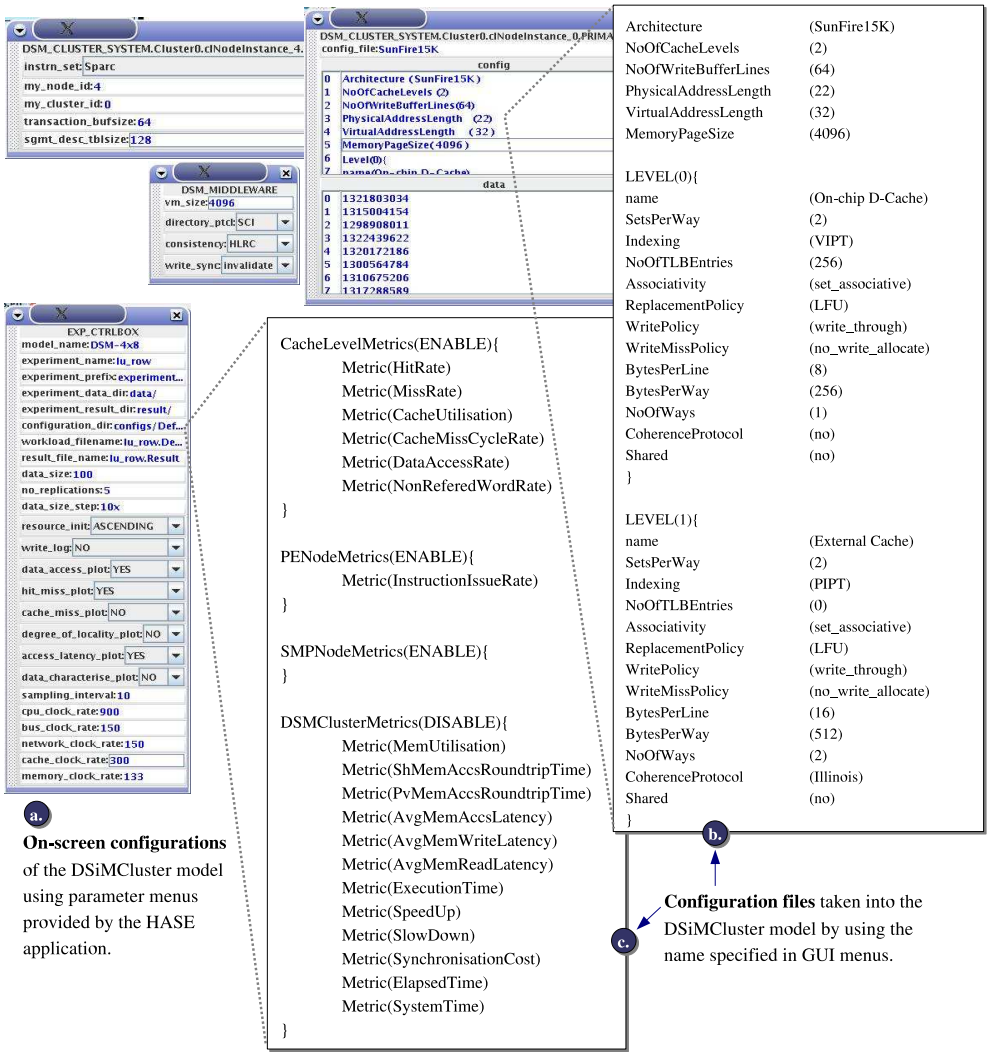


Figure 5.4: DSiMCLUSTER configuration.

result can also be used to identify whether the DSIMCLUSTER simulator should stop or should run another simulation with different parameter values obtained from the configuration file. In the following sections, the implementation details of how the DSIMCLUSTER carries out each of these phases are presented.

5.2 Form of Workload Input

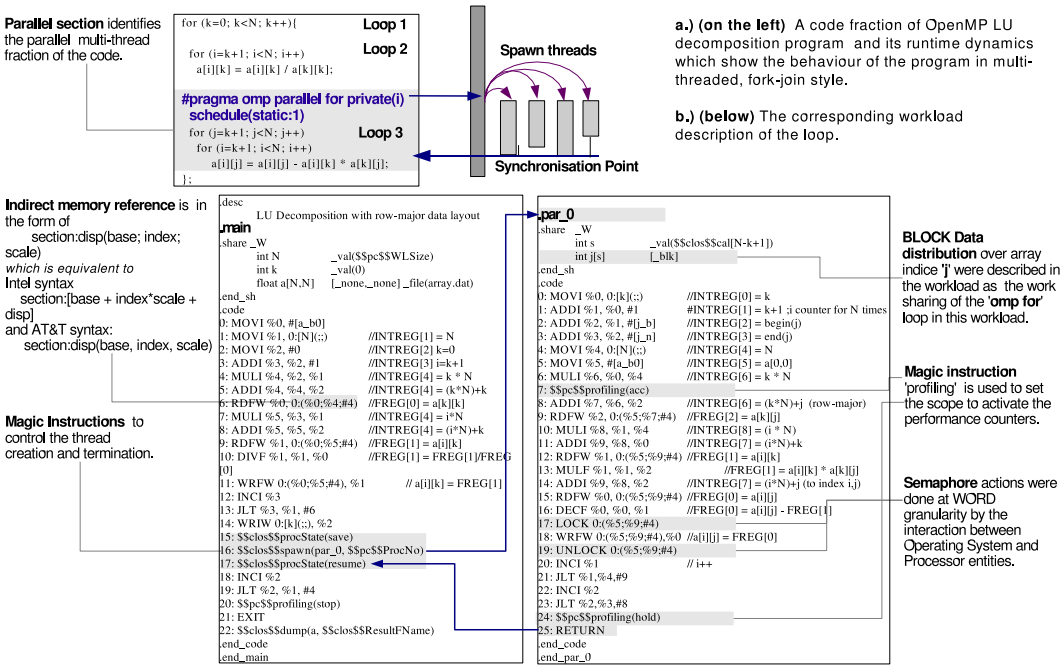


Figure 5.5: Workload file of LU decomposition.

The term *workload* is used to represent an executable program for a target-machine, which has to observe specific characteristics. In this work, the scientific workloads generated from source code written in OpenMP have been observed. Figure 5.5 shows the structure of a workload file used in DSIMCLUSTER. The original code fragment of an LU decomposition program with OpenMP C++ annotation, and its runtime characteristics, are shown in Figure 5.5 (a). Figure 5.5 (b) shows the corresponding workload file describing the main computation loop of the code fragment.

The type of workload files taken by the DSIMCLUSTER's OS Entity are the DSIMCLUSTER Workload Format (DWF) files. These DWF files hold code, data, and some DSIMCLUSTER special instructions suitable for recreating the OpenMP

parallel section and manipulating the multithreaded execution units. When the DSM_Manager Entity schedules a workload, the corresponding DWF file is parsed and converted into the simulated workload process in the node's OS Entity. Each section of the DWF file is copied into a `Workload` object to facilitate subroutine calls and the multithreaded context switches.

The DWF workload file format consists of the two following parts.

- **Description.** A description section begins by `.desc` followed by a one line string describing the project

- **Program basic units.** The program basic units define units of program execution, *i.e.* a subroutine, the main process or a thread.

The DWF program basic units include a *process section*, and one or more *thread or subroutine sections*. The section named *main* is regarded as the program's main process. Other section names are user defined, and these are regarded as thread sections. Each section comprises two types of basic segment: data segment and code segment. A *data segment* essentially contains a definition of global data. The header of the data segment includes directives to specify the segment type (*i.e.* private or shared), and its privilege level[3]. Both private and shared data segments contain variable declarations and information about where to find their initialisation values. Moreover, in a shared data section it is essential to identify the data distribution policy for each array variable. A code segment contains the execution code written in an assembly-like format. This code uses indirect memory references to access data defined in the data segment. It is important to note that, in a workload file, the position and order of sections and segments are not restricted.

A code segment contains a sequence of assembly instructions taken from an ELF[4] executable file (Section 2.1.2.1) using the ELF reader and disassemble utilities. Obtaining the workload definition involves manual code manipulation. Figure 5.6 shows the steps to obtain a workload file from an ELF executable.

[3] A three-bit code is used to grant privilege to local and remote access. The first bit identifies if the segment is shared. The second and third bits identify access rights for requests from local and remote processors respectively

[4] In this Chapter, the ELF file refers to the Executable and Linking Format object file (not the Entity Layout File of the HASE application), unless otherwise stated.

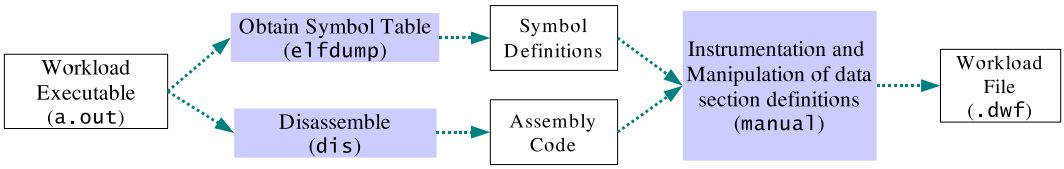


Figure 5.6: Steps to obtain a workload file from an ELF executable.

Since DSiMCluster targets parallel multithreaded applications, expressing parallelism using OpenMP sentinels, threads are dynamically created and terminated in fork-join style. This means that when an execution hits a parallel section (the highlight in blue in Figure 5.5 (a), a number of threads will be created, each of them sharing the computation load according to the prespecified description. As shown in Figure 5.5 (a), when one of these threads has finished, the thread will wait until the others have also finished their computation load. Once all of the threads have reached the synchronisation point, they will terminate themselves and return control to the main process. To emulate this process, a control-transfer trap using service-request definitions has been used. This control-transfer trap is the first key operation introduced into DSiMCluster, as mentioned in Chapter 1 Section 1.4. To integrate this into a workload, the calls to the thread routines in a workload are replaced by sets of *magic instruction interfaces* to trap the control of the simulation run. When the Processor Entity recognises a magic instruction, control will be transferred to the thread management services of the simulated OS Entity. A magic instruction interface is defined in the form:

```
$$<service entity>$$<service function>(parameters)
```

This implementation uses a function pointer to locate the corresponding callback function. The processor action will be halted once a magic instruction trap is hit. DSiMCluster then uses the combination of the name string of both the service entity and the service function as an index to retrieve the corresponding function pointer from a service table. Once this is done, the service function call performs the tasks listed below prior to transferring control back to the Processor Entity:

1. create or terminate a number of threads for running parallel tasks according to the parameters specified;

2. in the case of thread creation, the value of variables declared outside the parallel section will be passed to the newly created thread[5];
3. set the profiling scope according to the specified parameters.

Currently, DSIMCLUSTER can input the DWF file written in a DEFAULT instruction set[6] or a Sun SPARC V 9 instruction set. In both instruction set types, the workload file manipulation involves (1) defining data segments and their distribution, (2) renaming references to variables in the code segment, and (3) removing any system calls for I/O operations and other instructions that are not essential to the computation of interest. Once the code has been edited, it is instrumented with magic instructions to specify thread creation/termination, synchronisation, parameters transferred by value, and profiling scope. To create a DWF file from a SPARC execution file involves further manipulation to the code section as following:

- create the main and subroutine sections
- replace the calls to `__mt_` functions with `spawn` magic instructions
- replace the `sethi` instructions with `mov` instructions
- replace the `call` instructions with magic instructions
- reorder the codes as the effects of a branch delay slot
- resolve the branch target addresses
- replace a branch target address with the relative line number

Once the above manipulation has been done, the workload file is ready to be read by the simulation. The following section presents how the DSIMCLUSTER components interface to a workload.

5.3 Workload Interface

In this section the second key operation underlying the DSIMCLUSTER mechanism is described, *i.e.* an emulation of a multithreaded runtime within the simulation process's space. To implement this, three interface modules are used to emulate a multithreaded runtime environment. These are: an operating-system (OS) emulation mod-

[5] According to OpenMP specification, unless the variables were defined as *shared*, they will be passed to the parallel section by value. [OpenMP, 2002]

[6] *i.e.* the instruction set that is used for testing purposes by the DSIMCLUSTER model

ule, an address-translation module, an instruction translation module. These modules have been implemented in light-weight threads belonging to DSIMCLUSTER's kernel process. Firstly, the OS emulation module reflects the activity that controls the behaviour of the workload at runtime. Secondly, the address-translation module provides the flexibility to support a virtual-memory (VM) workload and reflects the cost of activities for the physical-logical memory address mapping. This module also plays an important role in emulating the implementation of software DSM which commonly uses VM address mapping to identify shared data. Lastly, the instruction translation module provides a set of supported instructions and the execution sequence of each of them.

5.3.1 Operating System Emulation Module

In the OS module, the characteristics of the workload's executable units (process/thread) and the allocation of resources (*e.g.* the number of requested processors) are emulated. These essential functionalities of an OS, which reside on each SMP node, have been implemented in an OS Entity. This entity provides users[7] with an interface to submit a workload file into the simulation kernel, allowing access to the functional capabilities of the simulated architectural components. Once a user requests a workload to run, the OS Entity performs a number of operations to transform the workload file into an executable unit. These operations are listed below.

- Parsing the workload file
- Creating an entry in the Process Table for this workload (Figure 5.7 (a.1))
- Allocating requested resources such as a private virtual address segment and a link to shared data segments (Figure 5.7 (a.2-3))
- Generating the memory footprint to initialise the memory snapshot
- Expanding data-distribution macros to calculate the index and chunk size
- Loading and linking data segments

Central to the OS Entity is the emulation of the functionalities of memory management and process management. Regarding memory management, a segmented-demand paged allocation scheme is used. In this scheme, the segment size is allocated using dynamic allocation, consequently, the segment size may not be equal to

[7] From this point on the *users of* DSIMCLUSTER *simulation model* will simply be referred as *users*.

the page/frame size. Page-frame allocation is carried out using a memory map table (MMT) together with a last-in first-out (LIFO) policy.

In process management, the OS Entity generates a representation of a workload executable unit based on a single-program multiple-data (SPMD) paradigm. Therefore, in DSIMCLUSTER, a workload's code segments (which are scheduled to one or many SMP nodes) are identical. Once an executable unit has been generated and all of its required resources have been allocated, it will be scheduled to the available processors according to the scheduling policy specified. During the scheduling process, the OS Entity updates the entries of the segment descriptor table (SDT) of the corresponding processors to provide the most recent address mapping information.

To handle system calls, the OS Entity provides a mapping table to redirect a request/signal to its handling routine. These requests include: an error signal, a segmentation fault signal, an interrupt service request, a request for thread creation/termination, a semaphore request for exclusive access to a shared-memory region and a thread-synchronisation request. The OS Entity also records the time spent in the handling routine to reflect the overhead cost of the operating system management.

5.3.2 Address Translation Module

In this module, the mechanism to locate where the data reside, and in which SMP's memories, is emulated. The address translation module involves two sets of translation tables, a set of segment descriptor tables (SDTs) and a set of translation look-aside buffers (TLBs). Firstly, a set of SDTs residing in each Processor Entity (*e.g.* one SDT per one executable unit) is used to control an access to a memory segment. Each SDT maps an indirect-address reference to an absolute virtual address called an effective address. It also checks the access rights of that request to the memory segment, and identifies whether the segment is shared or private. In case of an access violation, SDT causes an error signal to be sent to the OS Entity.

The second set of tables, TLBs, resides in the memory management unit (MMU) in each Processor Entity and also in each Cache Entity if it is virtually indexed, physically tagged[8]. A TLB holds and manages the absolute address mapping between the

[8]In *virtually-indexed, physically-tagged* (VIPT) cache, virtual addresses are used to index into the cache tag and data arrays while accessing the TLBs; the resulting tag is compared against the translated physical address to determine cache hit.

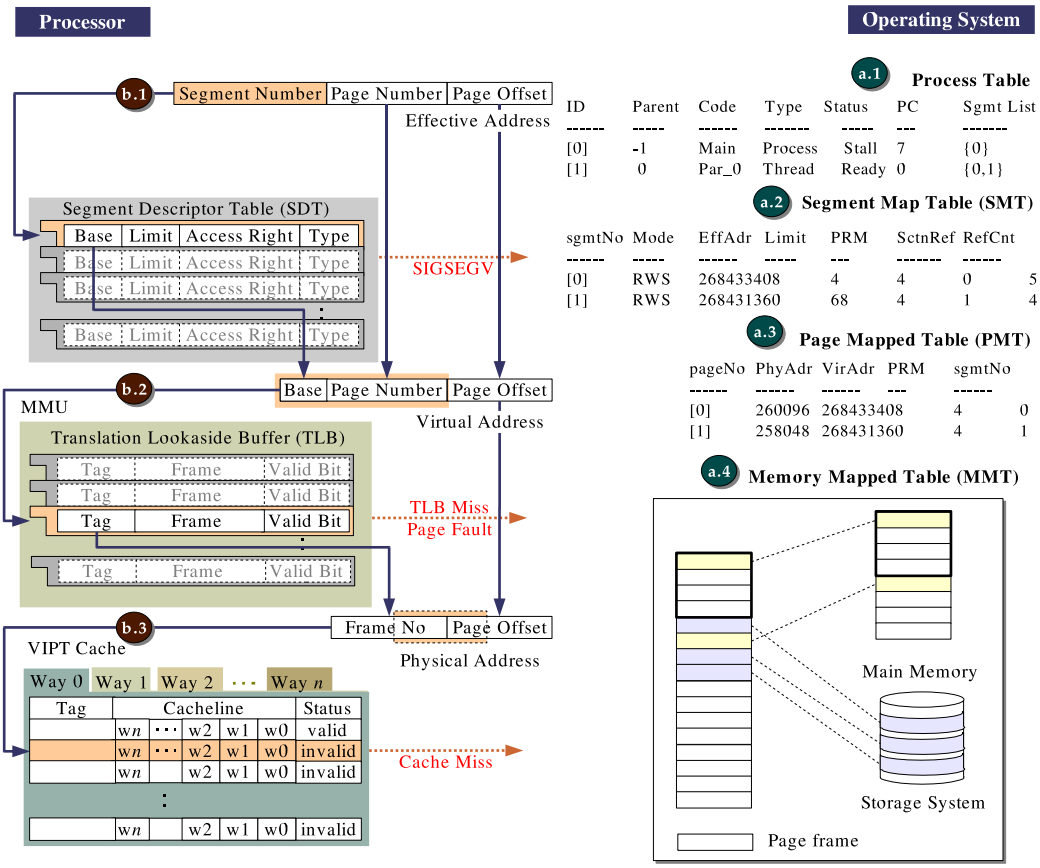


Figure 5.7: Address translation mechanism implemented in DSiMCLUSTER.

virtual and physical address space. In order to implement the TLB in the DSIMCLUSTER simulation model, the following assumptions are applied:

- There is no instruction TLB in the model.
- A TLB miss forces a cache miss at the cache which uses this TLB for address translation.
- TLB entries are stored and accessed in a fully associative manner.
- When a TLB page translation is replaced, all the corresponding cache entries (*i.e.* the entries which their addresses are in the victimised page) are invalidated.
- When a TLB page translation is replaced, the victim entry is determined using true least-recently-used (LRU) criteria.

Similarly to the profiling of an OS emulation module, the time spent on the address translation routines is recorded to reflect, in particular, the overhead cost and consequence of fault handling.

5.3.3 Instruction Translation Module

In the last module, instruction translation, the functionality of a processor when executing a target instruction are implemented. DSIMCLUSTER provides two different instruction-set implementations and a mechanism to plug in a new implementation. The workload instructions are translated during the simulation run and executed using the semantic functions of the instruction-set class. Each instruction-set implementation class is inherited from a set of interface classes defined in a `libclp` library. Figure 5.8 illustrates the class diagram related to the instruction set definition and its mechanism to plug the new instruction-set implementation into the DSIMCLUSTER kernel.

As mentioned earlier, DSIMCLUSTER currently supports two instruction sets: a DEFAULT instruction set and part of the Sun SPARC V9 instruction set (see Appendix C). Using the class hierarchy demonstrated above, the instruction set can be chosen before running a simulation. The instructions defined in the `.code` section of the DWF workload file are translated and executed by the Processor Entity. The instruction execution steps and their actual results stored in data registers are emulated in detail.

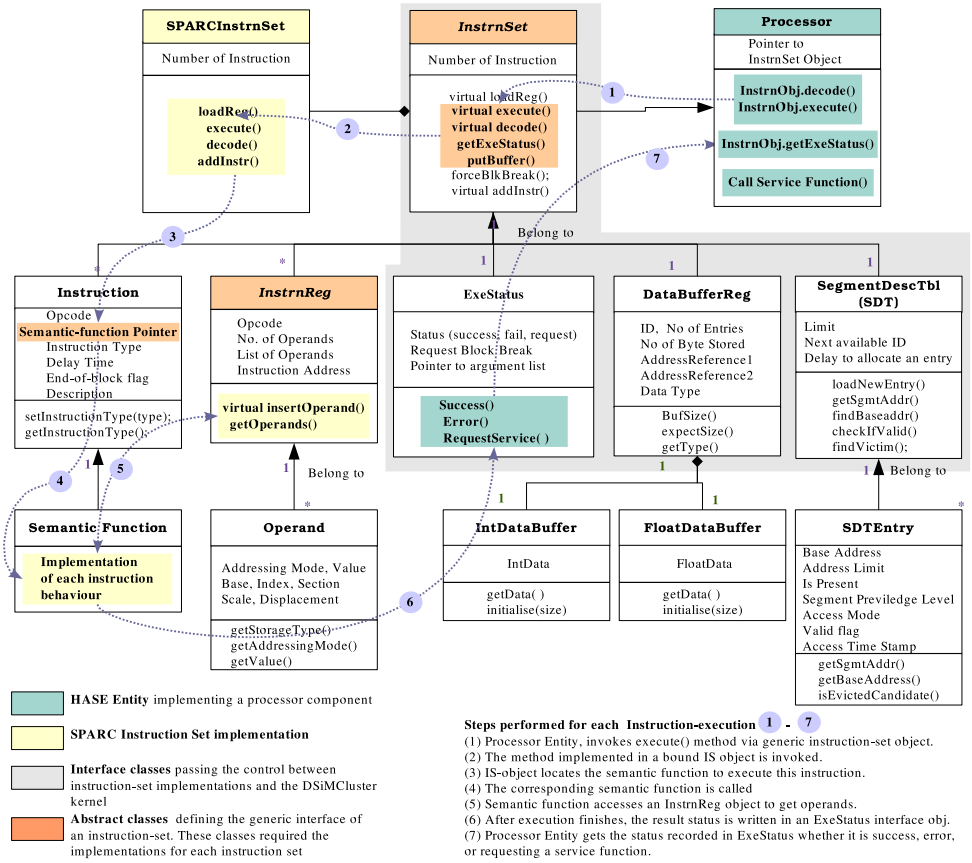


Figure 5.8: Class diagram illustrating the instruction-emulation classes in `clp` library and their interaction with the Processor Entity of the DSiMCluster kernel.

The Processor Entity continuously reads and executes each one-line string instruction from the `.code` section of the workload. After an instruction string is fetched, it is passed to the instruction set object to be decoded. The instruction set object then recognises the instruction mnemonic and operands, and stores these information into an `InstrnReg` object. At this step the instruction mnemonic is translated into an integer code ready to be mapped to the callback function which implements the execution of the instruction.

Once a decoding step is finished, the Process Entity calls the `execute()` method the instruction set object to actually execute of the instruction. At the `execute()` method, the instruction set object uses the translated code of the instruction mnemonic to get the pointer to the function which implements the semantic routine of this instruction (so-called the semantic function). The instruction set object uses this pointer to call the semantic function.

The semantic function performs the tasks required for this instruction[9] and records the results into data registers. Every semantic function stores the status of an execution into an `ExeStatus` object at the end of its tasks.

Once the `execute()` method is finished, the Processor Entity checks the execution status from the `ExeStatus` object. If the returned status is a 'success', the Processor Entity will fetch the next instruction and continue translating the next instruction. In case of a branch instruction, the status returned is a 'basic block break' which causes the Processor Entity to stop the execution cycle at this clock tick period.

If the instruction is a stack or a memory access operation, the instruction set object stores the relevant information into the `DataBufferReg` object prior to setting the corresponding service code (*e.g.* MEM_RD, MEM_WR, STACK_PUSH *etc.*) into the `ExeStatus`. In this case, the execution status will be set to 'request for service' instead of 'success', causing the Processor to pause the instruction cycle and perform the service as requested. The following pseudo code of the Processor Entity demonstrates this process.

```
1 InstrnSetObj = createInstrnSet( name )
  CALL InstrnSetObj.loadSDT( userSDT )
```

[9] *e.g.* performs an addition of two integer values from the specified integer registers in the case of an ADDI instruction of the DEFAULT instruction set, or updates the value of the program counter register in the case of a JUMPI instruction of the DEFAULT instruction set

```
DO
     code    = InstrnCache[PC]
     PC  = PC+1
     CALL InstrnSetObj.decode(code)
     CALL InstrnSetObj.execute()
     status  = InstrnSetObj.getExeStatus( &requestCode )
WHILE ( status EQU success )
IF ( status EQU request for service )
THEN CALL serviceRoutineMap[requestCode]()
```

5.4 Simulation of DSM Components

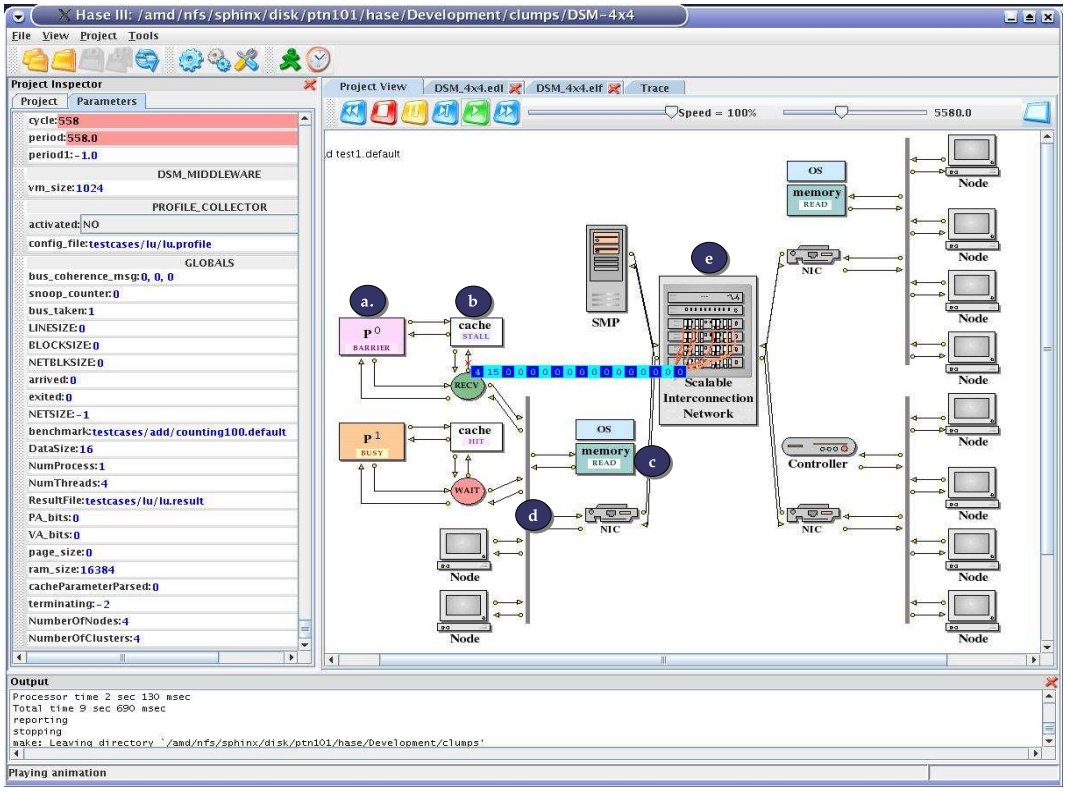


Figure 5.9: DSIMCLUSTER screenshot of a 4×4 DSM model.

This section gives a brief explanation of how the architectural components are simulated. These components represent the hardware mechanism of a DSM cluster.

Each architectural component that belongs to the framework group is implemented using a light-weight thread as a HASE Entity. The ones that belong to a parameter object group are implemented as C++ objects in a user-defined library. Figure 5.9 shows the relationships among these components by a screenshot of the 4×4 DSM model using DSIMCLUSTER. The specification in Table 4.2 (page 119) listed the architectural components covered in the DSIMCLUSTER simulation as well as their parameter objects. The following paragraphs give brief descriptions concerning the implementation issues of some of the components.

5.4.1 Processor

In DSIMCLUSTER, a processor is driven by the workload by continually interpreting instructions and performing instruction execution in an instruction cycle. As for the steps mentioned earlier, in every clock period, the Processor entity will execute the instructions until (a) a service requested has been placed or (b) the end of basic block is reached. The end of a basic block is specified in the implementation of an instruction set object, and is normally defined by a *control-transfer instruction* or *synchronisation instruction*. In this model, the instruction cache is assumed to always hit and a detailed analysis of instruction-level parallelism is not covered.

Prior to a decoding step, the Processor Entity checks if the instruction is a magic instruction (*e.g.* thread creation, the profiling activity set/unset). If a magic instruction is found, the corresponding service function will be invoked.

5.4.2 Memory Hierarchy

Since a cache plays an important role in memory access performance, detailed cache parameters have been implemented. The cache configuration is read from a text file (see Figure 5.4 (c)) into the simulation and the architecture configured accordingly before the simulation starts. Cache organisation and its parameters have been implemented in separate classes and grouped into a `libclc` library. The implementations included in the library are the classes `CacheLine`, `CacheWay`, `CacheOrganisation`, and `WriteBuffer`. The organisation of each cache level is implemented in a class called `CacheOrganisation`. Therefore, the number of cache levels, the associativities, the number of ways, line size, cache size, indexing method, and replacement

policy are fully re-configurable at runtime. Moreover, an address translation module (via TLB) is plugged in only when the cache level is configured to be VIPT.

5.4.3 Emulation of DSM System

The components of interest in the DSM implementation have been shown in Table 2.8 (page 50). DSIMCLUSTER implements the DSM management feature in a HASE Entity called DSM Manager. This entity acts as a control centre emulating the functionalities of a single software image of a DSM system. A centralised DSM management node, *i.e.* a dedicated node in a cluster that runs only the DSM management software, has been implemented which includes a job queue receiving batch job requests. An analyser may use a batch job request to define a number of workload file names to be fed into the simulation. The DSM Manager will pick up these names sequentially from the workload list to locate the workload file. Once the file is found, the DSM manager will allocate the shared data area from a DSM memory mapped table and schedule a process to an OS Entity using the scheduling policy defined. During the lifetime of the workload, the DSM manager will maintain memory coherence by listening to the signal from the OS Entity when a shared-data segment has been written. The transition of a shared data state will activate a procedure to maintain coherence according to the specified policy.

5.5 Specification-based Parameter-Model Interaction

In this section the description of the third key operation regarding verification applicability in DSIMCLUSTER is presented. The main concept embraces the automatic verification of coherence protocol emulation and the way to include this verification in a simulation model. A Specification-based Parameter-Model Interaction (SPMI) is proposed as a technique to verify and co-simulate a coherence protocol with the DSIMCLUSTER framework. This technique includes the protocol specification using a Protocol State-transition Description (PSD) language (see Section 4.3.1 page 121), and the ways to direct the model's behaviour using the semantics obtained from this specification. The concepts of how a PSD specification can be mapped to the implementation of the DSIMCLUSTER has been presented in Section 4.3.2.1. This section

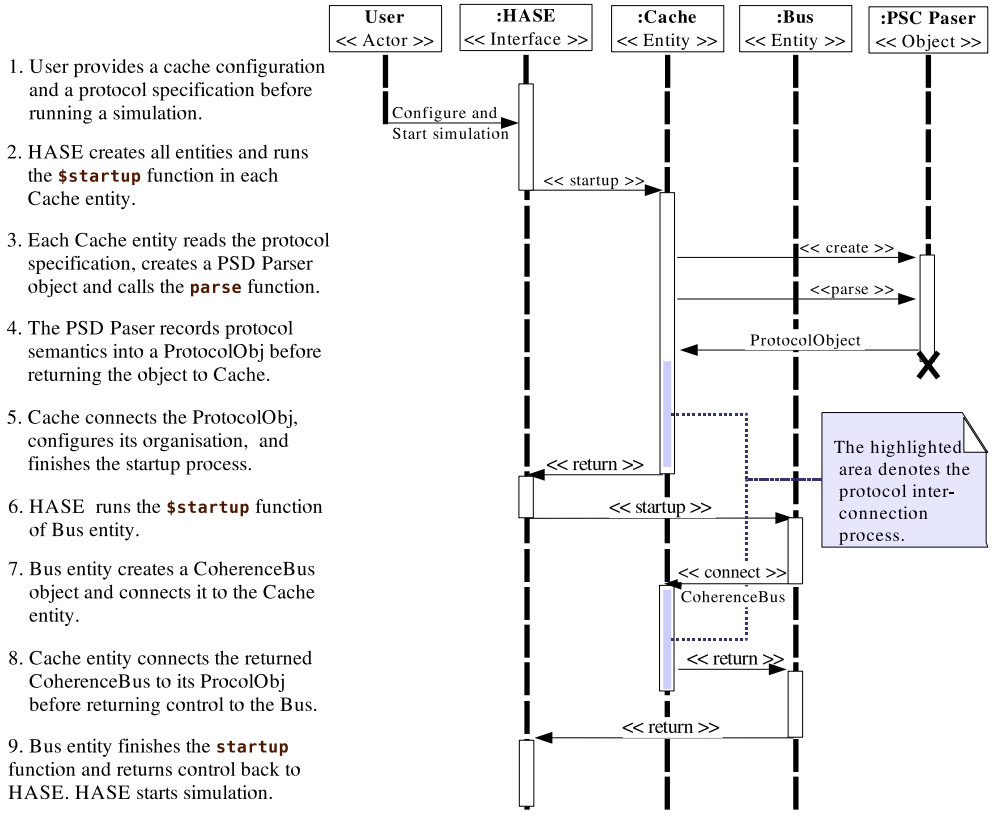


Figure 5.10: A UML sequence diagram showing the SPMI steps.

gives the implementation detail of how the PSD specification has been included and mapped to the DSiMCluster simulation.

The Specification-based Parameter-Model Interaction (SPMI) technique is a systematic means to add a specific feature of a parameter to a simulation model using the semantics obtained from a well-formed specification. Figure 5.10 shows a sequence diagram demonstrating the steps involved in the SPMI technique. These steps can be grouped into four major operations: (a) defining a protocol specification (step 1); (b) building the specification semantics into a coherence protocol object using the PSD parser (steps 2-4); (c) connecting the coherence protocol object to a simulation model (steps 5-8); (d) emulating the protocol behaviour (step 9, also described in Figure 5.13).

In Section 4.3.1, the definition of PSD language and the process to check the semantics of a coherence protocol from a PSD specification against three common errors have been described. As shown in the sequence diagram, once the well-formed protocol has been parsed, the object representing the coherence protocol is returned from the PSD parser and is ready to be connected to the Cache Entity. The following subsection explains the organisation of the coherence protocol object.

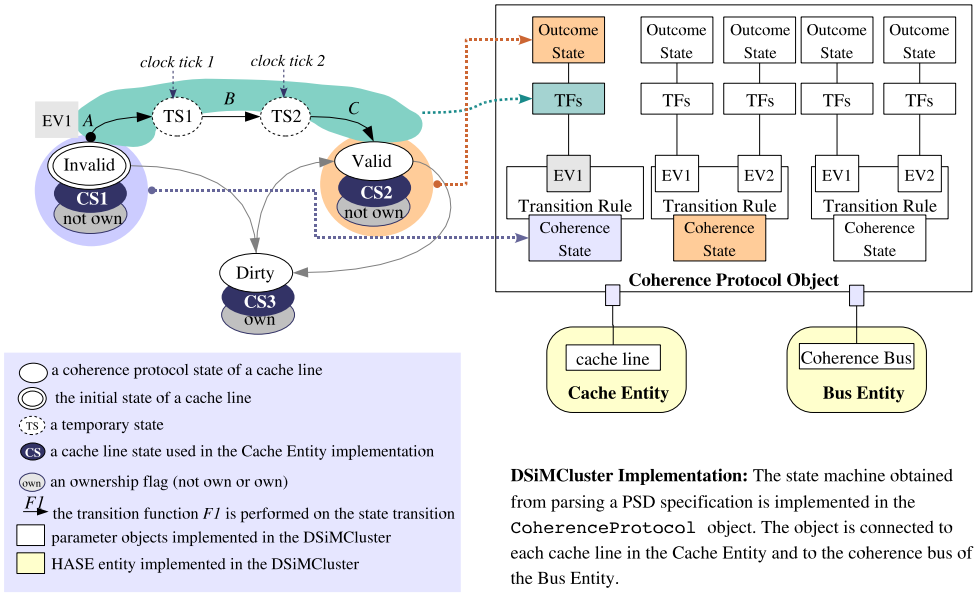


Figure 5.11: The state machine represented using the `CoherenceProtocol` object.

5.5.1 Coherence Protocol Object

The SPMI technique employs the concept of object-oriented simulation by using an object to represent both a semantic and a behavioural interface to different modules of the model implementation. An object of a `CoherenceProtocol` class is the central channel of interconnection between the protocol behavioural emulation and the semantics obtained from a protocol specification. The `CoherenceProtocol` represents the state machine obtained from parsing a PSD specification. Figure 5.11 shows the representation of a state machine using the CoherenceProtocol object and the connection of the object to the Cache and Bus framework components. This `CoherenceProtocol` class contains two important member classes, namely, the `TransitionRule` object and the `CoherenceController` object. Figure 5.12 (b) and (c) illustrates the two objects and their interconnection.

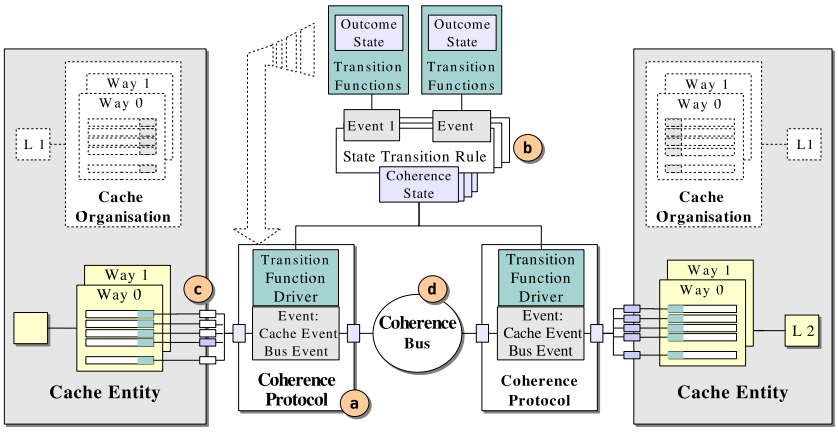


Figure 5.12: Protocol objects and their connections.

5.5.2 The PSD Re-entrant Parser

The semantic values of a protocol specification are recognised by a PSD re-entrant parser, a Generalized LR (GLR) parser[10]. In general, a re-entrant parser or pure parser is an independent object plugged into a program that employs its function. In DSIM-CLUSTER, the PSD parser has been implemented as an external object to be plugged into the Cache Entity framework component. The parser takes a PSD specification

[10]The PSD parser has been implemented using the Bison utility (http://www.gnu.org/software/bison/)

and generates an object representing the protocol (the `CoherenceProtocol` object) which is returned to the DSIMCLUSTER kernel. The parsing process of the parser has been presented in detail in Section 4.3.1.2. In the implementation, the parser works in collaboration with a lexical scanner[11]. The lexical scanner performs lexical analysis by recognising tokens and passing these accepted tokens to the parser. The parser then perform the syntactic analysis by matching the sequence of tokens against the PSD grammar rules. (In Appendix B Section B.1.2, the complete grammar rules of the PSD language are presented.) If a grammar rule is recognised, the semantic action of this rule will be invoked. Each semantic action constructs each constituent of a `CoherenceProtocol` object based on the semantic values of its parts. This process continues until the complete and well-formed specification has been successfully parsed. If there is a syntax error or if any erroneous definitions are detected, the parser will stop the simulation.

Once a well-formed specification is obtained, the PSD parser verifies the soundness and liveness properties of the specification using the steps explained in Section 4.3.2 and 4.3.3. If the specification has passed all of the sixteen PSD test conditions (summarised in Table 4.4), the specification is ready to be plugged in to the simulation kernel. Note that the tests of soundness and liveness properties have been implemented as methods in the `CoherenceProtocol` object.

5.5.3 Parameter-Model Interconnection

Once a specification semantic is successfully parsed (the state-transition information has been stored in a `CoherenceProtocol` object), the connection process begins. Each Cache Entity makes a copy of the `CoherenceProtocol` object, creates a handle to its copy, before passing this handle to the constructor of a `CacheOrganisation` object. A `CacheOrganisation` object[12] creates `CacheWays` and `CacheLines` according to the configuration, and also creates a `CoherenceController` (Figure 5.12 (c)) linked to each `CacheLine`. The handle of `CoherenceProtocol` is then attached to every `CoherenceController` (Figure 5.12 (a)). Once all Cache Entities have fin-

[11] The PSD lexical scanner is implemented using the Flex utility (http://www.gnu.org/software/flex/) with the *bison-bridge* option.

[12] *i.e.* an object representing the structure of cache, its contents, and associativity according to a configuration

ished these steps, the Bus Entity[13] communicates with each of the Caches to get their `CoherenceProtocol` handles. The Bus Entity then creates an instance of a `CoherenceBus` object and links this object with the obtained `CoherenceProtocol` handles (Figure 5.12 (d)). Figure 5.12 depicts the complete connection of two protocol objects to the cache hierarchy of a 1×2 DSM model which maintains cache coherence at level 2 caches. This figure shows the model of the DSIMCLUSTER representing a DSM architecture of the type illustrated in Figure 2.7 (b).

5.5.4 Specification-directed Emulation

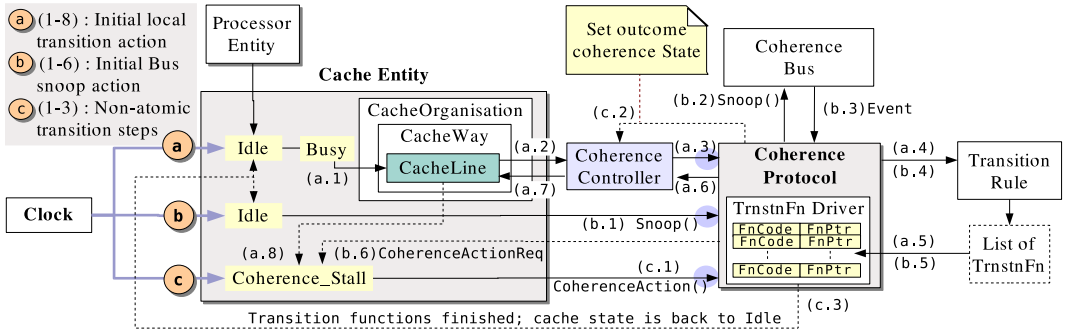


Figure 5.13: Block diagram shows the steps of the protocol behaviour emulation.

During a simulation run, Cache Entities play an important role in emulating the behaviour of the coherence protocols using the `CoherenceProtocol` objects. As shown in Figure 5.13, the emulation process is activated by the simulation clock signal sent to a Cache Entity. Coherence actions can be activated in two ways, *(a)* by a CPU access causing a hit/miss action at a cache line (Figure 5.13 (a)), or *(b)* by a BUS notified event snooped from the coherence bus (Figure 5.13 (b)).

The state of a Cache Entity is used to direct a sequence of protocol actions that require more than one clock cycle to finish, a so-called *non-atomic action* (Figure 5.13 (c)). It is important to note that the DSIMCLUSTER model uses a two-phase clock to model a Cache Entity. In this figure only the action of clock phase_0 has been illustrated. In clock phase_1, each Cache Entity directs its `CoherenceProtocol` object to snoop on the bus and to send an acknowledgement to a bus transaction if required.

[13] A Bus Entity in the DSIMCLUSTER represents a bus and its behaviour.

This is to prevent the case of *deadlock*, in which other caches could be waiting for a bus acknowledgement forever.

5.6 Hierarchical Profiling

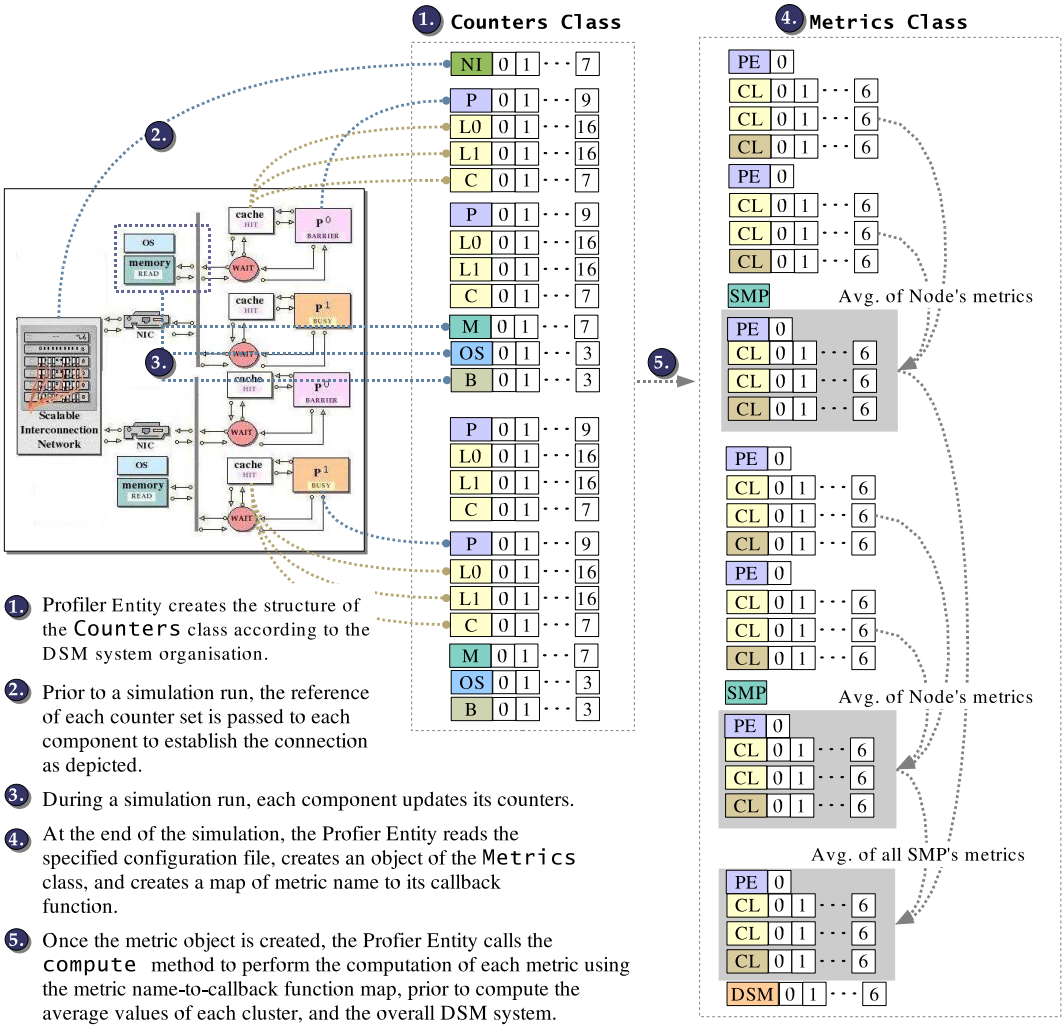


Figure 5.14: Structure and connections of the `Counters` and `Metrics`.

In a DSM system, performance can be observed at three levels: in each processing element (processor-cache), in each SMP node, and at a global, DSM system level. Performance estimation at each level is carried out via a group of performance metrics. Generally, performance metrics are used as the means by which analysers

measure and predict aspects of a system's functionality and resource usage, as a response to executing a workload. A performance metric can be described as a function of performance counters and the other metrics.

DSiMCluster provides a fixed set of performance counters for each framework component, and a configuration template to describe the statistical calculation of user-defined performance metrics based on these counters (See Appendix C, Section C.4). The structure of the performance metrics is defined in a class called `MetricMachine`, which defines a map of metric names and pointers to metric functions. Two member classes are defined in `MetricMachine`, including a `Counter` class and a `Metric` class. These member classes serve as containers to store the values of performance counters and metrics for each component indexed by the combination of an SMP ID and Processor ID.

Performance metrics for each level of a DSM cluster are defined in a configuration file (see Figure 5.4 (c)). Each of these metric names is stored in the `MetricMachine` class as a key to locate a `MetricCalculation` function implementing the calculation based on the available performance counters. This mechanism allows for user-defined metric functions to be dynamically plugged into the simulation kernel.

DSiMCluster calculates the metrics at the end of the simulation run. Metric calculation is done in a bottom-up style, from each cache level of an individual processing node up to the DSM cluster layer (Figure 5.4 (c)). Once the calculation is finished, the result of the metric calculation is dumped into a text file.

5.7 Model Verification Data

The specifications of eight bus-based coherence protocols have been developed. The verification using the PSD parser has been presented in the previous chapter. The result of the parsing step verification has shown that the PSD language is able to describe a state machine of a complicated protocol like MOESI. Some unsafe conditions have been declared and excluded from the specification using the PSD test conditions during the test of soundness and liveness properties.

However, these test conditions do not guarantee that the protocols will produce the correct result, since errors may creep into the definition of the transition functions. Therefore, this experiment aims firstly to validate the result of one coherence protocol

against measurement, and secondly to use the results obtained from the validated protocol to verify the result of the other protocols.

Table 5.1: Steps to perform the verification experiment.

Step 1: Screening of the workload
◦ Run LU Class A on the SunFire machine
◦ Select the dominant function (BUTS)
Step 2: Obtain a micro workload file
◦ Create a program calling the BUTS function
◦ Obtain the assembly codes of the program (using dis, CC -s, and elfdump)
Step 3: Create a DWF file
◦ Create the main and subroutine sections
◦ Create the data sections
◦ Manipulate codes as described in Section 5.2
Step 4: Perform measurement experiments
◦ Obtain initial data values
◦ Obtain the results of BUTS function
◦ Obtain the run profiles
Step 5: Perform simulation experiments
◦ Set the initial data values to the initialised data files
◦ Obtain the results of simulation runs
◦ Obtain the simulation run profiles

The experiment has been performed in five steps as summarised in Table 5.1. In the first step, a workload obtained from the NAS NPB 2.3 benchmark was run on a SunFire 15K machine to select the function which dominates the runtime. In the second step, the benchmark program is modified in order to obtain the input and output of the selected function. After the input and output of the selected function has been obtained, in the third step, the function is then implemented into a DSIMCLUSTER workload formate (DWF) file. In the fourth step, the measurement experiments have been performed by running the modified benchmark program (obtained from step 2) on the SunFire15K machine. The conditions used for the measurement experiments are presented in Table 5.2. In the last step, the simulation experiment has been performed. The DWF workload file (obtained from step 3) is simulated by the DSIMCLUSTER. In all simulations, the DSIMCLUSTER has been customised to model a SunFire 15K configuration as summarise in Table 5.3.

5.7.1 Experimental Design

A 1×4 DSM model with the SPMI technique has been chosen for preliminary model verification. Table 5.1 lists the steps used to perform this experiment. In this test, three different coherence protocols have been used with the same workload and compared with the results of one protocol against measurements obtained from a real system. To carry out these tests, three protocols with different numbers of states and behavioural characteristics have been selected: Synapse, Illinois, and MOESI. As SunFire maintains cache coherent by using the MOESI protocol, the first step is to validate the results of the MOESI specification. The validated result is then used to verify the Synapse and Illinois specifications. Each of these protocols has been examined using the OpenMP LU-decomposition workload obtained from the NAS NPB 2.3 benchmark [Jin et al., 1999]. All experiment runs used the class A benchmark, *i.e.* a small-scale workload with a three-dimensional matrix size of $64 \times 64 \times 64$. The values of hardware counters of a SunFire 15K machine have been collected using the same workload configuration[14] and the results obtained compared against the simulation results.

Model configuration. The simulation is based on a 1×4 DSM model which has been configured after the node architecture of the SunFire 15K server and Sun Fireplane system interconnect [Charlesworth, 2001]. Table 5.2 shows the configurations of the target machine used for running the measurement-based experiments. In the target machine, each processing node has a two-level cache. The level 1 Data Cache (DC) is a 64 KB on-chip cache with a line size of 32 bytes and a write-through, no-write-allocate policy. The DC is indexed by virtual address and tagged by physical address. The Level 2 or External Cache (EC) is an 8 MB external cache with a coherence control at a granularity of 64 bytes. The Level 2 cache is indexed and tagged by physical address with a write-back, write-allocate policy. The coherence policy of the SunFire system is MOESI, maintained at level-2 cache using the bus provided by the Sun Fireplane system interconnect [Charlesworth, 2001].

The DSIMCLUSTER was configured to represent the SunFire 15K configurations described above. However, due to the limitation of address length, half of the virtual and physical address space was simulated, *i.e.* the 32-bit virtual address and

[14] Using 4 OpenMP parallel threads running on four processors, one thread per processor

Table 5.2: Configurations of the SunFire 15K used for this experiment.

Component	Features
Processor	52 x 900MHz UltraSPARC III
Instruction Set	SPARC v9 64-bit RISC Architecture
Address Mapping	Two TLBs of 16 and 128 entries
	64-bit virtual address
	43-bit physical address
L1 Data Cache (DC)	a 4-way, 64KB-size, VIPT,
	32 bytes cacheline
L2 External Cache (EC)	a direct-mapped, 8MB-size, PIPT
	64 bytes coherence granularity
	256 bytes cacheline
OS	Unix operating system (Solaris 2.9)
Page Size	8KB
Compiler	CC
Flags	-x02 -xopenmp
Environments	OMP_NUM_THREADS=4
Profiler Tool	Collect (`collect`)
Flags	
1	-S off -p on -h EC_snoop_cb,,EC_snoop_inv
2	-S off -p on -h dcw,,ecim
3	-S off -p on -h dcwm,,ecref
4	-S off -p on -h ecm,,ecrm

22-bit physical address. Table 5.3 shows the configurations of the DSIMCLUSTER components used in this experiment.

Table 5.3: DSIMCLUSTER configurations for verification experiments.

Parameters	Features
Instruction Set	SPARC
Address Mapping	Data TLB of 128 entries
	32-bit virtual address
	22-bit physical address
Cache Parameters	NoOfLevels=2
Level 1 DC	a 4-way, 64KB-size, VIPT
	32 bytes cacheline
	CoherenceProtocol=no
Level 2 EC	a direct-mapped, 8MB-size, PIPT
	64 bytes coherence granularity
	256 bytes cacheline
	CoherenceProtocol=MOESI
Global Parameters	NumProcess=1
	NumThreads=4
	DataSize=64
	PageSize=8192(Bytes)
	Benchmark=LU-Buts-A.dwf
Experiment Control Parameters	WriteLog=Yes
	DataAccessPlot=Yes
	no_replications=6
	cpu_clock=900 (MHz)
	bus_clock=150 (MHz)
	network_clock=150 (MHz)
	cache_clock=300 (MHz)
	memory_clock=133 (MHz)

5.7.2 Verification of Multi-Level Inclusion

In a multiprocessor system, a multiple-level cache hierarchy has an *inclusion* or *Multi-Level Inclusion (MLI)* property if "the contents of a cache at level $i+1$, C_{i+1}, is a superset of the contents of all its child caches, C_i, at level i" [Baer and Wang, 1988]. Therefore, when a coherence protocol *invalidates* a content of C_{i+1}, the corresponding content in C_i should also be invalidated. Subsequently, this content should not be seen by the processor.

In a system with multiple level caches, the updating of a cache state must be finished before the next access to the lower level cache. Thus, the cache access is blocked until the actions of the coherence protocol have been completed. In DSIM-CLUSTER, the state `Coherence_Stall` (See Section 5.5.4) is used to prevent the lower level cache from being accessed by the processor prior to the completion of the coherence actions. Once the coherence actions are finished, the cache lines that have been invalidated by the coherence actions are marked as invalidated by inclusion. The following access misses to these cache lines are recorded separately as inclusion misses. The record of inclusion misses shows that if a simulation does not consider the inclusion property (*i.e.* allows a processor to access the lower-level caches during coherence actions on the higher-level cache), memory access characteristics obtained from such a simulation will be incorrect. The percentages of inclusion misses show the extent of the errors which would occur in a faulty simulation system.

5.7.3 Verification Results

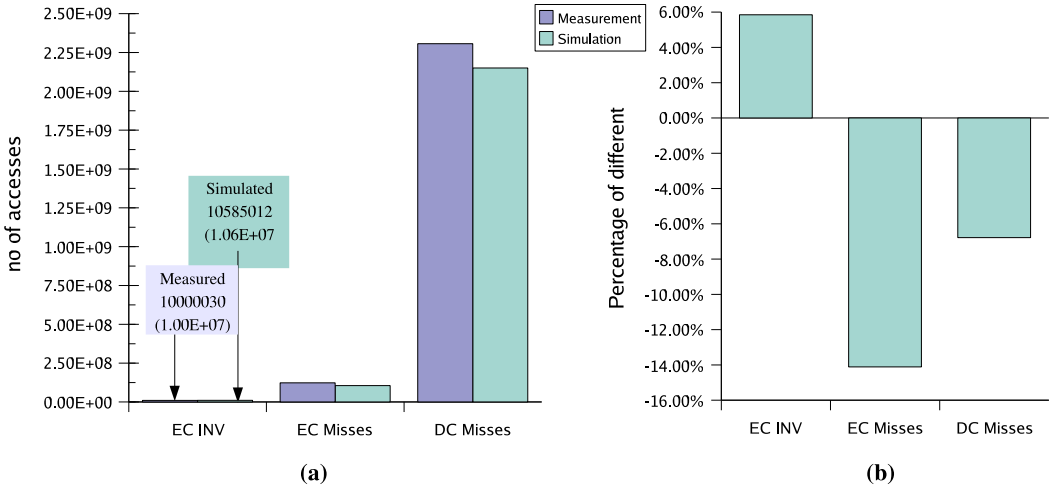


Figure 5.15: Comparison of simulation-measurement Results.

Figure 5.15 shows the comparison of simulation results against the results of measurement when running a particular function inside the LU program, called BUTS, which dominates the run time. Note that the measurement has been scoped at a particular function to make the results comparison feasible.

Figure 5.15 (a) shows the total number of invalidation and cache miss events occurring at the EC and Level 1 DC. Figure 5.15 (b) shows the percentage difference between the simulation results and the measurement results. Both the EC invalidation and DC Misses results show similarity between the simulation and the measurement results ($\pm 5-6\%$ in both cases). However, the EC misses show a noticeable difference (14.50 percentage points different). This is because of in the SunFire15K machine, EC is a unified cache (*i.e* it also accommodates instructions) while the simulation did not emulate instruction caching. Therefore the number of EC misses obtained during the simulation is far smaller. Nevertheless, as processors only read the content of instructions, they produced no invalidation events based on violation of cache update.

Table 5.4: Verification results of three protocols.

Protocol	**Safety**		**Liveness**		**Inclusion**	
	No.INV Global State	Critical State	Trapped State	Wait-Ack Matching Pairs	Percent EC INV	Percent MLI INV
Synapse	2	Dirty	None	14	13.02%	4.57%
Illinois	4	Dirty,VldExcl	None	18	12.84%	4.87%
MOESI	7	Mod,Excl	None	29	8.75%	5.17%

Table 5.4 summarises the obtained results corresponding to the assessment used in the verification process for the safety, liveness and inclusion properties. The protocol specifications have been examined to *prevent* erroneous cases for the safety and liveness properties as described earlier in Chapter 4. Moreover, in this experiment, the inclusion property is checked during each simulation run. To do so, the total number of events that invalidate a content of all the Level 2 caches (or External Cache: EC) are counted and confirmed with the total number of subsequent invalidations that occur in Level 1 (as the percentage shown in the last two columns in the table). All the simulation runs terminated successfully, and produced the same results as the measurements.

These two figures highlight the fact that the SPMI technique allows the DSIM-CLUSTER to reflect a correct projection of the simulation results with a very small fraction of time spent on specification parsing and erroneous cases detection. These results correspond to the measurement results obtained from a real machine with the difference of less than 10% in the data cache accesses.

5.8 Summary

In this chapter the implementation of the DSIMCLUSTER simulation model was described. The model was developed by integrating the framework components with the HASE++ DES engine, using the facilities provided by the HASE application. By doing so, the model structure and parameters can be reconfigured by the support facilities of the HASE infrastructure.

During each simulation run, the model reads a DSIMCLUSTER Workload Format (DWF) file that holds code, data, and some special instructions suitable for recreating the OpenMP parallel section and manipulating the multithreaded execution units. The DSM_Manager Entity assigns the workload to one copy of the OS Entity to create an execution environment for the workload. The OS Entity creates the memory segments prior to scheduling the workload's process to a processor selected from the available resource queue. The Processor Entity executes the workload by iterating a modified instruction cycle (*i.e.* fetch, decode, execute & store results, read the execution result, and perform service if requested). The decoding and execution are done by an instruction set object chosen as a parameter before running the simulation.

Prior to the decode step, the Processor Entity checks whether the instruction is a *magic instruction*, *i.e.* a command used to direct the Processor Entity to transfer control to the other simulation components. A magic instruction, `spawn`, is used to allow the OS Entity to emulate a multithreaded runtime environment during a simulation run. If such a magic instruction is used, the new threads will be created, linked, and allocated using the mechanisms implemented in the OS Entity. The status of the parent process/thread is kept, temporary results are put in the kernel stack, and the process/thread is halted waiting for all of the child threads to finish.

During the normal execution cycles, an implementation of a specification-based parameter-model interaction (SPMI) verification technique is used to simulate the bus-based coherence protocols, thus keeping the cache coherence. The hierarchical performance metrics used to obtain the workload execution profile have also been described.

A verification experiment of the DSIMCLUSTER simulation model against a Sun-Fire 15K machine has been presented. A workload of a particular function, BUTS, obtained from the LU decomposition program of the NPB 2.3 benchmark (class A)

has been run on both the real machine and the DSIMCLUSTER model with similar configurations. The number of cache misses obtained from both machines have been compared, resulting in a difference of less than $\pm 5-6\%$ in average, and 14.5% in the worst case at the L2 external cache. This result has confirmed that the model is working correctly, thus the model is ready to be used for some further performance evaluation experiments on a wider range of design parameters. In the next chapter, the expansion of the DSIMCLUSTER model to support the studies of the impact of data layout and protocol ownership on DSM locality will be presented.

Chapter 6

Preliminary Experiments

The verification experiment using a particular function of the LU decomposition workload has shown the accuracy of results obtained from the DSIMCLUSTER model. This chapter describes how the LU application workload was used in the analysis of two factors that may impact on memory locality. Since operations on array elements are common in many scientific codes, the impact of array memory layout and data access direction on memory locality was first analysed (Section 6.1). In this experiment, two workload files of the LU application were implemented in the `DEFAULT` instruction set format, each using a row-major and a column-major order memory layout. Experiment results obtained from running these workloads show that mis-arrangement of multidimensional array in memory may cause poor spatial locality, suggesting the use of compiler optimisation for minimising the effect. A DSIMCLUSTER workload file of the LU application from the NPB benchmark was obtained from the optimised object code generated on a SunFire 15K machine. Section 6.2 presents the characteristics of this workload on an SMP model of four processors. This workload was used in an experiment to study whether using the ownership-based technique in a bus-based coherence protocol impacts memory locality. Section 6.3 demonstrates this analysis and shows that, with a multiple readers access pattern, the ownership-based protocols achieve higher degree of locality than their non-ownership counterparts. Performance of the simulation model itself has been analysed (Section 6.4), in particular the sensitivity of the simulation run time and the size of event queues to the number of experimental factors. In Section 6.5, the content presented in this chapter is summarised.

6.1 Analysis of Locality *v* Data Layout

Operations on array elements are used in many scientific codes and benchmarks. Multi-dimensional arrays are generally stored in a linear memory address space by compilers using the traditional matrix representation, also known as canonical data layouts [Lin et al., 2002]. Two-dimensional arrays are generally arranged in memory in row-major order (for C, Java[1] *etc.*) or column-major order (for FORTRAN). Compilers map array indices into the linear address space in order to access array elements. For example, if there is a two-dimensional array, $A[M_{row}][N_{col}]$, the offset of memory address of an element $A[i][j]$ to the beginning of the array A can be derived from the row-major (RM) data layout function [Cierniak and Li, 1995],

$L_{RM}(i, j; M_{row}, N_{col}) = (i \times N_{col} + j) \times Element\_Size$ (Bytes)

or the column-major (CM) data layout function

$L_{CM}(i, j; M_{row}, N_{col}) = (j \times M_{row} + i) \times Element\_Size$ (Bytes).

Traversing an array in the direction that contradicts the layout order (*e.g.* traversing a row-major array in column-major order) leads to poor spatial locality. The performance loss in accessing large arrays can be a factor of 10 or more [Thiyagalingam and Kelly, 2002]. The cost can be more noticeable for higher dimensional arrays [Lin et al., 2002]. To alleviate this, a number of multidimensional array representations have been proposed along with a number of compiler optimisation techniques for loop and data layout transformation. These studies have shown that the ways array elements are laid out and accessed impact on memory locality. Therefore, in the first preliminary experiment, the DSIMCLUSTER model was tested by using it as an experimental platform to study the extent to which different data layout schemes affect the overhead of memory locality.

6.1.1 Experimental Methodology

To carry out these tests, the OpenMP LU-decomposition program was selected. The workload comprises a computation on three 128×128 matrices and was written in the `DEFAULT` instruction set used in the DSIMCLUSTER model. The experimental factor is the data layout that can be configured (a) in row-major form (b) in column-

[1] A particular interface known as Java BLAS (Basic Linear Algebra Subprograms) is used to support the column-major order in Java.

major form. In this experiment, the full-factorial design with six replications [Jain, 1991] was used, thus involving a total of 12 simulation runs (as shown in Table 6.1). On each simulation run, the processor IDs were randomly placed in the available resource queue. Table 6.1 also shows the sequence of processor IDs recorded from each simulation run.

Table 6.1: Experiment design for data layout study.

Set	Data Layout	List of Available Processor IDs					
		R1[a]	R2	R3	R4	R5	R6
1	Row-major	0,1,2,3	2,0,1,3	3,2,1,0	1,3,0,2	2,1,3,0	0,2,3,1
2	Column-major[b]	0,1,2,3	2,0,1,3	3,2,1,0	1,3,0,2	2,1,3,0	0,2,3,1

[a] R=Experiment Run

[b] The same order of available processor IDs is used for the column-major runs instead of using the random number.

Model size. The simulation is based on a 4-node SMP model (*i.e.* the 1×4 DSM) using a fully-associative single-level cache with the following features: a line size of 16 bytes; a write-invalidate protocol; a copy-back policy with write allocation. Finally, the cache is assumed to be large enough to accommodate at least the maximum number of elements dictated by the matrix specified in the experiment boundary. The latter implies that there will be no cache replacements.

6.1.2 Experiment Results

Figure 6.1 shows the memory access latency of the four processors (`P0` to `P3`) obtained in two different ways: (a) the expected latency calculated from the pattern of accesses to the cache recorded in the simulation trace file (b) the actual latency recorded in terms of simulation clock cycles. To compute the expected latency, the trace of memory accesses to caches (*i.e.* the records of all hits and misses) were multiplied by the average time penalty spent for each action. The time penalty of one and four clock cycles were used as the average latency of a cache hit and miss, respectively. Thus, the expected latency does not represent any delay caused by bus arbitration or cache coherence protocol.

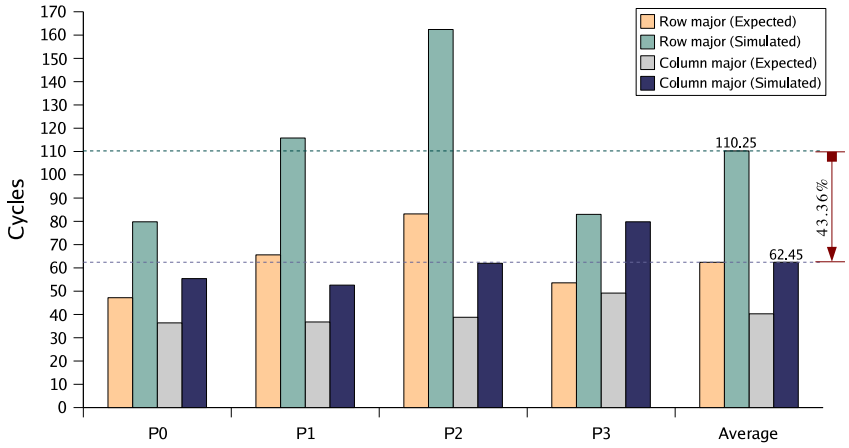


Figure 6.1: Memory access latency.

Both these results show that the data layout significantly affects the time to fetch the data. The difference between these two is caused by the overhead of coherence actions to handle false sharing accesses (occurring when `P1` and `P2` attempt to write data residing in the same cache line). The significant overhead is especially acute when using row-major layout at `P2`. The results show that using a column-major layout offers the reduction of average latency by 43.36 percentage points.

Figure 6.2 and 6.3 show the effect on cache misses and cache utilisation (*i.e.* the proportion of cached words that are actually accessed) when using the two different layouts. In essence the two layouts have different memory locality characteristics, with the row-major layout causing a greater degree of cache bubble, *i.e.* words brought into the cache which are not actually used.

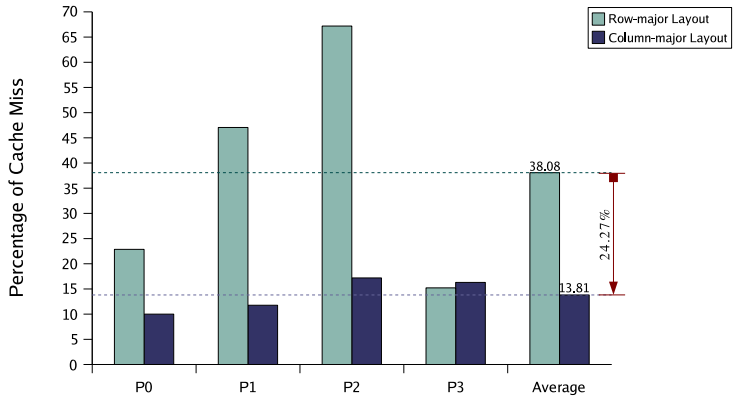


Figure 6.2: Percentage of cache misses.

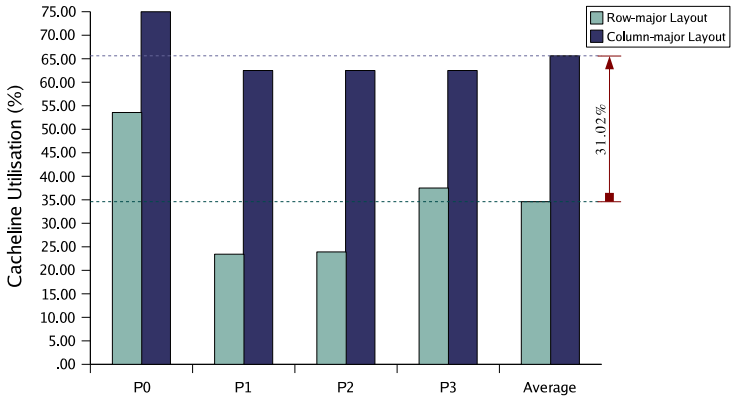


Figure 6.3: Percentage of cache utilisation.

The results show that using a column-major layout in the LU decomposition workload reduces the average cache miss rate from the row-major counterpart by 24.3 percentage points and also improves the cache utilisation by 31 percentage points. These two figures highlight the fact that a column-major data layout is a better choice than its row-major counterpart for this application. These results correspond to what would be expected from an analysis of the algorithm as the direction of data access is in column-major order. Mis-arrangement of multidimensional array in memory against the data access direction can cause poor spatial locality. In other words, the degree of locality is sensitive to the choice of memory alignment used by compilers. This finding suggested that to consider the impact of hardware alternatives and DSM policies on the memory locality, compiler optimised codes should be used to minimise the effects from data layouts. The results obtained from this experiment show that the model is working correctly. This gives confidence in configuring the DSIM-CLUSTER to run further performance evaluation experiments using a larger range of design parameters with the application workload from a scientific benchmark.

6.2 Workload Characterisation

In this work, the NAS Parallel Benchmarks (NPB) version 2.3 [Jin et al., 1999] (OpenMP C version) was chosen. The benchmark is well known, and is one of the common benchmarks used for the performance study of multiprocessor systems [Gustafson and Todi, 1998]. The NPB 2.3 benchmark suite was derived from a set of aerospace

Table 6.2: Characterisation of the LU workload file (DWF) *v* the Sun Sparc object file.

Sun Sparc Object File		DSIMCLUSTER Workload File	
OpenMP Directives[a]	Assembly Routines	Magic Instruction	Description
Parallel and Dynamic Threads Creation			
parallel	__mt_MasterFunction_	SPAWN	create parallel threads
private(i)	__mt_MasterFunction_	SPAWN	pushes i to stack
master	__mt_MasterFunction_	CALL	with Master flag
single	__mt_single_begin_	SPAWN	noOfThreads = 1
Work Sharing Directives			
for	__mt_WorkSharing_	using array index distribution	
schedule(static)	__mt_WorkSharing_	defined at the data segment	
nowait	__mt_WorkSharing_	SPAWN	with the 'nowait' flag
flush(flag)	__mt_WorkSharing_	FLUSH	write and notify all threads
Barrier and Synchronisation Directive			
critical	__mt_BeginCritSect	LOCK	lock and unlock
barrier	__mt_Barrier	BARRIER	wait for all threads

[a] all directives begin by #pragma omp

applications that simulate a class of calculations in computational fluid dynamics (CFD). The benchmark comprises five kernels and three simulated CFD applications. In the preliminary experiments which will be presented in the next section, a kernel benchmark called LU has been chosen. LU is a simulated CFD application that uses a symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretisation of the Navier-Stokes equations in 3-D by splitting it into block Lower and Upper triangular systems [Jin et al., 1999]. LU factorises the equation into lower and upper triangular systems. The systems are solved by the SSOR algorithm. In the NPB 2.3 OpenMP benchmark, the system has been implemented in the following iteration loop.

```
1     DO ISTEP=1,ITMAX
          CALL COMPUTE_RHS
3         CALL JACLD
          CALL BLTS
5         CALL JACU
          CALL BUTS
```

```
7           CALL ADD
        END DO
```

From the algorithm above, the blocked tridiagonal array is first calculated (RHS). Then the lower-triangular and diagonal systems are formed (JACLD) and solved (BLTS), followed by the upper-triangular system (JACU and BUTS). Lastly, the solution is updated. In solving the triangular systems, the solution at (i,j,k) depends on those at $(i+e, j, k)$, $(i, j+e, k)$ and $(i, j, k+e)$ where $e = -1$ for BLTS and $e = 1$ for BUTS. Nine different OpenMP directives have been used in the main program and in all of the subroutines listed above.

The LU workload was derived from the same method presented in the verification experiment in Chapter 5. Moreover, compiler optimisation is used to minimise the impact of memory layout and the direction of data access to memory locality overhead. Table 6.2 summarises the general information on the workload file.

6.2.1 Data Access Characterisation

The workload file obtained has been characterised by running it on a four-node SMP model. The SMP model has been chosen because it does not produce the overhead caused by data replication on to a remote node. Data access characterisation is done by analysing the addresses accessed. The trace of memory addresses that were accessed can be obtained in one of three ways: generated by a simulator, using hardware monitoring on real machine, or using the estimation produced by some advanced compilers [Grbic, 2003]. The choice of hardware monitor is not applicable for the SunFire 15K machine as, to the best of the author's knowledge, monitoring of accesses on a per-block basis is not supported by hardware counters of the UltraSparc III processors. In this work, the first method is used by extending the DSIMCLUSTER model to capture all effective addresses issued by each processor. The recording of accesses to effective addresses was done periodically, based on a reconfigurable sampling interval. Read and write accesses are classified. Figure 6.4 shows the plots of accesses from each of the four processors to the effective address space as a function of simulation time, sampling at every 100 processor cycles. A smaller interval is also possible, however, it may cause the file size to reach to the maximum limit defined at the host machine.

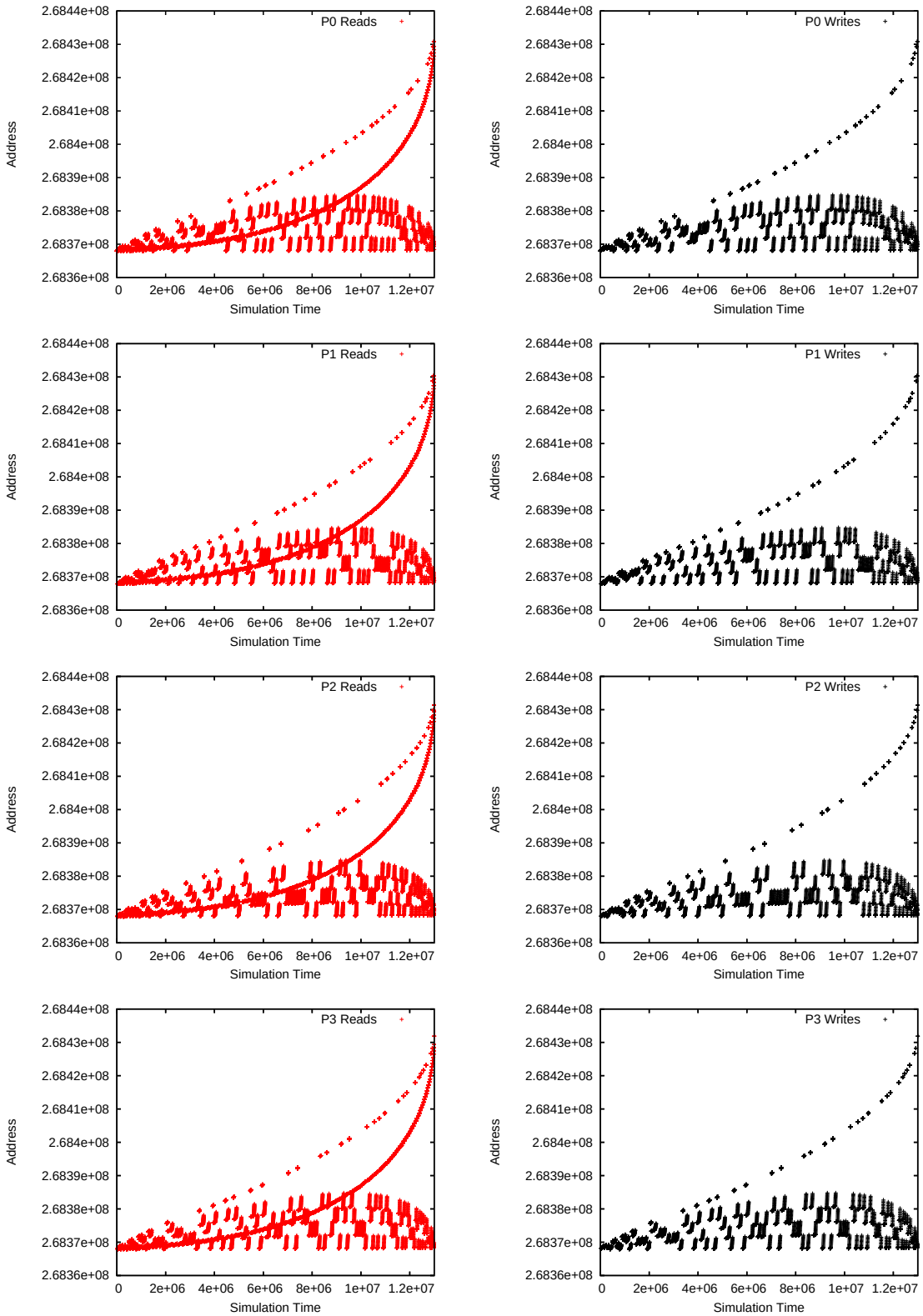


Figure 6.4: Data access pattern of the LU workload on four processors (P_0 to P_3).

Data accesses of LU as depicted in Figure 6.4 show similarity on both reads and writes among all four processors. A large number of accesses were surrounded in a shared data area which was allocated at the first data segment (lower memory address). Data accesses progress to the higher addresses (showed by two thin curves on both read and write panes) as a result of private data-segment allocation for each parallel section.

6.2.2 Data Access Patterns

To capture the patterns of data access, accesses to each effective address are classified by access type and the number of processors accessing the addresses during each sampling interval. These actions were classified into one of six patterns described below.

Single Reader (SR): The effective address was read multiple times by one processor. In addition, there were no write accesses to the address during the sampling period.

Multiple Readers (MR): The effective address was read by more than one processor. There were no write accesses to the address during the sampling period.

Single Writer (SW): The effective address was written to multiple times by one processor. There were no read accesses to the address during the sampling period.

Multiple Writers (MW): The effective address was written to by more than one processor. There were no read accesses to the address during the sampling period.

Single Reader Single Writer (SRSW): The effective address was read and written to multiple times by the same processor.

Multiple Readers Single Writer (MRSW): The effective address was read by many processors, but was written to by one processor which may or may not have been one of the readers. The numbers of reads and writes from the same processor are not related.

Single Reader Multiple Writers (SRMW): The effective address was read by one processor, but was written to by many processors which may or may not have

been the reader. The numbers of reads and writes from the same processor are not related.

Multiple Readers Multiple Writers (MRMW): The effective address was read by many processors, and was written to by many processors which may or may not have been among the readers. The numbers of reads and writes from the same processor are not related.

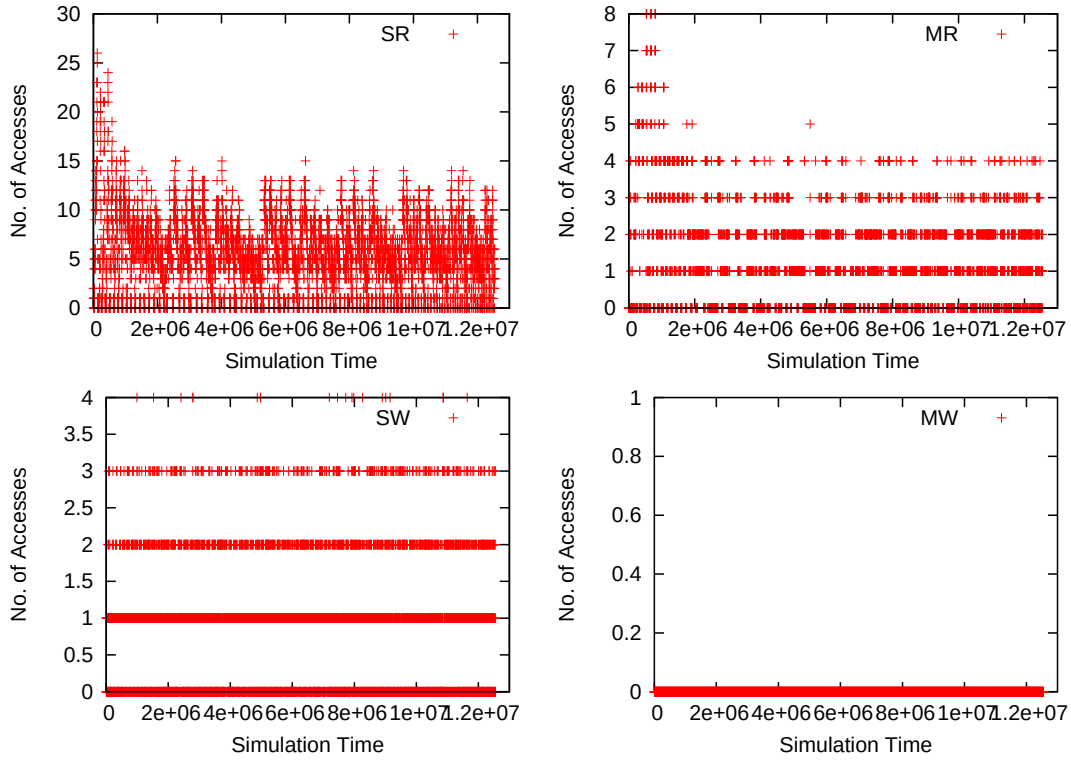


Figure 6.5: LU data access characterisation (SR, MR, SW, MW).

The number of effective memory addresses classified by each pattern were counted, and recorded into a plot file as a function of simulation time. These accesses were sampled at an interval of 100 simulated processor cycles. Figure 6.5 shows the plots of the number of effective addresses that were accessed in the SR, MR, SW, or MW patterns, respectively. In the figure, the large number of accesses in the SR plot demonstrates that between one and eighteen memory addresses were accessed by a single reader. The sharing of one to eight memory addresses among multiple readers in the MR plot was observed consistently across the simulation runtime. One to four

memory addresses were written by the same writers (shown in the SW plot) along the simulation runtime. No addresses fall into the multiple writers category observed from the LU workload.

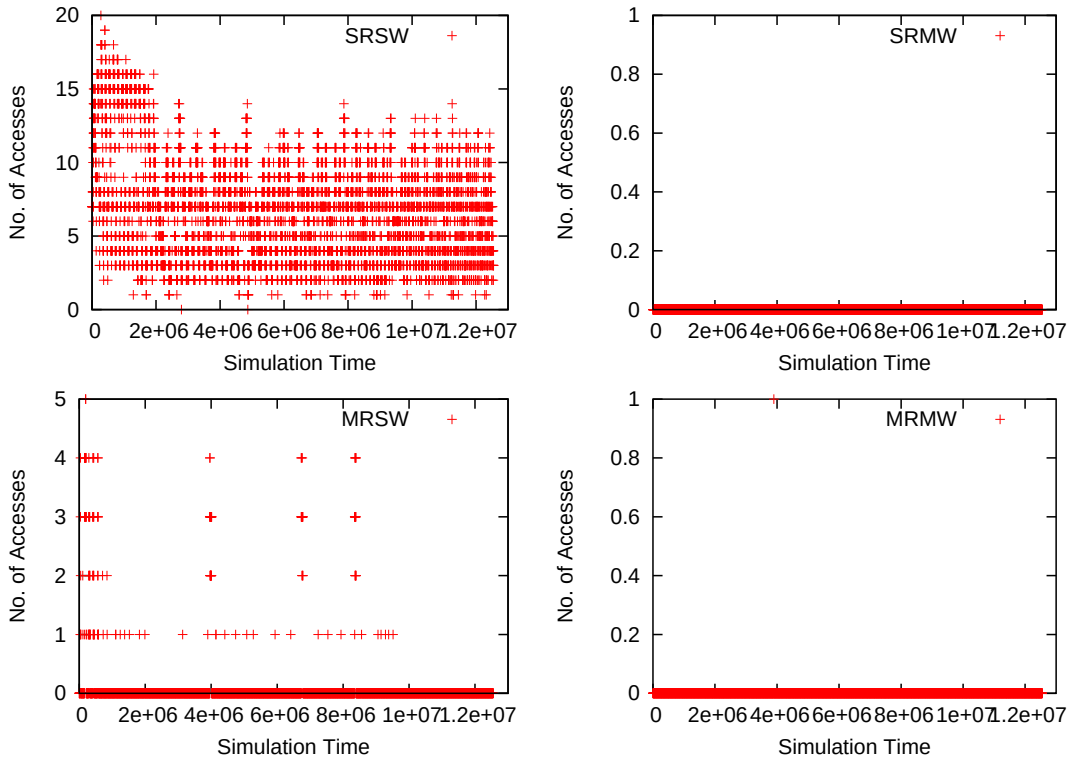


Figure 6.6: LU data access characterisation (SRSW, MRSW, SRMW, MRMW).

Figure 6.6 shows the records of the number of effective addresses that were requested for both read and write during the sampling interval. The figure shows the number of effective addresses that were accessed in the SRSW, SRMW, MRSW, or MRMW patterns, respectively. The large number of accesses in the SRSW plot shows that one to eighteen memory addresses were read and written to by the same processors (regardless of the sequence). This happened quite consistently across the application run time. Few addresses were accessed in the MRSW and MRMW patterns. There is no SRMW observed from this workload.

Remarks. It has been observed that the LU workload shows the reader-based characteristics (SR, MR, SRSW, SW) rather than writer-based. Read and write accesses progress in the blocked pattern, from a small block to the larger block size across the

application run time. Parallel threads use private data segments to keep some temporary data and the value of array indices assigned by the work sharing directive. The majority of memory accesses are to the shared memory segment which holds all array elements used for most of the computation. In the read-based access pattern, multiple cache replicas stay in a Shared Read state. If one of the replica has been modified, the source of update values (*e.g.* from memory or from other caches) plays an important role in relation to the degree of memory locality. It has been shown that invalidate-based protocols are a better choice for the reader-based access pattern since the cost of invalidate and write-through is less expensive than the broadcast update [Grbic, 2003] . However, many invalidate-based protocols use different schemes to identify the source of updated data value upon a read-miss request. Therefore, this issue is addressed in the second part of the analysis.

6.3 Locality *v* Coherence Protocol Ownership

In the second case study, the impact of using an ownership-based technique in the bus-based coherence protocol on memory locality is analysed. To do so, an experiment to evaluate locality effects on a 4-node SMP model using four different invalidate coherence protocols was conducted. The LU workload presented in the previous section was used. Therefore, the analysis is based on an application exhibiting the SR, SW, MR and SRSW data access patterns.

6.3.1 Experiment Methodology

In this experiment, four coherence protocols were divided into groups of two. The first group comprises the `Write Invalidate (WI)` and `Synapse (SNP)` protocols, which always deliver data from a shared memory when there is an access fault. These protocols are called *non-ownership* protocols. In contrast, the second group, called *ownership-based* protocols, uses another technique to handle access faults. In the case of an access fault, an ownership-based protocol will first identify which component *owns* the data and will ask the owner to deliver it. Two varieties of ownership-based protocols were chosen for this experiment, namely the `Illinois (IL)` and `Berkeley (BKLY)` protocols.

Table 6.3: Experimental factor and levels of the ownership study.

Factor	Levels
Coherence Protocol[a]	NO-WI, NO-SNP, OWN-IL, OWN-BKLY
OMP Threads	1, 2, 4

[a]NO=No-ownership, OWN=ownership-based

To evaluate the impact of the ownership technique on memory locality, four different configurations were set to include each of the chosen protocols in the DSIM-CLUSTER model. All of these configurations used the LU workload file described in the previous section. In order to avoid the bias from work-sharing characteristics in the workload, three different tests on each of these configurations were created. These three tests varied the work-sharing patterns by specifying different numbers of threads created per parallel section[2]. The tests include 1-thread, 2-thread and 4-thread, each of which allocates 1, 2, and 4 threads per parallel section to the workload. In this experiment, the 1-thread test generated 128 simulated parallel threads at runtime, while the 2-thread, and 4-thread tests generated 254, and 509 simulated parallel threads respectively. Once these tests had been set up, each of the tests was conducted in five replicas. Each replica was run as a single process randomly scheduled to one of the four simulated processors. Table 6.4 shows the experiment design for this test using the full-factorial with five replicas method. The experiment includes 12 run sets each of which comprises five replicas.

Model Configuration. The simulation is based on the 4-node SMP model using two-level caches. The primary cache is a two-way set-associative VIPT cache with a write-through policy. The line size is 16 bytes and the cache size is 1024 bytes. The secondary cache is a direct-mapped PIPT cache with a copy-back policy. In this case the line size is 64 bytes and the cache size is 4KB. Cache coherence is maintained at the secondary caches (Level-2 caches) using a shared coherence bus. One of the different varieties of coherence protocol was connected into the model at these secondary caches. Finally, the memory is assumed to be large enough to accommodate the maximum memory space requirement specified in the experiment

[2]In OpenMP, the number of threads per parallel section can be specified by three methods: (1) setting the OMP_NUM_THREADS environment variable (2) setting the number of threads at compile time and (3) specifying the number of threads in the code at the parallel section declaration.

Table 6.4: Experiment design for locality-ownership study.

Run set	Coherence Protocol[a]	OpenMP Threads	Order of processor IDs in the available queue				
			R1[b]	R2	R3	R4	R5
1	NO-WI	1	0, 1, 2, 3	1, 3, 0, 2	2, 1, 0, 3	3, 2, 0, 1	2, 0, 3, 1
2	NO-WI	2	2, 3, 0, 1	0, 3, 2, 1	0, 1, 3, 2	3, 0, 1, 2	1, 2, 3, 0
3	NO-WI	4	0, 2, 1, 3	3, 2, 0, 1	2, 1, 3, 0	2, 0, 3, 1	2, 1, 0, 3
4	NO-SNP	1	1, 2, 0, 3	1, 3, 2, 0	2, 3, 0, 1	0, 1, 2, 3	3, 0, 2, 1
5	NO-SNP	2	1, 3, 0, 2	2, 3, 1, 0	1, 2, 0, 3	0, 3, 1, 2	3, 0, 2, 1
6	NO-SNP	4	2, 1, 0, 3	1, 2, 0, 3	3, 0, 1, 2	0, 2, 1, 3	1, 2, 3, 0
7	OWN-IL	1	3, 2, 0, 1	3, 2, 1, 0	1, 3, 0, 2	2, 1, 0, 3	0, 1, 3, 2
8	OWN-IL	2	0, 3, 2, 1	1, 3, 2, 0	2, 0, 1, 3	0, 1, 3, 2	3, 0, 1, 2
9	OWN-IL	4	3, 0, 1, 2	2, 1, 0, 3	3, 2, 1, 0	1, 2, 0, 3	0, 2, 3, 1
10	OWN-BKLY	1	2, 0, 1, 3	0, 3, 1, 2	2, 3, 0, 1	1, 3, 2, 0	0, 1, 2, 3
11	OWN-BKLY	2	1, 2, 0, 3	1, 3, 0, 2	0, 3, 1, 2	2, 3, 1, 0	3, 0, 2, 1
12	OWN-BKLY	4	3, 0, 1, 2	1, 3, 0, 2	1, 2, 3, 0	2, 1, 3, 0	3, 1, 2, 0

[a]NO=No-ownership, OWN=ownership-based

[b]R=Run Number

boundary. The latter implies that there will be no paging due to the limitation of the physical memory.

6.3.2 Experiment Results

Firstly, the impact of protocol ownership on cache utilisation was analysed by cache miss percentage. Figure 6.7 illustrates the preliminary analysis as a histogram showing the cache miss percentages on each cache level and the percentage of inclusion misses[3]. The results obtained show a similarity of cache miss percentages regardless of the ownership technique. To confirm this, the Paired T-Test method was used to analyse the statistical correlation of data between two groups. The statistics obtained are presented in Appendix D, Section D.1. The statistics show that across all 4 processors, cache miss percentages increased by between 0.5 and 1 percentage point on average if using ownership identification. However, the caches clearly do not change the inclusion miss property over different tests; on average, there was a drop of only 0.01 percentage points when using ownership identification. The standard deviations reveal that level-2 caches were more variable with respect to cache miss percentage

[3]A cache miss is defined as an *inclusion miss* if the miss occurs at an invalid cache line in level-1 and the invalidation was forced by invalidation of the corresponding line in the level-2 cache

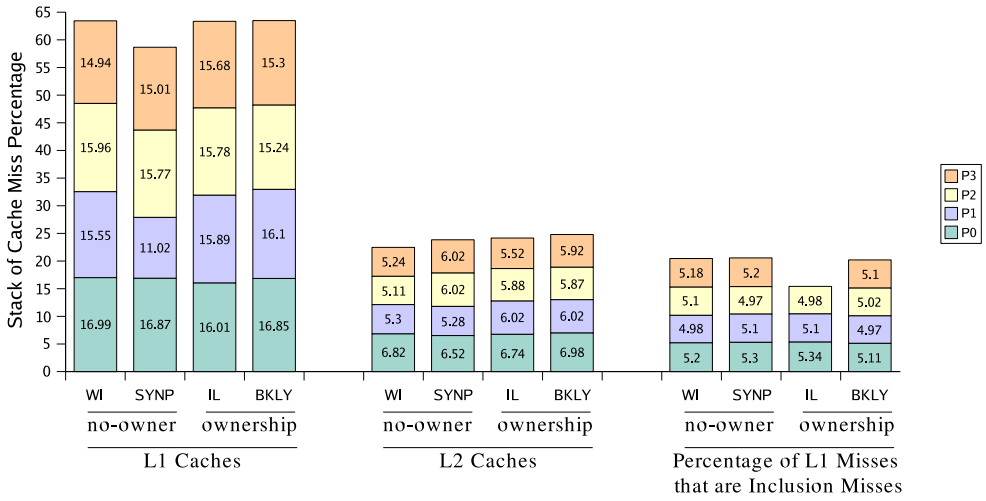


Figure 6.7: Distribution of cache miss percentage.

than level-1 caches. Moreover, the significance value greater than 0.10^4 for changes of cache miss percentage of all categories shows that using ownership technique in the cache coherence protocol did not significantly affect the cache miss pattern. Thus, in this case, it can be concluded that the ownership technique does not affect cache utilisation.

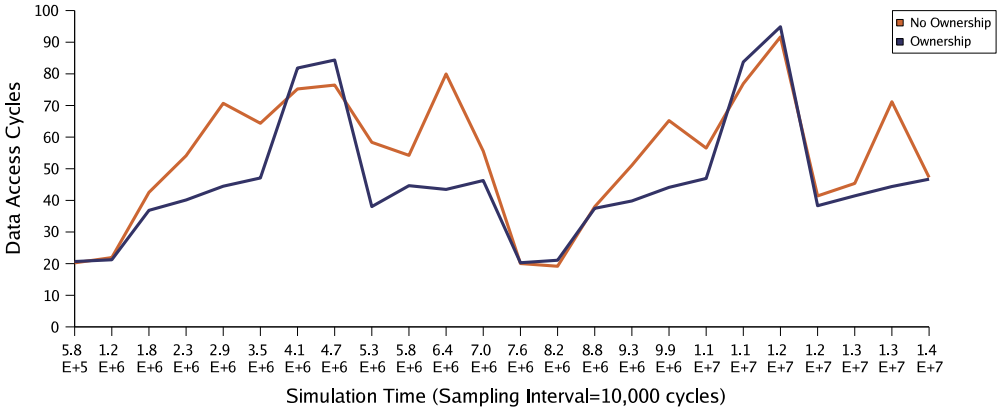


Figure 6.8: Average data access cycles against simulation time.

The second analysis focused on the overhead caused by the location of the data source, as shown by the average data access latency. Figures 6.8 and 6.9 shows the

[4]The t-values are 0.718, -1.022, and 0.260 with the resulting significance values from 2-thread test being 0.496, 0.341, 0.802 for the data obtained from level-1 caches, level-2 caches and inclusion miss percentage respectively.

average data access latency[5] using the two groups of protocols. The first chart illustrates a trace of the average latency recorded during the simulation run. This plot shows that the system with ownership protocols generally generated lower latency than the non-ownership counterpart. Figure 6.9 shows a histogram plotting the distribution of relative frequency against a range of latencies per simulated thread. To analyse this, the Paired T-Test method is also used for identifying the statistical significance of the results. The T-Test statistics show that across all 4 processors, data access latency was reduced by 17.76 and 28.86 percentage points on average if using the ownership-based protocols on 2- and 4- thread tests respectively. The standard deviations reveal that results obtained from the 4-thread tests were more variable with respect to latency than the 2-thread test results.

The results of the 4-thread test show a very high correlation of data access cycles between the systems with ownership-based and non-ownership based protocols[6]. Unlike the 2-thread test, in the 4-thread test, all processors saw reduced latencies and did so quite consistently. Since the significance value for change in latencies (testing at 95 percent confidence interval) is far less than 0.05 on both tests[7], it can be concluded that the average latency reduction of 17.76 and 28.86 percentage points per each processor is not due to chance variation, and can be attributed to the ownership technique used in the coherence protocols. The higher the number of threads per parallel section, the more consistently this improvement is achieved.

6.3.3 Discussion

LU decomposition is a well-known problem that represents a reader-based access pattern with a coarse grain spatial access pattern. Accessing data in a reader-based pattern implies that most of the time there is only one processor writing to a shared line (*i.e.* most of the time, there is only one *dirty copy* among the shared copies). In this situation, if using ownership, data is more likely to be delivered from one of the neighbouring caches owning the dirty copy than from the main memory. Consequently, the requester saves latency when obtaining data. Therefore it is hypothesised that with a

[5] From now on the *data access latency* will simply be referred to as *latency*.

[6] The correlation at -0.807, the absolute value nearly 1

[7] The significance values of 0.008 and 0.003 for 2-thread and 4-thread tests respectively

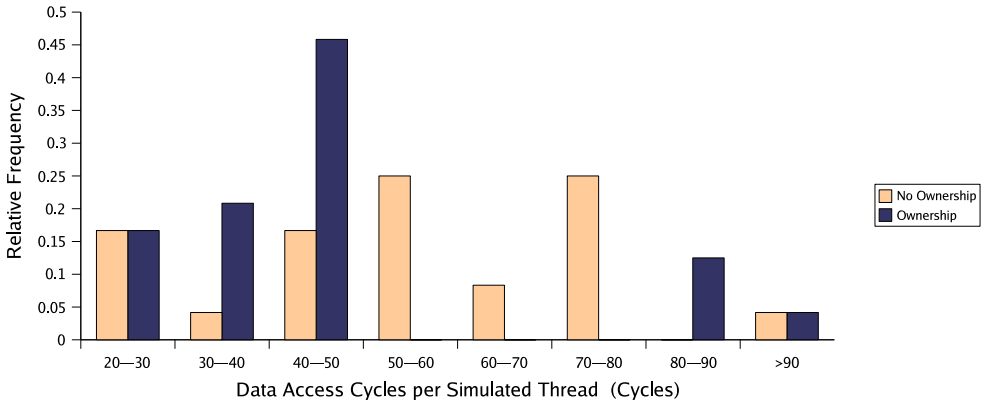


Figure 6.9: Relative frequency of data access cycle per simulated thread.

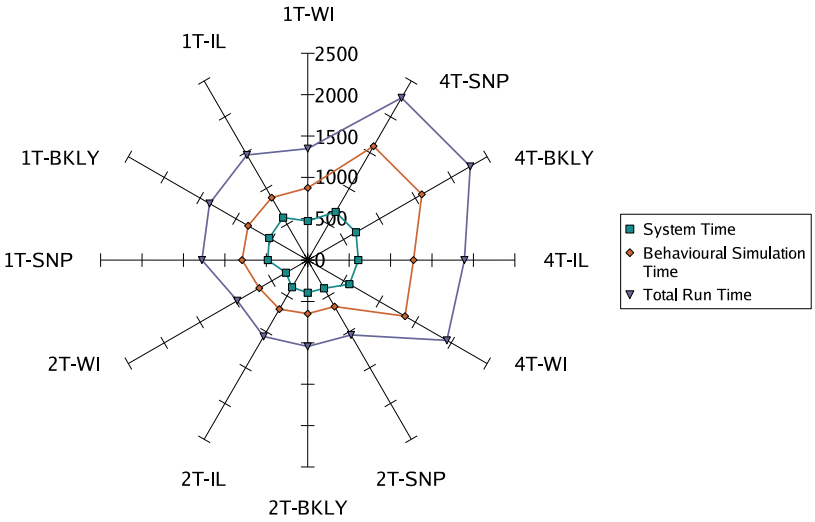
reader-based workload, ownership-based protocols should outperform non-ownership protocols, but should show no statical significance of difference in cache access pattern. Statistical analysis of the results has confirmed that this hypothesis is true. Besides, the impact is more consistent when there are more threads per parallel section (*i.e.* larger working group), which implies more read-shared copies.

This second case study has shown that DSIMCLUSTER is able to model a hierarchy of experimental factors using a parameter object interface. The results obtained give the confidence in the accuracy of the behavioural emulation based on the functional description of the components. In the next section, the evaluation of the DSIMCLUSTER model as a platform to conduct these experiments is discussed.

6.4 Simulation Model Evaluation

Using the run profile obtained from the previous experiment, the performance of DSIMCLUSTER can be evaluated using two criteria (a) the time to conduct an experiment (b) the resources consumed during a simulation run. Using the first criterion, the experimental times have been measured during the following three stages: customisation, reconfiguration, and running the simulation. In the first stage, the *customisation time* represents the time spent by the user modifying the DSIMCLUSTER to model the target architecture. In the second stage, the *reconfiguration time* represents the time spent by the user in setting up different design parameters for each test. In the last stage, the *time to run* is the elapsed wall-clock time that is required to simulate

a workload. The second criterion, the resources consumed, gives another perspective on the model performance. This criterion has been quantified by using the length of both the `Future` and `Deferred` event queues against the number of events processed per simulation cycle. This queue length reflects the size of memory reserved for maintaining the event management that is essential in the DES engine.



The axis shows the time to run simulation in seconds.

Figure 6.10: Distribution of the simulation runtime for each experimental factor.

For the second experiment, the customisation and reconfiguration time were very small[8]. Therefore, this time has been discarded and only the time to run a simulation has been analysed. Figure 6.10 illustrates the distribution of times to run a simulation for each experimental factor used in this experiment. The time is broken down into (1) *system time* (*i.e.* the time spent on managing Entity Threads in the DES kernel), and (2) *behavioural simulation time* (*i.e.* the time spent to emulate the target system behaviour).

The graph clearly shows that the 4-thread test of every protocol took over twice as long compared with the time spent on experiments involving the 1-thread, and 2-thread counterparts. This is because the number of dynamic threads created on the

[8] Average customisation time measured from the time to generate HASE Entities' codes until the 1×4 SMP model was successfully built, was less than 30 seconds. Likewise, the average reconfiguration time measuring from when each Entity creates parameter objects according to the model parameters until the parameters were successfully built, was 2.5 seconds, and was consistent regardless of experimental factors.

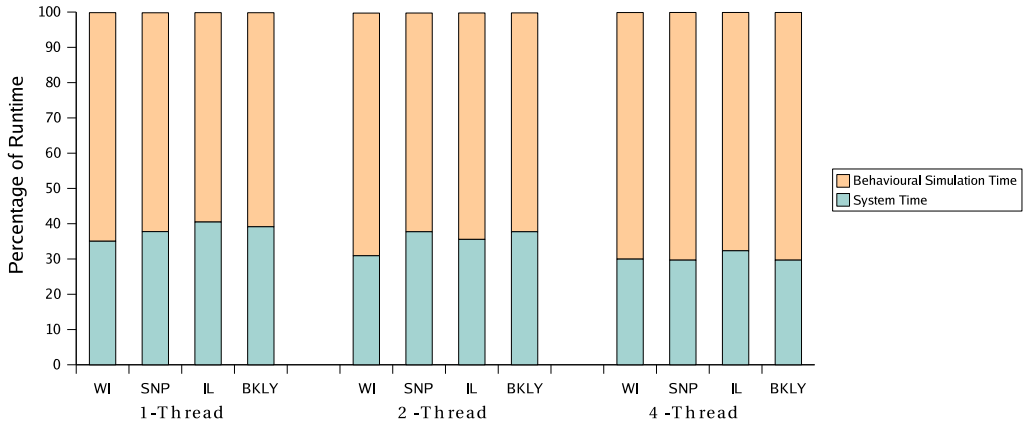


Figure 6.11: Proportion of runtime on the emulation of each experimental factor.

4-thread test consumed a large proportion of runtime to emulate the operating system functionalities. Despite that, the proportions of time spent on the DES engine and on the behavioural simulation did not vary so much across different experimental factors (as shown in Figure 6.11). This implies that in the DSIMCLUSTER model, the simulation runtime depends not only on how much work the simulator must carry out per memory reference, but also on the number of threads created for each parallel section in a workload. In addition, an average of 30–40 percentage of time is devoted to the mechanism at the DES kernel, regardless of the experimental factors.

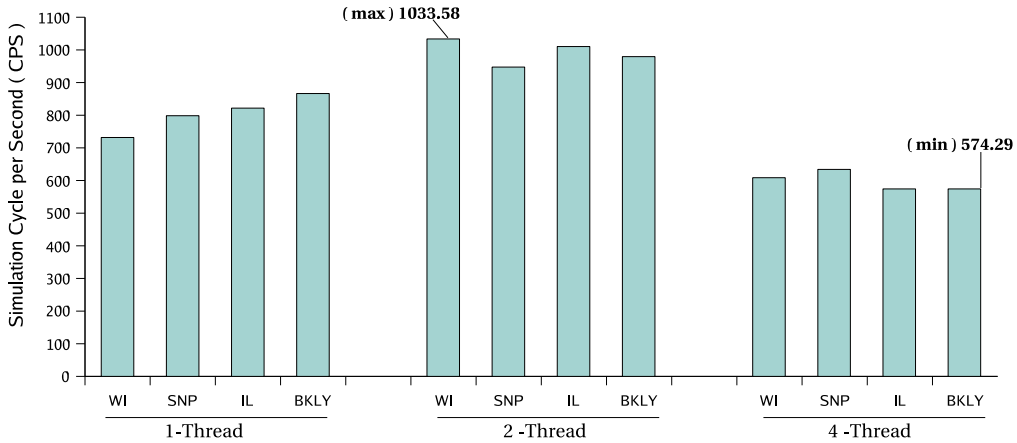


Figure 6.12: Effects of experimental factors on DSIMCLUSTER's performance.

Performance of the DSIMCLUSTER model in terms of runtime is described by the number of simulation cycles processed per second (CPS). The histogram in Figure

6.12 shows that from these experiments, a maximum performance of 1033.58 CPS on an Intel Xeon uniprocessor host at 1994.00 MHz CPU speed has been obtained. However, as shown in the histogram, when the number of simulated parallel threads is doubled, the CPS drops by an average of 50 percentage points. Therefore, the runtime slowdown is associated with the number of simulated parallel threads per workload.

The last evaluation criterion concerns the resources used in running an experiment on DSIMCLUSTER. Figure 6.13 shows a histogram illustrating the average number of events processed per simulation cycle, and the number that stay in the `Deferred` and `Future` queues obtained from using each experimental factor. The plot shows that if the number of simulated parallel threads is doubled, the total number of processed events tends to drop between 11–40 percentage points. In contrast, the number of events remaining in both queues tends to increase by 20–22.9 percentage points on average. Therefore, it can be concluded that a higher number of simulated parallel threads will caused DSIMCLUSTER to consume more memory. However this effect on memory is not linear and is less than the effect on simulation time.

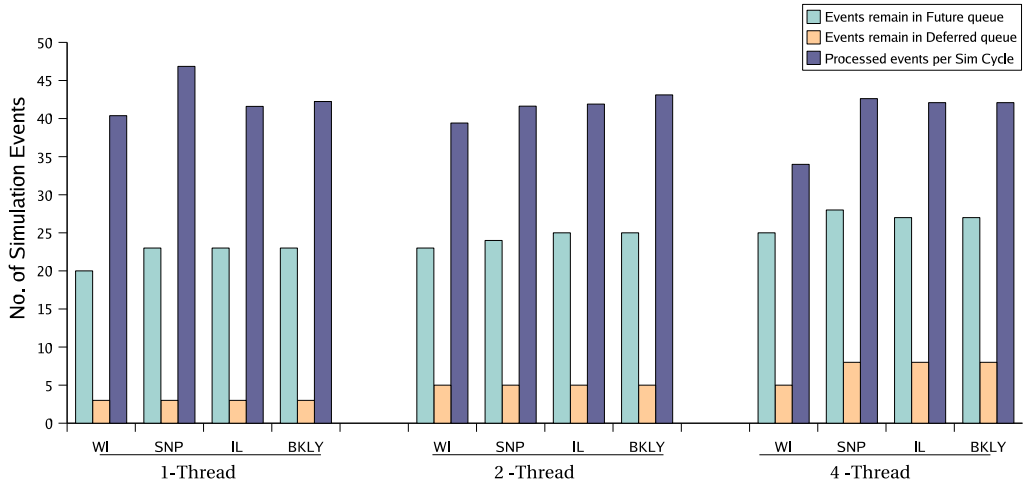


Figure 6.13: Event queue profile on each experimental factor.

Remarks. The two sets of preliminary experiments conducted at this stage have been based on a small scale SMP model with no interference from the interconnection network. The processing nodes have equal access to the shared bus. One objective of these preliminary experiments and the model evaluation was to verify the model behaviour and to cover the post-simulation results obtained. The characteristics of

the DSIMCLUSTER model are summarised as shown in Table 6.5. Note that the first three columns presented in this table are the same as those used in Table 3.1.

6.5 Summary

This chapter has introduced the framework of the DSIMCLUSTER model to assess the overhead caused by the reduction of memory locality. It consists of the data access characterisation and two preliminary experiments on memory locality analysis. At the beginning of this Chapter, an analysis of the impact of data layout on memory locality has been presented. This experiment shows the potential of DSIMCLUSTER to reflect the characteristics of memory at different levels. The results agree with existing literature showing that mis-arrangement of multidimensional arrays in memory may cause poor spatial locality, in this case it was showed by the reduction of cache utilisation. This finding suggested the use of compiler optimisation for minimising this effect.

The DSIMCLUSTER was extended further to support the LU application workload obtained from an optimised object file produced from the SunFire 15K machine. The workload has been characterised in both data access pattern and characteristics of memory performance using a 4-node SMP model. Data access patterns of the LU workload show a high frequency of read accesses, both in the single reader and multiple readers forms. Sharing data on read operations occurs quite constantly, and the sequence of read-writes from the same processors is common.

The analysis of impact of coherence protocol ownership on memory locality based on the multiple readers data access patterns has been demonstrated. The results show that the higher degree of locality, reflected by the average latency reduction of 17 to 28 percentage points per processor, can be attributed to the ownership technique used in the coherence protocols. The higher the number of parallel threads, the more consistently this improvement is achieved.

At the end of this chapter, the performance of the simulation model itself has been considered. The evaluation results are summarised based on the sensitivity of simulation performance to the various experiment factors. It was found that the performance of DSIMCLUSTER is especially susceptible to the execution dynamics of parallel workloads: the number of parallel threads created per each parallel section, data

access frequency and data sharing patterns. DSIMCLUSTER has gained model extensibility shown by the numbers of customisations and re-configurations for conducting experiments on different workload files, using different instruction sets, without having to recompile the source files. Based on its customisability, the DSIMCLUSTER has been extended to support further detailed analysis of memory locality problems. This analysis will be presented in the next chapter.

Table 6.5: Five features of DSIMCLUSTER model.

Feature	Characteristics		DSIMCLUSTER
	Technique	**Ways to integrate to a simulation**	
Workload Injection	HLL program	preprocess, instrument and compile	-
	Assembly code	instrument, assemble	√
	Executable file	instrument by binary code editing	-
	Trace	obtain by a tracer program	√[a]
Driving Technique	Direct Exec.	run a workload executable on the host	-
	Interpretation	translate or use application-driven	√
	Trace-driven	response to commands in trace	-[b]
Supports for Reconfi g-uration	Static Config.	have to modify the simulator code	for adding new counters
	Recompilation	modify the model desc. & recompile	for changing model architecture
	Library link	link with a new custom library	for customised instruction set
	Config File	set a config file before running a sim.	for caches, coherence protocols
	GUI menu	configure via the graphical interface	for other components
Profi ling and Performance Statistics	HW Counters	provide the fixed or selectable counters	fixed set of HW counters provided
	Perf.Metrics	calculate metrics (fixed or extensible)	extensible metric calculation module
	Run Profiles	get exe. profile of the workload's process	√
	Graphs or Plots	create graphs or plots	√
Verifi cation	Tracing	record logs of the simulated behaviour	√
	Specification	use specification-based verification	√ (for bus-based coherence protocols)
	Visualisation	verify module connections on screen	√ (HASE feature)
	Animation	verify sim. behaviour via animation	√ (HASE feature)

[a] Can be configured by using instruction set class

[b] Traces are used for verification purposes.

Chapter 7

Analysis of DSM Memory Locality

Chapter 6 showed evidence for increased degree of locality of the LU workload due to (a) matching the direction of array elements layout in RAM to the direction of data accesses and (b) using an ownership-based coherence protocol. In this chapter, the investigation is extended to find the answers to the two objectives stated in Section 1.2.1: (1) to determine which factors are most influential on the degree of locality, (2) to identify which tuning might lead to stability of performance. Two sets of experiments were conducted using three workloads (LU, CG, and FT) from the NPB benchmark. This chapter presents the study considerations, experimental methods, results, and discussion of the findings.

The chapter begins by describing the characteristics of the two workloads, CG and FT. In Section 7.2, three metrics used for observing memory performance of the workloads on different DSMs are presented: cache miss statistics, access latency, and degree of locality. In this section, several parameters that might contribute to the loss of memory locality are also stated. Since many parameters or factors can be contributors, a strategy of conducting experiments in two steps has been adopted. In the first step, a screening experiment was conducted using ten parameters chosen from the architectural design space, workload characteristics, and DSM management policies (Section 7.3). The results of the screening experiment, summarised by using the analysis-of-variance (ANOVA) statistical model, identified that the five most important factors are the DSM architecture, consistency technique, cache architecture, coherence unit and the number of threads created per parallel section.

In Section 7.4, the second step of the experiments, a full-factorial experiment

involving only two dominant factors, the DSM architecture and the consistency technique, is presented. Statistical analysis of the experiment results shows that locality gain exhibited in the three workloads, shown by the reduction of average latency, is more consistent on the DSM clusters composed of medium-scale SMPs (8 to 16 processors per node).

A high percentage of false access misses was found to be associated with the data invalidation technique used in the DSM consistency policy, suggesting the advantage of employing write update for the multiple readers access pattern. In Section 7.5, a conclusion on the experiment results and findings is given.

7.1 Workload Characterisation

Two kernel codes of the NPB benchmark were chosen for further experiments on the memory locality analysis. These comprise the implementation of (a) a Conjugate Gradient method (CG) and (b) a 3-D fast Fourier Transform-based spectral method (FT). The reasons these two workloads were chosen are the lower complexity in function calls and the length of object files in comparison to the other applications. Despite that, extra functionalities to cover different OpenMP directives from those used in the LU workload have been included (listed in Table 7.1).

Table 7.1: OpenMP directives of the CG and FT workloads.

Features	Implementation in DSiMCluster
Calls to random functions	using a magic instruction recognised at the decoding stage to perform the function at the Processor Entity
#pragma omp for reduction(+:a,b)	accumulate the value of a and b on memory write; the SunFire compiler translates this instruction into read-modify-write operations inside a critical region

7.1.1 CG

Conjugate gradient method is an iterative technique to solve sparse linear systems. The CG workload uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. This kernel tests unstructured grid computations and communications by using a matrix with randomly

generated locations of entries [Jin et al., 1999]. As shown in Table 7.1, the calls to random functions have been simulated by using a magic instruction (RANDOM) implemented as part of the Processor Entity. The parallelisation of CG is mostly performed by the loops inside the conjugate gradient iteration loop (shown in line 4 of the pseudo code below). The conjugate gradient function (CONJ_GRAD) consists of one sparse-matrix vector product, two reduction sums and several vector updates. Norms are calculated (via reduction) after the iteration loop (line no. 5). The reduction[1] operation is translated into the read-modify-write operations inside a critical region. DSIMCLUSTER's LOCK and UNLOCK are used to control concurrent accesses to the critical region.

```
    CALL MAKEA
2   CALL CONJ_GRAD
    DO ITER=1, NITER
4       CALL CONJ_GRAD
    CALCULATE NORM (with OpenMP's reduction directive)
6   DO J=1, LASTCOL-FIRSTCOL
        CALCULATE X[J] FROM NORM AND Z[J]
8       END DO
    END DO
```

7.1.2 FT

FT contains the computational kernel of a 3-D fast Fourier Transform (FFT)-based spectral method. FT performs three one-dimensional (1-D) FFTs, one for each dimension. Benchmark FT performs the spectral method with first a 3-D fast Fourier transform and then the inverse in an iterative loop. In the FFT function, the arguments 1 and -1 (line 2 and 5 of the pseudo code below) identify the direction of computation on the tridimensional matrices (u0, u1), *i.e.* which dimension will be calculated first. The parallelisation of FT is performed at the outer-most loop of the tridimensional matrix calculation inside the FFT iteration loop. The parallel operations used in the LU and CG workloads cover those that are used in the FT computation.

[1] using the directive, #pragma omp for reduction(+:sum) private(d)

```
1    CALL SETUP
     CALL FFT(1)
3    DO ITER=1, NITER
         CALL EVOLVE
5        CALL FFT(-1)
         CALL CHECKSUM
7    END DO
```

The difficulty in translating the FT workload into a DSIMCLUSTER workload file is how to represent the C 'dcomplex' data type (*i.e.* a structure of two 'double' values). The data type is generated by compilers as consecutive double words in the workload's assembly form. However, in the DSIMCLUSTER workload file format, the declaration of data segments remain in a high-level language form. In the data segments of a DSIMCLUSTER workload file, the variables are declared along with their data types (*e.g.* double, int, float, long). To declare a variable of the 'dcomplex' type, an array of two 'double' elements is used. Thus, the tridimensional arrays of the dcomplex type were translated into four-dimensional arrays of the double type. For example, the original declaration of array u1 is as follow.

```
1    dcomplex u1[NZ][NY][NX];
```

The declaration of array u1 in the DSIMCLUSTER workload file is:

```
1    int NZ  _val{64}
     int NY  _val{64}
3    int NX  _val{64}
     double u1[NZ,NY,NX,2] [_none,_none,_none,_none]
```

7.1.3 Data Access Characterisation

Using the method described in Section 6.2.2, the data access pattern characterisation of CG and FT workloads is shown in Figures 7.1, 7.2, 7.3 and 7.4. The numbers of effective memory addresses classified by each pattern are counted, and recorded into a plot file as a function of simulation time. These accesses were sampled at an interval of 100 simulated processor cycles.

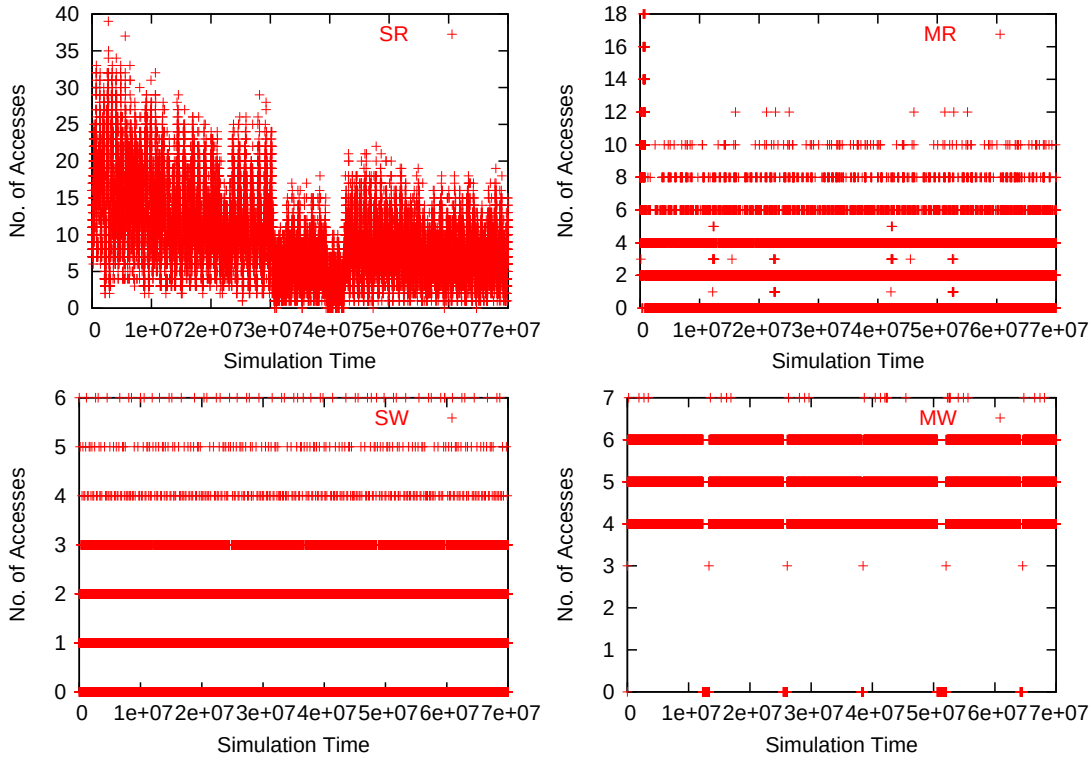


Figure 7.1: CG data access characterisation (SR, MR, SW, MW).

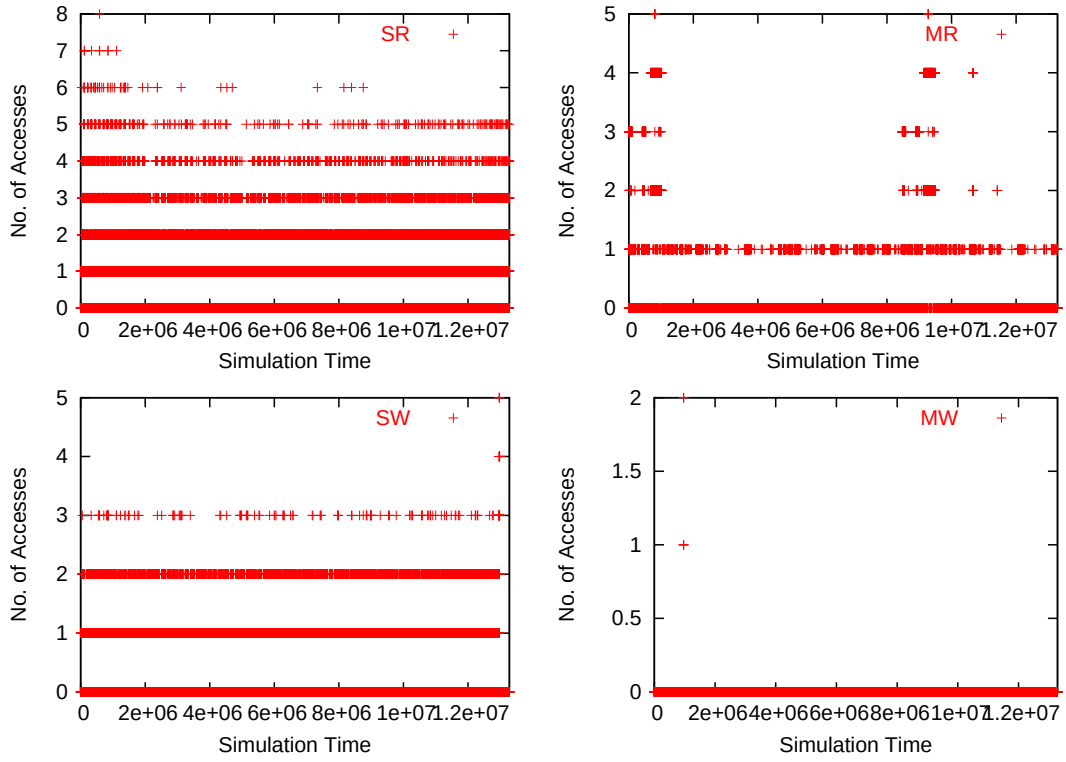


Figure 7.2: FT data access characterisation (SR, MR, SW, MW).

Figures 7.1 and 7.2 show the plots of the numbers of effective addresses that were accessed in the SR, MR, SW, or MW patterns of the CG and FT workload, respectively. In both workloads, the SR plots show the high numbers of memory addresses that were accessed by a single reader. The plots of the CG workload show higher numbers of addresses being accessed in all SR, MR, SW and MW patterns than that of the FT workload. This could be related to the different calculation patterns between the two workloads as well as the randomly generated location of data used in the CG codes.

The CG and FT workloads show the sharing of up to eighteen and five memory addresses, respectively among multiple readers in the MR plots. In both workloads, multiple-reader accesses were observed consistently across the simulation runtime. Up to six and five memory addresses were written by the same writers (shown in the SW plot) during the simulation runtime of the CG and FT workloads, respectively. Very few addresses fall into the multiple writers category observed from the FT workloads. However, higher numbers of addresses were accessed by multiple writers in the CG workload.

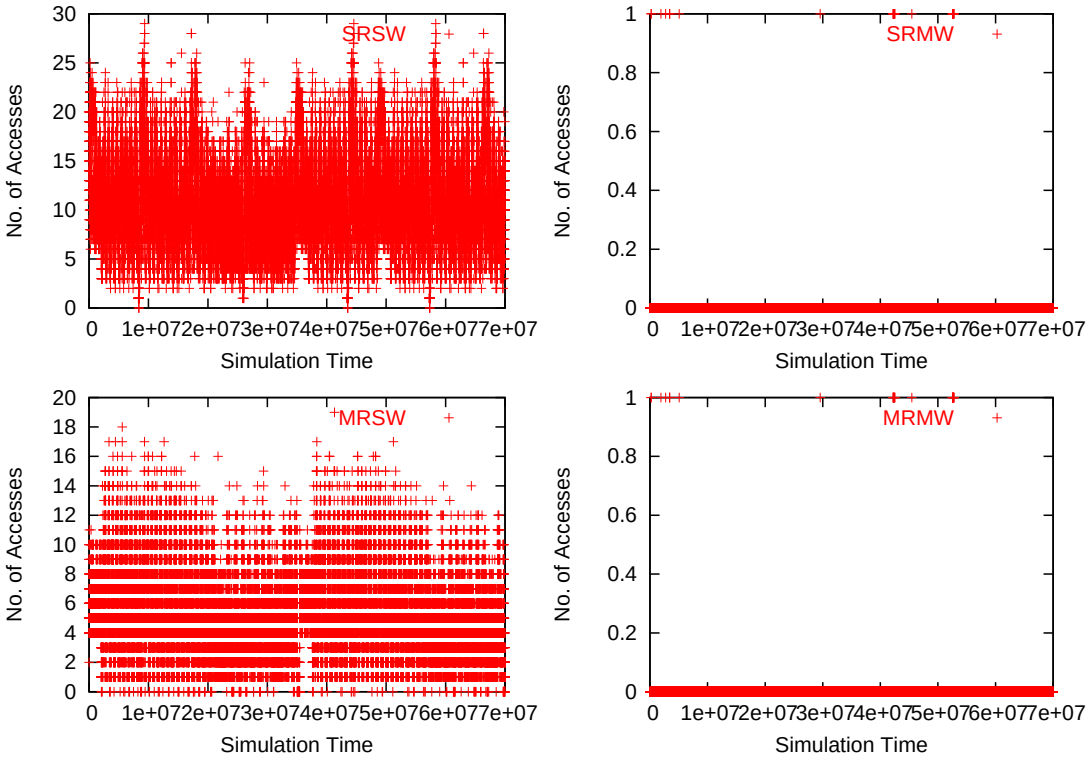


Figure 7.3: CG data access characterisation (SRSW, SRMW, MRSW, MRMW).

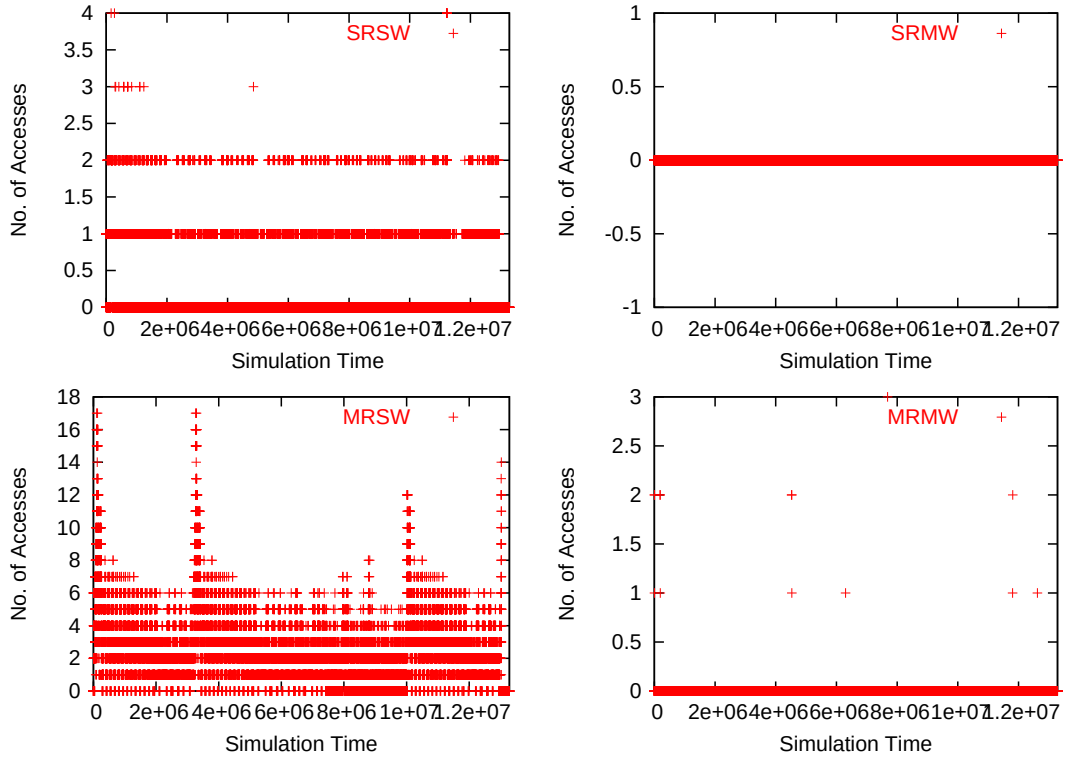


Figure 7.4: FT data access characterisation (SRSW, SRMW, MRSW, MRMW).

Figure 7.3 and 7.4 show the records of the numbers of effective addresses that were requested for both read and write during the sampling interval. The figures show the number of effective addresses that were accessed in the SRSW, SRMW, MRSW, or MRMW patterns, respectively. In contrast to LU (described in Section 6.2.2), both CG and FT workloads exhibit the high frequency of accesses to memory addresses in a multiple-reader single-writer pattern as shown in the MRSW plots of both figures. Up to twenty and eighteen addresses were shared among multiple readers, with one writer updating it, in CG and FT, respectively.

The large numbers of accesses in the SRSW plot of the CG workload show that 1 to 29 memory addresses were read and written to by the same processors (regardless of the sequence). This happened quite consistently across the application run time. In both workloads, few addresses were accessed in the MRMW patterns. There is no SRMW observed from the FT workload and very few SRMW accesses were observed from the CG workload.

Remarks. It has been observed that the CG and FT workloads show high frequencies of read-shared accesses (MRSW, SRSW, SR and MR) with single writer rather than the multiple writers accesses. In both workloads, parallel threads use the private data segments to keep some temporary data and the value of array indices assigned by the work-sharing directive. The majority of memory accesses are to the shared memory segment which holds all array elements used for most of the computation. The high frequency of shared accesses in SR, SRSW and MRSW patterns in the CG workloads caused the high data access latency in comparison to the FT workload. The next section considers the factors which impacted on the planning of experiment design to analyse memory locality effect in different DSM architectures.

7.2 Experiment Planning

7.2.1 Performance Metrics

There are two primary performance metrics normally used in the literature to describe memory performance: cache miss statistics and data access latency. The first describes how frequently memory accesses were not satisfied at primary caches. The increased number of cache misses may be an effect of the reduction of spatial locality. The second metric, data access latency, represents the amount of time required to read or write a given location in memory. When using simulation, a smaller latency (obtained from the same control environments running the same workload) shows the locality gain due to the behaviour of the observed experimental factors. The results obtained in Chapter 6 showed that the larger the number of parallel threads, the more consistently the locality yield can be seen. Moreover, in this work, the impact of different parameters on memory locality is also explained by the *degree of locality* recorded at each memory access during a simulation run. The following sections give the explanation of each of these metrics.

7.2.1.1 Cache miss statistics

The overall cache miss statistics can be divided into several categories according to the cause of miss as described in Chapter 2. In terms of performance, access misses belonging to the same category can cause different time penalties depending on both

the execution condition and the relationship between two consecutive misses. Hristea and Lenosky introduced a micro benchmark suite for measuring memory hierarchy performance in both uniprocessor optimisations and the contention and coherence effects of multiprocessors [Hristea et al., 1997]. From this research, cache misses have been further described by the relationship between two misses in the following terms.

- *in-isolation misses* are isolated in time from one another so that the fill time is less than the time to the next miss.
- *back-to-back misses* are occurrences of two consecutive misses (m_1 and m_2) with a minimal separation, and these two misses are *dependent* (*i.e.* the address of m_2 depends on the data returned by m_1).
- *pipelined misses* are occurrences of two consecutive misses that are independent and have minimal separation, so that performance is limited by whichever resource is a bottleneck during the cache fills.

The studies presented in this chapter refer to cache misses in terms of in-isolation misses as there is no outstanding transaction in any cache controllers. The processors are halted waiting for each access to complete before issuing a new command. Thus, the misses are isolated in time from one another.

7.2.1.2 Data Access Latency

Data access latency represents the amount of time required for a memory access to be completed, in the unit of CPU cycles or nanoseconds. Two different cases were used to measure memory access latencies on a real system in order to quantify contention and to balance memory load on hardware DSM multiprocessors [Nikolopoulos, 2003].

- *Back-to-back latency* is the non-overlapped latency[2] of a cache miss which must be served from main memory. The latency is taken by measuring from the time the cache miss occurs until the time the entire requested cache line is brought into the cache.

[2]Overlapped latency is memory access latency that is measured on processors that allow multiple outstanding memory accesses to overlap with the execution of computation.

- *Restart latency* is similar to back-to-back latency. The difference is that the restart latency is taken by measuring from the time the cache miss occurs until the time the requested word (instead of the entire cache line) is brought into the cache and the processor can restart execution. This latency is taken by measuring an in-isolation miss.

All experiments presented in this chapter measured the restart latency. The interval of memory access latency shows the difference between the time at which the requested data was returned to the processor and the time at which the request for this data was issued.

7.2.1.3 Degree of Locality

In this work, the term *degree of locality* has been introduced as a response variable describing the percentage of accesses to a DSM shared-memory region which are resolved at (a) local cache hierarchy (b) local memory and (c) remote memory. To capture the degree of locality, each memory access package issued from the processors contains an integer variable called `LocalityDegree` which is initialised to zero. Each component involved in supplying data for each memory accesses (*i.e.* the Cache, Memory, and OS Entity) increments the value of the `LocalityDegree` variable once the memory access package has arrived. Six basic degrees of locality are given below.

- Degree 1 (*Local Cache, Shared Clean*) represents the number of accesses satisfied by the local cache hierarchy while the cache lines were in a Shared Clean state.
- Degree 2 (*Local Cache, Dirty*) represents the number of accesses satisfied by the local cache hierarchy while the cache lines were in a Dirty state. Thus, actions to maintain memory coherence take place before the accesses can be completed.
- Degree 3 (*Local Memory, Shared Clean*) represents the number of accesses that were misses at the local caches, but which were satisfied at local memory when the data blocks were in Shared or Private Clean state. Thus, no coherence actions are involved before the accesses are completed.

- Degree 4 (*Local Memory, Dirty*) represents the number of accesses that were misses at the local caches, but which were satisfied at local memory when the data blocks were in a Dirty state. Thus, the data block must be updated before it can be accessed.
- Degree 5 (*Remote Memory, Shared Clean*) represents the number of accesses that were misses at both local caches and local memory as the data blocks were kept in a remote node. At the time of an access, the data blocks were in a Shared or Private Clean state. Thus, data replication or migration takes place before the access can be satisfied.
- Degree 6 (*Remote Memory, Dirty*) represents the number of accesses that were misses at both local caches and local memory as the data blocks were kept in a remote node. At the time of an access, the data block is in a Dirty state. Thus, the coherence actions to get the most updated value of the data blocks take place prior to data replication or migration to the local memory of the requested node. After that the access is satisfied at the node's local memory.

7.2.2 Parameters of Interest

Scheduling. For each simulation run, the `DSM_Manager` selected the first available SMP node in the system resource queue to accommodate the application process. The node's OS manages the process, allocates memory space, and schedules the process to the first available processor ID in the node's resource queue. The node's OS always tries to allocate all parallel threads within the node (1 thread per processor). If the number of parallel threads exceeds the number of processors in an SMP node, the OS will request the `DSM_Manager` to accommodate the remaining threads in the remote node(s).

DSM Manager. Memory pages are allocated to the workload process and replicated to a remote node on the first-touched basis. At the time of reads and writes to a critical region, the home-based and homeless consistency models work differently as described in Chapter 2. In both local read and local write operations in a home-based DSM system, a processor issues a request for an effective address whose home memory is on the same SMP node. Regardless of using the data update or data invalidate technique, all accesses are satisfied at local caches or local memory because the home

memory is always valid.

Cache Architecture and Coherence Protocols. Two models of cache architecture were included in this study. Both of the models comprise two level caches. The primary cache is a large VIPT data cache of 64KB divided into cache lines of 16 bytes size. The secondary cache is a 4MB PIPT data cache with 512 bytes cache line size. The L1 caches are direct-mapped caches with write-through policy. The L2 caches are fully-associative with copy-back caches. A four-state snoopy protocol, MESI, is used in both architectures. The content of secondary caches and memories are kept coherent among the SMP nodes using a directory protocol. In cache architecture A, the intra-node cache coherence is maintained at the secondary caches (L2 caches) using a shared coherence bus. However, in cache architecture B, the secondary cache is a shared cache. The snoopy coherence protocol is used to maintain the intra-node cache coherence at L1 caches.

7.2.3 Experimental Strategy

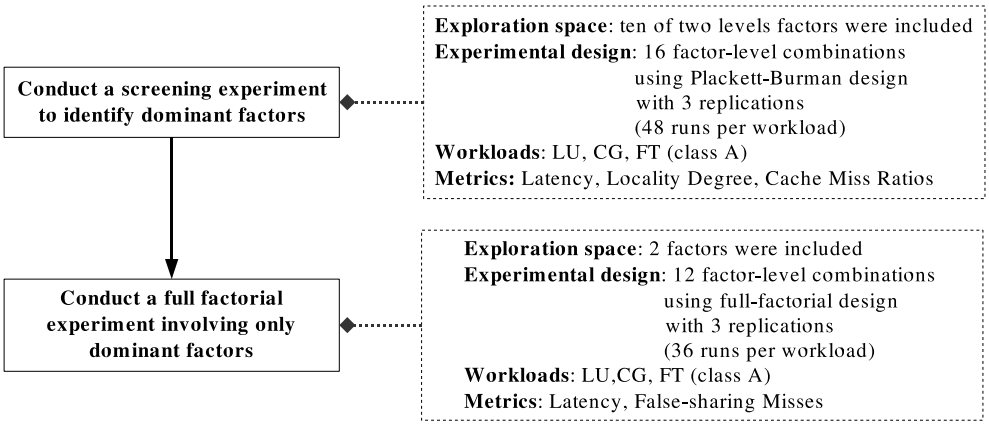


Figure 7.5: Experimental design of the memory locality study.

A strategy of conducting experiments in two steps was adopted to analyse memory locality against the large number of factors which can affect it. Figure 7.5 depicts the experimental design of the memory locality study. In the first step, a screening experiment was conducted to identify the small number of dominant factors out of the ten possible factors. The results obtained from the screening experiment were analysed using the statistical model to identify the dominant factor at 95% confidence

interval. After the dominant factors had been identified, the second step of the experiments was conducted using full-factorial design involving only the dominant factors. The following sections describe these two steps in the experiments, their results and the discussion of the findings.

7.3 Dominant factor identification

The goal of this experiment was to identify a small number of factors that most affect the memory locality problem. All parameters considered in this study, as presented in the previous section, were grouped into ten factors. These factors are listed, along with the levels chosen for consideration, in Table 7.2. Three applications used in this study are LU, CG and FT. Each SMP node has a two-level cache according to the configuration A and B as described above. The bus and interconnection clock rates are 150 MHz. Main memory, cache hierarchy, and processor clock frequencies are 133MHz, 300MHz and 900MHz, respectively. The latency of a page fault penalty, *i.e.* the cost of system calls to obtain the requested page, is 1000 processor cycles. Finally, the memory is assumed to be large enough to accommodate the maximum memory space requirement specified in the experiment boundary. The latter implies that there will be no paging due to the limitation of the physical memory.

Table 7.2: Experimental factor-levels of the memory locality study.

Factor		Levels	
No.	Name	-1	+1[a]
1.	DSM Architecture	4×8	16×2
2.	Cache Architecture	A	B
3.	Coherence Unit	cache line quarter(Q)	a cache line (C)
4.	Bus Architecture	Split-transaction (SS)	Transaction-Bus (TF)
5.	Snoopy Protocol	Update-based (UPD.)	Invalidate-based (INV.)
6.	Directory Protocol	Central directory (CD)	SCI
7.	Interconnection Routing	Store-and-forward	Wormhole
8.	Consistency Model	LRC	HLRC
9.	Consistency Technique	Update (UPD.)	Invalidate (INV.)
10.	OpenMP Threads	2	8

[a]The symbols '-1' and '+1' are the coded factor levels corresponding to the generator listed in Table 7.3

A sixteen-run Plackett-Burman screening design was used [Mason et al., 2003]. Six more runs than the number of factors is chosen, following the Mason *et al.*'s recommendation, to avoid the statistical errors of too few test runs. The coded factor levels of the Plackett-Burman design generator are shown in Table 7.3. The first row in Table 7.3 lists the Plackett-Burman design generator of sixteen test runs. The elements in each column are the coded factor levels: a minus sign denotes one level of a factor and a plus sign denotes another level (corresponding to the values listed in Table 7.2). The construction of a sixteen-run design is done by continuously performing a "rotate-left" of the coded factor levels to obtain the succeeding rows of the fifteen test runs. Then, a final row of minus signs is added. As recommended by Mason *et al.*, this experiment used six fewer factors than test runs, so five columns of the generated design are discarded. Only the highlighted columns are used to construct the experiment.

Table 7.3: Screening experiments generator.

Run	Experimental Factor ID														
No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	+1	-1	-1	-1	+1	-1	-1	+1	+1	-1	+1	-1	+1	+1	+1
2	-1	-1	-1	+1	-1	-1	+1	+1	-1	+1	-1	+1	+1	+1	+1
3	-1	-1	+1	-1	-1	+1	+1	-1	+1	-1	+1	+1	+1	+1	-1
4	-1	+1	-1	-1	+1	+1	-1	+1	-1	+1	+1	+1	+1	-1	-1
5	+1	-1	-1	+1	+1	-1	+1	-1	+1	+1	+1	+1	-1	-1	-1
6	-1	-1	+1	+1	-1	+1	-1	+1	+1	+1	+1	-1	-1	-1	+1
7	-1	+1	+1	-1	+1	-1	+1	+1	+1	+1	-1	-1	-1	+1	-1
8	+1	+1	-1	+1	-1	+1	+1	+1	+1	-1	-1	-1	+1	-1	-1
9	+1	-1	+1	-1	+1	+1	+1	+1	-1	-1	-1	+1	-1	-1	+1
10	-1	+1	-1	+1	+1	+1	+1	-1	-1	-1	+1	-1	-1	+1	+1
11	+1	-1	+1	+1	+1	+1	-1	-1	-1	+1	-1	-1	+1	+1	-1
12	-1	+1	+1	+1	+1	-1	-1	-1	+1	-1	-1	+1	+1	-1	+1
13	+1	+1	+1	+1	-1	-1	-1	+1	-1	-1	+1	+1	-1	+1	-1
14	+1	+1	+1	-1	-1	-1	+1	-1	-1	+1	+1	-1	+1	-1	+1
15	+1	+1	-1	-1	-1	+1	-1	-1	+1	+1	-1	+1	-1	+1	+1
16	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Table 7.4 shows the decoded combinations of factor levels obtained from replacing the coded factor levels listed in Table 7.3 with the actual values presented in Table 7.2. These sixteen combinations were used to configure the DSIMCLUSTER model for running three workloads: LU, CG, and FT. Three replications of each run

was conducted. At each replication, the processor IDs were randomly placed in the available resource queue. To minimise the possibility of bias effects due to the run order, the experimental test sequence was randomised. Three performance metrics (response variables) were captured: data access latency, cache miss characteristics, and the degree of locality.

Table 7.4: Screening experiments for the memory locality study.

Factor	Experimental Factor									
Level	1	2	3	4	5	6	7	8	9	10
Combi-	DSM	Cache	Coh.	Bus	Snpy	Dir.	Inter-	Consis.	Consis.	OMP
nation	Arch.	Arch.	Unit	Arch.	Ptcl.	Ptcl.	connect	Model	Tech.	Thrd
1	16×2	A	Q	SS	INV	CD	Store.	HLRC	INV	2
2	4×8	A	Q	TF	UPD	CD	Worm.	HLRC	UPD	8
3	4×8	A	C	SS	UPD	SCI	Worm.	LRC	INV	2
4	4×8	B	Q	SS	INV	SCI	Store.	HLRC	UPD	8
5	16×2	A	Q	TF	INV	CD	Worm.	LRC	INV	8
6	4×8	A	C	TF	UPD	SCI	Store.	HLRC	INV	8
7	4×8	B	C	SS	INV	CD	Worm.	HLRC	INV	8
8	16×2	B	Q	TF	UPD	SCI	Worm.	HLRC	INV	2
9	16×2	A	C	SS	INV	SCI	Worm.	HLRC	UPD	2
10	4×8	B	Q	TF	INV	SCI	Worm.	LRC	UPD	2
11	16×2	A	C	TF	INV	SCI	Store.	LRC	UPD	8
12	4×8	B	C	TF	INV	CD	Store.	LRC	INV	2
13	16×2	B	C	TF	UPD	CD	Store.	HLRC	UPD	2
14	16×2	B	C	SS	UPD	CD	Worm.	LRC	UPD	8
15	16×2	B	Q	SS	UPD	SCI	Store.	LRC	INV	8
16	4×8	A	Q	SS	UPD	CD	Store.	LRC	UPD	2

7.3.1 Experimental Results

Figure 7.6 shows the average data access latency against each factor-level combination obtained from each simulation run of the LU, CG and FT workloads. The lines in the three plots shows the average latency among the three runs. Higher latencies were observed at the CG workloads in comparison with the LU and FT workloads, due to the higher percentage of data accesses, as depicted in the plots of data access characterisations in Section 7.1.3. Generally, the latencies of CG workloads are quite consistent across all factor-level combinations.

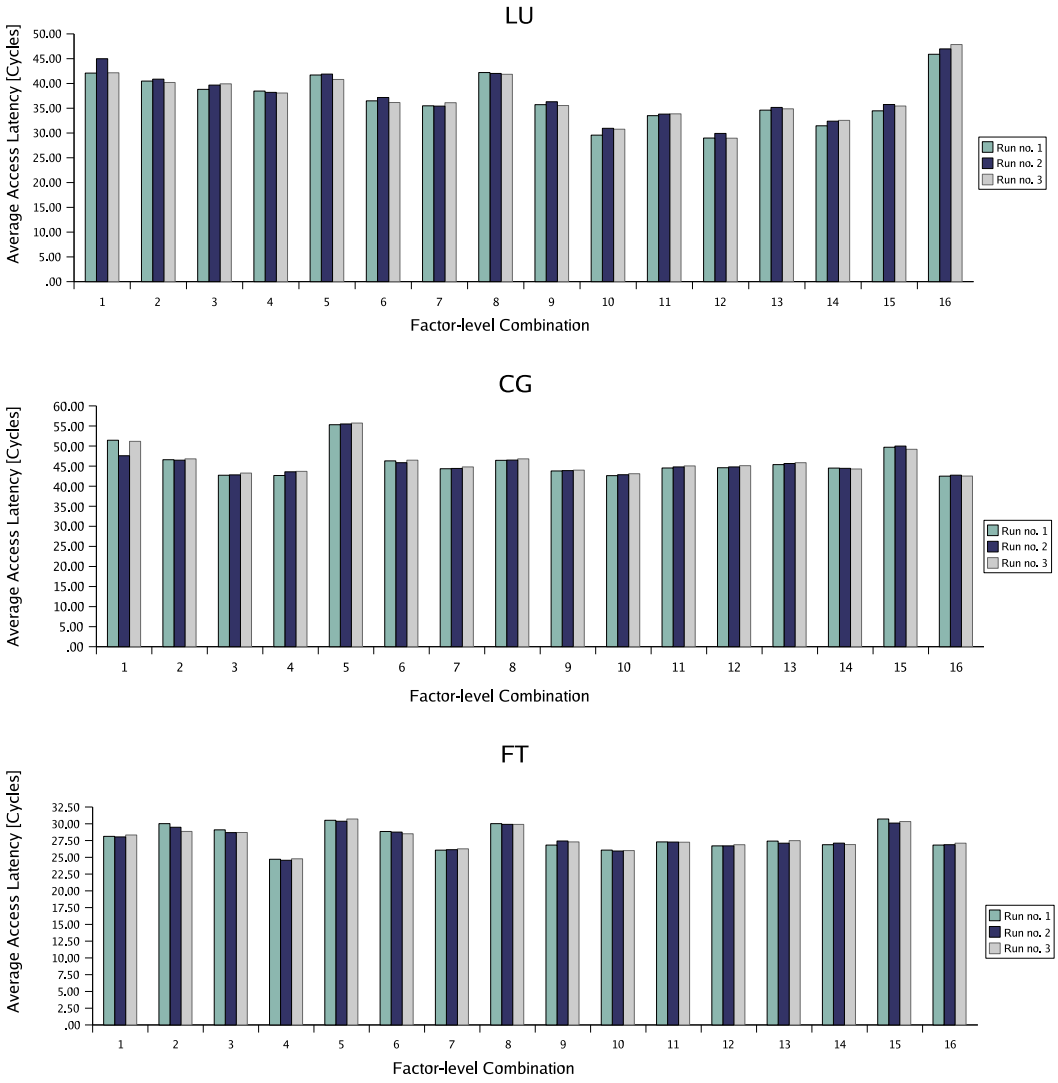


Figure 7.6: Average data access latency of LU, CG and FT.

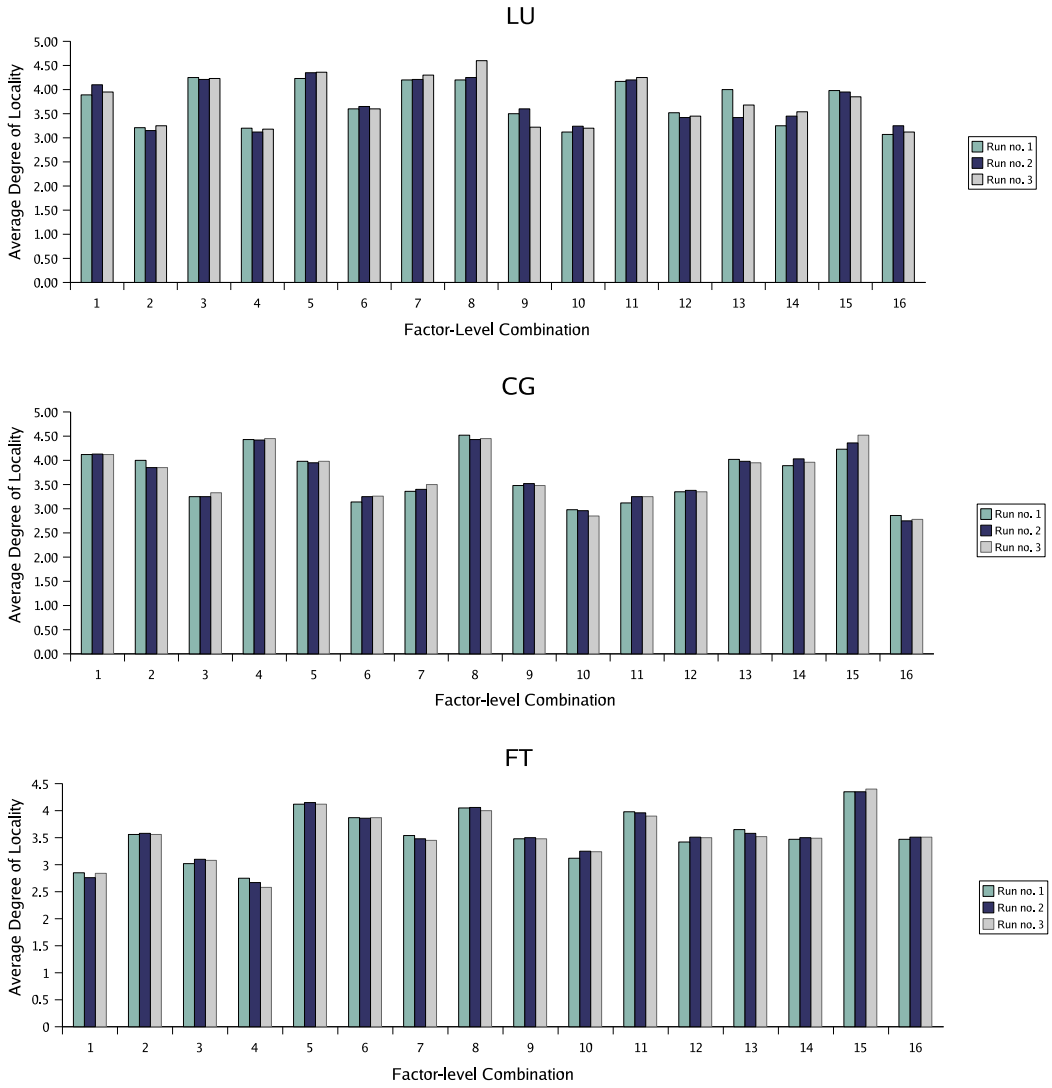


Figure 7.7: Average degree of locality of LU, CG and FT.

Figure 7.7 illustrates the average degree of locality obtained from the LU, CG, and FT workloads for each factor-level combination. In contrast to the latency plots shown in Figure 7.6, all locality degree plots show more variation. The degree of locality falls between 2 and 5 degrees on average, reflecting that data accesses are mostly resolved in the requesters' local memories. The variation in the average locality degrees of between 3 and 4, observed from the LU, CG, and the majority of combinations of the FT workloads, describes the high frequency of changing the data state between Clean and Dirty. This evidence can be associated with data access characterisation showing the high frequency of SRSW and MRSW patterns, *i.e.* sharing of the same addresses among readers and writers.

Figure 7.8 and 7.9 show the percentage of cache misses and inclusion misses of the three workloads against each factor-level combination. The average cache miss percentages are quite consistent across all factor-level combinations. The average of between 13 and 18 percentage points of cache misses was observed in the three workloads. Figure 7.9 shows the percentage of L1 cache misses that are caused by invalidating the L2 cache lines, *i.e.* the percentage of inclusion misses. The LU and CG workloads have lower percentages of inclusion misses than the FT workload, in general. However, the average plots of the LU and CG are more variable in comparison to the FT average plot. This evidence can be attributed to the greater variation in data sharing patterns in the LU and CG calculations than in the FT code.

All figures described earlier show the overall response of different factor-level combination models to the performance metrics. To analyse these results, the analysis-of-variance (ANOVA) statistical model is used to summarise the variation of each performance metric due to the assigned factor-level combinations. Table 7.5 presents the resulting ANOVA table obtained from running a One-way ANOVA test on each factor against the three metrics: data access latency, degree of locality and cache misses ratio.

The ANOVA table provides a formal F test for the factor effect. The F-statistic is the mean square for the factor divided by the mean square for the error. To select the probability of observing the factor effect on the performance metric at greater than 95%, the factor is significant at less than 5% level (the value in column *Sig.* is less than 0.05). Thus, the less the significant level, the more probability to observe the factor effect to the performance metric.

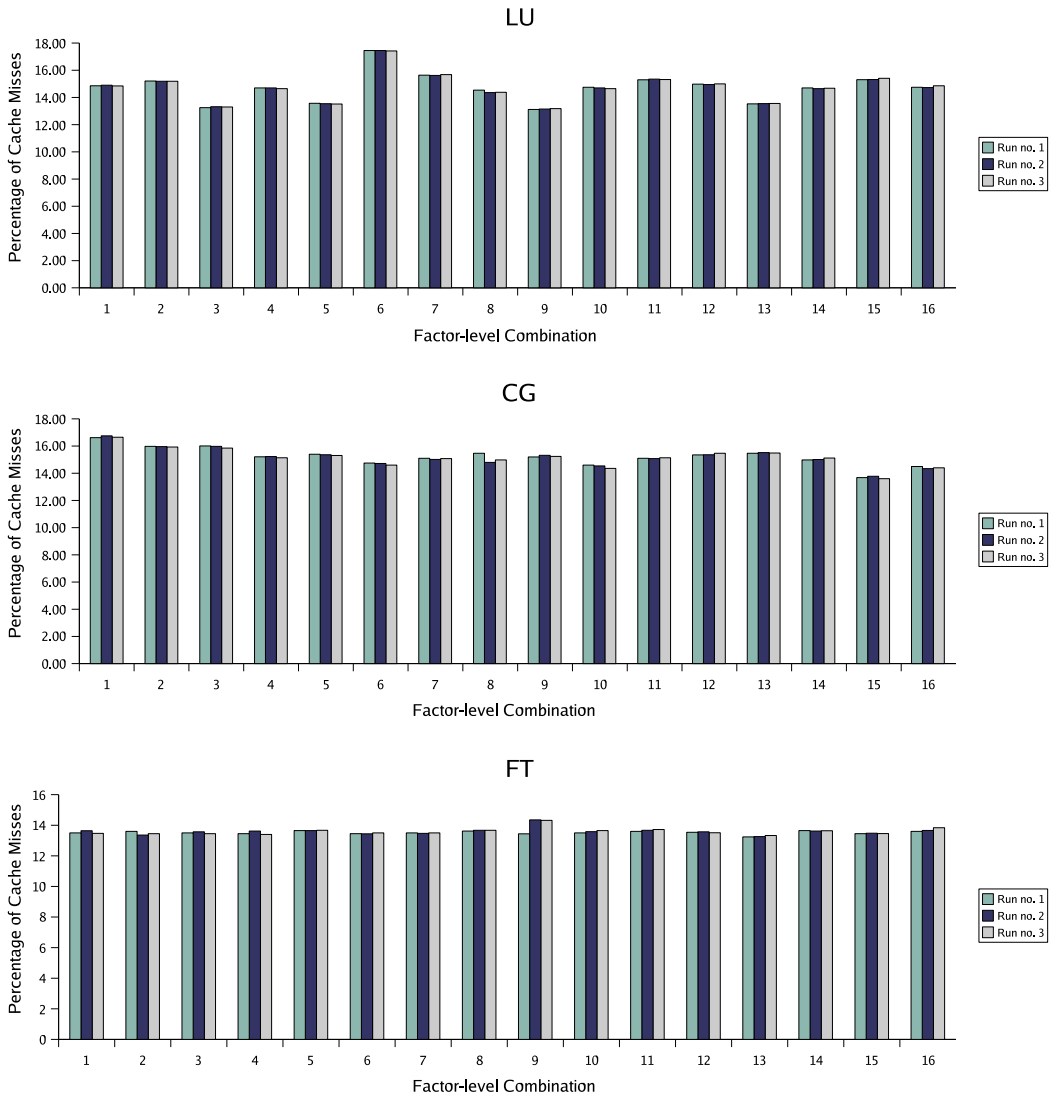


Figure 7.8: Percentage of cache misses.

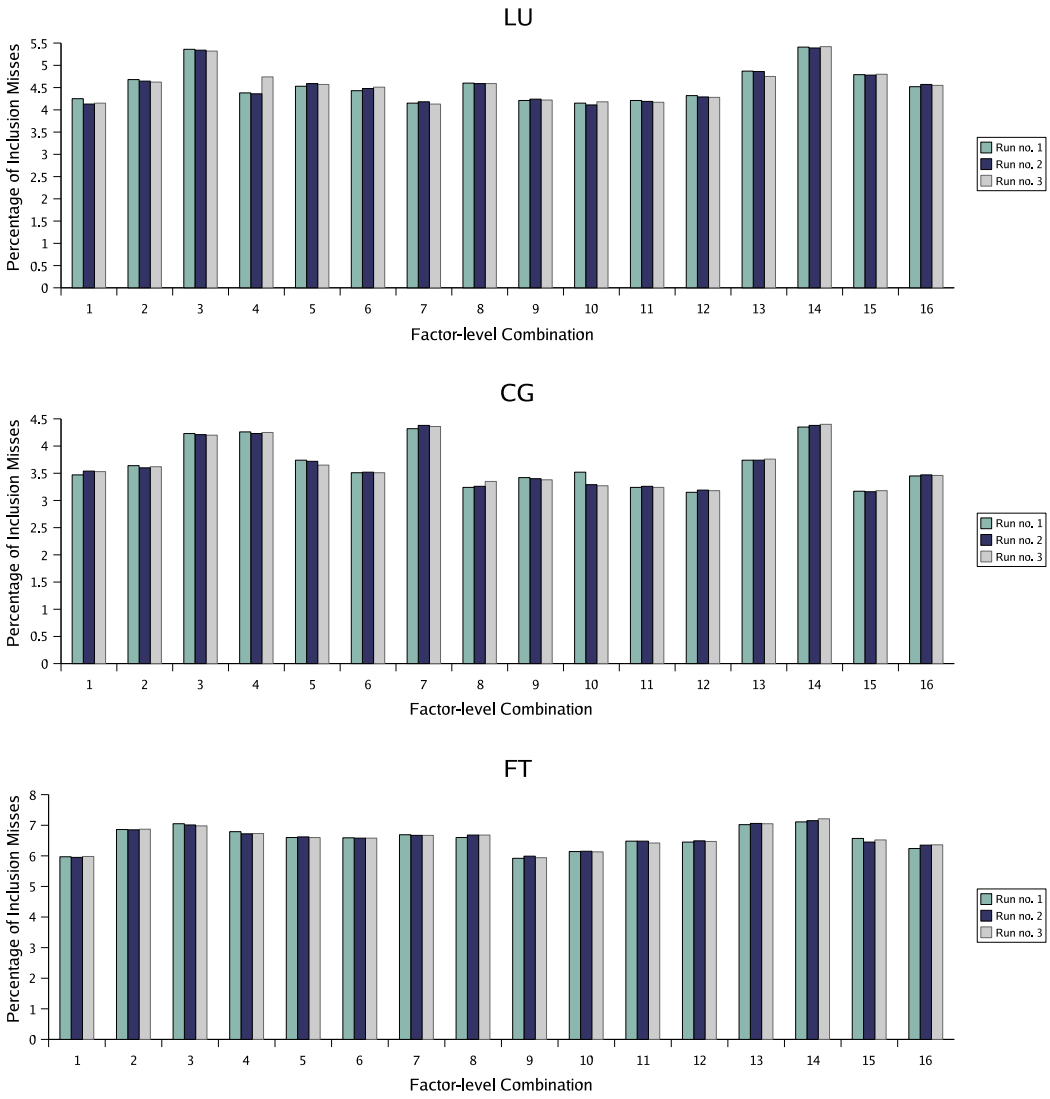


Figure 7.9: Percentage of invalidate causing misses (inclusion misses).

As shown in Table 7.5, several of the factors are statistically significant in terms of latency and degree of locality. To select the important factors, the 90% probability (*i.e.* 0.1 significance level) is used in testing hypotheses, *i.e.* whether changing the levels of the factors can be attributed to the change in memory locality (reflected by the latency, degree of locality and cache miss ratio).

As highlighted in the ANOVA table, the high level of data access latency can be contributed by the cache architecture and coherence units (as shown by the significance value being less than 0.1). Three factors show a contribution to the high level of locality degree including the DSM architecture, consistency technique and the number of OpenMP threads. No significance value of less than 0.1 is observed from the ANOVA table as the response to cache misses percentage.

7.3.2 Remarks

A performance study [Figueiredo and Fortes, 2000] has shown that, in heterogeneous DSM systems, processor performance has the greatest effect on the DSM performance, followed by cache size, memory latency, and network latency. The experiments described in this thesis aim to describe the memory characteristics from another perspective. That is, in DSM systems composed of homogeneous processors and the same cache size, which system components are the dominant contributors to memory latency.

In the screening experiments, when using 90% probability to observe the effect (at 0.1), five dominant factors have been selected out of the ten factors based on the One-way ANOVA test. These factors have been shown to contribute to the data access latency and the degree of locality. From the screening experiments, the highlighted factors that are sensitive to the memory locality (order by the signifance levels) are the consistency technique, DSM architecture, the number of OpenMP threads, coherence unit and cache architecture.

The statistical results shown in Table 7.5 also highlight some information on which group of factors are most related to each metric. If using the probability of observing the factor effect on each metric at greater than 70% (less than 0.3 significance level), six factors are found to be related to the degree of locality[3].

[3] *i.e.* consistency technique, DSM architecture, number of OpenMP thread, bus architecture, snoopy protocol, and cache architecture (order by the signifance levels)

Table 7.5: ANOVA Table for Tests of Dominant Factors.

Factor	df	Data Access Latency				Degree of Locality				Cache Misses Ratio			
		SS	MS	*F*	Sig.	SS	MS	*F*	Sig.	SS	MS	*F*	Sig.
DSM Architecture	1	91.012	91.012	1.366	.244	6.782	6.782	33.643	.000	1.802	1.802	1.789	.183
Cache Architecture	1	220.671	220.671	3.359	.069	.277	.277	1.118	.292	.744	.744	.733	.393
Coherence Unit	1	281.233	281.233	4.309	.040	.249	.249	1.006	.317	.007	.007	.007	.935
Bus Architecture	1	.040	.040	.001	.981	.446	.446	1.810	.181	.965	.965	.952	.331
Snoopy Protocol	1	44.223	44.223	.661	.418	.345	.345	1.398	.239	.000	.000	.000	.998
Directory Protocol	1	41.174	41.174	.615	.434	.023	.023	.093	.760	.381	.381	.375	.541
Interconnection Routing	1	1.412	1.412	.021	.885	.219	.219	.882	.349	.727	.727	.716	.399
Consistency Model	1	14.964	14.964	.223	.638	.096	.096	.384	.536	2.198	2.198	2.187	.141
Consistency Technique	1	155.875	155.875	2.356	.127	4.512	4.512	20.740	.000	1.569	1.569	1.554	.215
OpenMP Threads	1	6.605	6.605	.098	.754	1.902	1.902	8.062	.005	1.884	1.884	1.871	.174

However, using the same selection criterion, only four and three factors are found to be highly related to the data access latency[4] and cache miss ratio[5] respectively.

Consistency technique (update or invalidate) used at the software DSM system has been found important for all of the performance metrics, at different levels. The consistency technique is most related to the degree of locality (*i.e.* it contributes most to the decision of where the data is placed). However, in terms of time spent to access data, the coherence unit and cache architecture have shown more impact than the consistency technique. The coherence unit denotes the frequency of actions required to maintain cache coherence. In this experiment, different memory addresses in the shared data segment are accessed constantly by processors. Therefore, the larger the coherence unit, the more frequently the coherence actions are required.

The cache architecture in this experiment is used to test whether caches of the same size but organised in two different ways will cause an impact on the three performance metrics. The statistic results shows that the activities required to maintain cache coherence and the organisation of cache hierarchy contributes to data access time more than the location of data segments at main-memory level.

The choice of DSM architecture (16×2 or 4×8 DSMs) used to organise the processing load between inter-node and intra-node sharing has also been found to be important for all performance metrics. In this experiment, the main process of the workload was scheduled to one of the processors in an SMP node. The later, dynamically created parallel threads were scheduled within the same node first. Based on this scheduling criteria, it is found that the number of processors in an SMP node shows more impact on the location where data would be placed (the degree of locality) than on the time spent on accessing the data (the data access latency).

A detailed analysis of cache performance shows that the overhead of the coherence protocol is associated with the data access patterns and the parallel constructs used in the workload applications [Marathe et al., 2004]. In this screening experiment, the parallel construct used to identify the number of parallel threads (OMP_NUM_THREADS) has been included with the other factors. The results give more insight into the role of the parallel construct. The results show that the parallel construct studied in this experiment impacts on the location of data and cache miss

[4] coherence unit, cache architecture, consistency technique, and DSM architecture

[5] consistency model, the number of OpenMP threads, and the DSM architecture

ratio. However, it has a lower probability[6] of affecting the time spent on accessing data in comparison to the other factors.

To carry out the detailed analysis on the most dominant factors, the smaller significance level of 0.05 (95% confidence interval) is used in order to select only the factors with higher probability to have effects on the performance metrics. Therefore, two factors, the DSM architecture and consistency technique, were chosen. The next step, analysing the DSM locality problem using these two dominant factors, is described in the following section.

7.4 Detailed analysis of dominant factors

In the second step of locality analysis experiments, the impact of different DSM architectures and consistency techniques on memory locality is analysed. To do so, an experiment to evaluate locality effects on six 128-node DSM models using two DSM consistency policies was conducted. The three workloads (LU, CG, and FT) used in the previous experiment were used. Therefore, the analysis is based on an application exhibiting the SR, SW, MR, SRSW, and MRSW data access patterns.

7.4.1 Experiment Methodology

In this experiment, the 128-node DSM models were divided into groups of three including clusters of small-scale, medium-scale and large-scale SMPs. The first group comprises the 64×2 and 32×4 DSM models. The other two groups comprise the 16×8 and 8×16 DSM models and the 4×32 and 2×64 DSM models respectively. Two varieties of the home-based LRC consistency models were chosen for this experiment: the update-based and invalidate-based protocols. The list of factors and levels used in this experiment is shown in Table 7.6.

Table 7.6: Experimental factor and levels of the detail analysis of dominant factors.

Factor	Levels
DSM Architecture	64×2, 32×4, 16×8, 8×16, 4×32, 2×64
Consistency Policy	HLRC-Update, HLRC-Invalidate

[6] at 24.6% of probability (.754 significance level) to observe the effect of the number of OpenMP threads on data access latency

To evaluate the impact of the DSM cluster architecture on memory locality, six different configurations were set to include each of the chosen consistency policies in the DSIMCLUSTER model (as shown in Table 7.6). All of these configurations used the LU, CG, and FT workload files of 8 threads created per each parallel section. Once these tests had been set up, each of the tests was conducted in three replicas. Table 7.7 shows the experiment design for this test using the full-factorial method. The experiment includes 12 run sets each of which comprises three replicas, *i.e.* 36 runs in total for each workload.

Table 7.7: Experiment design for detail study of dominant factors.

Run set	DSM Architecture		Consistency Model
	cluster of	Model	
1	Small-scale SMPs	64×2	Update-based HLRC
2		64×2	Invalidate-based HLRC
3		32×4	Update-based HLRC
4		32×4	Invalidate-based HLRC
5	Medium-scale SMPs	16×8	Update-based HLRC
6		16×8	Invalidate-based HLRC
7		8×16	Update-based HLRC
8		8×16	Invalidate-based HLRC
9	Large-scale SMPs	4×32	Update-based HLRC
10		4×32	Invalidate-based HLRC
11		2×64	Update-based HLRC
12		2×64	Invalidate-based HLRC

7.4.2 Experiment Results

7.4.2.1 Data Access Latency

Firstly, the impact of both DSM architectures and consistency policies on data access latency was analysed. Figure 7.10 shows a histogram plotting the data access latency obtained from each factor-level combination. The results obtained show a consistent latency on the DSM systems composed of small-scale and medium-scale SMPs (64×2 to 8×16) on the LU and CG workloads. The shift of average latency is noticeable for the 4×32 DSM model (the large-scale SMPs), showing the sensitivity of latency due to the intra-node bus contention. To confirm this, the Paired T-Test method was

used to analyse the statistical correlation of data between the change of SMP size to the average data access latency.

DSM architectures *v* Data Access Latency. Figure 7.11 shows the comparison of means of data access latency on each test pair[7]. The T-Test statistics show two findings as listed below.

a. Changing the node sizes from small-scale to medium-scale SMPs (Pair 1) can reduce the access latency by 7.3 percentage points. All processors saw reduced latencies and this happens quite consistently[8].

b. When enlarging the node size to large-scale SMPs (in both Pair 2 and 3), data access latency is significantly increased by 34.8 and 45.4 percentage points on average.

Since the significance value for a change in latencies (testing at 95 percent confidence interval) is far less than 0.05 on all three-pair tests[9], it can be concluded that the findings (a) and (b) listed above are not due to chance variation. These latency changes can be attributed to the change of the number of processors per SMP nodes in a DSM cluster. The results show that upgrading from small-scale to medium scale SMPs shows latency gain while upgrading to large-scale SMPs shows latency reduction. This suggests that, in terms of data access latency, using a medium-scale SMPs in a DSM cluster offers a better performance (based on the LU, CG, and FT workloads and the system conditions used in this experiment).

Consistency Techniques *v* Data Access Latency. Figure 7.11 also shows the comparison of means of data access latency against the change of DSM consistency model from using the invalidate-based to the update-based technique (Pair 4). The statistics show the finding as listed below.

a. When using the update-based technique, processors saw reduced latencies by 9 percentage points and did so quite consistently (the correlation of 0.977, nearly 1).

[7]Results obtained from running the Paired T-Test analysis method using a statistical software, SPSS
[8]shown by the high correlation of 0.771, or at absolute value nearly 1
[9]The significance values of 0.000, 0.000, and 0.005 for pair 1, 2 and 3 tests respectively

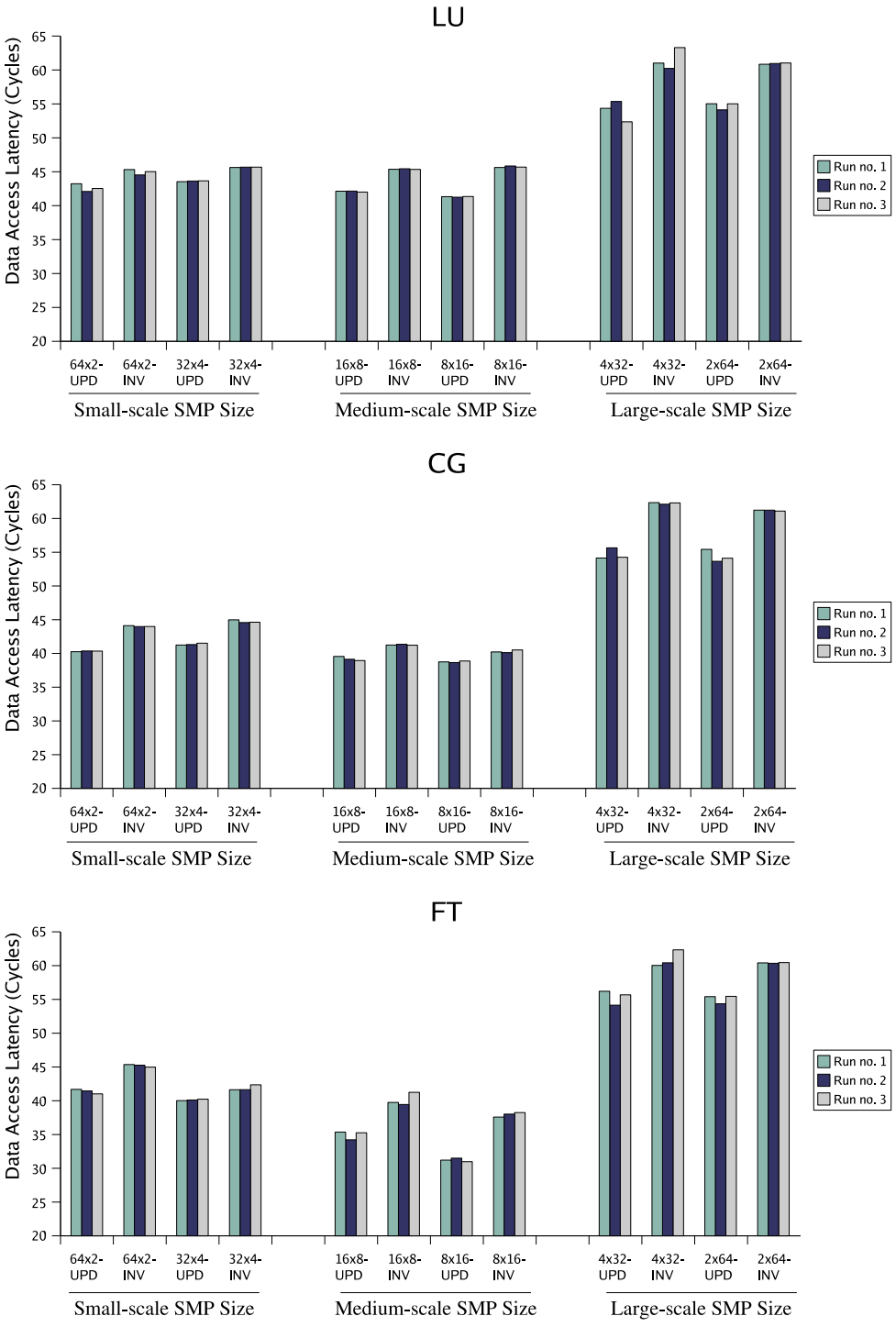


Figure 7.10: Average data access latency.

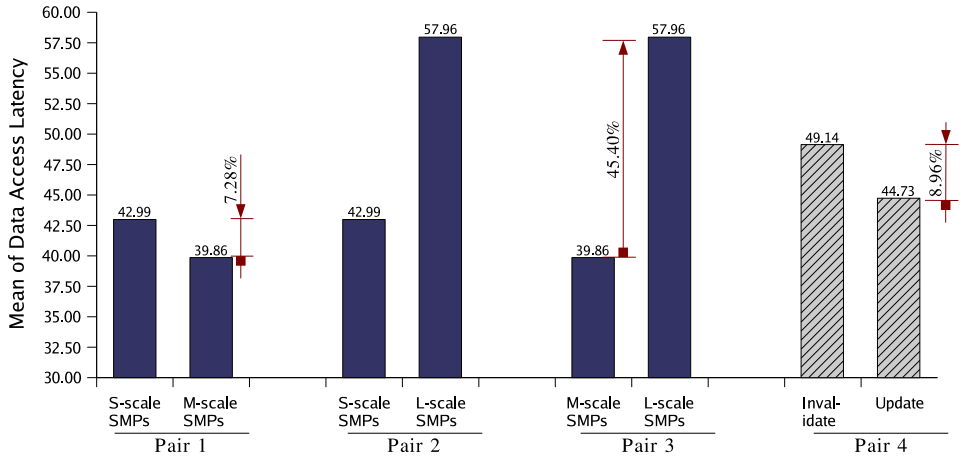


Figure 7.11: Mean of latency when changing the SMP sizes and consistency policy.

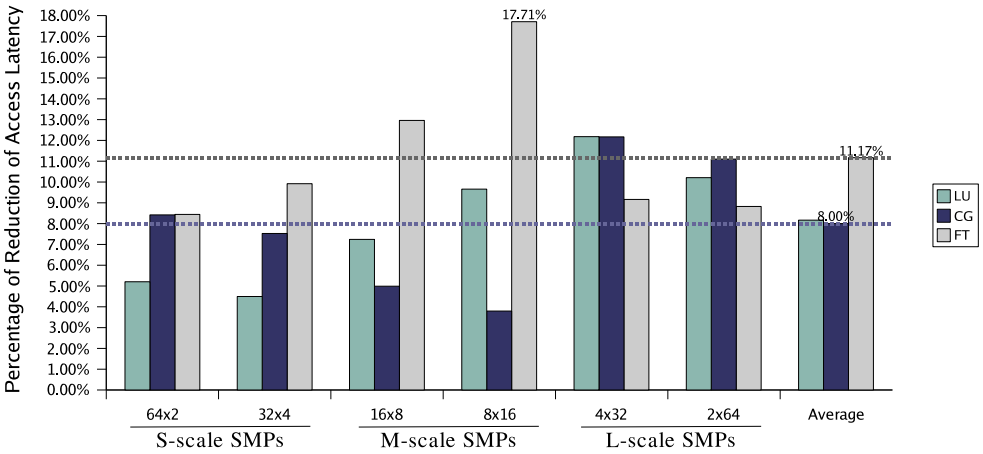


Figure 7.12: Percentage of latency reductions of using update-based policy.

b. The standard deviation reveals that changing the consistency technique cause less variation with respect to data access latency than changing the number of processors in an SMP node.

c. The variation of latency reduction for each workload (shown in Figure 7.12) shows the average latency reduction when using the update-based policy instead of the invalidation-based policy by between 8 and 11.17 across the three workloads.

If testing at 95 percent confidence interval, the significance value for change in latencies due to consistency technique is far less than 0.05. Therefore, it can be concluded that the findings (a), (b) and (c) listed above is not due to chance variation, and can be attributed to the use of update-based technique in a DSM manager.

7.4.2.2 False-sharing Misses

The second analysis focused on the overhead caused by false sharing, as shown by the number of false-sharing caused misses recorded at L1 caches. Figure 7.13 shows the number of false access misses recorded from three workloads using the two protocols. From the three histograms, among different SMP scales using the same consistency policy, the similarity of average numbers of false sharing misses is observed in general. However, the number of misses caused by false-sharing is significantly increased from the update-based protocol to the invalidate protocol. To analyse this, the Paired T-Test method is also used for identifying the statistical significance of the results.

DSM architectures *v* False-sharing Misses. The statistic tests of Pair 1 to Pair 3 (Figure 7.14) show the findings as listed below.

a. Across all processors, the number of false-sharing misses was reduced by 0.6 and 3.8 percentage points on average if upgrading the number of processors in an SMP node (Pair 1 to 3).

b. The standard deviations reveal that the results of changing from small-scale SMPs to both medium- and large- scale SMPs (Pair 1 and 2) were more variable with respect to the number of misses than the Pair 3 results.

The significance value of less than 0.05 for changes of false-sharing misses of all three pairs shows that the number of processors per SMP node affects the false-

sharing misses pattern. It can be concluded that the reduction of false-sharing misses by up to 4 percentage points can be attributed to the scale of the SMP nodes in a DSM cluster.

Consistency Techniques *v* False-sharing Misses. Figure 7.14 also shows the comparison of means of the number of false-sharing misses against the change of DSM consistency model from using the invalidate-based to the update-based technique (Pair 4). The statistic test shows the findings as listed below.

a. When using the update-based technique, processors saw reduced false-sharing misses by 9.5 percentage points and did so quite consistently (with a correlation of 1).
b. The variation of false-sharing misses (shown in Figure 7.15) shows the average reduction of false-sharing misses when using the update-based policy by between 7.4 and 19.4 across the three workloads.

If testing at 95 percent confidence interval, the significance value for the change in false-sharing misses due to consistency technique is far less than 0.05. Thus, in this case, it can be concluded that using the update-based technique can reduce the number in false-sharing misses on the CG, LU and FT workloads. However, the standard deviation reveals that changing the consistency technique was more variable with respect to false-sharing misses than changing the number of processors in an SMP node. This suggests that, in terms of false-sharing misses, changing the consistency technique shows greater impact then changing the SMP node size.

7.4.3 Discussion

The LU, CG, and FT workloads represent a reader-based access characteristic with a coarse grain spatial access pattern. This experiment used eight threads per each parallel section, causing a variety of sharing patterns on both the intra-node and extra-node memories. In the DSM models composed of 2-way and 4-way SMPs, the overhead of sharing an intra-node bus is less than that of the DSM models composed of larger-scale SMPs, as shown by the average latency and the change of latencies for upgrading the SMP size. The more processors in the SMP nodes, the more variation of data access latencies should be seen. Statistical analysis of the results has confirmed that

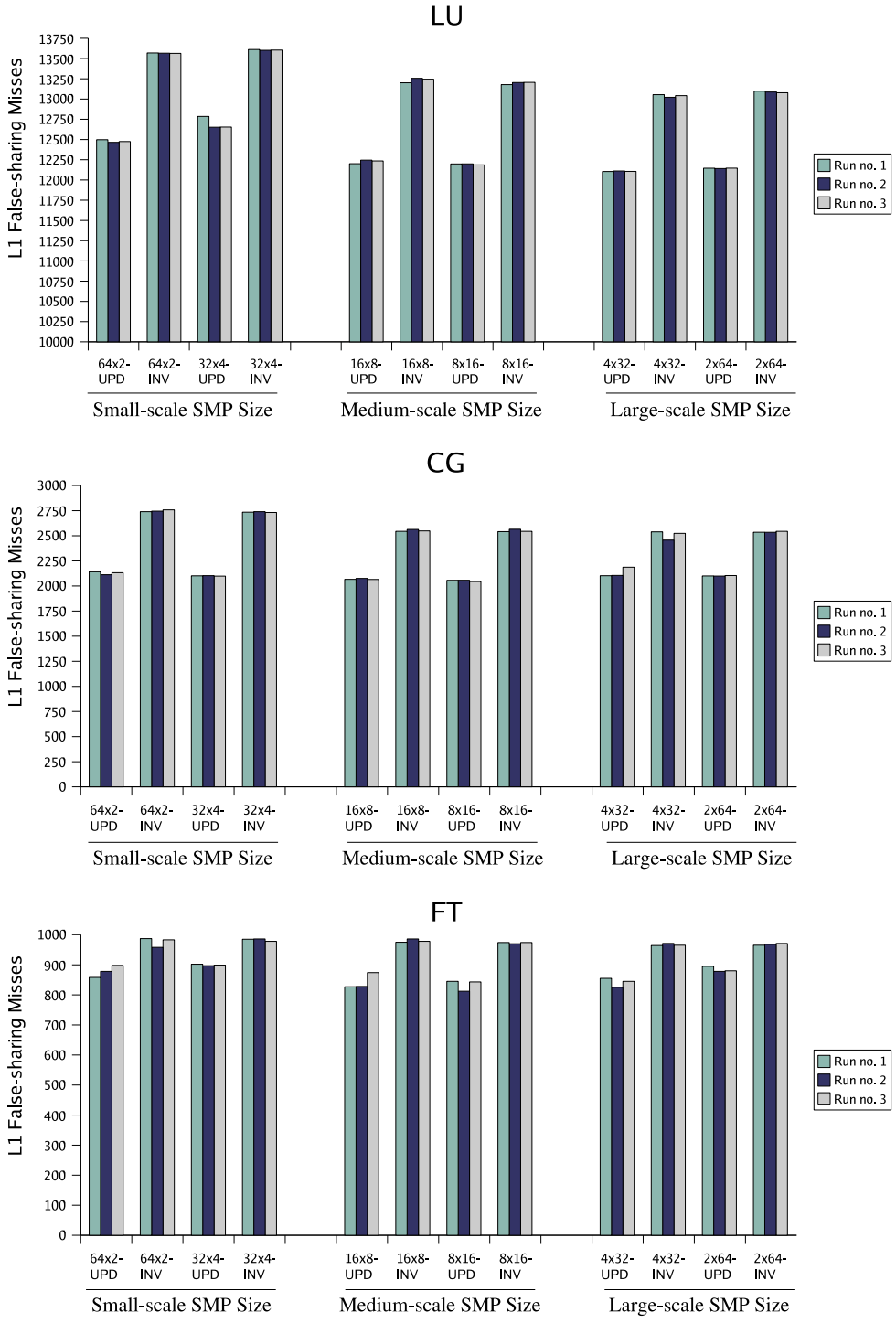


Figure 7.13: False-sharing misses at L1 caches.

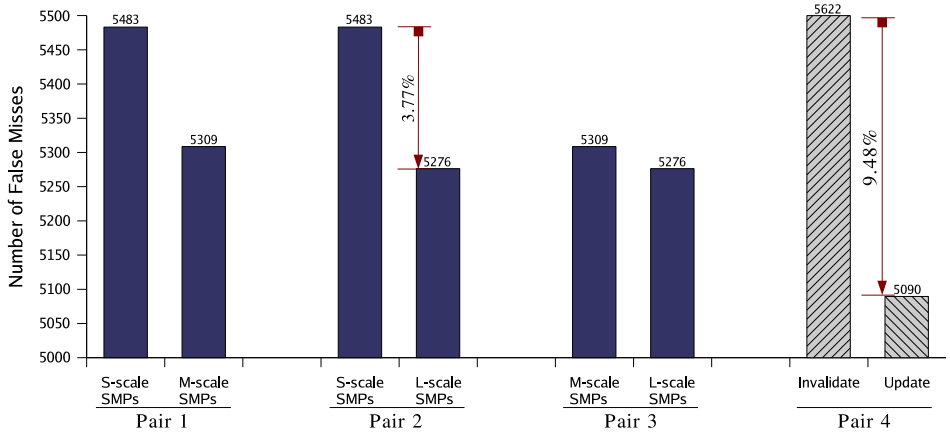


Figure 7.14: Mean of false-sharing misses.

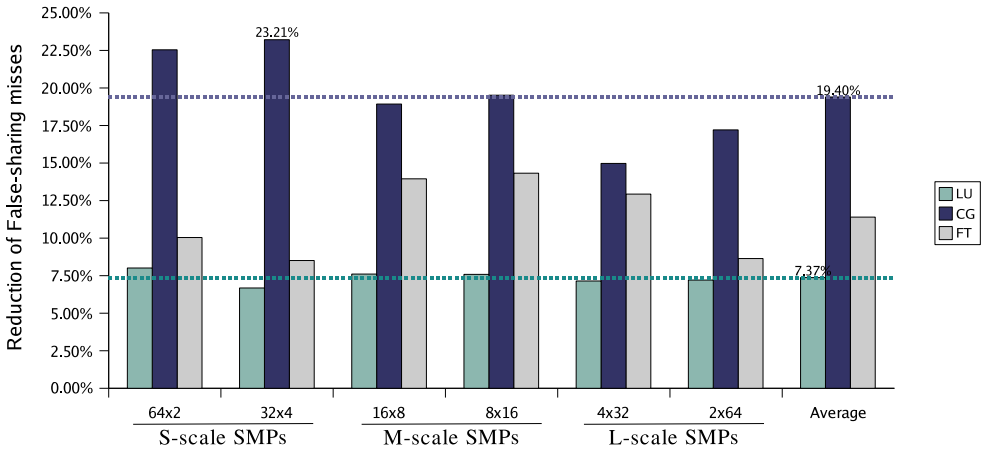


Figure 7.15: Percentage of false-miss reductions when using update-based policy.

this hypothesis is true. Besides, the impact is more noticeable in large-scale SMPs due to the impact on intra-node bus contention.

In terms of false-sharing misses, it was observed that the CG workload shows more sensitivity to the use of the updated technique in the consistency model. This observation is coherent with the data access characterisation showing more accesses by multiple writers than in the LU and FT workloads. If using an update policy, when an updated content is brought into the L2 caches, it does not always invalidate the L1 cache line. Thus, when two or more processors attempt to write to different blocks of the same L2 cache line, the one that wins the acquire request will write first and broadcast the updated content. In this case, the L1 cache line of the other processor will not be invalidated as the updated address of the L2 cache line is not included in the L1 cache line. Therefore, the reduction of false-sharing misses at L1 cache lines can be expected. Statistical analysis of the results has confirmed this claim, suggesting the use of an update-based technique for the reader-based workloads.

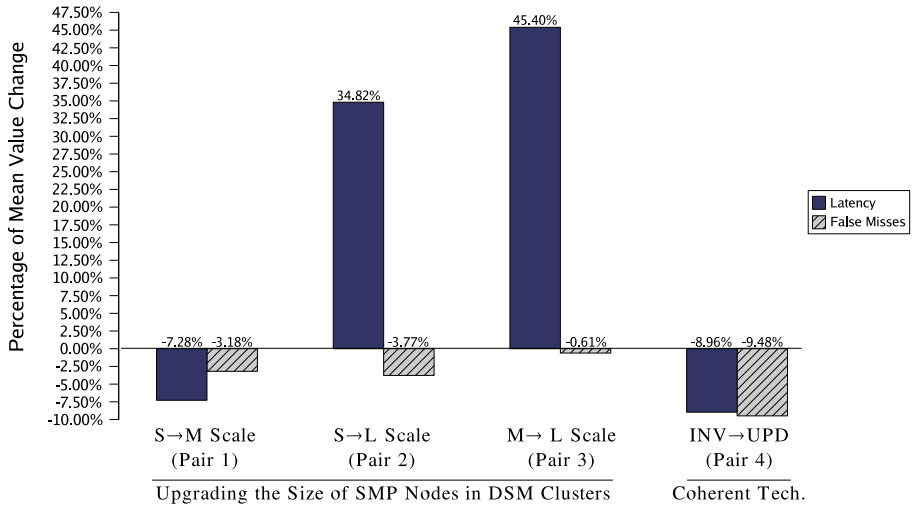


Figure 7.16: Percentage of mean value difference of the latency and false misses.

The percentage of change in the mean value of false misses and data access latency is shown in Figure 7.16. This statistical summary shows more variation in data access latency due to changing the number of processors in SMP nodes, rather than being due to the consistency technique. The high percentage of change suggests a careful selection of SMP size to match the pattern of data access characteristics could yield more stability in performance. Upgrading the size of SMPs to large-

scale shows a significantly increased latency, while upgrading the size of SMPs to medium-scale shows a latency reduction. This suggests that DSM clusters composed of medium-scale SMPs can offer better performance than the other two choices. The lower variation in false misses reduction and latency gain shown in using the update-based technique suggested that the performance gain is quite consistent. Therefore, based on running the LU, CG, and FT workloads on the system conditions used in this experiment, carefully choosing the SMP size and using the update-based consistency technique in the DSM manager can lead to stability of performance regarding the memory locality.

7.5 Summary

The results outlined in this study demonstrate that the consistency technique used in the DSM manager and the structure of the DSM clusters dominate the degree of locality compared with other parameter variations for all three workloads based on the 32-node DSM models. These observations suggested detailed analysis experiments on the two most dominant factors.

The results of the analysis show the dominant factors affecting memory locality are the number of processors per SMP node and the choice of technique used in the consistency policy. This is manifested by the change of data access latency and the number of false misses when changing the two factors. T-Test statistics were used to confirm the significance of the results. The results show that DSM clusters composed of medium-size SMPs offer more advantage regarding the data access latency with a very small impact on the false-sharing misses. The higher degree of locality, reflected by the average latency reduction of between 8 to 11.2 percentage points, can be attributed to the consistency technique (update or invalidate) used in the DSM manager. These results suggest that using medium-scale SMPs in a DSM cluster and data update technique at the DSM manager can cause less impact on memory locality, yet lead to more stability in performance.

Chapter 8

Conclusions

In distributed-shared memory systems, the memory locality problem is considered as critical in terms of performance and portability of DSM applications. Existing works have shown the potential of simulation modelling as a tool to conduct experiments to study different DSM systems, mostly in terms of the impact of different hardware designs. However, the rapid advances of DSM implementations in software have made two issues of simulation model design ever more important: the limitation of simulation frameworks on *model extensibility* and the lack of *verification applicability* during a simulation run causing a delay in the verification process. This thesis has focused on techniques to enable rapid construction of simulation models for the analysis of the memory locality problem on various DSM architectures. In terms of simulation techniques, this thesis has contributed to both identifying the ways to achieve model extensibility and proposing a technique to apply verification during a simulation run. Various experiments conducted by using a simulation model, DSIMCLUSTER, have demonstrated the potential of these techniques to support the analysis of memory locality problems. From these experiments, the memory consistency technique used in the DSM management software, and the number of nodes per SMP in a cluster have been found to be associated with the higher degree of locality loss. Detailed analysis of these two factors suggested that using the update based policy for the multiple readers access pattern, and using a medium size (8 to 16) SMP with the update-based consistency technique could contribute to locality gain on the three workloads by up to 17 percentage points. This chapter draws conclusions on the key contributions and discusses the directions for future research.

8.1 Thesis Summary

This thesis focuses on simulation modelling techniques to support the analysis of the overhead caused by memory locality in a variety of DSM systems, a compelling platform for high performance computing. Existing researches have shown that the performance loss of DSM systems caused by the reduction of memory locality is crucial, and is associated with many factors. Nevertheless, the diversity of the hardware architectures and the heterogeneity of DSM implementations has made the problem difficult to analyse, especially in terms of design concepts comparison. This constraint has forced the re-evaluation of techniques used to construct a simulation model to allow the exploration of the generic design concepts.

As described in Chapter 3, many previous simulation modelling techniques have attempted to allow rapid construction of multiprocessor simulation models using direct execution, abstraction of details, timing-first approach, object-oriented approach or model reuse. Some have been constructed with generality using interpretation-driven simulation to support a large exploration space. Others have focused on the simulation speed and allowed the simulated details to be adjusted. The missing pieces that may allow the simulation model to cover both software and hardware activities are (a) the exploration on *model extensibility* by separating the model from the simulation engine, and (b) ways to integrate the verification of components into a simulation (*verification applicability*).

This thesis studied the model extensibility achieved by separating the simulation model from the discrete event simulation engine. A construction framework, HASE, and its DES engine HASE++, were used to develop the key components of a DSM system (referred to as the *framework components*) based on the model specifications defined by DEVS-based language. The resulting simulation model, DSIMCLUSTER, has been implemented and equipped with the four important operations as follows:

1. a mechanism to transfer control among the simulation kernel, the workload execution unit, and different model parameters;

2. an emulation of a multi-threaded runtime environment to simulate parallel multi-threaded workloads within the simulation address space;

3. a specification-based verification technique to simulate a vital component, the

bus-based coherence protocol;

4. a hierarchical organisation of performance metrics.

Verification applicability has been incorporated into the DSIMCLUSTER using the Specification-based Parameter-Model interaction technique. This technique has been demonstrated by verifying the simulated behaviour of bus-based coherence protocols. The verification involves the specification checking the model semantics obtained from parsing a text-based specification written in a Protocol State-transition Definition (PSD) language. The correctness of the model is validated through a comparison study against the SunFire 15K machine. The comparison of cache miss percentages obtained showed that the results of the simulation were in accord with the real machine, at a difference of $\pm 5 - 6\%$ on average and 14.5% in the worst case.

Experiments were conducted using the DSIMCLUSTER model to study the importance of two groups of parameters (the architectural design choices and the DSM management policies) to the degree of locality reduction. The study was separated into four series of experiments. The first two experiments allowed preliminary analysis of the impact of data layout and protocol ownership on memory locality using a particular workload, LU decomposition. The results showed that the improvement of spatial locality (reflected by the improvement of cache utilisation) of 31 percentage points can be gained from matching the direction of accesses to the array data layout in RAM. The results from the coherence protocol ownership showed that with the multiple readers access pattern, systems with ownership-based protocols could reduce the loss of locality (reflected by the reduction of data access latency) by an average, 17.8 and 28.9 percentage points. The statistical *t-test* has shown that the higher the number of threads per parallel section, the more consistently this improvement is achieved.

The last two series of experiments were aimed at identifying the factors that dominate the degree of locality in various DSM architectures, and analysing the impact of these factors in detail. To identify the most dominant factors, sixteen combinations of ten 2-level factors were used to conduct a series of screening experiments on DSIMCLUSTER using (a) the Plackett-Burman design generator with three replications, (b) three workloads obtained from the NPB 2.3 benchmark (LU, CG, FT), and (c) three performance metrics (cache miss statistics, data access latency and the

degree of locality). Results of the screening experiment, summarised by using the analysis-of-variance (ANOVA) statistical model, identified four important factors out of ten: the write-synchronisation policy, consistency model, DSM architecture, and coherence unit. The highest correlation factors associated with the high-level degree of locality loss are (a) the memory consistency model used in the DSM management software and (b) the cluster architecture, in particular the number of nodes in an SMP.

Results of the analysis experiments on these two factors, using full-factorial design, demonstrated that using a medium size (8 to 16) SMP could contribute to the locality gain (reflected by the reduction of memory access latency) on the three workloads by up to 17 percentage points, or an average of between 8 and 11.2 percentage points. The results also showed that data invalidation caused by consistency models is responsible for a high percentage of false access misses at local caches by up to 23 percentage points with an average of between 7.3 and 19.4 percentage points on the three workloads, suggesting that using write update on shared data is crucial for a multiple readers access pattern. Tuning these factors may relax the unnecessary coherence actions that could be a major cause of locality overhead.

Using the HASE construction model, DSIMCLUSTER gained model extensibility shown by the numbers of customisation and re-configuration options for conducting experiments on different workloads without having to recompile the source files. The graphical representation of the model and the post-mortem animation have also contributed to the verification of interconnection and communication among model components. The performance (measured in terms of simulation run time) of the simulation approach used in DSIMCLUSTER is especially susceptible to the execution dynamics of parallel workloads: the number of parallel threads created for each parallel section, data access frequency and data sharing patterns (inter-node and intra-node accesses). However, improvement of simulation run time can be gained by switching off the tracing, behaviour log, and plot file generation facilities.

8.2 Implications of the Simulation of Model

The development of DSIMCLUSTER based on the two objectives (model extensibility and verification applicability) has demonstrated the prevalence of model specification. In Chapter 1, it has been shown that less than ten components were included

in the basic diagram of a generic DSM multiprocessor. However, the large number of tunable features and operational sequences (presented in Chapter 2) have made the strategy used to filter the essential attributes of such a system ever more challenging. The text-based specification describing each component used in this work, which later on had been developed into the DEVS-based specifications, is found to be valuable in two aspects. Firstly, it summarises the attributes of the components essential for the implementation of a simulation model. The DEVS specifications used in this work (shown in Appendix A) were modified concurrently with the expansion of the model, as more features and parameters were gradually included. As a result, the final specification could describe the model behaviour and the possibility to extend the model in the future. Secondly, the list of state transitions in a DEVS specification allows the complete model to be debugged and tested relatively easily. Each item in this list includes the initial conditions, the operation to be performed and the possible outcome states. The functional relationship among different parts of the components have been defined and, as a consequence, each functional part can be tested individually.

Four sets of experiments with various numbers of factors have been carried out by running the workloads on the custom DSIMCLUSTER models to represent different architectures. The model customisations were mostly done without having to re-compile the simulation kernel again. This shows that the organisation of the DSIMCLUSTER model allows the simulation to be custom-tailored relatively easily. Besides, as most of the parameters were implemented as custom libraries, the ways to include the new parameters can be done by plugging a new library in to the existing model.

8.3 Future Work

While the work presented in this thesis provides a basis for the investigation of the memory locality problem in different DSM systems, several avenues of research are indicated by the limitations of the current study. Three key issues that should be addressed are a) the extension to a larger exploration space, b) the optimisation of the time spent on preparing a workload, and c) the optimisation of the simulation runtime. In the first issue, the validity of the simulation framework when extended to a wider variety of existing machine architectures is of importance to the continued

use of DSIMCLUSTER to support different target architectures, *e.g.* a heterogeneous cluster. In the second issue, an examination of how the role of compilers used to obtain valid inputs to a simulation can reduce the time to investigate the large variety of workloads. Lastly, in the third issue, the integration of the analytical process into a behavioural model implemented in a simulation (via interfacing with an analytical tool) can alleviate the delay of detailed simulation, particularly on the components with actions that can be predicted by mathematical models, like buses and system interconnections. Continued work into the simulation of input/output modules and more detailed work on network interconnection is required. Some possible challenges are described below.

8.3.1 Validation

Validation, as seen in this work, can be done by comparing results with a real machine. There is a body of work on verifying the correctness of results obtained from the simulation with regard to coherence protocol semantics. Unfortunately, much work remains to be done to validate models in terms of timing and cycle accurate simulation. As this study focused on memory performance, work that verifies the properties of the processor is lacking. A possible way to achieve more accuracy in processor cycles is to replace the instruction cache used in the current model with the detailed instruction fetching components.

8.3.2 Analysis of a Large Scale, Realistic Workload

Although the design of cluster architectures and the activity of coherence protocols show high and consistent impact against the utilisation of spatial locality, in the presence of multiple readers, the impact from more varieties of workloads have not yet been included. More importantly, memory paging due to the physical space limitation or memory failure were not addressed in the current study. It is proposed, therefore, that the following work be carried out to assess these issues. One possible way is to add into the instruction set implementation some more instructions related to the input/output data transfer. One way to simulate this on the current DSIMCLUSTERmodel is to include some magic instructions to support the system calls. These magic instructions provide the way for the Processor Entity to pass control to the OS

entity on the fly. Thus, the interaction between the Processors and the Operating System Entities for I/O transfers, memory paging due to the physical space limitation, or memory failure could be addressed using this technique.

8.3.3 Parallel Constructs and Compiler-directed Techniques

During the initial phase of the current study, it was noted that general DSM performance was found to be degraded due to different parallel constructs and ways the parallel workloads are distributed as directed by compilers. For further investigation of this, a broad spectrum of parallel constructs (both user defined and compiler directed) should be used and related to the results obtained by simulation. In DSIM-CLUSTER, the impact of OMP_NUM_THREADS using static scheduling has been identified as important to the degree of memory locality measured. From this data, a better understanding of load balancing schemes, parallel workload distribution and correlation of code- and data- affinity overheads could be effected. Similar to using magic instructions to emulate the procedure calls or multithreading, the use of magic instructions for different forms of parallel constructs would also be a useful tool.

8.3.4 Locality Analysis on Larger Exploration Space

One potential avenue of research was indicated in Chapter 7: the DSM system reacted differently with two consistency techniques employed in the DSM manager. Memory page replications are managed by concurrent activities of coherence directory managers. An exploration of the effect of unbalanced workloads, fault-tolerance schemes, and the interception of high priority jobs may be warranted. This extension can be done by adding new features to the implementation of the DSM manager. Integrating the probability models to generate different patterns of incoming workloads could be useful for the study of load balancing. In this case, an instruction set object implementing the higher level instruction formats or trace could be used to replace the current detailed instruction set implementation.

8.4 Conclusions

DSM performance can be readily degraded by the loss of locality caused by various activities used to maintain memory consistency. The levels of locality loss are dependent less on the activities of individual hardware components than on the software policy and structure of a cluster. The overhead caused by locality loss can be controlled by the matching of write-synchronisation policy (update or invalidate) to the potential data access patterns with an adjustable coherence granularity. As architectures become more hierarchically complex, future simulation methodology will increasingly rely on the integration of a generic, effective DES development tool, ways to verify a model of new components, and analytical methods to simplify the unnecessary details. The results presented here, obtained by conducting experiments on the DSIMCLUSTER, are supportive of the hypothesis that the investigation of DSM memory locality problem can be achieved by using an extensible, automatically verified simulation model as a tool. The use of such simulation has led to an insightful understanding of the architecture with a high level of accuracy, and is unlikely to pose a significant resource requirement on the host machine for conducting experiments.

Appendix A

The DSIMCLUSTER Specification

A.1 The Coupled DEVS definitions of DSIMCLUSTER

Coupled model Node

components: {Processor; Cache; BusInterface}

internal coupling:

BusInterface.toProcessor $\longrightarrow$ Processor.fromBus

BusInterface.toCache $\longrightarrow$ Cache.fromBus

Processor.toCache $\longrightarrow$ Cache.fromProcessor

Processor.toBus $\longrightarrow$ BusInterface.fromProcessor

Cache.toBus $\longrightarrow$ BusInterface.fromCache

Cache.toProcessor $\longrightarrow$ Processor.fromCache

end Node

Coupled model Controller

components: {Memory; OperatingSystem}

internal coupling:

end Controller

Coupled model SMP

components: {Coupled:Node; Coupled:Controller;

Bus; NetworkInterface}

internal coupling:

Node.BusInterface.toMemory ⟶ Bus.fromNode
Controller.Memory.toNode ⟶ Bus.fromMemory
Bus.toMemory ⟶ Controller.Memory.fromBus
Bus.toNode ⟶ Node.BusInterface.fromMemory

end SMP

Coupled model CLUMPS
components: {Coupled:SMP; InterconnectionNetwork}
internal coupling:
SMP.NetworkInterface.toNetwork
⟶ InterconnectionNetwork.fromCluster
InterconnectionNetwork.toCluster
⟶ SMP.NetworkInterface.fromNetwork

end CLUMPS

Coupled model DSiMCluster
components: {DSMManager, Profiler, Coupled:CLUMPS}
internal coupling:

end DSiMCluster

A.2 The Atomic DEVS definitions

A.2.1 Processor

Atomic Component Processor

inports: {fromCache; fromBus};

outports: {toCache; toBus};

State Variables:

status with range {IDLE, BUSY, MEMRD, MEMWR,
BARRIER, STALL}

Atomic Operations:

A1: Fetching instructions from the instructionCache,
performs the execution in the instruction cycle
until one of these conditions has been satistified:
a) the end of basic block is reached,
b) an exception has occured, or
c) an interrupt signal has arrived.

A2: Sending request for data access (memory read/write) and wait until
a) receiving data (in case of read), or
b) receiving acknowledgement (in case of write)

A3: Sending request to pass a barrier and wait until
a) all waiting processors have hit the barrier

A4: Acquiring an exclusive lock for a memory location and wait until
a) the lock has been acquired

A5: Releasing an exlusive lock

A6: Retriving and invoking a sequence of service routines

initial condition:

(status := IDLE; clockPhase = 0)

external transition:

{ (clockPhase=0,status = IDLE)
⇒ A1(status = BUSY)
⇒status={IDLE, MEMRD, MEMWR,
BARRIER, LOCK}}

{ (clockPhase=0, status = MEMRD)⇒ A2() ⇒ status={MEMRD, IDLE}}

{ (clockPhase=0, status = MEMWR)⇒ A2() ⇒ status={MEMWR, IDLE}}
{ (clockPhase=0, status = BARRIER)
⇒ A3() ⇒ status={BARRIER, IDLE}}
{ (clockPhase=0, status = LOCK)⇒ A4() ⇒ status={LOCK, IDLE}}

internal transition:

output function:

{ recordProfile();}

A.2.2 Cache Hierarchy

Atomic Component Cache

inports: {fromProcessor; fromBus};

outports: {toProcessor; toBus};

State Variables:

status with range {C_IDLE, C_RD_HIT, C_RD_MISS,
C_WR_HIT, C_WR_MISS,
C_RM_STALL, C_WR_STALL
C_COHERENCE_STALL, C_STALL}

Atomic Operations:

A1:Listening to CPU command,
perform cache lookup
until one of these conditions has been satistified:
a) access hit, or
b) access miss

A2:Listening to event from Bus,
a) receiving data (in case of read), or
b) receiving and updating data (in case of write miss), or
c) receiving acknowledgement (in case of write through)
and passing the result to CPU

A3:Sending a request for data to Memory

A4:Listening to event from the coherence controllers
a) perform coherence actions, and/or
b) sending an acknowledge

A5:Sending a request to coherence controller
to broadcast the cache update actions

A6:Continue cache lookup in the next level

A7:Update the states of cache line due to cache hierarchy actions

A8:Writing data and sending acknowledge to CPU

A9:Sending the requested data to CPU

A10:Checking coherence state-transition status

initial condition:

(status := IDLE; clockPhase = 0)

external transition:

{(clockPhase=0, status=IDLE)
⇒ A1(status=BUSY), A5()
⇒ status={C_WR_HIT, C_RD_HIT, C_RD_MISS,
C_WR_MISS, C_RM_STALL, C_WR_STALL}
{(clockPhase=1, status=IDLE)
⇒ A4(status=BUSY), A7()
⇒ status={IDLE, C_COHERENCE_STALL}}
{(clockPhase=0, status=C_STALL)
⇒ A2(status=BUSY)
⇒ status={C_WR_HIT, C_RD_HIT, C_STALL}}

internal transition:

{(clockPhase=1, status=C_WR_HIT)
⇒ A8(status=BUSY), A7(), A5()
⇒ status=IDLE}
{(clockPhase=1, status=C_RD_HIT)
⇒ A9(), A5()
⇒ status=IDLE}
{(clockPhase=1, status=C_RD_MISS)
⇒ A6(), A5()
⇒ status={C_RD_HIT, C_RD_MISS, C_RM_STALL}}
{(clockPhase=1, status=C_WR_MISS)
⇒ A6(), A5()
⇒ status={C_WR_HIT, C_WR_MISS, C_WR_STALL}}
{(clockPhase=1, status={C_WM_STALL, C_RM_STALL})
⇒ A3(), A5()
⇒ status={C_STALL}}
{(clockPhase=1, status = C_COHERENCE_STALL)
⇒ A10(), A7()
⇒ status={IDLE, C_COHERENCE_STALL}}

output function:

{ recordProfile();}

A.2.3 Bus Interface

Atomic Component BusInterface

inports: {fromCache; fromProcessor};

outports: {toCache; toProcessor};

State Variables:

status with range {BI_IDLE, BI_WAITING, BI_SENDING_DATA,
BI_SENDING_CMD, BI_RECEIVING_DATA,
BI_RECEIVING_CMD)}

Atomic Operations:

A1: Listening for a request to use bus from Processor
a) if there is a request, set Processor as the bus master
b) if no request from Processor, checking at the Cache
if there is a request, set Cache as the bus master
c) if no request from Processor, nor Cache
checking at the Bus if there is a request
set Bus as the bus master

A2: Receiving command from the master

A3: Listening for a bus-grant signal

A4: Sending the request package to bus

A5: Sending data package to bus

A6: Receiving data from the master, perform one of the following
a) if the data is to return to Processor or Cache, sending data
b) if the data is to send to bus, perform bus arbitration

initial condition:

(status := BI_IDLE; clockPhase = 0)

external transition:

{(clockPhase=0, status = BI_IDLE)
⇒A1()
⇒status={BI_IDLE, BI_RECEIVING_CMD}}
{(clockPhase=1, status = BI_RECEIVING_CMD)
⇒ A2()
⇒ status={BI_RECEIVING_DATA}}
{(clockPhase=0, status = BI_RECEIVING_DATA)

⇒A6()
⇒status={BI_WAITING, BI_IDLE}}
{ (clockPhase=1, status = BI_WAITING)
⇒ A3()
⇒ status={BI_SENDING_CMD, BI_WAITING}}
{ (clockPhase=0, status = BI_SENDING_CMD)
⇒ A4()
⇒ status={BI_SENDING_DATA}}
{ (clockPhase=1, status = BI_SENDING_DATA)
⇒ A5()
⇒ status={BI_SENDING_DATA, BI_IDLE}}

internal transition:

output function:

{ recordProfile();}

A.2.4 Bus

Atomic Component Bus

inports: {fromNode[1]; fromMemory};

outports: {toNode; toMemory};

State Variables:

status with range {B_IDLE, B_COMMAND, B_DATA, B_TEST}

Atomic Operations:

A1: Listening for an arbitration request from node or memory
if there is a request, send bus grant and set the master

A2: Receiving command from the master

A3: Receiving data from the master and getting the slave address

A4: testing if the slave entity is ready
if the slave is ready, send the command and data

initial condition:

(status := B_IDLE; clockPhase = 0)

external transition:

{(clockPhase=0, status = B_IDLE)
⇒A1() ⇒status={B_IDLE, B_COMMAND}}

{(clockPhase=1, status = B_COMMAND)
⇒ A2() ⇒ status={B_DATA}}

{(clockPhase=0, status = B_DATA)
⇒A3() ⇒ status={B_TEST}}

{ (clockPhase=1, status = {B_TEST})
⇒ A4()
⇒ status={{B_TEST, B_IDLE}}}

{ (clockPhase=0, status = {B_TEST})
⇒ A4()
⇒ status={{B_TEST, B_IDLE}}}

internal transition:

output function:

{ recordProfile();}

A.2.5 Memory

Atomic Component Memory

inports: {fromBus};

outports: {toBus};

State Variables:

status with range {M_IDLE, M_GET_CMD, M_READ, M_WRITE,
M_ARBITRATE, M_SEND, M_WAITING}

Atomic Operations:

A1: Listening for a request from the Bus

A2: Receiving command from the Bus

A3: Receiving data from the Bus

A4: Performing memory read

A5: Performing memory write and notify clDSMManager

A6: Sending a bus arbitration signal

A7: Checking for a bus grant signal

A8: Sending a data or acknowledge package Bus

initial condition:

(status := M_IDLE; clockPhase = 0)

external transition:

{(clockPhase=0, status = M_IDLE)
⇒A1() ⇒status={M_IDLE, M_GET_CMD}}

{(clockPhase=1, status = M_GET_CMD)
⇒ A2()
⇒ status={M_GET_CMD, M_READ}}

{(clockPhase=0, status = M_GET_CMD) ⇒A3() ⇒ status=M_WRITE}

{ (clockPhase=0, status = M_READ) ⇒ A4() ⇒ status=M_ARBITRATE}

{ (clockPhase=0, status = M_WRITE) ⇒ A5() ⇒ status=M_ARBITRATE}

{ (clockPhase=1, status = M_ARBITRATE) ⇒ A6() ⇒ status=M_WAITING}

{ (clockPhase=0, status = M_WAITING) ⇒ A7() ⇒ status=M_SEND}

{ (clockPhase=1, status = M_SEND) ⇒ A8() ⇒ status=M_IDLE}

internal transition:

output function:

{ recordProfile();}

Appendix B

Specification-based Parameter-Model Interconnection

B.1 Protocol State-transition Description (PSD)

B.1.1 Lexical Rules

```
%option bison-bridge reentrant
PROTOCOL_FNS writeData|hold|waitForUpdateData|getUpdateData|fillCache|
             sendAcknowledge|checkPriority|waitForAcknowledge
CACHE_STATES gb_DIRTY|gb_INVALID|gb_EXCLUSIVE|gb_SHARED|
             gb_READ_ONLY|gb_MIN_COUNTER|gb_VICTIMISED|
             gb_MINIMUM
COMPONENT    CACHE|BUS|CPU|MEM
ACTION       readMiss|readHit|writeMiss|writeHit
BUS_FNS      toBUS.(broadcast|supplyData)
CPU_FNS      toCPU.supplyData
MEM_FNS      toMEM.(readData|flushCacheline)

%%
∧(”//”).*\n                  /* eat up one-line comment */
[−+]?[0−9]+                  {yylval_param→num=atoi(yytext);
                              return NUMBER;}
```

”− >”	{ return TO; }
” ==” | ” >” | ” <” | ” >=” | ” <=”	{ yylval_param→charstr = strdup(yytext); return OPERATOR;}
Rule	{ return RULETAG;}
Header	{ return HEADERTAG;}
Name	{yylval_param→charstr = (char*) strdup(yytext); return NAMETAG;}
State	{ return STATETAG;}
Ownership	{return OWNERTAG;}
CacheStateMap	{return CACHESTATETAG;}
Ignore	{return IGNORETAG;}
Priority	{return PRIORITYTAG;}
Verification	{ return VERIFYTAG; }
InvalidGlobalState	{ return INVGST_TAG; }
InvalidTransition	{ return INVTRNSTN_TAG;}
yes|no	{ yylval_param→charstr = (char*) strdup(yytext); return FLAG;}
CACHE_STATES	{ yylval_param→charstr = (char*) strdup(yytext); return CACHESTATE;}
COMPONENT	{ yylval_param→charstr = (char*) strdup(yytext); return COMP;}
ACTION	{ yylval_param→charstr = (char*) strdup(yytext); return ACT;}
BUS_FNS	{ yylval_param→charstr = (char*) strdup(yytext); return BUSFN;}
CPU_FNS	{ yylval_param→charstr = (char*) strdup(yytext); return CPUFN;}
MEM_FNS	{ yylval_param→charstr = (char*) strdup(yytext); return MEMFN;}
PROTOCOL_FNS	{ yylval_param→charstr = (char*) strdup(yytext); return FN;}
“=”	{ return EQU; }
“{”	{ return OCB; }

```
"}"                                 { return CCB; }
"("                                 { return ORB; }
")"                                 { return CRB; }
"["                                 { return OSB; }
"]"                                 { return CSB; }
";"                                 { return SEMCOL; }
","                                 { return SEP;}
"::"                                { return OR;}
":"                                 { return COL;}
"?"                                 { return QTAG; }
"."                                 { return DOT; }
[\t\n]+                             /* eat up whitespace */
[a-zA-Z"_"][0-9a-zA-Z"_"]*+
                                    {yylval_param→charstr = (char*) strdup(yytext);
                                     return NAME;}
%%
```

B.1.2 PSD Grammar

```
%locations
%pure_parser
%parse-param {clumps::CoherenceProtocol *ptcl}
%parse-param {yyscan_t yyscanner}
%lex-param   {yyscan_t yyscanner}

%%
protocol_def : header_def transition_def_list verification_def { }
             ;
header_def   : HEADERTAG '{' name_def state_def owner_def cache_state_def '}'
             ;
name_def     : NAMETAG '=' NAME
             ;
state_def    : STATETAG '(' NUMBER ')' '=' state_list ';'
             ;
state_list   : NAME
             | NAME ',' state_list
             ;
owner_def    : OWNERTAG '(' FLAG ')' '=' flag_list ';'
             ;
flag_list    : FLAG
             | FLAG ',' flag_list
             ;
cache_state_def : CACHESTATETAG '=' cache_state_list ';'
                ;
cache_state_list : CACHESTATE
                 | CACHESTATE ',' cache_state_list
                 ;
transition_def_list : transition_def
                    | transition_def_list transition_def
                    ;
transition_def : STATETAG '(' NAME ')' '{' rule_def ignore_event_def priority_def '}'
```

```
                ;
rule_def        : RULETAG '(' NUMBER ')' '{' event_rule_list '}'
                ;
event_rule_list : event_trnstn_rule
                | event_trnstn_rule event_rule_list
                ;
event_trnstn_rule : event '(' NUMBER ')' OSB trnstn_fn_list CSB '→' '{' out_state_def '}'
                  ;
event : COMP DOT ACT
      ;
trnstn_fn_list : trnstn_fn
               | trnstn_fn ',' trnstn_fn_list
               ;
trnstn_fn      : /* empty */
               | CPUFN
               | MEMFN
               | BUSFN
               | FN
               | FN '(' param_def ')'
               | BUSFN '(' param_def ')'
               ;
param_def      : NUMBER
               | conditional_trans_fn
               ;
conditional_trans_fn : OPERATOR NUMBER '?' trnstn_fn '::' trnstn_fn
                     ;
out_state_def : NAME
              | NAME '::' NAME
              ;
ignore_event_def : IGNORETAG '(' NUMBER ')' '{' event_list '}'
                 ;
event_list       : /* empty */
                 | event
```

```
                    | event ‘,’ event_list
                    ;
priority_def        : PRIORITYTAG ‘(’ FLAG ‘)’ ‘{’ outstate_priority_map_list ‘}’
                    ;
outstate_priority_map_list : outstate_priority_map
                           | outstate_priority_map ‘,’ outstate_priority_map_list
                           ;
outstate_priority_map      : /* empty */
                           | ‘(’ NAME ‘:’ NUMBER ‘)’
                           ;
verification_def : VERIFYTAG ‘{’ invalid_gblstate_list invalid_trnstn_list ‘}’
                 ;
invalid_gblstate_list : INVGST_TAG ‘(’ NUMBER ‘)’ ‘{’ number_list ‘}’
                      ;
number_list : /* empty */
            | NUMBER
            | NUMBER ‘,’ number_list
            ;
invalid_trnstn_list : INVTRNSTN_TAG ‘(’ NUMBER ‘)’ ‘{’ inv_state_trnstn_list ‘}’
                    ;
inv_state_trnstn_list : state_trnstn_def
                      | state_trnstn_def ‘,’ inv_state_trnstn_list
                      ;
state_trnstn_def : /* empty */
                 | ‘(’ NAME ‘→’ NAME ‘)’
                 ;
%%
```

B.2 Bus-based Coherence Protocol Specifications

B.2.1 Illinois

B.2.1.1 Summary of Characteristics

No of waiting-acknowledge matching pairs:
from State [Invalid] = 8 pairs
from State [Exclusive] = 2 pairs
from State [Shared] = 6 pairs
from State [Private_Dirty] = 2 pairs
Total = 18 pairs
No of possible transitions:
from State [Invalid] = 3 possible transitions
from State [Exclusive] = 7 possible transitions
from State [Shared] = 8 possible transitions
from State [Private_Dirty] = 7 possible transitions
Total = 25 possible transitions:

B.2.1.2 The PSD Specification

```
Header{
// note: single writer
  Name= Illinois
  State(4) = Invalid, Exclusive, Shared, Private_Dirty;
  Ownership(yes) = no, yes, no, yes;
  CacheStateMap = gb_INVALID, gb_EXCLUSIVE, gb_SHARED, gb_DIRTY;
}
// "Invalid" state flag 0001
State(Invalid){
      Rule(2){
          CPU.readMiss(4)[toBUS.broadcast,
                          waitForUpdateData(> 0? ::toMEM.readData),fillCache,
                          toCPU.supplyData]→{Shared::Exclusive}
          CPU.writeMiss(4)[toBUS.broadcast,
```

```
                                waitForUpdateData(> 0?getUpdateData::
                                toMEM.readData),writeData, fillCache]→{Private_Dirty}
        }
        Ignore(6){
            CPU.readHit, CPU.writeHit, BUS.readHit,
            BUS.writeHit, BUS.readMiss, BUS.writeMiss
        }
        Priority(yes){
            (Exclusive:1), (Private_Dirty:1), (Shared:0)
        }
}
// "Exclusive" state flag 0010
// CPU.readMiss and CPU.writeMiss occure when the cacheline is victimised.
// This state also called 'Read Private'
State(Exclusive){
        Rule(6){
            CPU.readHit(1)[toCPU.supplyData]→{Exclusive}
            CPU.writeHit(2)[writeData, fillCache]→{Private_Dirty}
            CPU.readMiss(4)[toBUS.broadcast,
                                waitForUpdateData(> 0? ::toMEM.readData),
                                fillCache, toCPU.supplyData]→{Shared::Exclusive}
            CPU.writeMiss(4)[toBUS.broadcast,
                                waitForUpdateData(> 0?::toMEM.readData),
                                writeData, fillCache]→{Private_Dirty}
            BUS.readMiss(1) [toBUS.supplyData]→{Shared}
            BUS.writeMiss(1)[toBUS.supplyData]→{Invalid}
        }
        Ignore(2){
            BUS.readHit, BUS.writeHit
        }
        Priority(yes){
            (Invalid:0) }
}
```

```
// “Shared” state flag 0100
// CPU.readMiss and CPU.writeMiss occure when this cacheline is victimised.
State(Shared){
    Rule(7){
        CPU.readHit(1)[toCPU.supplyData]→{Shared}
        CPU.writeHit(2)[toBUS.broadcast, waitForAcknowledge(==0),
                        writeData, fillCache]→{Private_Dirty}
        CPU.readMiss(4)[toBUS.broadcast,
                        waitForUpdateData(> 0? ::toMEM.readData), fillCache,
                        toCPU.supplyData]→{Shared::Exclusive}
        CPU.writeMiss(4)[toBUS.broadcast,
                         waitForUpdateData(> 0? ::toMEM.readData), writeData,
                         fillCache]→{Private_Dirty}
        BUS.writeHit(0)[]→{Invalid}
        BUS.readMiss(1)[ checkPriority(== 1?toBUS.supplyData::) ]→{Shared}
        BUS.writeMiss(1)[ checkPriority(==1?toBUS.supplyData::) ]→{Invalid}
    }
    Ignore(1){
        BUS.readHit
    }
    Priority(yes){
        (Exclusive:1), (Private_Dirty:1), (Invalid:0)
    }
}
// “Private_Dirty” state (flag 1000)
// CPU.readMiss occures when the cache is victimised.
// This state also called ’Modified’
State(Private_Dirty){
    Rule(6){
        CPU.readHit(1)[toCPU.supplyData]→{Private_Dirty}
        CPU.writeHit(2)[writeData, fillCache]→{Private_Dirty}
        CPU.readMiss(4)[toMEM.flushCacheline, toBUS.broadcast,
                        waitForUpdateData(> 0? ::toMEM.readData),
```

```
                    fillCache, toCPU.supplyData]→{Shared::Exclusive}
        CPU.writeMiss(5)[toMEM.flushCacheline, toBUS.broadcast,
                    waitForUpdateData(> 0?::toMEM.readData),
                    writeData, fillCache]→{Private_Dirty}
        BUS.readMiss(2)[toBUS.supplyData, toMEM.flushCacheline]→{Shared}
        BUS.writeMiss(1)[toBUS.supplyData]→{Invalid}
    }
    Ignore(2){
        BUS.readHit, BUS.writeHit
    }
    Priority(yes){
        (Invalid:0)
    }
}
// '*' represents a "don't care" bit, 'M' represents multiple copies are not allowed
// e.g. '**11' means it is invalid to have the co-existance of cacheline replicas
// that one is in the Shared state and another in the Private_Dirty state
Verification{
    InvalidGlobalState(4){**11,***M, *M**, *11*}
    InvalidTransition(1){(Invalid → Invalid)}
}
```

B.2.2 Berkeley

Berkeley is a four-state write-allocation protocol with ownership-based technique. The Berkeley protocol includes the Invalid, Read_Only, Shared_Dirty and Private_Dirty states.

The possible sources of data inconsistency are:

C1 a Shared_Dirty cache co-exists with caches in state Private_Dirty;
C2 a Read_Only cache co-exists with caches in state Private_Dirty;
C3 there is more than one Private_Dirty cache.

Unsafe condition	Global State Flag (Invalid, Read_Only, Shared_Dirty, Private_Dirty)
C1	**11
C2	*1*1
C3	***M

B.2.2.1 Summary of Characteristics

No of waiting-acknowledge matching pairs:
from State [Invalid] = 8 pairs
from State [Read_Only] = 9 pairs
from State [Shared_Dirty] = 9 pairs
from State [Private_Dirty] = 2 pairs
Total = 28 pairs
No of possible transitions:
from State [Invalid] = 2 possible transitions
from State [Read_Only] = 6 possible transitions
from State [Shared_Dirty] = 7 possible transitions
from State [Private_Dirty] = 6 possible transitions
Total = 21 possible transitions

B.2.2.2 The PSD Specification

Header{
Name = **Berkeley**
// Berkeley also called Berkeley-SPUR

```
        State(4) = Invalid, Read_Only, Shared_Dirty, Private_Dirty;
        Ownership(yes) = no, no, yes, yes;
        CacheStateMap = gb_INVALID, gb_EXCLUSIVE, gb_SHARED, gb_DIRTY;
}
        // flag 0001
State(Invalid){
        Rule(2){
                CPU.readMiss(4)[toBUS.broadcast,
                                waitForUpdateData(≥0?::toMEM.readData),fillCache,
                                toCPU.supplyData]→{Read_Only}
                CPU.writeMiss(4)[toBUS.broadcast,
                                waitForUpdateData(≥0?::toMEM.readData),
                                writeData,
                                fillCache]→{Private_Dirty}
        }
        Ignore(6){
                CPU.readHit, CPU.writeHit, BUS.readHit,
                BUS.writeHit, BUS.readMiss, BUS.writeMiss
        }
        Priority(no){
        }
}
        // flag 0010
        // also called 'Valid' or 'unowned'
State(Read_Only){
        Rule(6){
                CPU.readHit(1)[toCPU.supplyData]→{Read_Only}
                CPU.readMiss(4)[toBUS.broadcast,
                                waitForUpdateData(≥0?::toMEM.readData),
                                fillCache,
                                toCPU.supplyData]→{Read_Only}
                CPU.writeHit(4)[toBUS.broadcast,
                                waitForAcknowledge(== 0?::hold),
```

```
                                writeData,
                                fillCache]→{Private_Dirty}
            CPU.writeMiss(4)[toBUS.broadcast,
                                waitForUpdateData(≥0?::toMEM.readData),
                                writeData,
                                fillCache]→{Private_Dirty}
            BUS.writeHit(1)[sendAcknowledge(-1)]→{Invalid}
            BUS.writeMiss(1)[sendAcknowledge(-1)]→{Invalid}
        }
        Ignore(2){
            BUS.readHit, BUS.readMiss
        }
        Priority(no){}
}
        // flag 0100
        // so called 'Owned non exclusive'
State(Shared_Dirty){
        Rule(7){
            CPU.readHit(1)[toCPU.supplyData]→{Shared_Dirty}
            CPU.readMiss(5)[toMEM.flushCacheline,toBUS.broadcast,
                                waitForUpdateData(≥0?::toMEM.readData),
                                fillCache,
                                toCPU.supplyData]→{Read_Only}
            CPU.writeHit(4)[toBUS.broadcast,
                                waitForAcknowledge(== 0?::hold), writeData,
                                fillCache]→{Private_Dirty}
            CPU.writeMiss(5)[toMEM.flushCacheline,toBUS.broadcast,
                                waitForUpdateData(≥0?::toMEM.readData),
                                writeData,
                                fillCache]→{Private_Dirty}
            BUS.readMiss(1)[toBUS.supplyData]→{Shared_Dirty}
            BUS.writeHit(1)[toBUS.supplyData]→{Invalid}
            BUS.writeMiss(1)[toBUS.supplyData]→{Invalid}
```

```
        }
        Ignore(1){
                BUS.readHit
        }       Priority(no){
        }
}
        // flag 1000
        // This state also called 'Dirty' or 'Owned Exclusive'
State(Private_Dirty){
        Rule(6){
                CPU.readHit(1)[toCPU.supplyData]→{Private_Dirty}
                CPU.readMiss(4)[toMEM.flushCacheline, toBUS.broadcast,
                                waitForUpdateData(≥0?::toMEM.readData),
                                fillCache,
                                toCPU.supplyData]→{Read_Only}
                CPU.writeHit(2)[writeData,
                                fillCache]→{Private_Dirty}
                CPU.writeMiss(5)[toMEM.flushCacheline, toBUS.broadcast,
                                waitForUpdateData(≥0?::toMEM.readData),
                                writeData,
                                fillCache]→{Private_Dirty}
                BUS.readMiss(1)[toBUS.supplyData]→{Shared_Dirty}
                BUS.writeMiss(1)[toBUS.supplyData]→{Invalid}
        }
        Ignore(2){
                BUS.readHit, BUS.writeHit
        }
        Priority(no){
        }
}
Verification
{
        InvalidGlobalState(3){***M, **11, *1*1}
```

InvalidTransition(3){(Invalid→Invalid),
(Read_Only→Shared_Dirty), (Invalid→Shared_Dirty)
}
}

B.3 Summary of Protocol Definitions

B.3.1 Write Invalidate

Write Invalidate is a three-state protocol including, Valid, Invalid and Dirty states. The possible sources of data inconsistency are listed below.

Unsafe Condition	PSD Global State (Invalid, Valid, Dirty)
A Dirty cache co-exists with caches in state Valid.	*11
There are more than one data replicas in Dirty state.	**M

Write Invalidate	**States**: Invalid, Valid, Dirty
	Invalid Global State: **M, *11
	Invalid Transition : 1
	(Invalid) → (Invalid)
	No of waiting-acknowledge matching pairs:
	from State [Invalid] = 6 pairs
	from State [Valid] = 4 pairs
	from State [Dirty] = 2 pairs
	Total = 12 pairs
	No of possible transitions:
	from State [Invalid] = 2 possible transitions
	from State [Valid] = 6 possible transitions
	from State [Dirty] = 6 possible transitions
	Total = 14 possible transitions

B.3.2 Synapse

Synapse is a write-allocation protocol with three states, namely, Invalid, Valid and Dirty. The possible sources of data inconsistency are listed below.

Unsafe Condition	PSD Global State (Invalid, Valid, Dirty)
A Dirty cache co-exists with caches in state Valid.	*11
There are more than one data replicas in Dirty state.	**M

Synapse	**States**: Invalid, Valid, Dirty
	Invalid Global State: *11, **M
	Invalid Transition : 1
	(Invalid) $\rightarrow$ (Invalid)
	No of waiting-acknowledge matching pairs:
	from State [Invalid] = 6 pairs
	from State [Valid] = 6 pairs
	from State [Dirty] = 2 pairs
	Total = 14 pairs
	No of possible transitions:
	from State [Invalid] = 2 possible transitions
	from State [Valid] = 6 possible transitions
	from State [Dirty] = 6 possible transitions
	Total = 14 possible transitions:

B.3.3 DEC Firefly

DEC Firefly is a four-state write-allocation protocol with ownership-based technique. The protocol states are Invalid (INV), Read_Private (RdPrv), Read_Shared (RdSh) and Private_Dirty (PrvDrt) states. The possible sources of data inconsistency are listed below.

Unsafe Condition	PSD Global State (INV, RdPrv, RdSh, PrvDrt)
A PrvDrt cache co-exists with caches in state RdSh.	**11
A PrvDrt cache co-exists with caches in state RdPrv.	*1*1
There is more than one Read_Private replica	*M**
There is more than one Private_Dirty replica	***M

Firefly	**States**: Invalid, Read_Private, Read_Shared, Private_Dirty
	Invalid Global State: **11, *1*1, *M**, ***M
	Invalid Transition : 4
	(Invalid) → (Invalid)
	(Read_Private) → (Invalid)
	(Read_Shared) → (Invalid)
	(Private_Dirty) → (Invalid)
	No of waiting-acknowledge matching pairs:
	from State [Invalid] = 8 pairs
	from State [Read_Private] = 4 pairs
	from State [Read_Shared] = 9 pairs
	from State [Private_Dirty] = 2 pairs
	Total = 23 pairs
	No of possible transitions:
	from State [Invalid] = 4 possible transitions
	from State [Read_Private] = 8 possible transitions
	from State [Read_Shared] = 10 possible transitions
	from State [Private_Dirty] = 8 possible transitions
	Total = 30 possible transitions:

B.3.4 Dragon

Dragon is a five-state write-allocation protocol. The protocol states are Invalid (I), Read_Private (RP), Shared_Clean (SC), Shared_Dirty (SD) and Private_Dirty (PD) states. The possible sources of data inconsistency are listed below.

Unsafe Condition	PSD Global State (I, RP, SC, SD, PD)
A PD cache co-exists with caches in state SD	***11
A PD cache co-exists with caches in state SC	**1*1
A PD cache co-exists with caches in state RP	*1**1
A RP cache co-exists with caches in state SC	*11**
A RP cache co-exists with caches in state SD	*1*1*
There is more than one Read_Private replica	*M***
There is more than one Private_Dirty replica	****M

Dragon	**States**:Invalid, Read_Private,
	Shared_Clean, Shared_Dirty, Private_Dirty
	Invalid Global States: ***11, **1*1, *1**1,
	*11**, *1*1*, ****M, *M***
	Invalid Transition : 5
	(Invalid) → (Invalid)
	(Read_Private) → (Invalid)
	(Shared_Clean) → (Invalid)
	(Shared_Dirty) → (Invalid)
	(Private_Dirty) → (Invalid)
	No of waiting-acknowledge matching pairs:
	from State [Invalid] = 10 pairs
	from State [Read_Private] = 2 pairs
	from State [Shared_Clean] = 9 pairs
	from State [Shared_Dirty] = 9 pairs
	from State [Private_Dirty] = 2 pairs
	Total = 32 pairs
	No of possible transitions:
	from State [Invalid] = 4 possible transitions
	from State [Read_Private] = 9 possible transitions
	from State [Shared_Clean] = 8 possible transitions
	from State [Shared_Dirty] = 11 possible transitions
	from State [Private_Dirty] = 9 possible transitions
	Total = 41 possible transitions

B.3.5 MESI

MESI is a four-state write-once protocol with ownership-based technique. The acronym MESI denotes the four states of the protocol, namely, Modified (M), Exclusive (E), Shared (S) and Invalid (I). The state Exclusive implies cache line ownership. The possible sources of data inconsistency are listed below.

Unsafe Condition	PSD Global State (M, E, S, I)
An Exclusive cache line co-exists with a Shared line.	*11*
An Exclusive cache line co-exists with a Modified line.	11**
A Modified cache line co-exists with a Shared line.	1*1*
There are multiple cache lines in the Exclusive states.	*M**
There are multiple cache lines in the Modified states.	M***

MESI **States**:Modified, Exclusive, Shared, Invalid
Invalid Global States: 1*1*, M***,
Invalid Transition : 1
(Invalid) $\rightarrow$ (Invalid)
No of waiting-acknowledge matching pairs:
from State [Modified] = 0 pair
from State [Exclusive] = 4 pairs
from State [Shared] = 3 pairs
from State [Invalid] = 4 pairs
Total = 11 pairs
No of possible transitions:
from State [Modified] = 6 possible transitions
from State [Exclusive] = 6 possible transitions
from State [Shared] = 6 possible transitions
from State [Invalid] = 2 possible transitions
Total = 20 possible transitions:

Appendix C

DSIMCLUSTER Workload and Configuration Files

C.1 DSIMCLUSTER Workload Format (DWF)

The type of workload files taken by the DSIMCLUSTER's `OS` Entity are the DSIMCLUSTER Workload Format (DWF) files. A DWF file is composed of three types of sections as listed below.

Description Section. `.desc`

A one-line string contains as a short description of the workload.

Main process Section. This section is regarded as the main program and used to create the main process of the workload. The main section is separated into subsections, defining using the structure shown below.

`.main`
{*One or more Data Sections*}
`.code`
{*Assembly-liked format codes*}
`.end_code`
`.end_main`

Thread or subroutine section. A section whose name started with "`.sub_`" is regarded to as a subroutine or a thread section. This section is invoked by the

using the magic instructions, `CALL` or `SPAWN`. The magic instruction `CALL`, manages a subroutine call while the `SPAWN` magic instruction creates multiple parallel threads using the specified "`.sub_`" section. Similar to the structure of the main section, the subroutine section is separated into subsections, defining several data sections and one code section using the structure shown below.

`.sub_`*`section_name`*
{*One or more Data Sections*}
`.code`
{*Assembly-liked format codes*}
`.end_code`
`.end_sub_`*`section_name`*

Table C.1: DWF Supported Data Type and Declaration Formats.

C/C++ declaration	DSIMCLUSTER
int i;	int i _val([*initial value*])
double d;	float d _val([*initial value*])
int arr[9][9][3];	int arr[9,9,3][a] distribution inititial_values
double arr[4][3];	float arr[4,3] distribution initial_values

[a] no space allows between commas

C.2 The DEFAULT Instruction Set

C.2.1 Register Sets

Table C.2: The DEFAULT Instruction Set Registers.

Symbol Object	Name	Comment
Integer Registers	%0 . . . %31	32-bit Registers
Floating Points Registers	%0 . . . %31	32-bit Registers
General Purpose Register	%g0 . . . %g31	64-bit Registers
Stack-pointer register	%sp	
Segment-pointer register	%fp	

Note that, in the DEFAULT Instruction Set, the instruction mnemonics (ending with F-Floating Point or I-Integer) are used to identify which register set is to be used. For example, if using, MOVF %0,%1, the value of floating point register %1, will be copied to the floating point register %0. While using, MOVI %0,%1, will select the integer register set instead.

C.2.2 Addressing Memory

An indirect addressing mode in the DSIMCLUSTER workload file is specified in the format similar to the AT&T definition, as follow.

$$Section : Displacement(base : index : scale)$$

The effective address is calculated by the following equation,

$$effective\_address = SectionAddress + (base + (index \times scale)) + displacement$$

In this case, the term *base* and *index* refer to the registers which keep the addresses to be used as base address and index address, respectively.

C.2.3 Instructions

The DSIMCLUSTER model supports a custom instruction set, DEFAULT. The instruction set has twenty seven instructions as listed in the Table C.3 below.

Table C.3: The DEFAULT assembly instructions.

ADDF	ADDF dest, opr1, opr2	[dest ← opr1+opr2]
ADDI	ADDI dest, opr1, opr2	[dest ← opr1+opr2]
BARR	Barrier	
DECF	DECF dest, val	[dest ← dest-val]
DECI	DECI dest, val	[dest ← dest-val]
DIVF	DIVF dest, opr1, opr2	[dest ← round(opr1/opr2)]
DIVI	DIVI dest, opr1, opr2	[dest ← round(opr1/opr2)]
EXIT	EXIT	
INCI	INCI dest	[dest++]
IPOP	IPOP ireg	[ireg ← value from user stack]
IPUSH	IPUSH ireg	[user stack ← ireg]
JGE	JGE reg1, reg2, desc	[jump to dest when (reg1) ≥ (reg2)]
JLT	JLT reg1, reg2, desc	[jump to dest when (reg1) < (reg2)]
KPOPI	KPOPI ireg	[ireg ← value from kernel Stack]
KPUSHI	KPUSHI ireg	[kernelStack ← ireg]
KTOPI	KTOPI ireg	[ireg ← value from kernel Stack]
LOCK	LOCK mem	[request LOCK at memory address mem]
MOVF	MOVF dest, srce	[dest ← source]
MOVI	MOVI dest, srce	[dest ← source]
MULF	MULF dest, opr1, opr2	[dest ← round(opr1*opr2)]
MULI	MULI dest, opr1, opr2	[dest ← opr1*opr2]
RDFW	RDFW dest reg, src mem	[reg_addr ← mem_addr]
RDIW	RDIW dest reg, src mem	[reg_addr ← mem_addr]
RETURN	RETURN	[return the program control to the caller]
UNLOCK	UNLOCK	[UNLOCK the previous lock]
WRFW	WRFW src mem, des reg	[mem_addr ← reg_addr]
WRIW	WRIW src mem, des reg	[mem_addr ← reg_addr]

C.2.4 An Example DEFAULT Object File

C.2.4.1 Main Program Section.

.desc

LU algorithm using the column-major data layout

.main

.share _W

int N _val(?'DataSize'?)

float a[N,N] [_none,_none] _file(data/A128x128)

.end_sh

.code

```
0       $$<pc>profiling{start}
1       MULI %0, #[a_b0], #4
2       MOVI %1, #128
3       MOVI %8, %1
4       DECI %8, #1
5       MOVI %2, #0
6       ADDI %3, %2, #1
7       MULI %4, %2, %1
8       ADDI %6, %4, %2
9       RDFW %0, 0:[a](%0;%6;#4)
10      ADDI %5, %3, %4
11      RDFW %1, 0:[a](%0;%5;#4)
12      DIVF %2, %1, %0
13      WRFW 0:[a](;%5;#4), %2
14      INCI %3
15      JLT %3, %1, #10
16      KPUSHI %8
17      KPUSHI %0
18      KPUSHI %1
19      KPUSHI %2
20      $$<clos>stacktop{int,k}
21      $$<clos>spawn{sub_par0}
22      KPOPI %2
23      KPOPI %1
24      KPOPI %0
25      KPOPI %8
26      INCI %2
27      JLT %2, %8, #6
28      $$<pc>profiling{stop}
29      EXIT
```

.end_code

.end_main

C.2.4.2 Parallel Thread Section.

.sub_par0
.share

```
int S _val(?‘$$<clos>calValue{N-k}’?)
int j[S] [_blk]
```

.end_sh
.code

```
0     KTOPI %0
1     ADDI %1, %0, #1
2     MOVI %2, #[j_b0]
3     MOVI %3, #[j_n]
4     MOVI %4, #0
5     MOVI %5, #128
6     MOVI %12, %5
7     DECI %12, #1
8     MOVI %6, %1
9     MULI %7, %2, %1
10    ADDI %8, %7, %0
11    RDFW %2, 0:[a](;%8;#4)
12    MULI %9, %0, %1
13    ADDI %10, %9, %6
14    RDFW %3, 0:[a](;%10;#4)
15    MULF %3, %3, %2
16    ADDI %11, %7, %6
17    RDFW %4, 0:[a](;%11;#4)
18    DECF %5, %4, %3
19    LOCK 0:[a](;%11;#4)
20    WRFW 0:[a](;%11;#4), %5
21    UNLOCK 0:[a](;%11;#4)
22    INCI %6
```

```
23              JLT %6,%12,#13
24              INCI %2
25              JGE %2,%12,#28
26              INCI %4
27              JLT %4,%3,#9
28              RETURN
.end_code
.end_sub_par0
```

C.3 The SUN SPARC Instruction Set

The DSIMCLUSTER model supports a part of Sun Sparc instruction set format. The supported registers and instructions are listed in this section.

C.3.1 Register Sets

Table C.4: Sun SPARC special symbols and registers.

Symbol Object	Name	Comment
General-purpose (GP) registers	%r0 . . . %r31	
GP global registers	%g0 . . . %g7	Same as %r0 . . . %r7
GP 'out' registers	%o0 . . . %o7	Same as %r8 . . . %r15
GP local registers	%l0 . . . %l7	Same as %r16 . . . %r23
GP 'in' registers	%i0 . . . %i7	Same as %r24 . . . %r31
Stack-pointer register	%sp (%sp = %o6 = %r14)	
Frame-pointer register	%fp (%fp = %i6 = %r30)	

Note that all registers are 64-bit registers.

C.3.2 Instructions

The DSIMCLUSTER model supports twenty-eight instructions of the Sun Sparc V9 instruction set. The supported instructions are listed in TableC.5 below.

C.3.3 Example Object Files

This section presents a fraction of a CG workload object file implemented by using the Sun Sparc instruction set. This example shows the call to a `makea` subroutine from the main program. This example workload also demonstrates the use of a `SPAWN` magic instruction to create of multiple parallel threads (of `sub__$d1V644.makea`) in the subroutine `makea` to carry out some computations in parallel.

C.3.3.1 Main Program Data Sections

.desc

CG from NPB2.3 class A

.main

.shareW (bss)

float RCOND _val(0.1)
int NA _val(14000)
int NONZER _val(11)
int NITER _val(15)
float SHIFT _val(20.0)
int NZ _val(2198000)
int naa _val(14000)
int nzz _val(2198000)
int firstrow _val(1)
int lastrow _val(14000)
int firstcol _val(1)
int lastcol _val(14000)
int colidx[2198001]
int rowstr[14002]
int iv[28004]
int arow[2198001]
int acol[2198001]
double v[14002]
double aelt[2198001]
double a[2198001]

double x[14003]
double z[14003]
double p[14003]
double q[14003]
double r[14003]
double w[14003]
double amult _val(1220703125.0)
double tran _val(314159265.0)

.end_sh

.private _W (local)
int i _val(0)
int j _val(0)
int k _val(0)
int it _val(0)
int nthreads _val(1)
double zeta _val(0.0)
double rnorm _val(0.0)
double norm_temp11 _val(0.0)
double norm_temp12 _val(0.0)
double t _val(0.0)
double mflops _val(0.0)
double zeta_verify_value
double epsilon

.end_pr

C.3.3.2 Main Program Code Sections

.code

0 ld 0:[NA](;;),%r0
1 ld 0:[NONZER](;;),%r1
2 ld 0:[NITER](;;),%r2

```
3     ld 0:[SHIFT](;;),%r3
4     ld 0:[NZ](;;),%r4
5     st %r0,(%sp)
6     st %r1,(%sp)
7     st %r2,(%sp)
8     st %r0,0:[naa](;;)
9     st %r4,0:[nzz](;;)
10    st 10.36,1:[zeta_verify_value](;;)
11    st RANDOM{314159265;1220703125},1:[zeta](;;)
12    mov %r3, %sp
13    mov %r4, %sp
14    $$<clos>call{sub_makea}
      :
      : calls to the other subroutines
      :
      $$<pc>profiling{stop}
      exit
.end_code
.end_main
```

C.3.3.3 A Subroutine Section

```
.sub_makea
.private _W (data1)
            int i _val(0)
            int nnza _val(0)
            int iouter _val(0)
            int ivelt _val(0)
            int ivelt1 _val(0)
            int irow _val(0)
            int nzv _val(0)
            double size
            double ratio
```

```
                double scale
                int jcol
.end_pr
.code
0               ld 0:[naa](;;), %r0
1               st %r0,(%sp)
2               $$<clos>stacktop{int,n}
3               $$<clos>spawn{sub__$d1V644.makea}
4               ldd 0:[SHIFT](;;),%f0
5               ld 0:[iv](;;),%r1
6               ld 0:[v](;;),%r2
7               ld 0:[aelt](;;),%r3
8               ld 0:[acol](;;),%r4
9               ld 0:[arow](;;),%r5
10              ld 0:[RCOND](;;),%f1
11              ld 0:[lastcol](;;),%r6
12              ld 0:[firstcol](;;),%r7
13              ld 0:[lastrow](;;),%r8
14              ld 0:[firstrow](;;),%r9
15              ld 0:[NONZER](;;),%r10
16              ld 0:[rowstr](;;),%r11
17              ld 0:[colidx](;;),%r12
18              ld 0:[a](;;),%r13
19              ld 0:[nzz](;;),%r14
20              ret
.end_code
.end_sub_makea
```

C.3.3.4 A Parallel Thread Section

```
.sub__$d1V644.makea
.private
                        int ittr[n] [_blk]
```

```
.end_pr
.code
0                       mov %sp, i4
1                       add ittr_b0,1,%i0
2                       mov ittr_n,%i2
3                       add %i4,%i0, %i1
4                       acquire [colidx]
5                       st 0,[colidx](;%i1;4)
6                       release [colidx]
7                       add 1, %i0, %i0
8                       cmp %0, %i2
9                       bl 3
10                      ret
.end_code
.end_sub__$d1V644.makea()
```

C.4 DSIMCLUSTER Configuration

C.4.1 Example Configuration Files

C.4.2 DSIMCLUSTER's Counters and Plot files

Table C.5: The SPARC instructions implemented in DSIMCLUSTER.

Instruction	Mneumonic	Operands	Description
add	add	reg_rs1, reg_rs2, reg_rd	add
and	and	reg_rs1, reg_or_imm, reg_rd	logical and
be	be{,a}	target_addr	branch if equal
bg	bg{,a}	target_addr	branch if greater than
bge	bge{,a}	target_addr	branch if greater than or equal to
bl	bl	target_addr	branch if less than to target_addr
ble	ble{,a}	target_addr	branch if less than or equal to
bne	bne{,a}	target_addr	branch if not equal
call	call	target_name	call to subroutine
cmp	cmp	reg_rs, reg_rd	test if rs EQU rd
faddd	add	freg_rs1, freg_rs2, freg_rd	add
fdivd	fdivd	freg_rs1, freg_rs2, freg_rd	divide
flush	cmp	reg_rs, reg_rd	test if rs EQU rd
fmuld	fmuld	freg_rs1, freg_rs2, freg_rd	multiply
fsubd	fsubd	freg_rs1, freg_rs2, freg_rd	subtraction
ld	ld	mem_rs, reg_rd	load word
ldd	ldd	mem_rs, reg_rd	load double word
mov	mov	reg_rs, reg_rd	move word
or	or	reg_rs1, reg_or_imm, reg_rd	inclusive or
ret	ret		return to caller
sll	sll	reg_rs1, reg_or_imm, reg_rd	shift left logical
sra	sra	reg_rs1, reg_or_imm, reg_rd	shift right arithmatic
srl	srl	reg_rs1, reg_or_imm, reg_rd	shift right logical
st	ld	reg_rs, mem_rd	store word
sub	sub	reg_rs1, reg_rs2, reg_rd	subtraction
udiv	udiv	reg_rs1, reg_rs2, reg_rd	unsigned div
umul	umul	reg_rs1, reg_rs2, reg_rd	unsigned mul
xor	xor	reg_rs1, reg_or_imm, reg_rd	exclusive or

Table C.6: Examples of two configurations of cache organisation.

Parameter	Example 1	Example 2
Architecture	(A)	(B)
NoOfCacheLevels	(2)	(2)
NoOfWriteBufferLines	(64)	(64)
PhysicalAddressLength	(22)	(22)
VirtualAddressLength	(32)	(32)
MemoryPageSize	(4096)	(4096)
LEVEL(0){		
name	(On-chip D-Cache)	(On-chip D-Cache)
SetsPerWay	(1)	(1)
Indexing	(VIPT)	(VIPT)
NoOfTLBEntries	(256)	(256)
Associativity	(direct_mapping)	(direct_mapping)
ReplacementPolicy	(LFU)	(LFU)
WritePolicy	(write_through)	(write_through)
WriteMissPolicy	(no_write_allocate)	(no_write_allocate)
BytesPerLine	(16)	(16)
BytesPerWay	(16384)	(16384)
NoOfWays	(4)	(4)
CoherenceProtocol	(no)	(MESI)
CoherenceUnit	(4)	(4)
SharedCache	(no)	(no)
}		
LEVEL(1){		
name	(External Cache)	(External Cache)
SetsPerWay	(2)	(2)
Indexing	(PIPT)	(PIPT)
NoOfTLBEntries	(0)	(0)
Associativity	(set_associative)	(set_associative)
ReplacementPolicy	(LFU)	(LFU)
WritePolicy	(copy_back)	(copy_back)
WriteMissPolicy	(write_allocate)	(write_allocate)
BytesPerLine	(512)	(512)
BytesPerWay	(262144)	(262144)
NoOfWays	(16)	(16)
CoherenceProtocol	(MESI)	(no)
SharedCache	(no)	(yes)
}		

Table C.7: Example of a configuration file for the Profiler Entity.

```
CacheLevelMetrics    (ENABLE)/(DISABLE)
{
                     Metric(HitRate)
                     Metric(MissRate)
                     Metric(CacheUtilisation)
                     Metric(CacheMissCycleRate)
                     Metric(DataAccessRate)
                     Metric(NonReferedWordRate)
}

PENodeMetrics        (ENABLE)/(DISABLE)
{
                     Metric(InstructionIssueRate)
}

SMPNodeMetrics       (ENABLE)/(DISABLE)
{
//performing the average of PE node metrics if set to 'ENABLE'
}

DSMClusterMetrics    (ENABLE)/(DISABLE)
{
                     Metric(MemUtilisation)
                     Metric(ShMemAccsRoundtripTime)
                     Metric(PvMemAccsRoundtripTime)
                     Metric(AvgMemAccsLatency)
                     Metric(AvgMemWriteLatency)
                     Metric(AvgMemReadLatency)
                     Metric(ExecutionTime)
                     Metric(SpeedUp)
                     Metric(SlowDown)
                     Metric(SynchronisationCost)
                     Metric(ElapsedTime)
                     Metric(SystemTime)
}
```

Table C.8: Counters and plot files provided in DSiMCluster.

Index	**Counters of each architectural component**				
	Processor	**Cache Level**	**Cache hierarchy**	**Memory**	**Workload**
0	Instructions no.	TotalAccess	Idle cycles	TotalMemAccess	No. of Threads
1	Instruction executed	Read hits	Read hits	WriteAccess	Stall cycles
2	Read accesses	Read misses	Read misses	ReadAccess	Ctx switching stalls
3	Write accesses	Write hits	Write hits	SharedMemAccess	Accesses to stacks
4	I/O write accesses	Write misses	Write misses	SharedMemRead	Segments allocated
5	Branch to blk	Lines cached	Read miss stalls	SharedMemWrite	Page allocated
6	Branch from blk	Words accessed	Write miss stalls	PageFragment	Page faults
7	Total cycles	Access hits	Total cycles	Stall cycles	Segment access faults
8	Sync. Cycles	Access misses			
9	Avg. access cycles	HitToMissGap			
10		MissToHitGap			
11		TLB misses			
12		Avg. miss penalty			
13		Coherence Msg			
14		CacheLineSize			
15		Write accesses			
16		Read accesses	Auto-generated plot files		
17		Startup misses	Sim time *v* access latency		
18		Capacity misses	Sim time *v* no. of hits before a miss		
19		Inclus. False miss	Sim time *v* shared/privated read/write accesses		
20		Inclus. True miss	Sim time *v* degree of locality		
21		False misses	Sim time *v* access characterisation		
22		True misses	Sim time *v* cache misses breakdown		

Appendix D

Statistical Tables

D.1 Protocol Ownership Analysis

Table D.1: Paired Samples Statistics of Cache Misses (Protocol Ownership Tests).

Paired Samples Statistics

Cache Level	Factors	Mean	N	Std. Deviation	Std. Err Mean
L1	NOOWNER	15.8617	8	0.2610	0.0923
	OWNER	15.7927	8	0.1981	0.0700
L2	NOOWNER	6.1305	8	0.2148	0.0759
	OWNER	6.1990	8	0.2694	0.0953
Inclusion Misses					
L1	NOOWNER	5.3097	8	0.0422	0.0149
	OWNER	5.3040	8	0.0431	0.0152

Table D.2: Paired Samples Correlations of Cache Misses (Protocol Ownership Tests).

Paired Samples Correlations

No.	Factor	N	Correlation	Sig.
Pair 1	NOOWNER *v* OWNER L1	8	0.323	0.435
Pair 2	NOOWNER *v* OWNER L2	8	0.715	0.046
Inclusion Misses recorded at L1 Caches.				
Pair 3	NOOWNER *v* OWNER	8	-0.058	0.892

Table D.3: Paired Samples Statistics of Access Latency (Protocol Ownership Tests).

Paired Samples Statistics

No.	Factors	Mean	N	Std. Deviation	Std. Error Mean
Pair 1	NOOWNER	48.2525	16	10.2431	2.5608
	OWNER 1-Thread	36.6330	16	3.1217	0.7804
Pair 2	NOOWNER	41.4846	8	6.2927	2.2248
	OWNER 2-Thread	34.1168	8	2.1961	0.7764
Pair 3	NOOWNER	55.0203	8	8.9743	3.1729
	OWNER 4-Thread	39.1493	8	1.2598	0.4454

Table D.4: Paired Samples Correlations of Access Latency (Protocol Ownership Tests).

Paired Samples Correlations

No.	Factor	N	Correlation	Sig.
Pair 1	NOOWNER *v* OWNER 1-Thread	16	0.527	0.036
Pair 2	NOOWNER *v* OWNER 2-Thread	8	0.454	0.258
Pair 3	NOOWNER *v* OWNER 4-Thread	8	-0.807	0.016

Table D.5: Paired Samples Test Results on Data Access Latency of Coherence Protocol Ownership Study.

(a) Cache Misses

Cache OpenMP Level	Pair	Paired Differences					*t*	df	Sig. 2-tailed
		Mean	Std. Dev.	Std. Error Mean	95% Confidence Interval				
					Lower	Upper			
L1	NOOWNER *v* OWNER	0.06903	0.27199	0.09616	-0.15836	0.29642	0.718	7	0.496
L2	NOOWNER *v* OWNER	-0.06849	0.18960	0.06703	-0.22700	0.09001	-1.022	7	0.341
Inclusion Misses									
L1	NOOWNER *v* OWNER	0.00571	0.06206	0.02194	-0.04618	0.05759	0.260	7	0.802

(b) Data Access Latency

No. OpenMP Thread	Pair	Paired Differences					*t*	df	Sig. 2-tailed
		Mean	Std. Dev.	Std. Error Mean	95% Confidence Interval				
					Lower	Upper			
1	NOOWNER *v* OWNER	11.6194	8.9993	2.2498	6.8240	16.4148	5.165	15	0.000
2	NOOWNER *v* OWNER	7.3679	5.6452	1.9959	2.6484	12.0874	3.692	7	0.008
4	NOOWNER *v* OWNER	15.8709	10.0181	3.5419	7.4956	24.2462	4.481	7	0.003

D.2 Test of Dominant Factors

Table D.6: ANOVA Table for Tests of Dominant Factors.

Factor	df	Data Access Latency				Degree of Locality				Cache Misses Ratio			
		SS	MS	*F*	Sig.	SS	MS	*F*	Sig.	SS	MS	*F*	Sig.
DSM Architecture	1	91.012	91.012	1.366	.244	6.782	6.782	33.643	.000	1.802	1.802	1.789	.183
Cache Architecture	1	220.671	220.671	3.359	.069	.277	.277	1.118	.292	.744	.744	.733	.393
Coherence Unit	1	281.233	281.233	4.309	.040	.249	.249	1.006	.317	.007	.007	.007	.935
Bus Architecture	1	.040	.040	.001	.981	.446	.446	1.810	.181	.965	.965	.952	.331
Snoopy Protocol	1	44.223	44.223	.661	.418	.345	.345	1.398	.239	.000	.000	.000	.998
Directory Protocol	1	41.174	41.174	.615	.434	.023	.023	.093	.760	.381	.381	.375	.541
Interconnection Routing	1	1.412	1.412	.021	.885	.219	.219	.882	.349	.727	.727	.716	.399
Consistency Model	1	14.964	14.964	.223	.638	.096	.096	.384	.536	2.198	2.198	2.187	.141
Consistency Technique	1	155.875	155.875	2.356	.127	4.512	4.512	20.740	.000	1.569	1.569	1.554	.215
OpenMP Threads	1	6.605	6.605	.098	.754	1.902	1.902	8.062	.005	1.884	1.884	1.871	.174

D.3 Detailed Analysis of Dominant Factors

Table D.7: Detail Statistics of the Paired T-Test on False Misses.

Paired Samples Statistics

No.	Factors	Mean	N	Std. Deviation	Std. Error Mean
Pair 1	Small-scale SMPs	5,483.1389	36	5,499.30111	916.55018
	Medium-scale SMPs	5,308.5833	36	5,351.59152	891.93192
Pair 2	Small-scale SMPs	5,483.1389	36	5,499.30111	916.55018
	Large-scale SMPs	5,276.2778	36	5,288.87792	881.47965
Pair 3	Medium-scale SMPs	5,308.5833	36	5,351.59152	891.93192
	Large-scale SMPs	5,276.2778	36	5,288.87792	881.47965
Pair 4	Invalidate-based	5,622.4815	54	5,514.30979	750.40251
	Update-based	5,089.5185	54	5,178.89669	704.75857

Table D.8: Paired Samples Correlations of the Paired T-Test on False Misses.

Paired Samples Correlations

No.	Factor	N	Correlation	Sig.
Pair 1	Small *v* Medium Scale	36	1.000	0.000
Pair 2	Small *v* Large Scale	36	1.000	0.000
Pair 3	Medium *v* Large Scale	36	1.000	0.000
Pair 4	Invalidate *v* Update	54	1.000	0.000

Table D.9: Detail Statistics of the Paired T-Test on Access Latency.

Paired Samples Statistics

No.	Factors	Mean	N	Std. Deviation	Std. Error Mean
Pair 1	Small-scale SMPs	42.9894	36	1.91875	0.31979
	Medium-scale SMPs	39.8608	36	3.91990	0.65332
Pair 2	Small-scale SMPs	42.9894	36	1.91875	0.31979
	Large-scale SMPs	57.9578	36	3.41893	0.56982
Pair 3	Medium-scale SMPs	39.8608	36	3.91990	0.65332
	Large-scale SMPs	57.9578	36	3.41893	0.56982
Pair 4	Invalidate-based	49.1375	54	8.88590	1.20922
	Update-based	44.7344	54	7.65233	1.04135

Table D.10: Paired Samples Correlations of the Paired T-Test on Access Latency.

Paired Samples Correlations

No.	Factor	N	Correlation	Sig.
Pair 1	Small *v* Medium Scale	36	0.771	0.000
Pair 2	Small *v* Large Scale	36	0.723	0.000
Pair 3	Medium *v* Large Scale	36	0.462	0.005
Pair 4	Invalidate *v* Update	54	0.977	0.000

Table D.11: Paired Samples Test Results of the False Misses and Data Access Latency.

(a) False Misses

No.	Factor	Paired Differences					*t*	df	Sig.
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval				
					Lower	Upper			
Pair 1	Small *v* Medium Scale	174.55556	165.71860	27.61977	118.48445	230.62666	6.320	35	0.000
Pair 2	Small *v* Large Scale	206.86111	223.91467	37.31911	131.09929	282.62294	5.543	35	0.000
Pair 3	Medium *v* Large Scale	32.30556	78.61182	13.10197	5.70714	58.90397	2.466	35	0.019
Pair 4	Invalidate *v* Update	532.96296	365.13791	49.68898	632.62643	433.29950	10.726	53	0.000

(b) Data Access Latency

No.	Factor	Paired Differences					*t*	df	Sig.
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval				
					Lower	Upper			
Pair 1	Small *v* Medium Scale	3.12869	2.72888	0.45481	2.20537	4.05201	6.879	35	0.000
Pair 2	Small *v* Large Scale	-14.96833	2.42721	0.40454	-15.78958	-14.14708	-37.001	35	0.000
Pair 3	Medium *v* Large Scale	-18.09703	3.83061	0.63844	-19.39312	-16.80093	-28.346	35	0.000
Pair 4	Invalidate *v* Update	4.40309	2.14262	0.29157	4.98791	3.81827	15.101	53	0.000

Bibliography

[Abandah and Davidson, 1998] Abandah, G. and Davidson, E. (1998). Characterizing distributed shared memory performance: a case study of the Convex SPP1000. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):206–216.

[Acacio et al., 2002] Acacio, M. E., González, J., García, J. M., and Duato, J. (2002). Owner prediction for accelerating cache-to-cache transfer misses in a cc-NUMA architecture. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–12, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Acquaviva and Jalby, 2000] Acquaviva, J. T. and Jalby, W. (2000). Hardware prediction for data coherency of scientific codes on DSM. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 41, Washington, DC, USA. IEEE Computer Society.

[Adve and Gharachorloo, 1996] Adve, S. V. and Gharachorloo, K. (1996). Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76.

[Adve et al., 1999] Adve, S. V., Pai, V. S., and Ranganathan, P. (1999). Recent advances in memory consistency models for hardware shared memory systems. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):445–455.

[Agarwal and et. al., 1991] Agarwal, A. and et. al. (1991). The MIT Alewife machine : A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic.

[Agerwala et al., 1995] Agerwala, T., Martin, J. L., Mirza, J. H., Sadler, D. C., Dias, D. M., and Snir, M. (1995). SP2 system architecture. *IBM Systems Journal*, 34(2).

[Alam, 2004] Alam, S. (2004). *Simulation of the UKQCD Computer*. PhD thesis, Institute of Computing Systems Architecture, School of Informatics, University of Edinburgh.

[Ammar, 2005] Ammar, M. (2005). Why we STILL don't know how to simulate networks. In *In Proceedings of the 38th Annual Simulation Symposium*.

[Anderson et al., 2003] Anderson, J. H., Kim, Y.-J., and Herman, T. (2003). Shared-memory mutual exclusion: major research trends since 1986. *Distrib. Comput.*, 16(2-3):75–110.

[Austin et al., 2002] Austin, T., Larson, E., and Ernst, D. (2002). SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67.

[Baer and Wang, 1988] Baer, J.-L. and Wang, W.-H. (1988). On the inclusion properties for multi-level cache hierarchies. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 73–80. IEEE Computer Society Press.

[Baylor et al., 1997] Baylor, S., Ekanadham, K., Jann, J., Lim, B.-H., and Pattnaik, P. (1997). Lazy home migration for distributed shared memory systems. In *High Performance Computing, 1997. Proceedings. Fourth International Conference on*. IEEE.

[BBN Laboratories, 1986] BBN Laboratories (1986). Butterfly parallel processor overview. BBN Report 6149, BBN Laboratories, Cambridge, MA.

[Bell, 1999] Bell, G.; van Ingen, C. (1999). DSM perspective: another point of view. *Proceedings of the IEEE*, 87:3:412–417.

[Bennett et al., 1996] Bennett, A. J., Field, T., and Harrison, P. (1996). Modelling and validation of shared memory coherency protocols. *Performance Evaluation*, 27-28:541–563.

[Bilardi et al., 1996] Bilardi, G., Herley, K. T., Pietracaprina, A., Pucci, G., and Spirakis, P. (1996). BSP vs LogP. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 25–32, New York, NY, USA. ACM Press.

[Bilas et al., 2003] Bilas, A., Jiang, D., and Singh, J. P. (2003). Shared virtual memory clusters: bridging the cost-performance gap between SMPs and hardware DSM systems. *J. Parallel Distrib. Comput.*, 63(12):1257–1276.

[Bilas et al., 1999] Bilas, A., Jiang, D., Zhou, Y., and Singh, J. P. (1999). Limits to the performance of software shared memory: a layered approach. In *Fifth International Symposium On High-Performance Computer Architecture*, pages 193–202. IEEE.

[Brewer et al., 1991] Brewer, E. A., Dellarocas, C. N., Colbrook, A., and Weihl, W. E. (1991). PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts.

[Brunie and Lefevre, 1996] Brunie, L. and Lefevre, L. (1996). A DSM-based structural programming environment for distributed and parallel processing. In *in Proceedings of the Third International Conference on High-Performance Computing (HiPC '96)*.

[Bull and O'Neill, 2001] Bull, J. M. and O'Neill, D. (2001). A microbenchmark suite for OpenMP 2.0. *SIGARCH Comput. Archit. News*, 29(5):41–48.

[Buyya and Murshed, 2002] Buyya, R. and Murshed, M. (2002). GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing. *The Journal of Concurrency and Computation: Practice and Experience (CCPE)*, 14. Wiley Press.

[Carlson et al., 1999] Carlson, W. W., Draper, J. M., Culler, D. E., Yelick, K., Brooks, E., and Warren, K. (1999). Introduction to UPC and language specification. Ccs-tr-99-157, LLNL.

[Casanova, 2001] Casanova, H. (2001). SimGrid: A toolkit for the simulation of application scheduling. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 430. IEEE Computer Society.

[Cassandras and Lafortune, 1999] Cassandras, C. and Lafortune, S. (1999). *Introduction to Discrete Event Systems*. Kluwer Academic Publishers.

[CCGrid, 2005] CCGrid (2005). *The 5th IEEE Symposium on Cluster Computing and the Grid (CCGrid 2005)*. IEEE/Computer Society.

[Chame and Dubois, 1993] Chame, J. and Dubois, M. (1993). Cache inclusion and processor sampling in multiprocessor simulations. In *SIGMETRICS '93: Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 36–47. ACM Press.

[Charlesworth, 2001] Charlesworth, A. (2001). The sun fireplane system interconnect. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 7–7. ACM Press.

[Chaudhuri et al., 2003] Chaudhuri, M., Heinrich, M., Holt, C., Singh, J. P., Rothberg, E., and Hennessy, J. (2003). Latency, occupancy, and bandwidth in DSM multiprocessors: A performance evaluation. *IEEE Transactions on Computers*, 52(7):862–880.

[Cierniak and Li, 1995] Cierniak, M. and Li, W. (1995). Unifying data and control transformations for distributed shared-memory machines. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 205–217, New York, NY, USA. ACM Press.

[Clarke, 2004] Clarke, B. (2004). Solemn: Solaris emulation mode for Sparc Sulima. In *In Proceedings of the 37th Annual Simulation Symposium*, pages 64–71.

[Clarke et al., 2002] Clarke, B., Czezowski, A., and Strazdins, P. (2002). Implementation aspects of Sparc v9 complete machine simulator. In *Proceedings of the 25th Australasian Computer Science Conference*, pages 23–32. Monash University.

[Coe et al., 1998] Coe, P., Howell, F., Ibbett, R., and L.M.Williams (1998). A hierarchical computer architecture design and simulation environment. *ACM Transactions on Modeling and Computer Simulation*, 8(4):431 – 446.

[Coe, 2000] Coe, P. S. (2000). *Simulation Model of Shared-Memory Multiprocessor System*. PhD thesis, University of Edinburgh, United Kingdom.

[Connelly and Ellis, 1995] Connelly, C. and Ellis, C. S. (1995). A workload characterization for coarse-grain multiprocessors. In *The 9th International Symposium on Parallel Processing*, pages 393–397. IEEE.

[Coulouris et al., 2001] Coulouris, G., Dollimore, J., and Kindberg, T. (2001). *Distributed Systems Concepts and Design*, chapter 16 Distributed Shared Memory. Addison-Wesley, third edition edition.

[Courtney and Chevalier, 2004] Courtney, T. and Chevalier, F. (2004). Work in progress: Low level simulation of back-end storage network using HASE. In *In Proceedings of the IEEE conference on Mass Storage systems and Technology*, Maryland, USA.

[Culler et al., 1993] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. (1993). LogP: towards a realistic model of parallel computation. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, New York, NY, USA. ACM Press.

[Davis and Goldschmidt, 1990] Davis, H. and Goldschmidt, S. R. (1990). Tango: A multiprocessor simulation and tracing system. Technical Report CSL-TR-90-439, Stanford University, Computer Systems Laboratory.

[Delzanno, 2003] Delzanno, G. (2003). Constraint-based verification of parameterized cache coherence protocols. *Form. Methods Syst. Des.*, 23(3):257–301.

[DeRose et al., 2002] DeRose, L., Ekanadham, K., and Hollingsworth, J. K. (2002). SIGMA: A simulator infrastructure to guide memory analysis. In *Conference on High Performance Networking and Computing (SC2002)*, Baltimore, USA.

[Desikan et al., 2001] Desikan, R., Burger, D., and Keckler, S. W. (2001). Measuring experimental error in microprocessor simulation. In *Proceedings of the 2001 symposium on Software reusability*, pages 266–277. ACM Press.

[Dill et al., 1992] Dill, D. L., Drexler, A. J., Hu, A. J., and Yang, C. H. (1992). Protocol verification as a hardware design aid. In *ICCD '92: Proceedings of the 1991*

IEEE International Conference on Computer Design on VLSI in Computer & Processors, pages 522–525. IEEE Computer Society.

[Dongarra et al., 2005] Dongarra, J., Sterling, T., Simon, H., and Strohmaier, E. (2005). High-performance computing: Clusters, constellations, MPPs, and future directions. *Computing in Science & Engineering*, 7(2):51–59.

[Dwarkadas et al., 1999] Dwarkadas, S., Hardavellas, N., Kontothanassis, L., Nikhil, R., and Stets, R. (1999). Cashmere-VLM: Remote memory paging for software distributed shared memory. In *13th International and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, pages 153–159.

[Eskicioglu et al., 1999] Eskicioglu, M., Marsland, T., Hu, W., and Shi, W. (1999). Evaluation of the JIAJIA software DSM system on high performance computer architectures. In *Proceedings of the 32nd Annual Hawaii International Conference on*.

[Field et al., 1998] Field, A., Harrison, P., and Kanani, K. (1998). Automatic generation of verifiable cache coherence simulation models from high-level specifications. In *Australian Computer Science Communications*, volume 20, pages 261–275.

[Figueiredo and Fortes, 2000] Figueiredo, R. and Fortes, J. (2000). Impact of heterogeneity on DSM performance. In *Proceedings the Sixth International Symposium on High-Performance Computer Architecture (HPCA-6)*, pages 26–35.

[Flynn, 1995a] Flynn, M. J. (1995a). *Computer Architecture: Pipelined and Parallel Processor Design*, chapter 8 Shared Memory Multiprocessor. Jones and Bartlett Publishers, Barb House, Barb Mews, London, W6 7PA.

[Flynn, 1995b] Flynn, M. J. (1995b). *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett Publishers, Barb House, Barb Mews, London, W6 7PA.

[Foglia, 2001] Foglia, P. (2001). An algorithm for the classification of coherence related overhead in shared-bus shared-memory multiprocessors. *IEEE TCCA Newsletter*.

[Foster and Kesselman, 1998] Foster, I. and Kesselman, C., editors (1998). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, first edition edition.

[Frank et al., 1993] Frank, S., Burkhardt, H., I., and Rothnie, J. (1993). The KSR 1: bridging the gap between shared memory and MPPs. In *Compcon Spring'93*, pages 285–294. IEEE. Digest of Papers.

[Fuller et al., 1973] Fuller, S. H., Siewiorek, D. P., and Swan, R. J. (1973). Computer modules: An architecture for large digital modules. In *ISCA '73: Proceedings of the 1st annual symposium on Computer architecture*, pages 231–237. ACM Press.

[Gharachorloo et al., 1998] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. (1998). Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 376–387. ACM Press.

[Grbic, 2003] Grbic, A. (2003). *Assessment of Cache Coherence Protocols in Shared-Memory Multiprocessors*. PhD thesis, Graduate Department of Electrical and Computer Engineering, University of Toronto.

[Grujić et al., 1996] Grujić, A., Tomaŝević, M., and Milutinović, V. (1996). A simulation study of hardware-oriented DSM approaches. *the IEEE Parallel & Distributed Technology: Systems & Applications*, 4(11):74–83.

[Gustafson and Todi, 1998] Gustafson, J. L. and Todi, R. (1998). Conventional benchmarks as a sample of the performance spectrum. In *in the Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, pages 514–523.

[Heinrich et al., 1999] Heinrich, M., Soundararajan, V., Hennessy, J., and Gupta, A. (1999). A quantitative analysis of the performance and scalability of distributed shared memory cache coherence protocols. *IEEE Transactions on Computers*, 48(2):205–217.

[Hennessy et al., 1999] Hennessy, J., Heinrich, M., and Gupta, A. (1999). Cache-coherent distributed shared memory: perspectives on its development and future challenges. *Proceedings of the IEEE*, 87(3):418–429.

[Herrod, 1993] Herrod, S. A. (1993). Tango lite: A multiprocessor simulation environment introduction and user's guide. Technical report, Computer Systems Laboratory, Stanford University.

[Herrod, 1998] Herrod, S. A. (1998). *Using Complete Maching Simulation to Understand Computer System Behavior*. PhD thesis, Department of Computer Science, Standford University.

[Holt et al., 1996] Holt, C., Singh, J. P., and Hennessy, J. (1996). Application and architectural bottlenecks in large scale distributed shared memory machines. In *ISCA '96: Proceedings of the 23rd annual international symposium on Computer architecture*, pages 134–145. ACM Press.

[Howell, 1997] Howell, F. (1997). HASE++(version 1.0) a discrete event simulation library for c++. http://www.dcs.ed.ac.uk/home/hase/userguide/edl/hasepp.ps.

[Howell and McNab, 1998] Howell, F. and McNab, R. (1998). SimJava: a discrete event simulation package for java with applications in computer systems modelling. In *The First International Conference on Web-based Modelling and Simulation*. Society for Computer Simulation.

[HPF Forum, 1997] HPF Forum, editor (1997). *High Performance Fortran Language Specification Version 2.0*. Rice University.

[Hristea et al., 1997] Hristea, C., Lenoski, D., and Keen, J. (1997). Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–12, New York, NY, USA. ACM Press.

[Hsiao and King, 2000] Hsiao, H.-C. and King, C.-T. (2000). MICA: A memory and interconnect simulation environment for cache-based architectures. In *the 33rd IEEE Annual Simulation Symposium (SS 2000)*, pages 317–325. IEEE.

[Hsieh et al., 1996] Hsieh, W. C., Kaashoek, M. F., and Weihl, W. E. (1996). Dynamic computation migration in DSM systems. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 44.

[Huang, 2000] Huang, J. (2000). The simulator for multithreaded computer architecture release 1.2. Technical Report ARCTiC-00-05, University of Minnesota.

[Hughes et al., 2002] Hughes, C., Pai, V., Ranganathan, P., and Adve, S. (2002). RSIM: simulating shared-memory multiprocessors with ilp processors. *Computer*, 35(2):40–49.

[Hwang, 1993] Hwang, K. (1993). *Advanced Computer Architecture Parallelism Scalability Programmability*, chapter 5 Bus, Cache, and Shared Memory. McGraw-Hill, international edition edition.

[IEEE-SA Standards Board, 1993] IEEE-SA Standards Board (1993). IEEE standard for scalable coherent interface (SCI). Ieee std 1596-1992, IEEE.

[Ikodinovic et al., 1999] Ikodinovic, I., Milenkovic, A., and Milutinovic, V. (1999). Limes: A multiprocessor simulation environment for PC platform. In *PRAM'99*, Kazimierz Dolny, Poland.

[Iosevich and Schuster, 2004] Iosevich, V. and Schuster, A. (2004). A comparison of sequential consistency with home-based lazy release consistency for software distributed shared memory. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 306–315. ACM Press.

[Jain, 1991] Jain, R. (1991). *The art of Computer System Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. John Wiley & Sons, Inc.

[Jégou, 2003] Jégou, Y. (2003). Implementation of page management in Mome, a user-level DSM. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, pages 479– 486. IEEE Computer Society.

[Jiang and Chaudhary, 2004] Jiang, H. and Chaudhary, V. (2004). Process/thread migration and checkpointing in heterogeneous distributed systems. In *In Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*.

[Jin et al., 1999] Jin, H., M.Frumkin, and Yan, J. (1999). The OpenMP implementation of NAS parallel benchmarks and its performance. NAS Technical Report NAS-99-011, NASA Ames Research Center.

[Kee et al., 2004] Kee, Y.-S., Kim, J.-S., and Ha, S. (2004). Memory management for multi-threaded software DSM systems. *Parallel Computing*, 30:121–138.

[Keleher et al., 1992] Keleher, P., Cox, A. L., and Zwaenepoel, W. (1992). Lazy release consistency for software distributed shared memory. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 13–21, New York, NY, USA. ACM Press.

[Kodi, 2005] Kodi, A. K. (2005). Design of a high-speed optical interconnect for scalable shared memory multiprocessors. *IEEE Micro Special Issue Hot Interconnects 12*, 25(1):41–49.

[Kontothanassis et al., 1997] Kontothanassis, L., Hunt, G., Stets, R., Hardavellas, N., Cierniak, M., Parthasarathy, S., Meira, Jr., W., Dwarkadas, S., and Scott, M. (1997). VM-based shared memory on low-latency, remote-memory-access networks. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 157–169, New York, NY, USA. ACM Press.

[Kudlur and Govindarajan, 2004] Kudlur, M. and Govindarajan, R. (2004). Performance analysis of methods that overcome false sharing effects in software DSMs. *J. Parallel Distrib. Comput.*, 64(8):887–907.

[Lai and Falsafi, 1999] Lai, A.-C. and Falsafi, B. (1999). Memory sharing predictor: the key to a speculative coherent DSM. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 172–183. IEEE Computer Society.

[Lai and Falsafi, 2000] Lai, A.-C. and Falsafi, B. (2000). Comparing the effectiveness of fine-grain memory caching against page migration/replication in reducing traffic in DSM clusters. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 79–88. ACM Press.

[Lamport, 1979] Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*.

[Law and Kelton, 1991] Law, A. M. and Kelton, W. D. (1991). *Simulation Modeling & Analysis*, chapter 1 Basic Simulation Modeling, pages 8–10, 116–117. McGraw-Hill, Inc, second edition. Time-Advance Mechanism.

[Li, 1986] Li, K. (1986). *Shared virtual memory on loosely coupled multiprocessors*. PhD thesis, Yale University.

[Liang et al., 2002] Liang, T.-Y., Shieh, C.-K., and Li, J.-Q. (2002). Selecting threads for workload migration in software distributed shared memory systems. *Parallel Comput.*, 28(6):893–913.

[Lin et al., 2002] Lin, C.-Y., Liu, J.-S., and Chung, Y.-C. (2002). Efficient representation scheme for multidimensional array operations. *IEEE Trans. Comput.*, 51(3):327–345.

[Liu et al., 2004] Liu, Y.-T., Liang, T.-Y., Kuo, Z.-H., and Shieh, C.-K. (2004). Involving memory resource consideration into workload distribution for software DSM systems. In *CCGrid 2004: Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, pages 395–402.

[Lucas, Jr., 1971] Lucas, Jr., H. C. (1971). Performance evaluation and monitoring. *Computing Surveys*, 3(3):79–91.

[Magnusson and Werner, 1995] Magnusson, P. and Werner, B. (1995). Efficient memory simulation in SIMICS. In *Proceedings of the 28th Annual Simulation Symposium*, pages 62–73, Santa Barbara, California. IEEE.

[Magnusson and et al., 2002] Magnusson, P. S. and et al. (2002). Simics: A full system simulation platform. *Computer*, 35(2):50–58.

[Mallet et al., 2002] Mallet, F., Alam, S., and Ibbett, R. (2002). An extensible clock mechanism for computer architecture simulations. In *Internation Conference on Modeling and Simulation*, Marina del Rey, California, USA.

[Manoj et al., 2004] Manoj, N. P., Manjunath, K. V., and Govindarajan, R. (2004). CAS-DSM: a compiler assisted software distributed shared memory. *Int. J. Parallel Program.*, 32(2):77–122.

[Marathe et al., 2004] Marathe, J., Nagarajan, A., and Mueller, F. (2004). Detailed cache coherence characterization for OpenMP benchmarks. In *ICS'04*. ACM.

[Marco Ballette, 2005] Marco Ballette, Antonio Liotta, S. M. R. (2005). Execution time prediction in DSM-based mobile grids. In *In Proceedings of the 5th IEEE Symposium on Cluster Computing and the Grid (CCGrid 2005)*.

[Mascarenhas et al., 1995] Mascarenhas, E., Knop, F., and Rego, V. (1995). ParaSol: A multithreaded system for parallel simulation based on mobile threads. In *1995 Winter Simulation Conference*. SCS.

[Mason et al., 2003] Mason, R. L., Gunst, R. F., and Hess, J. L. (2003). *Statistical Design and Analysis of Experiments*, chapter 7. Fractional Factorial Experiments, pages 228–270. John Wiley & Sons, Inc., second edition edition.

[Mauer et al., 2002] Mauer, C. J., Hill, M. D., and Wood, D. A. (2002). Full-system timing-first simulation. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 108–116. ACM Press.

[Miller et al., 2004] Miller, J., Baramidze, G., Sheth, A., and Fishwick, P. (2004). Investigating ontologies for simulation modeling. In *In Proceedings of the 37th Annual Simulation Symposium*, pages 55–63.

[Milojíčić et al., 2000] Milojíčić, D., Douglis, F., Paindaveine, Y., Wheeler, R., and Zhou, S. (2000). Process migration. *ACM Comput. Surv.*, 32(3):241–299.

[Morin and Puaut, 1997] Morin, C. and Puaut, I. (1997). A survey of recoverable distributed shared virtual memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 8:959–969.

[Mukherjee et al., 2000] Mukherjee, S., Reinhardt, S., Falsafi, B., Litzkow, M., Hill, M., Wood, D., Huss-Lederman, S., and Larus, J. (2000). Wisconsin Wind Tunnel II: a fast, portable parallel architecture simulator. *Concurrency, IEEE [see also IEEE Parallel & Distributed Technology]*, 8:12–20.

[Mukherjee et al., 1997] Mukherjee, S. S., Reinhardt, S. K., Falsafi, B., Litzkow, M., and Huss-Lederman, S. (1997). Wisconsin Wind Tunnel II: A fast and portable parallel architecture simulator. In *Workshop on Performance Analysis and Its Impact on Design (PAID)*.

[Nair, 2002] Nair, R. (2002). Effect of increasing chip density on the evolution of computer architectures. *IBM Journal of Research and Development*, 46(2/3) Scaling CMOS to the Limits.

[Nanda et al., 1998] Nanda, A. K., Hu, Y., Ohara, M., and Benveniste, C. D. (1998). The design of compass: An execution driven simulator for commercial applications running on shared memory multiprocessors. In *The 12th. International Parallel Processing Symposium (IPPS'98)*, Orlando, Florida. Computer Society.

[Ng and Wong, 1999] Ng, M. C. and Wong, W. F. (1999). Adaptive schemes for home-based DSM systems. In *Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*.

[Nguyen et al., 1996] Nguyen, A.-T., Michael, M., Sharma, A., and Torrellas, J. (1996). The Augmint multiprocessor simulation toolkit for Intel x86 architectures. In *Proceedings of 1996 International Conference on Computer Design*.

[Nieplocha et al., 2005] Nieplocha, J., Krishnan, M., Palmer, B., Tipparaju, V., and Zhang, Y. (2005). Exploiting processor groups to extend scalability of the GA shared memory programming model. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 262–272, New York, NY, USA. ACM Press.

[Nikolopoulos, 2003] Nikolopoulos, D. S. (2003). Quantifying contention and balancing memory load on hardware DSM multiprocessors. *J. Parallel Distrib. Comput.*, 63(9):866–886.

[Nitzberg and Lo, 1991] Nitzberg, B. and Lo, V. (1991). Distributed shared memory: A survey of issues and algorithms. *Computer*.

[Niwa et al., 2000] Niwa, J., Matsumoto, T., and Hiraki, K. (2000). Comparative study of page-based and segment-based software DSM through compiler optimization. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 284–295, New York, NY, USA. ACM Press.

[Numrich and Reid, 1998] Numrich, R. and Reid, J. (1998). Co-Array Fortran for parallel programming. *Fortran Forum*.

[OpenMP, 2002] OpenMP (2002). OpenMP C and C++ application program interface version 2. Technical report, OpenMP Architecture Review Board.

[Pai et al., 1997] Pai, V. S., Ranganathan, P., and Adve, S. V. (1997). RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. *IEEE TCCA Newsletter*.

[Pinto et al., 2003] Pinto, R., Bianchini, R., and Amorim, C. (2003). Comparing latency-tolerance techniques for software DSM systems. *Parallel and Distributed Systems, IEEE Transactions on*, 14:1180–1190.

[Pong and Dubois, 1997] Pong, F. and Dubois, M. (1997). Verification techniques for cache coherence protocols. *ACM Comput. Surv.*, 29(1):82–126.

[Pong and Dubois, 2000] Pong, F. and Dubois, M. (2000). Formal automatic verification of cache coherence in multiprocessors with relaxed memory models. *IEEE Trans. Parallel Distrib. Syst.*, 11(9):989–1006.

[Pousa et al., 2005] Pousa, C., Goes, L., and Martins, C. (2005). Reconfigurable object consistency model. In *19th IEEE International Symposium on Parallel and Distributed Processing*. IEEE.

[Protić et al., 1995] Protić, J., Tomaŝević, M., and Milutinović, V. (1995). A survey of distributed shared memory systems. In *Proceedings of the 28th IEEE/ACM Hawaii International Conference on System Sciences*, pages 74–84. IEEE.

[Qin and Baer, 1997] Qin, X. and Baer, J.-L. (1997). A performance evaluation of cluster architectures. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 237–247. ACM Press.

[Raina, 1992] Raina, S. (1992). Virtual shared memory: A survey of techniques and systems. Technical report, University of Bristol, Bristol, UK, UK.

[Reinhardt et al., 1993] Reinhardt, S. K., Hill, M. D., Larus, J. R., Lebeck, A. R., Lewis, J. C., and Wood, D. A. (1993). The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. *SIGMETRICS Perform. Eval. Rev.*, 21(1):48–60.

[Robinson et al., 2004] Robinson, S., Nance, R., Paul, R., Pidd, M., and Taylor, S. (2004). Simulation model reuse: definitions, benefits, and obstacles. *Simulation Modelling Practice and Theory*, 12:479–494.

[Roh et al., 2000] Roh, Y., Seong, B. H., and Park, D. (2000). Hiding latency through bulk transfer and prefetching in distributed shared memory multiprocessors. In *In Proceedings of the Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, pages 164 – 166.

[Salehi et al., 1995] Salehi, J., Kurose, J., and Towsley, D. (1995). The performance impact of scheduling for cache affinity in parallel network processing. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing 1995*, pages 66–77. IEEE.

[Sandri et al., 2004] Sandri, A. L., Gonçalves, R. A. L., and ao A. Martini, J. (2004). SMS - tool for development and performance analysis of parallel applications. In *Proceedings of the 37th Annual Simulation Symposium*, page 196. IEEE Computer Society.

[Shan and Singh, 2000] Shan, H. and Singh, J. P. (2000). A comparison of three programming models for adaptive applications on the Origin2000. In *Proceedings of SC2000*, Dallas, TX, USA.

[Shen, 2000] Shen, X. (2000). *Design and Verification of Adaptive Cache Coherence Protocols*. PhD thesis, Massachusetts Institute of Technology, The Stata Center, 32 Vassar Street, Cambridge, Massachusetts 02139.

[Shi et al., 1997] Shi, W., Hu, W., and Tang, Z. (1997). An interaction of coherence protocols and memory consistency models in DSM systems. *SIGOPS Oper. Syst. Rev.*, 31(4):41–54.

[Shi and Tang, 1998] Shi, W. and Tang, Z. (1998). Using confidence interval to summarize the evaluation results: A case study. In *Technical Committee on Computer Architecture (TCCA) Newsletter*. IEEE Computer Society.

[Singh et al., 1992] Singh, J., Weber, W.-D., and Gupta, A. (1992). Splash: Stanford parallel applications for shared-memory. In *Computer Architecture News*.

[Singhal and Shivaratri, 1994] Singhal, M. and Shivaratri, N. G. (1994). *Advanced Concepts in Operating Systems Distributed, Database, and Multiprocessor Operating Systems*, chapter 10 Distributed Shared Memory, pages 236–258. McGraw-Hill, Inc., international edition edition.

[Skadron et al., 2003] Skadron, K., Martonosi, M., August, D. I., Hill, M. D., Lilja, D. J., and Pai, V. S. (2003). Challenges in computer architecture evaluation. *Computer*, 36(8):30–36.

[Sorin et al., 2002] Sorin, D. J., Plakal, M., Condon, A. E., Hill, M. D., Martin, M. M. K., and Wood, D. A. (2002). Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. *IEEE Trans. Parallel Distrib. Syst.*, 13(6):556–578.

[Steinke and Nutt, 2004] Steinke, R. C. and Nutt, G. J. (2004). A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849.

[Stoy et al., 2001] Stoy, J. E., Shen, X., and Arvind (2001). Proofs of correctness of cache-coherence protocols. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 43–71. Springer-Verlag.

[Stumm and Zhou, 1990] Stumm, M. and Zhou, S. (1990). Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5):54–64.

[Sulistio et al., 2005] Sulistio, A., Poduvaly, G., Buyya, R., and Tham, C.-K. (2005). Constructing a Grid simulation with differentiated network service using GridSim. In *Proc. of the 6th International Conference on Internet Computing (ICOMP'05)*, Las Vegas, USA.

[Sulistio et al., 2004] Sulistio, A., Yeo, C. S., and Buyya, R. (2004). A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools. *International Journal of Software: Practice and Experience*, 34:653–673. Wiley Press.

[Sun Microsystems, 2002] Sun Microsystems (2002). *Program Performance Analysis Tools Forte Developer 7*. Sun Microsystems.

[Sunada et al., 1998] Sunada, D., Glasco, D., and Flynn, M. (1998). ABSS v2.0: a Sparc simulator. In *the proceedings of the 8th Workshop on Synthesis And System Integration of Mixed Technologies (SASIMI '98)*.

[Szwed and et al., 2004] Szwed, P. K. and et al. (2004). SimSnap: Fast-forwarding via native execution and application-level checkpointing. In *in the Proceedings of*

the Eighth Workshop on Interaction between Compilers and Computer Architectures (INTERACT'04), pages 65–74.

[Takefusa et al., 1999] Takefusa, A., Matsuoka, S., Nakada, H., Aida, K., and Nagashima, U. (1999). Overview of a performance evaluation system for global computing scheduling algorithms. In *Proceedings of 8th IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 97–104. IEEE Computer Society.

[Talbot, 1999] Talbot, S. A. (1999). *Shared-Memory Multiprocessors with Stable Performance*. PhD thesis, Department of Computing, Imperial College, London.

[Tanenbaum, 1995] Tanenbaum, A. S. (1995). *Distributed Operating Systems*. Prentice-Hall, Inc.

[Tao et al., 2003] Tao, J., Schulz, M., and Karl, W. (2003). A simulation tool for evaluating shared memory systems. In *In Proceedings of the 36th Annual Simulation Symposium*.

[Tao et al., 2005] Tao, J., Schulz, M., and Karl, W. (2005). SIMT/OMP: A toolset to study and exploit memory locality of OpenMP applications on NUMA architectures. *Lecture Notes in Computer Science*, 3349:41–52.

[Tasiran et al., 2003] Tasiran, S., Yu, Y., and Batson, B. (2003). Using a formal specification and a model checker to monitor and direct simulation. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 356–361. ACM Press.

[Thaker and Chaudhary, 2003] Thaker, D. and Chaudhary, V. (2003). DSM*Sim*: A distributed shared memory simulator for clusters of symmetric multi-processors. In *Proceedings of the International Conference on Parallel and Distributed, Processing Techniques and Applications, PDPTA '03*, volume 4. CSREA Press.

[Thaker and Chaudhary, 2005] Thaker, D. D. and Chaudhary, V. (2005). Simulation tools to study a distributed shared memory for clusters of symmetric multiprocessors. *Future Generation Computer Systems*. In Press, Corrected Proof, Available online 27 June 2005.

[Thiyagalingam and Kelly, 2002] Thiyagalingam, J. and Kelly, P. H. J. (2002). Is Morton layout competitive for large two-dimensional arrays? In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 280–288, London, UK. Springer-Verlag.

[Tikir and Hollingsworth, 2004] Tikir, M. M. and Hollingsworth, J. K. (2004). Using hardware counters to automatically improve memory performance. In *SC '04: Proceedings of the Proceedings of the ACM/IEEE SC2004 Conference (SC'04)*, page 46, Washington, DC, USA. IEEE Computer Society.

[TIS Committee, 1995] TIS Committee (1995). Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2. Technical report, TIS Committee.

[Vachharajani et al., 2004] Vachharajani, M., Vachharajani, N., and August, D. I. (2004). The Liberty structural specification language: a high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 195–206. ACM Press.

[Valiant, 1990] Valiant, L. G. (1990). A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111.

[Veenstra and Fowler, 1994] Veenstra, J. E. and Fowler, R. J. (1994). MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 201–207.

[Verghese et al., 1996] Verghese, B., Devine, S., Gupta, A., and Rosenblum, M. (1996). Operating system support for improving data locality on CC-NUMA compute servers. In *In Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 279–289.

[Wainer et al., 2002] Wainer, G., Morihama, L., and Passuello, V. (2002). Automatic verification of DEVS models. In *Proceedings of SISO Spring Interoperability Workshop*. Simulation Interoperability Standards Organization's (SISO).

[Walsh and Sirer, 2004] Walsh, K. and Sirer, E. G. (2004). Staged simulation: A general technique for improving simulation scale and performance. *ACM Trans. Model. Comput. Simul.*, 14(2):170–195.

[Wilkinson and Allen, 1999] Wilkinson, B. and Allen, M. (1999). *Parallel Programming Techniques and Applications using Networked Workstations and Parallel Computers*. Prentice Hall.

[Woo et al., 1995] Woo, S., Ohara, M., Torrie, E., Singh, J., and Gupta, A. (1995). The SPLASH-2 Programs: Characterization and Methodological Considerations. In *22nd International Symposium on Computer Architecture*.

[Wunderlich et al., 2003] Wunderlich, R. E., Wenisch, T. F., Falsafi, B., and Hoe, J. C. (2003). SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. *SIGARCH Comput. Archit. News*, 31(2):84–97.

[Yelick et al., 1998] Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. (1998). Titanium: A high-performance java dialect. In *ACM 1998 Workshop on Java for High Performance Network Computing*, Stanford, CA.

[Yu et al., 2004] Yu, B.-H., Huang, Z., Cranefield, S., and Purvis, M. (2004). Homeless and home-based lazy release consistency protocols on distributed shared memory. In *Proceedings of the 27th conference on Australasian computer science*, pages 117–123. Australian Computer Society, Inc.

[Zeigler et al., 2000] Zeigler, B. P., Praehofer, H., and Kim, T. G. (2000). *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, chapter 4 Introduction to Discrete Event System Specification (DEVS). Academic Press, A Harcourt Science and Technology Company 525 B Street, Suite 1900, San Diego, CA 92101-4495, USA, 2nd edition.

Index