Simulation of the UKQCD Computer

Sadaf Alam

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2004

Abstract

Software simulations are extensively used in the investigation and exploration of complex systems and in the verification and advancement of scientific theories. Nowadays, simulation models are an indispensable exploratory tool, considered as vital as theoretical and observational methodologies in scientific research. Present supercomputing resources are not enough for a number of critical scientific simulations, collectively known as 'grand challenge' applications. Developing supercomputers for high-end scientific applications is itself a grand challenge. Increasingly complex system architectures, the enormous design space of supercomputer systems and complicated hardware-software interactions pose phenomenal design challenges for system architects. A flexible, cost-effective and efficient simulation framework is required in order to get an insight into the performance characteristics of high-end systems.

HASE, the Hierarchical computer Architecture design and Simulation Environment, allows for parameterised prototyping of computing systems at multiple abstraction levels encompassing system hardware and application software components. The UKQCD computer simulation research aims to explore the design space and to investigate the performance restricting features of a recent, application-specific supercomputer called QCDOC. Quantum Chromodynamics (QCD) is a particle physics theory and a 'grand challenge' application. The QCDOC (QCD-On-Chip) computer employs IBM System-On-Chip technology for a Teraflop-scale supercomputer design.

An application-driven simulation approach is introduced for application-specific supercomputer modelling in HASE. Parameterised hardware-software co-simulation models of the QCDOC machine have been created. Advantages of the application-driven co-simulation include a wider design space exploration of hardware components, not restricted by static workload configurations, and simulation metamodelling. Workload scalability and load balancing experiments have been conducted along with performance evaluation of QCDOC's custom-designed features. Moreover, together with HASE features, simulation metamodelling allows efficient generation of alternate simulation models with maximum component reuse and minimum design overhead. Experiments with HASE QCDOC and its successor Bluegene/L simulation models confirm that QCDOC design configurations are optimised for parallel QCD code.

Acknowledgements

It is a great pleasure to acknowledge the incredible amount of support and encouragement of many individuals during my PhD research at the University of Edinburgh.

I am truly grateful to my supervisor Professor Roland Ibbett for his help, guidance, patience and understanding — from project proposal to the final write-up. His insightful suggestions have been a stimulating challenge to me. If it were not for his guidance and persistent support, this thesis would not have been completed. My deepest gratitude to Professor Tony Kennedy, my second supervisor, who introduced me to the high performance QCD calculations, and whose enthusiasm for the QCD supercomputers motivated me to determine my research methodology. He used to answer my naive questions about QCD theory and parallel QCD simulations with great patience. I hope that in the future I will have the wonderful privilege to seek my supervisors' advice.

Dr Frederic Mallet, the HASE UKQCD computer simulation project research fellow, made significant contributions to the HASE platform for my research project. I shared my office with him for two years and he was always there to provide support and to share his recent PhD experience. I also wish to thank the UKQCD research team members at the School of Physics, University of Edinburgh and at Columbia University. Many thanks to Professor Richard Kenway who provided access to the QCDOC development resources and documentation. I am indebted to Dr Peter Boyle and Dr Balint Joo for giving me an access to their benchmark and operating system code, and for providing incredible insight into the QCDOC system.

I have met many interesting post-graduates and researchers at the School of Informatics who enriched my research experience. I am also thankful to the School of Informatics computing support team for their timely help and assistance.

I thank my husband for believing in me — always. My wee son, who grew from a baby to a young boy during this time, is my inspiration and the greatest source of joy in my life. I also thank my parents — they have always been there for me.

I acknowledge the EPSRC studentship (UKQCD computer simulation project grant GR/R27129) and Informatics Teaching Organisation teaching assistantship. Finally, I acknowledge all copyrights and trademarks.

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:

- S. Alam and R. Ibbett, "A Methodology for Simulating Scientific Supercomputing Systems", *Summer Simulation MultiConference*, USA, July 2004.
- S. Alam, R. Ibbett and F. Mallet, "Performance Evaluation of Local Communications: A Case-study", *Proceedings of 15th International Conference on Parallel and Distributed Computing and Systems*, USA, November 2003.
- S. Alam, R. Ibbett and F. Mallet, "Simulation of a Computer Architecture for Quantum Chromodynamics Calculations", *Crossroads, The ACM Student Magazine, Interdisciplinary Computer Science*, Issue 9.3, Spring 2003, pp. 18-23.
- F. Mallet, R.N. Ibbett and S. Alam, "An Extensible Clock mechanism for Computer Architecture Simulations", *13th International Conference on Modelling and Simulation*, USA, May 2002.
- S. Alam, "HASE: A Toolset for Modelling Massively-Parallel Computers: Parameterised Simulation Model of the QCDOC Machine", Poster presented at the Informatics Jamboree Workshop, UK, May 2002.

To my family.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Performance Studies of Computer Systems

The performance evaluation, analysis, prediction and measurement of existing and novel computer systems have become an increasingly challenging task. The achievable performance of an application over an underlying system architecture depends on a number of hardware and software factors including the operating system characteristics. Performance results from a study are unreliable if these factors are not taken into account. Detailed software simulations have therefore become essential for evaluating ideas in the computer architecture field, reported by Skadron *et. al.* [SMA$^+$03] in a recent discussion article.

A large fraction of computer architecture research papers presented during the last decade at the International Symposium of Computer Architecture (ISCA), used simulation for quantitative evaluation. Comparatively, about 20 years ago, only a small fraction of papers mentioned simulation as their research method. Figure 1.1 shows that there has been a shift toward simulation-based computer architecture research.

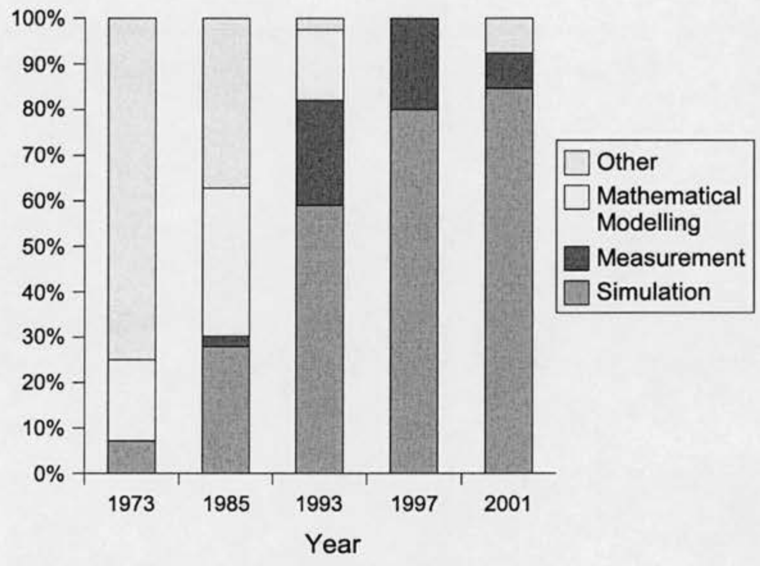


Figure 1.1: Performance evaluation methodologies used in a sampling of papers from the Proceedings of the ISCA

Simulation is even more useful in research to evaluate radical new ideas and to explore the design space of a computing system. In contrast, mathematical or analytical modelling schemes and direct system measurements have their limitations. Analytical modelling schemes do not capture the dynamics of a system while a physical machine for investigating a new design is not always available. Even within existing systems, profile management tools have constraints and they are not always non-intrusive.

With a wide-scale acceptance of simulation as a performance evaluation technique, there are a number of challenges and limitations — especially in the multiprocessor systems arena. Firstly, developing a high-fidelity software simulation is becoming a more time-consuming and complicated task with the ongoing rapid innovation of computer systems. Secondly, complete multiprocessor software simulations impose phenomenal computing and storage requirements on the host system.[1] Thirdly, the wall clock time taken to simulate a system imposes design trade-offs between the level of design detail of the target system captured by the simulated software and the time it takes to execute on a host system. Finally, the range of applications is becoming more

[1]Target system — the one being simulated. Host system — the one that runs the simulated program/software.

diverse; benchmarks and performance metrics for a range of commercial and scientific applications are simply not available. Skadron *et. al.* [SMA$^+$03] concluded that no benchmark suite can be a one-size-fits-all solution.

The scope of the research presented in this thesis involves the creation of parameterised HASE simulation models of a recent scientific supercomputer to explore its design space and to conduct performance studies of a highly-demanding scientific application. HASE, the simulation platform for the research, is a Hierarchical computer Architecture design and Simulation Environment that allows for parameterised software prototyping of computer systems at multiple abstraction levels encompassing system hardware and application software. The target high-end supercomputer, QCDOC, employs an innovative System-On-Chip (SOC) integration technology and has been custom-built for a class of applications called Quantum Chromodynamics (QCD). In this thesis, high-end scientific supercomputing systems performance modelling issues, and the flexibility, scalability and reusability provided by the HASE platform are identified, discussed and explored.

1.2 Challenges in Scientific Supercomputing

"Supercomputers and desktops are just not the same thing in spite of the fact that they both perform calculations."

The above statement by Dongarra *et. al.* [DSSS03] is directed at the ongoing debate about the future directions of scientific High Performance Computing (HPC). Scientific HPC applications include simulations in natural and physical sciences, for example, DNA and black hole simulations, modelling processes, for instance, shuttle launch and nuclear devices experiments and analysis of very large data sets like data from observatories and collected from nuclear colliders. These applications require the largest, fastest, and most expensive type of computers called supercomputers that are designed for and dedicated to executing complex calculations rapidly. Supercomputers do not differ from desktop computers in processing power, storage capacity and memory alone. It is the architectural and algorithmic changes, which define a supercomputer. The distinction is that supercomputers carry out large amounts of parallel and vector data processing.

Historically, scientific supercomputers were application-specific, i.e., they were not widespread and were hugely expensive for most research communities. They were considered as special-purpose computers, in terms of their hardware, software and operating system designs. In late eighties, with the advances in Very Large Scale Integration (VLSI) technologies, high-end computing power became available to larger scientific research communities. Low cost, high performance systems built with commodity off-the-shelf processors, like Beowulf clusters, and massively-parallel systems with small numbers of specialised components were considered the best design option. As processor power kept on increasing with Moore's law, and memory wall effects and communication latencies started dominating the overall parallel performance, the sustainable performance for application code was called into question.

Since the late 1990s, the difference between the theoretical peak performance and performance sustained by the application code over high-end supercomputers has been increasing. In other words, the sustained-to-peak performance ratio is declining, particularly for scientific HPC calculations. Scientific codes typically achieve 10% to 30% of the peak performance over main-stream cluster configurations. Therefore, it has been argued that the processing, storage and networking technologies optimised for database and web server applications are not meeting the demands of the high-end scientific computing. HPC systems require significant advances in critical component areas to continue to achieve increases in overall system performance, including memory bandwidth, network bandwidth and latency. The fastest supercomputers for the past two years, Japan's Earth Simulator, invoked a re-thinking of the way in which scientific HPC is progressing. Japan's Earth Simulator does not follow the current supercomputer design trends, i.e. it is not based on commodity design components. Instead, it is custom designed and optimised for earth simulation scientific applications.

Presently, there are two profound arguments for the direction of future scientific supercomputer designs and innovation: Bell and Gray [BG02] argue that it will be driven by market forces while Dongarra *et. al.* [DSSS03] and Simon *et. al.* [SMK$^+$03] are in favour of application-oriented supercomputer design efforts. A new class of supercomputer system designs will require advances in computer modelling, simulation, debugging and compilation strategies and researches. Simon *et. al.* [SMK$^+$03] asserted that commodity off-the-shelf processing are neither cost-effective nor a viable solution for a large number of scientific applications, and serious considerations should be given to optimising systems to a particular large-scale scientific application.

1.2.1 Design Space Exploration

Estimating the suitability and sustained performance of a high-end system for a parallel application is not a straightforward task. With increasing system complexity, achievable performance for an application code depends on exploiting parallelism that is available at various levels of a multiprocessor system design. For instance, a parallel application should take maximum advantage of the Instruction Level Parallelism (ILP) as well as inter-node parallelism. Hence an analytical, or static performance calculation may not produce a reliable and guaranteed outcome, since these studies do not incorporate code execution dynamics.

Simulation is identified as the most comprehensive performance analysis and exploration mechanism for microarchitecture research [SMA$^+$03]. A large-scale simulation design itself is a complicated task as it poses challenges to a simulation designer over its accuracy, flexibility and time to produce simulation results. Skadron *et. al.* [SMA$^+$03] comprehends the role of computer architecture simulation frameworks and challenges in computer architecture performance evaluation:

> "Quantitative evaluation of computer architectures relies heavily on simulators and simulator infrastructure, yet today's robust and publicly available simulation tools are by no means capable of supporting the full range of studies that the architecture community must pursue."

In particular, multiprocessor systems built with a range of heterogeneous components are increasingly difficult to evaluate and to experiment with. Performance bottlenecks arise from hardware and its interactions with the application software; what the relationship should be between the two within a simulation framework, so that these performance bottlenecks can be comprehensively exposed, is still an open question. A high-level abstraction of hardware and software characteristics is therefore necessary to validate, reuse, evaluate and extend multiprocessor simulation models and frameworks with a degree of flexibility and efficiency. Tightly-integrated scientific supercomputers, which are the focus of the research presented in this thesis, often have unique design features and are devoted to a few highly-demanding applications. Thus, highly flexible and extensible simulation frameworks are essential for modelling these supercomputers.

1.2.2 Divergence Problem

Supercomputing systems have been paramount in the research, innovation, validation and exploration of theoretical QCD, like many other disciplines in natural and physical sciences. QCD is considered as one of the most established scientific discipline that has been studied on high-end parallel computers over the last three decades. During this time, computation power has rocketed. Even desktop computers of today can offer more compute power and storage capacity than a supercomputer of twenty years ago. In this scenario, one can expect that a parallel computer based on current technology should be able to address the QCD 'Grand Challenge' application status, as suggested by Bowler [Bow98]. A grand challenge computing problem is one which cannot be solved in a reasonable amount of time on currently available parallel supercomputers.

Surprisingly, a large number of QCD calculations cannot be simulated within a reasonable approximation range on many high-end systems today. And QCD is not alone; a number of 'Grand Challenge' applications have their status unchanged even after 20 to 30 years. A combination of factors can be considered responsible for the lack of utilisation of ever-increasing compute, storage and communication power. Simon *et. al.* [SMK$^+$03] attributed the poor achievable performance of several scientific applications to the 'divergence problem'. Divergence means that the requirements of scientific applications are substantially different when compared to the mass commercial and industrial compute and communication requirements.

Thus, there ought to be close integration of application and architectural exploration for scientific supercomputers. A number of high-end Massively Parallel Processing (MPP) supercomputers with tens of thousands of processing nodes can score a large peak floating-point operations per second (Flops) rate, but sustain a modest fraction of the theoretical peak performance. The 'divergence problem' is again to blame as the supercomputers built with design components that are optimised for commercial applications do not scale and deliver a high performance for the scientific application codes. Computer simulations are considered as a tool that can assist not only in investigating an application code's execution behaviour over an underlying architecture but also in exploring the design space of a supercomputer architecture for a scientific application.

1.3 Motivation and Project Aims

Computer architecture simulations and their simulators have been used for performance evaluation and exploration of parallel systems for many years. A large number of simulators model low- and medium-scale parallel systems with shared and distributed-shared memories, and conduct experiments with standard benchmark suites that represent commercial workload characteristics. For the design space exploration and analysis of high-end supercomputers, analytical and static models have been employed; these models provide very little insight into application code software interaction and scaling with respect to parallel hardware configurations.

The performance of a parallel system is attributed to a range of its hardware components as well as the characteristics of the workload. A simulation model that encapsulates both hardware details with a degree of accuracy and software characteristics with the workload scaling properties is likely to provide a better understanding of the performance limiting factors. At the same time, the simulation model should be flexible enough to allow a user to experiment with a range of hardware configurations and workload variations. In practice, it is extremely difficult to develop a simulation model that captures all the above-listed requirements for a single processing node let alone replicate it for a larger system configurations. Thus, levels of hierarchical abstraction are necessary such that it is possible to zoom in and out of the details of the model. Furthermore, the level of detail captured by a simulation model should not have phenomenally long and impractical execution time that would ultimately prevent exploration of wide-range design options.

The key motivations for the UKQCD computer simulation project are the component reuse facilities and the degree of freedom offered by the HASE platform to a computer architecture simulation designer. A scalable simulation model design within an existing computer architecture simulation framework for performance studies of a recent scientific supercomputer is the design target. The main aim of the research presented in this thesis is to investigate the performance limiting factors of the QCDOC computer architecture for the execution of the QCD application code through parameterised HASE simulation models. A further aim is to explore the design parameter space of the model to investigate variations in performance against a range of architectural parameters in order to inform the design of subsequent generations of application-specific supercomputing systems.

1.4 HASE

HASE [HASa] is a Hierarchical computer Architecture design and Simulation Environment which allows for the rapid development and exploration of computer architectures at multiple levels of abstraction, encompassing both hardware and software. HASE has been considered as the most suitable platform for the UKQCD computer simulation studies because of its (a) rapid and flexible prototyping facilities and (b) support mechanisms for multiprocessor simulations. In addition to the parameterised and flexible prototyping facility for custom-built computer systems, a simulation trace file containing simulated components information during an experiment is generated by HASE. Analyses of a HASE simulation trace file enables a user to quantify results as well as to investigate the overall behaviour of various design components during a simulation experiment. Information in the simulation trace file can be animated to validate the individual components and their interactions within a HASE simulation model.

Performance evaluation and exploration of an application code on an MPP system require extensive instrumentation and experimentation. An in-depth and precise understanding of the behaviour of underlying system's components for the execution of the application code is necessary to identify the performance bottlenecks. For parallel systems, an additional metric called scalability is part and parcel of a performance evaluation exercise. These system hardware and application software requirements have been addressed in the parameterised HASE QCDOC model through hardware configuration parameters and application workload parameters. Application code execution, in a format specified by a simulation designer, can be incorporated in a HASE simulation model. Hence, a HASE model simulates execution of software over the hardware within a single model, thereby capturing the dynamics of control and data paths of a computer system. This HASE facility is exploited in conducting the scalability and load balancing experiments for parallel QCD benchmark software. In the QCDOC model, the workload parameters alter the computation and communication configurations for a QCDOC processing node or its parallel workload.

1.5 QCDOC — The UKQCD Computer

The UKQCD community is one of the world's leading field theory lattice groups with an established programme to study QCD, a 'Grand Challenge' computing application. The group is in the process of procuring a QCDOC [CCC$^+$01] computer, currently being constructed through a collaboration between IBM and Columbia University. QCDOC is a massively parallel Multiple Instruction Multiple Data (MIMD) machine which has been custom-built to perform QCD simulations. A QCDOC processing node is composed of an Application Specific Integrated Circuit (ASIC) and a Synchronous DRAM (SDRAM). The processing nodes are connected in a six-dimensional torus topology.

For the last three decades, a number of commercial and custom-built parallel machines have been employed to solve QCD calculations and they have successfully produced a number of results. The natural parallelism in QCD applications, resulting from its lattice formulation (called lattice QCD), made QCD one of the first applications to be tried out on early parallel computers. Yet QCD calculations still challenge the power of high-end supercomputers; continuing to push the boundaries of supercomputer architecture, QCD algorithms and parallel software techniques.

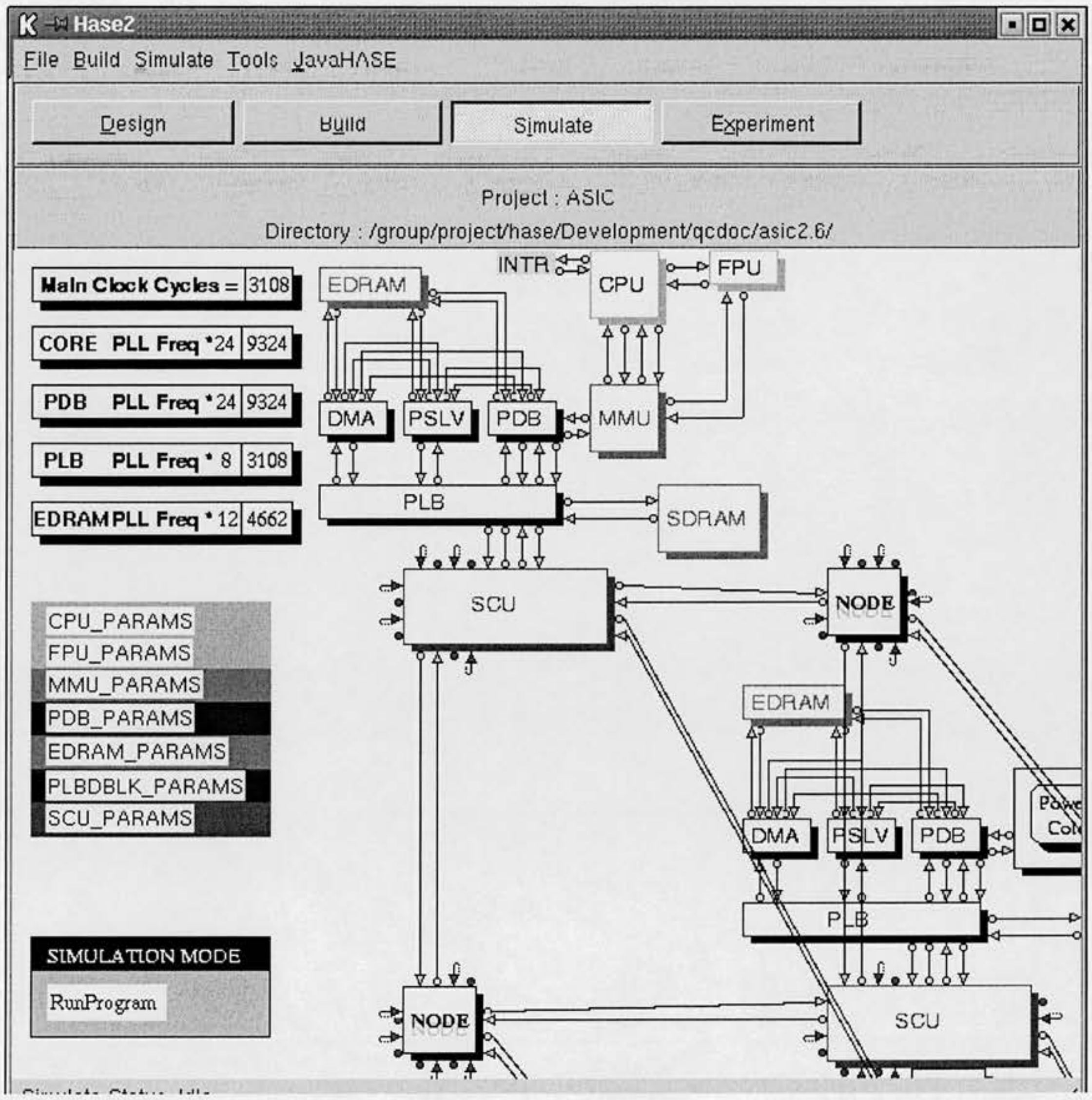
The task of developing the UKQCD computer simulation model takes into account the simulation practices in multiprocessor systems. Presently available multiprocessor simulators address the issues of mass market systems, i.e., shared memory database and transaction servers, and their representative workloads. Likewise, simulation models of the application-specific QCDOC system are benchmarked with parallel QCD workload.

1.6 Design Methodology

In order to achieve the project aims, an application-driven design approach was adopted for modelling the UKQCD. This is because tightly-integrated, high-end supercomputers like QCDOC are designed for and dedicated to a few highly-demanding applications. Thus, for a wide range of performance experiments, not only the QCDOC design components are parameterised but also the parallel QCD code characteristics including workload size are also made HASE design parameters. With the precise knowledge of hardware details of the architecture,[2] the initial target was to model, as precisely as possible, the design details of the system. A further aim was to explore the design space, in particular, the custom-designed components of the hardware. The design space exploration was facilitated with a range of hardware parameter options. As a massively-parallel system with numerous processing nodes, the design focus was an appropriate level of abstraction for running simulations. A hierarchical design approach was followed in abstracting out the simulation details as the experiments were performed at different levels.

Figure 1.2 shows the multiple abstraction levels of a QCDOC processing node.

[2]Access to the confidential design documents and numerous discussions with the QCDOC design team members.



Project : ASIC
Directory : /group/project/hase/Development/qcdoc/asic2.6/

Main Clock Cycles = 3108
CORE PLL Freq *24 9324
PDB PLL Freq *24 9324
PLB PLL Freq * 8 3108
EDRAM PLL Freq * 12 4662

SIMULATION MODE
RunProgram

Figure 1.2: QCDOC simulation model in the HASE design window

A user can zoom-in and out of the design details of the system through the graphical interface. For instance, on one hand, a processing node can be viewed as a single design entity, while on the other hand a user can inspect the Central Processing Unit (CPU) parameters by going down the hierarchical abstraction levels. In addition, altering the design parameter values for an entity is also done through the Graphical User Interface (GUI). Before a simulation run, a user can, for example, change the Level 1 cache configuration in the Memory Management Unit (MMU) entity via its parameter window. Likewise, software parameters of the model can be changed through a GUI, which is explained in detail in Chapter 3.

A key requirement of an application-driven system analysis is to understand and incorporate the application code characteristics in the model. Seltzer *et. al.* [SKSZ99] argued the case for application-specific benchmarking against standard benchmarking:

> "... the results of such benchmarks provide little information to indicate how well a particular system will handle a particular application. Such results are, at best, useless and, at worst, misleading."

The model was benchmarked with an optimised QCD application code developed by the QCDOC design team. The QCD kernel code is an indispensable part of a QCD application execution; thus it is parameterised and simulated with the QCDOC hardware model. This parameterised hardware-software co-simulation within the HASE environment not only enabled identification of performance limiting factors of QCDOC hardware for the QCD application but it also helped in analysis and scaling behaviour of the application workload.

Finally, the research presented in this thesis also aimed at informing future-generation MPP system optimisation for the QCD application code. A recent SOC-based scientific supercomputing system called Bluegene/L was studied, as part of the research, by employing an application-driven metamodelling design scheme. Meta-modelling enables efficient generation of alternate simulation models by employing maximum component reuse and minimum design overhead. A parameterised HASE metamodel was constructed, which was capable, by an appropriate selection of parameters, of simulating the QCDOC and Bluegene/L system characteristics. QCD experiments have been performed in order to compare and to quantify the impact of custom-designed components on the QCD code performance.

1.7 Related Research

A number of multiprocessor simulators allow a user to alter levels of memory hierarchy and network latencies. For instance, RSIM [PRA97] allows two memory hierarchy levels with a range of protocols for shared memory systems, while Wisconsin Wind Tunnel II [MRF+00] models shared and distributed-shared memory systems. These simulators are in general driven by industry-standard benchmarks. On the contrary, the Teraflops and Petaflops scale systems use analytical models or combine an analytical model with a detailed single node performance model. Performance prediction techniques have been employed in PACE [NKP+00] and in exploring large-scale parallel systems by Kerbyson *et. al.* [KHW02]. These technique create a simulation model of the system with pre-defined parameters and application code traces.

Almasi *et. al.* [ACC+02] demonstrated the scalability of molecular dynamics calculations on a cellular Petaflops computer using an analytical model of the application and functional modelling techniques for the underlying hardware. They claimed that it is not possible to simulate a 32K node system, therefore, a single node functional simulation is extrapolated for overall performance results. Timing information is extracted from the functional simulator output. Even though such a technique may be precise and efficient, extending and altering the hardware configuration involve three levels of changes: the analytical model, the functional simulator and perhaps the timing simulator, too. Consequently, a functional and timing simulator is necessary to provide a large degree of freedom to change the hardware configuration without altering the corresponding analytical model of the application code, as well as a timing simulator of the memory traces.

Like Almasi *et. al.*'s system-specific simulation model, the Bluegene/L design team has created a single-node complete system simulation called BGLSim [MCS+03]. It is based on SimOS [RBDH97] a complete system simulation approach that incorporates operating system and peripheral devices in the simulation models. The resulting simulations are therefore extremely slow and need powerful host platforms. BGLSim can run unmodified Bluegene/L code for a single processor; it is primarily used for tuning the application code and communication library development. Due to the enormous simulation execution requirements, the BGLSim runs on a Linux cluster.

Research in progress include a performance simulator called POEMS [ABB$^+$00] and *Extreme-scale* simulations [ABB$^+$01]. Performance Oriented End-to-end Modeling System (POEMS) project promises to assist with the performance study of large-scale computational systems across all stages of design. It is based on COMPASS [NYHG$^+$98], a COMponent-based Parallel System Simulator, that provides direct execution-driven, parallel simulation for performance prediction of parallel computation-, communication-, and I/O-intensive programs written using the Message Passing Interface (MPI) library. The POEMS project relies on task graphs of an application, a knowledge base for widely-based algorithms, analytical and simulating components of the model and a parallel compiler within the model. It is to be driven by an specification language, details of which are not yet available to examine the complexity and restrictions of the specification language. Another research project, the *Extreme-scale* simulations were introduced by Alexander *et. al.* [ABB$^+$01] in 2001. Their radical idea is based on a parallel discrete-event simulation engine using an object-oriented scheme, which would enable modelling of large-scale networks with thousands of processing nodes, in a manner similar to the Internet modelling. Its initial prototype was aimed at supporting studies of simulation performance and scaling rather than the design space of the simulated system. It is however not yet clear how low-level processing node design details, hardware-software interactions and single node performance search space will be modelled.

In designing a software simulation model, trade-offs are often made according to the purpose for which the simulation is built. For instance, BGLSim is a functional simulator because precise timing details are not significant for building compilers and other system tools. ASCI Q simulations are aimed at achieving the maximum possible fraction of peak performance for a predefined system configuration and a target workload. The research presented in this thesis, in contrast, focuses on performance search space of a grand challenge application on next-generation scientific supercomputing systems. The main advantage of this application-driven, parameterised metamodelling scheme over other simulation strategies is that the design parameter space of hardware components is not limited by static workload configurations and traces. A further advantage is that it does not require time consuming, off-line code/trace generation.

These techniques have allowed us to explore the performance search space of the parallel QCD code not only on the QCDOC but also on its successor, Bluegene/L, design configurations.

1.8 Thesis Overview

Chapter 1 has introduced the research presented in this thesis namely the UKQCD computer simulation project. The remainder of the thesis explains in detail the research background, the simulation platform, the target architecture together with design and implementation details of the HASE QCDOC model. In addition, a description of simulation metamodelling in HASE and simulation-based performance results are presented along with the discussions and key research contributions.

The thesis outline is as follows:

Chapter 2 details the background to parallel system simulations: techniques, frameworks and practices. Moreover, it describes the simulation framework for the UKQCD computer, HASE. Since the focus of the research is an application-specific supercomputer, chapter 2 provides background and related work in the field of parallel QCD computer designs and their software repositories. Finally, in the context of on-going HPC research in the field of QCD, the UKQCD computer, QCDOC, is introduced.

Chapter 3 presents the design and implementation details of the HASE QCDOC computer simulation model. Firstly, the hardware details of the QCDOC system is outlined, including its communication network design. Secondly, the modelling strategy of the QCDOC system in HASE is discussed. Identification of limitations in modelling QCDOC system in HASE and how they were rectified is presented next. This is followed by a description of the extensions made to the HASE platform as a result of requirements of QCDOC model. Lastly, issues in model validation and result gathering are discussed in the MPP-scale simulation model design space exploration.

Chapter 4 describes the experiments and analysis of the experimental results from the HASE QCDOC simulation model. First, the computation and communication logic of the QCDOC processing node is described. Next, an outline of the experimental setup is explained. This follows an analysis of parallel QCD code characteristics. Performance scalability and design space exploration experiments and results are discussed in this chapter. In the end, the chapter outlines the requirements for a higher abstraction-level model, which was designed, as part of the research, for experiments on a larger (10-Teraflops) scale system configurations.

Chapter 5 discusses the advantages of metamodelling and hardware-software co-simulation approach within the HASE framework, particularly in modelling SOC-based MPP computers like QCDOC and its successor Bluegene/L. The architecture of Bluegene/L system and its similarities with and differences from the QCDOC architecture are outlined. Parallel QCD workload distribution and mapping characteristics are explained next. This chapter explains in detail the simulation metamodelling of scientific supercomputers in HASE and advantages and shortcomings of this scheme. Finally, a description of the performance comparison experiments conducted on the QCDOC and Bluegene/L simulation models is provided.

Chapter 6 presents a summary of findings and contributions to the field of scientific supercomputer simulation. Furthermore, this chapter outlines the future prospects and directions of this research.

Chapter 2

Background

This chapter provides background information. Firstly, parallel supercomputer performance evaluation techniques and tools are briefly discussed along with the benefits of modelling and simulation of complex computer architectures. Secondly, the simulation platform for the UKQCD computer simulation project, HASE, is described. Lastly, an introduction to the history and current status of QCD calculations on parallel computers is presented. In the context of the on-going research efforts to built supercomputers for QCD calculations, a brief description of the architecture and design philosophy of the QCDOC, the UKQCD computer, is presented.

2.1 Computer Architecture Performance Studies

The tremendous complexity of computer systems and increasingly complicated interactions with application software is making performance evaluation, analysis and prediction of existing and novel systems a profoundly challenging task. Cost-effective, flexible, extensible and scalable tools and techniques are required to explore huge design space and to evaluate radical new ideas, particularly, when high-end multiprocessor supercomputer architectures and applications are considered. For instance, a single performance problem in the world's second fastest supercomputer was identified and subsequently rectified using an arsenal of performance analysis tools (table 2.1) [PKP03].

Technique	Description	Purpose
measurement	running full applications under various system configurations and measuring their performance	determine how well the application actually performs
micro- benchmarking	measuring the performance of primitive components of an application	provide insight into application performance
software simulation	running an application or benchmark on a software simulation instead of a physical system	examine a series of 'what if' scenarios, such as cluster configuration changes
analytical modelling	devising a parameterised, mathematical model that represents the performance of an application in terms of the performance of processors, nodes, and networks	rapidly predict the expected performance of an application on existing or hypothetical machines

Table 2.1: Performance analysis tools and techniques [PKP03]

Table 2.1 lists the tools and techniques used in improving the effective performance of ASCI Q, the world's second-fastest supercomputer. Using an arsenal of performance-analysis techniques including analytical models, custom micro-benchmarks, full applications, and simulators, Petrini *et. al.* [PKP03] succeeded in observing a serious performance problem, thereby identifying the source of the problem and eliminating it with up to a factor of 2 improvement in application performance. Petrini *et. al.* claim that their performance analysis methodology is immediately applicable to other large-scale supercomputers. However, detailed knowledge of the application and architecture are a prerequisite for their performance modelling techniques of the high-end systems.

The research presented in this thesis uses the HASE simulation framework [CHIW98] for modelling scientific supercomputing systems, thereby reducing the cost of building simulation models from scratch. HASE is a Hierarchical computer Architecture design and Simulation Environment which allows for flexible and efficient prototyping of computing systems at multiple abstraction levels encompassing both hardware and software design parameters. Further details of the HASE platform are presented in section 2.2. HASE extends the concept of 'software simulation' presented in table 2.1 by adding parameterisation so that not only existing system configurations (and a wider performance search space) but also hypothetical or ideal system configurations can be explored through experimentation.

A 'grand challenge' scientific application called QCD and its performance evaluation on a recent massively-parallel processing (MPP) supercomputer QCDOC is the scope of the research presented in this thesis. However, the analysis of performance studies mechanisms and approaches presented in this thesis can be applied to other high-end computing applications and architectures.

2.1.1 Why Simulate an MPP System

One of the main aims of the QCD research communities, including the UKQCD, is procurement of cost-effective, high performance supercomputers *specifically* designed for QCD calculations. Establishing the *suitability* of a supercomputer for a given application is not a straightforward task; memory bandwidth per flops ratio and network bandwidth requirements cannot be translated or mapped directly to a supercomputer design. The design and implementation of high-performance systems is a highly complex problem requiring knowledge of many factors, according to Kerbyson *et. al.* [KHW02]. The peak performance of a system results from the underlying hardware architecture including processor design, memory hierarchy, inter-processor communication system, and their interactions. Moreover, the achievable performance is dependent upon the workload for the system, and specifically how this workload utilises the resources within the system. Thus, optimising the peak performance of a system component is only valuable if it has an associated impact on the achievable performance of the workload. Culler *et. al.* [CGS98] comprehensively identify the hardware and software factors involved in parallel computer architecture design. To express supercomputer performance, the peak performance (maximum theoretical operation performance) is often used. Peak performance is an important index for supercomputers, indicating the maximum possible computer hardware performance. However, the performance achieved when running a real program varies depending on the characteristics of the program, and the peak performance is never achieved. Furthermore, overall performance also depends heavily on various factors such as the operating system, compiler, and I/O performance.

Performance analysis is therefore required in order to ascertain the impact on performance resulting from architectural evolution and innovation. Performance modelling and evaluation are central because they can provide information on the expected performance of a workload, given a certain architectural configuration. These studies are useful throughout a system life-cycle: starting at design when no system is available for measurement, in procurement for the comparison of systems, through to implementation and installation, and to examine the effects of updating a system over time. At each point, the performance models should provide an expectation and insight into the achievable performance with a reasonable fidelity. When considering Teraflops-scale systems, a large investment is required throughout the life-cycle and thus performance analysis studies are a requirement. Martin [Mar88] outlined the significance of supercomputer performance evaluation for a scientific application:

> "... applications are the centre of the universe; supercomputer systems exist for their benefit ..."

then he added:

> "The ability to assess and measure accurately the performance of new designs can promote development and avoid the possibility of overlooking fruitful ideas in architecture and algorithm design."

The performance analysis approach for the research described in this thesis is application-driven — similar to the design approach of high-end scientific supercomputers. An application-driven approach involves an understanding of the processing flow in the application, the communication and computation patterns, and how they use and are mapped onto the available resources. From this, simulation models are constructed that encapsulate and parameterise key performance characteristics.

The aim of the simulation is to provide an insight. By keeping the models as flexible as possible whilst not sacrificing accuracy, it may be used to explore the achievable performance in new situations — in hardware system design and workload modifications.

2.1.2 Design Trade-offs Exploration

A variety of techniques can be used to explore design trade-offs in a computer architecture and to evaluate its performance. Evaluating the performance of an existing system is in principle straightforward, as benchmarks can be run to measure the execution time. Without extensive instrumentation — involving both hardware and software — experiments on an existing system with rigid hardware configurations provide little understanding of the causes of performance limitations. Built-in system monitors offer a limited opportunity to measure the effects of varying architectural design configurations.

The complexity and cost of a parallel system are far greater than those of existing uniprocessor systems. A parallel system needs close scrutiny to establish the design and performance trade-off factors of the hardware and the application. A complete *existing* system in most cases is not available for establishing, for instance, parallel workload distribution strategies and communication characteristics. These can be determined by extrapolating single-node performance measures, but without taking into account parallel system/software interactions, one can have very little confidence in these performance predictions. Cost and performance analyses of parallel systems are therefore increasingly conducted using simulation models and analytical models.

2.1.2.1 Analytical Modelling Limitations

Analytical or static modelling is effective for a variety of multiprocessor system components. Usually, analytical models are driven by workload models rather than benchmarks or real applications and the results can be unreliable in situations where attempts are made to model complex systems containing a variety of components.

An example of an analytical model for parallel systems is the Bulk Synchronous Processing (BSP) model [SHM97]. BSP views a parallel machine as a set of processor and memory pairs with a global communication network and includes a method for synchronising the processors. While this model can be used to develop parallel applications and algorithms, and assess their performance, there are restrictions on how a parallel program can execute on a BSP, i.e., through pre-defined execution steps. The performance of the BSP-style program can be characterised by three parameters: p, L and g, where p stands for the number of processors, L is the cost of global synchronisation in units of time step, and g corresponds to network throughput (the ratio of the number of local computational operations performed per second by all processors to the total number of data words delivered per second by the router). A parallel algorithm is expressed in the BSP model as a sequence of parallel supersteps. Each superstep consists of a sequence of local computation steps plus any message exchanges, followed by a global synchronisation.

Culler *et. al.* [CKP+93] proposed LogP, which extends BSP by taking into account the communication cost. It includes four parameters to characterise the system: L, o, g and P, where P stands for the number of processors involved; o represents the software overhead associated with the transmission/reception of message; L is the upper bound on the hardware delay in transmitting a fixed but small size message between two end-points; and g is the minimal time interval between consecutive messaging events at a processor, which corresponds to the network throughput available to the processor. Later, Touyama and Horiguchi [TH99] introduced the LogPQ model which augments the LogP model on the stalling issue of the network constraint by adding buffer queues in the communication lines. The LogPQ model focuses on the communication network and protocol design study but is limited to communication parameters' characteristics.

Large system designs have traditionally used analytical models for performance predictions and scalability studies as analytical estimates can be performed relatively quickly. Nonetheless, analytical techniques are considered static — these studies give no information about the dynamics of a system, particularly when hardware-software interactions are considered. Thus, the analytical estimates cannot be considered as absolutely reliable.

One way to address the limit to analytical modelling is to build a small prototype system. This scheme has its own constraints and drawbacks; firstly, it is expensive to build prototypes for early stages of the design process. Secondly, the scalability analysis of parallel code cannot be performed on small versions of systems. Finally, a hardware prototype, even though accurate, provides limited flexibility in terms of design space and this stage cannot be started until most of the design considerations are made final. As a result, the designers are unlikely to explore a wider design space. In the presence of multiprocessor prototype design constraints, one needs to resort to some kind of modelling in order to study systems yet to be built and to estimate the scalability of these systems for a given application. Simulation has therefore become a popular approach to analyse dynamic behaviour and to predict performance of complex systems.

2.1.2.2 Simulation

A simulation is the execution of a model, represented by a computer program that gives information about the system being investigated. The simulation approach of analysing a model is experimental and dynamic compared to the analytical approach, where the method of analysing the system is purely theoretical and static. Software simulations are not only considered more accurate and reliable, but they also provide an inexpensive and flexible way to explore a wide range of design options [SMA+03].

Industry uses cycle-accurate and functional simulation extensively during processor and system design because it is the easiest and least expensive way to explore design options [SMA+03]. Simulation is even more important in research to evaluate radical new ideas and characterise the nature of the design space. It is increasingly considered as the *de facto* performance modelling method in the evaluation of shared and distributed-shared memory architectures. There are several reasons for this. The accuracy of analytical models in the past has been insufficient for the type of design decisions computer architects wish to make (for instance, what kind of caches coherence or branch prediction schemes are needed).

Analytical modelling schemes have been extensively used in multiprocessor performance studies, since performance results can be generated far more quickly compared to a multiprocessor software simulation. Researchers are however now resorting to alternative performance exploration techniques as the achievable performance of scientific codes over off-the-shelf clusters is steadily declining; parallel scientific codes typically achieve 10-30% of peak performance on main-stream cluster configurations. Software simulations are identified as the only tool that can provide an insight into the dynamic behaviour of an application over an underlying architecture [SMK$^+$03] thereby facilitating investigations of future system designs. Many research projects, most notably Alexander *et. al.* [ABB$^+$01] project at the Los Alamos National Laboratory, are addressing issues of *extreme-scale* simulations. The *extreme-scale* simulations approach for simulating computing architectures is applicable to high-end computing systems (with thousands of processing nodes) and to advanced, novel architectural configurations.

2.1.3 Computer Architecture Simulators

Simulators model existing or future machines or microprocessors. They are essentially a model of the system being simulated, written in a high level computer language, such as C or Java, and run on some existing machine. The machine on which the simulator runs is called the host machine and the machine being modelled is called the target machine. Computer architecture simulators provide an environment to study complex systems by simulating (in some cases by visually displaying) the organisation and underlying simultaneous events during a program execution. These simulators proved to be an effective tool to study and to simulate complex trade-offs of efficiency and performance inherent in parallel architectures. Furthermore, a computer architecture simulator enables computer architects to design and experiment with different forms of parallelism such as multiple pipelines, multiple processors, clustered computer systems, and distributed computing systems cost-effectively and efficiently. Researchers and students benefit with parallel architecture simulation tools and frameworks — not only do simulators facilitate flexible performance evaluation studies but they also provide insight into the working of a parallel system.

Computer architecture simulators vary widely in their applications. They are used by processor architects to evaluate uniprocessor design trade-offs, by operating system authors to debug their code and to evaluate operating system performance, by parallel system architects to assess the performance of large systems, and by end users to execute programs written for one system on a different host system. Simulators also vary in their performance and the level of detail they can model.

A significant metric in parallel computer simulations is the slow-down, or the average number of simulator host instructions executed per simulated instruction. In general, the more detail a simulator captures, the greater its slow-down. Slow but accurate simulators have the advantage of capturing subtleties of the target system. However, their slow speed limits the size of the system they can model and the number of simulated instructions they can execute. The simulator designer must choose a level of simulation detail that is fine enough to capture important performance artifacts and, at the same time, fast enough to model large systems and long running applications in an acceptable time frame.

Simulators can be functional simulators or timing simulators. There also exists trace-driven or execution-driven simulators. There can be simulators of components or of the complete system. Functional simulators simulate functionality of the target processor, and in essence provide a component similar to the one being modelled — without precise timing information. The register values of the simulated machine are available in the equivalent registers of the simulator. In addition to the values, the simulators also provide performance information in cycles count, cache hit ratios, branch prediction rates, etc. Thus the simulator is a virtual component representing the microprocessor or subsystem being modelled plus a variety of performance information. If performance evaluation is the only design objective, not the insight, one does not need to model the functionality. For instance, a cache performance simulator does not need to actually store values in the cache; it only needs to store information related to the address of the value being cached. Partially stored information is sufficient to determine a future hit or miss. While it is useful to have the values as well, a simulator with complete functionality together with performance is bound to be slower than a pure performance simulator.

2.1.3.1 Multiprocessor Simulators — The Benefits

There are several benefits of using a simulator for evaluating multiprocessor architectures considering the exponential cost and time requirements associated to build them.

- It is in principle more straightforward to augment a simulator with measurement and debugging features than its hardware counterpart.
- In a system design process, it can take several months/years from a 2-4 node prototype design to a full machine with thousands of processors. In a simulator, provided memory and system requirements of the host permit, this transition can be possible in days if not hours to estimate the scalability and operating system requirements of a system.
- On a simulator, it is possible to reflect the ideal and hypothetical states of a system by approximating the physical constraints, which is impossible in a real system.
- A multiprocessor system simulator is portable; this is a useful property if designs have to be exchanged among physically separate researchers.
- Prototype failures are less expensive in simulators in terms of cost and time.
- Depending on the memory and speed requirements of a simulator, it is possible to run multiple experiments on a single host concurrently.
- Additional benefits of simulators include the ability to generate address traces of user and system code non-intrusively, and to stress-test operating system software by causing the most serious and complex interrupt and exception conditions.

High-fidelity and an appropriate level of detail and efficiency of a simulation model are the key considerations for a model designer. Beside these, the model should be flexible enough to explore and investigate the design space of the target system. Multiprocessor simulation models are considerably challenging to model for two main reasons; firstly, their memory requirements, dimensions, and interconnection network topology, secondly, complex interactions between parallel software and parallel hardware. Scalability of a system is one of the most sought-after characteristics of a parallel system. Experiments to measure the scalability of a parallel system can only be performed by providing a flexible way to increase the size of the machine and its workload. Altering dimensions and size of a parallel machine can have serious implications; it may depend on the host memory capacity and can be limited by the host processing power. This is because an increase in the size of the simulating system not only multiplies its hardware requirements but also affects the interaction of parallel software with the hardware.

2.1.3.2 Types of Multiprocessor Simulators

Different representations of the parallel workload can be used to categorise computer system simulations. Ferrari [Fer78] listed four categories of computer system simulations:

1. **Distribution-driven simulation** — The workload is modelled by a stochastic model of the distribution of memory references. These models are extremely efficient but limited to overall performance measures and are unable to provide an insight into the system. Also, it is immensely difficult to identify stochastic models that can represent the dynamic behaviour of a parallel machine. Distribution-driven multiprocessor simulation has been largely used in interconnection network research.

2. **Trace-driven simulation** — Central to this mode is a trace file. A trace of memory references is generated once by means of executing the workload on a machine similar to the target system or by using a functional simulator. Results from simulations can be effective and accurate as long as the properties of the modelled machine and the machine from which the traces are obtained remain similar. In other words, these types of simulation are not suitable to explore a wider design space. Uhlig and Mudge [UM97] presented a comprehensive survey of trace-driven simulation techniques and tools.

3. **Program-driven simulation** — Generally considered as a complete simulation scheme because both processors and the memory system of the target system are simulated. The workload is executed on the simulated processors where each machine instruction of the target processor can take many machine cycles of the host computer. An instruction or program-driven simulation is an extremely accurate technique but is considered impractical for large multiprocessor simulators due to the speed and memory constraints inherent in parallel systems.

4. **Execution-driven simulation** — This multiprocessor simulation scheme was proposed by Covington *et. al.* [CDJ$^+$91] to address the inefficiencies of program-driven simulations. The workload in an execution-driven model is executed on a host computer rather than on the simulated processor. At events of interest, for example, shared memory references, control is transferred to the simulation software, which simulates the memory system. Although this scheme is fast, it provides little opportunity to comprehend microprocessor-level parameters like Instruction Level Parallelism (ILP). These issues have been addresses by Veenstra and Fowler's [VF94] program-driven shared memory simulator called MIPS INTerpreter (MINT). Rice University later built a new version of RSIM, the execution-driven simulator with ILP support [PRA97], for shared memory and uniprocessors simulation.

Coe [Coe00] in his PhD thesis provided a detailed account of the above four categories of multiprocessor simulators along with a detailed analysis of representative simulators of each of the above category.

Some simulators are hybrid, i.e. they provide a choice or mix-match between any of the above-mentioned two categories or combine functional simulator with timing simulator. Limes [IMMM99] is a hybrid trace and execution driven simulator. Tango Lite [Her93] is a functional and timing simulator. The SPASM [Siv97] simulator, which was used to study scalability of shared memory multiprocessors, combines analytical and execution-driven simulation techniques.

Most multiprocessor simulators report performance as a function of either memory hierarchy or interconnection networks of the system. Since the complexity of a processing node is increasing rapidly, and the effect of the sub-system cannot be completely ignored, a few full system simulations have recently modelled all system components including operating systems, device drivers and web servers. These include Rosenblum *et. al.* SimOS [RBDH97], a complete system simulator built at Stanford University. The SimOS, however, provides the user with no control over the underlying memory hierarchy configuration. Simics by Magnusson *et. al.* [MCE$^+$02] is a useful tool to study computer systems with up to a few nodes, but is not public and hence users have no control over the underlying system. The University of Wisconsin-Madison SimpleScalar [ALE02] is an execution-driven simulator, therefore, it compromises simulator's performance over accuracy.

2.1.4 Effect of System Design on a Simulation Technique

Selection of the most appropriate simulation techniques from those listed in the previous section depends on the hardware features of interest. For example, for shared-memory multiprocessors coherency protocols, memory references are the critical values and what happens in the execution pipelines is not as significant. On the other hand, when studying execution pipeline utilisation, each pipeline stage has to be modelled accurately. For the former example, an execution-driven simulation is a suitable candidate, for the latter, a program-driven simulation model is necessary. In addition to architecture-specific characteristics, simulation models can be compared on their accuracy, flexibility and efficiency (wall clock time to run a simulation).

In practice, a simulation technique may equally depend on the physical properties of a system, i.e. whether it is a physically distributed system or a shared-memory system. According to the famous Flynn classification, multiprocessors can be classified as Single-Instruction-Multiple-Data (SIMD) and Multiple-Instruction-Multiple-Data (MIMD) systems as shown in figure 2.1.

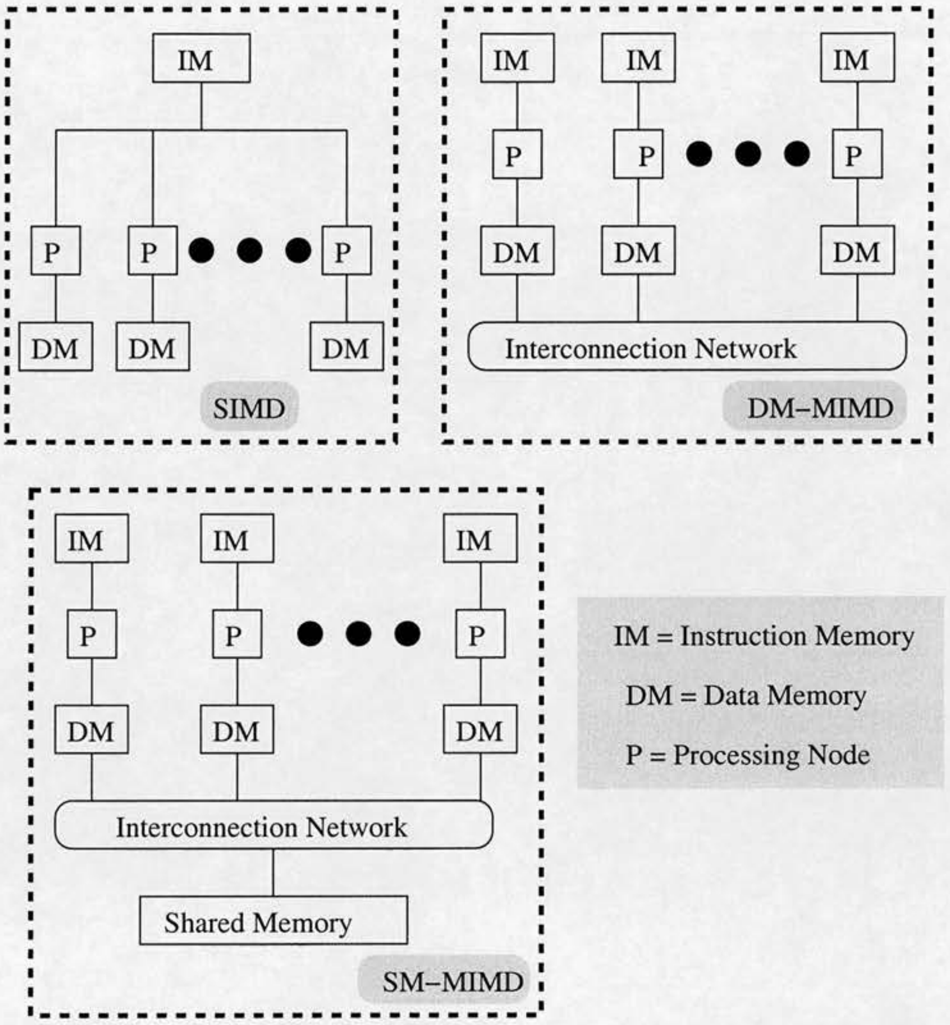


Figure 2.1: Classification of multiprocessor systems

In a shared-memory (SM-MIMD) system, for instance, Symmetric Multi-Processors (SMP) or distributed shared-memory multiprocessors (DSM-MIMD) like Cray T3E (physically distributed with a shared address space), most commercial database and transaction processing servers fall in this classification. There are simulators that use commercial workloads, for example, COMPASS [NYHG$^+$98], which is an execution-driven simulator for the study of commercial applications on shared memory multiprocessors. Mauer *et. al.* [MHW02] studied a full system including the operating system for database applications. Like shared memory and distributed-shared memory multiprocessors, their simulators normally scale to up to few tens of nodes.

Tightly-integrated scientific MPP systems with thousands of processing nodes are generally distributed memory — processing nodes are not connected to a single shared memory via a network. Moreover, these systems generally run a Single-Program-Multiple-Data (SPMD) programming models with explicit message passing communications, notably scientific applications as identified by Cypher *et. al.* [CHKM93]. Research in this area has been focused on the single processor optimisation techniques and analytical models of interconnection networks performance. Dally [Dal90] conducted the analytical study of k-ary n-cube networks. Simulator of Massive ARchitecture and Topologies (SMART) [PV97] considered a static or analytical representation of processing nodes. MIT's execution-driven parallel-computer simulator Proteus [BDCW92] models shared memory and MIMD computers employing direct execution and a static cache-memory representation.

Parallel computer architecture simulation is a growing research area; it is beyond the scope of this thesis to provide a complete account of efforts and contribution in this field. Nevertheless, the main research directions and significant research efforts have been identified that are likely to impact scientific supercomputer performance studies. Present researches suggest that simulation has proved a popular technique for exploring large number of alternate options available when designing a multiprocessor system. Simulations enable designers to evaluate the design alternatives without the expense and long design times associated with hardware construction, and simulation models offer a degree of flexibility that is not available with hardware. A flexible simulation model permits changes to be made to the architecture quickly and with a minimum of effort; for example, adding more processors, changing the cache configuration and altering the dimensions and topology of the interconnection network.

2.1.5 Challenges: Trends in Supercomputer Design

A wide range of parallel supercomputer architectures has evolved over the last decades. These supercomputers can roughly be divided into four major classes of parallel machines: vector machines, shared memory multiprocessors (SMMPs), distributed memory multiprocessors (DMMPs), and hybrid systems or clusters. Currently the most powerful machines are in the range of tens of Teraflops but major advances are expected in the near future. Japan's Earth Simulator project, the world's fastest supercomputer for the past two years, is a vector processor based parallel computer. It has an aggregate power of 40 Teraflops and achieves well above 50% of sustained performance for a range of scientific applications and performance benchmarks. The Earth Simulator supercomputer cannot be considered as cost-effective and is well beyond the reach of many research communities. From the system design view-point, the Earth Simulator is based on clusters of vector SMPs.

From late 1980s to the successful installation of the Earth Simulator, commodity off-the-shelf processor based MPPs were considered a more viable solution for supercomputer design and evolution. MPPs implement a memory architecture that is radically different from vector processors memory system. MPPs can deliver peak performance an order of magnitude faster than vector systems but often sustain performance lower than vector machines. A major challenge in MPP design is to enable a wide range of applications to sustain performance levels higher than on vector systems. MPPs typically use hundreds to tens of thousands of fast commercial microprocessors with the processors and memories paired into distributed processing elements (PEs). Scientific calculations are densely marked with loops; vector processors are optimised for loop iterations, hence, they tend to perform better as compared to scalar or superscalar processors as shown in figure 2.2.

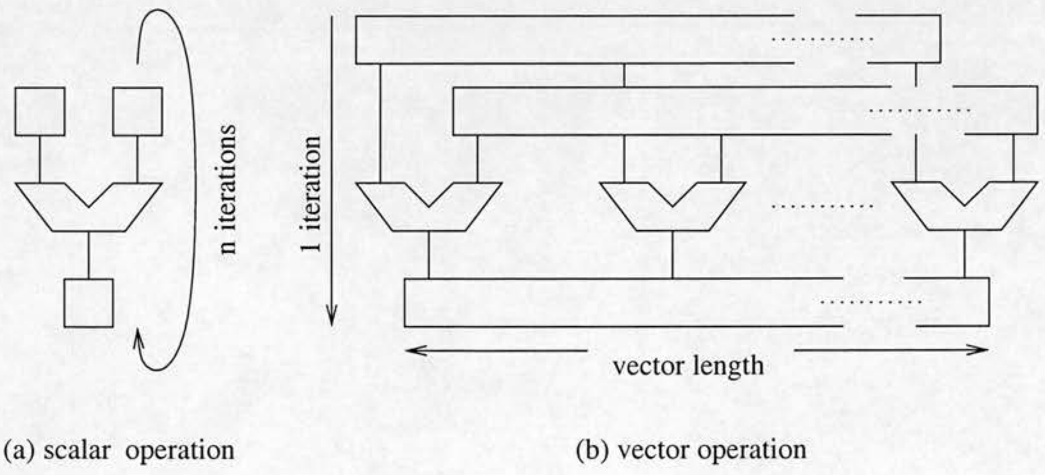


Figure 2.2: Difference between scalar and vector operations

MPP memory interconnects have tended to be slower than high-end vector memory interconnects. MPP interconnects have non-uniform memory access, i.e., the access speed (latency and bandwidth) from a processor to its local memory tends to be faster than the access speed to remote memories. Therefore, a goal for MPP design is to raise the efficiency of hundreds of microprocessors working in parallel to a point where they perform more useful work than can be performed on traditional parallel vector machines. Improving the microprocessor interconnection network will broaden the spectrum of MPP applications that have faster times-to-solution than on parallel vector systems.

In the presence of MPP design challenges, the Hybrid Technology Multi-threaded (HTMT) project in the US envisages building the first Petaflops machine by 2007. The architecture designed for this machine no longer belongs to any of the classes discussed above but comprises advanced new technologies such as Sterling and Zima's Gilgamesh [SZ02], a multithreaded Processor-in-Memory (PiM) architecture with smart memory and optical storage. PiM exploits recent advances in semiconductor fabrication processes that enable the integration of DRAM cell blocks and CMOS logic on the same chip. The benefit of PiM structures is that processing logic can have direct access to the memory blocks at an internal memory bandwidth. Because of the efficiencies derived from staying on-chip, power consumption can be an order of magnitude lower than comparable performance with conventional microprocessor based systems.

A number of Petaflop efforts are currently being undertaken at IBM. IBM's Petaflops machines, named Blue Gene [Tea01a], will be comprised of cellular computers and tightly integrated multithreaded chips [ACC$^+$02] (32 processor per memory chip). Cellular computing concept is similar to the PiM design concepts of integrating a large number of processor and memory components on-chip. However, the former offers both instruction-level and thread-level parallelism. Cascaval *et. al.* provide a comprehensive list and analysis of current and novel efforts in the processor designs directed for minimising memory latencies in their paper [CCC$^+$02].

It is evident from the latest design efforts that high performance on new generation supercomputers will be available at the cost of increased system complexity. Simon *et. al.* [SMK$^+$03] have doubts about achieving high fractions of sustained performance on MPPs based on commodity off-the-shelf components, especially for scientific applications. They argue that the computation requirements of scientific and commercial applications are *diverging*, and within the scientific high performance computing paradigm, applications are unique. It has been identified by a number of US High End Computing Revitalisation Task Force (HECRTF) white papers[1] that a better interfacing and instrumentation of scientific application software and hardware is needed. Software simulations have been identified as the only mechanism that provides a better understanding of software behaviour and execution on a hardware. In order to obtain a better fraction of sustained-to-peak performance, tools should be detailed enough to capture the design details with high fidelity, at the same time being driven by actual workloads. Moreover, the models should allow scalability studies for parallel workloads to be performed in the simulator without re-designing a whole new system. The HASE platform is considered capable of addressing these challenges and is the simulation framework for the application-specific QCDOC architecture exploration — the UKQCD computer simulation project.

2.2 HASE

A computer architecture and its implementation can be represented in a number of ways and at different levels of abstraction. At the highest level these could be multiprocessor systems, while at the lowest an architecture may consist of an ensemble of transistors on a VLSI chip (or set of chips). A variety of simulators has been designed for each different level of abstraction, and some design tools can generate implementations automatically at lower levels. The HASE enables computer architects to create architectural designs at different levels of abstraction and to explore designs through a graphical interface. It provides freedom to the simulation designer to choose an appropriate level of abstraction. The output of a simulation run can be used to

[1]Available at: <http://www.nersc.gov/news/hecrtf.html>

animate the graphical display of the design, allowing the user to visualise the behaviour and interactions between hardware components as programs (software) are run on it. Working at the processor architecture level, for example, a designer could investigate bottlenecks in the instruction pipelines and try alternative design strategies to eliminate them. In order to inspect the effect of alternate design configurations, HASE allows designers to incorporate parameterisation of hardware components. In addition to the parameter space, HASE provides infrastructure for the simulation of systems via a design hierarchy, permitting models to be designed, simulated and experimented at an appropriate level of abstraction.

Coe *et. al.* [CHIW98] described the HASE as an environment that permits rapid development and exploration of computer architectures at multiple levels of abstraction, encompassing both hardware and software. In a technical report [CIRW98], it was anticipated that HASE can be considered as a platform capable of supporting hardware software co-design. A recent extension to HASE is JavaHASE [MI03]. JavaHASE converts existing HASE simulation models into fully interactive simulation applets accessible via the WWW.

2.2.1 HASE Building Blocks

The design of the HASE environment centres around the concept of component-oriented programming, which is very close to the object-oriented programming paradigm. HASE has therefore been implemented in an object-oriented language C++. In HASE, design components of a computer system can be represented at different levels: primitive gate level, register transfer level and complex processing node level.

Furthermore, many complex systems of concurrently interacting components can be more easily understood in a graphical description rather than a textual description. Likewise, a static and textual description cannot precisely capture the dynamic behaviour of hardware and software interactions in complex uni- and multi-processor computer architectures. Thus, the user interface to HASE is via a graphical design window and users view the results of simulation runs through animation in the HASE design window.

The main features of the HASE environment include an Entity Description Language (EDL), an Entity Layout (EL) file, a discrete-event simulation engine Hase++, a visualisation mechanism and results gathering tools. The EDL definitions of the architectural components provide information about architecture parameter definitions, component parameters and ports, hierarchical on-screen structure, global system parameters and component interconnect. Predefined multiprocessor topology templates are also included in EDL to aid in the rapid development and exploration of multiprocessor networks. The EDL description, when combined with the Entity Layout file containing display information, completely describes the architecture to be simulated.

Once the architecture is loaded into the HASE system from an EDL file, the simulation executable can be generated by combining the architecture information and user defined parameters with the individual component behavioural descriptions. The behaviour of entities is described in Hase++. Hase++ is a process-oriented, multithreaded, discrete-event simulation engine. According to the *discrete-event* simulation approach, the components of a simulation model consist of events. These events are activated at certain points in time, thereby affecting the overall state of the system. Events are discrete so between the execution of two events nothing happens. Design and implementation details of various discrete-event simulation engines can be found in [SB01].

The library Hase++ is implemented in C++; it provides an event-based approach of defining an entity behaviour. Using Hase++ library functions, a user can create process-oriented, discrete-event simulation models. Within the *process-oriented* approach, the components of the program consist of entities, which combine several related events. Hase++ includes a set of library routines to provide for not only process-oriented, discrete-event simulation but also a run-time system for multithreading many objects in parallel and keeping track of simulation time.

2.2.1.1 HASE Operation Modes

Presently, there are four modes of operation in the HASE that facilitate in the design, implementation, testing and experimentation of a simulation model:

1. **Design Mode** — The operations available to the designer in Design mode allow a designer to furnish the graphical layout of a simulation model. A simulation designer can adjust the on-screen design components (HASE entities) and save the layout information.

2. **Build Simulation Mode** — Build mode creates an executable simulation of the architecture. The options available include the type of simulation to be created, for example, trace file generation or building in the debug mode. Moreover, a simulation designer can specify a trace level, which determines how the events generated during a simulation run are filtered for the simulation trace file.

3. **Simulate Mode** — In simulate mode, the trace file is generated as a result of a simulation run. From the trace file the graphical display of the design can be animated. Simulation mode allows system parameters to be changed and timing diagrams for system components to be viewed.

4. **Experiment Mode** — Multiple automatic executions of the simulation are performed in the experiment mode, with different parameter settings used in each execution.

Each mode provides the designer with a different set of menu options and depending on the option selected, different operations can be performed. The currently active mode determines the functionality of the menus attached to the components, for example, when in 'Simulate System' mode, memory components have an option to load new memory contents.

A HASE project includes `project.edl` and `project.elf` files for the project design and layout, HASE entities' behaviour files (`entity.hase`), and additional C/C++ files and libraries that may be specified by a simulation designer. During build project mode, HASE generates C++ files from the behavioural Hase++ files along with a Makefile, project's header and C++ files. 'Generate and make' option in the project build mode creates a simulation executable. Input parameters information and values are stored in the `.params` file. Figure 2.3 shows the steps of generating a HASE executable from various files.

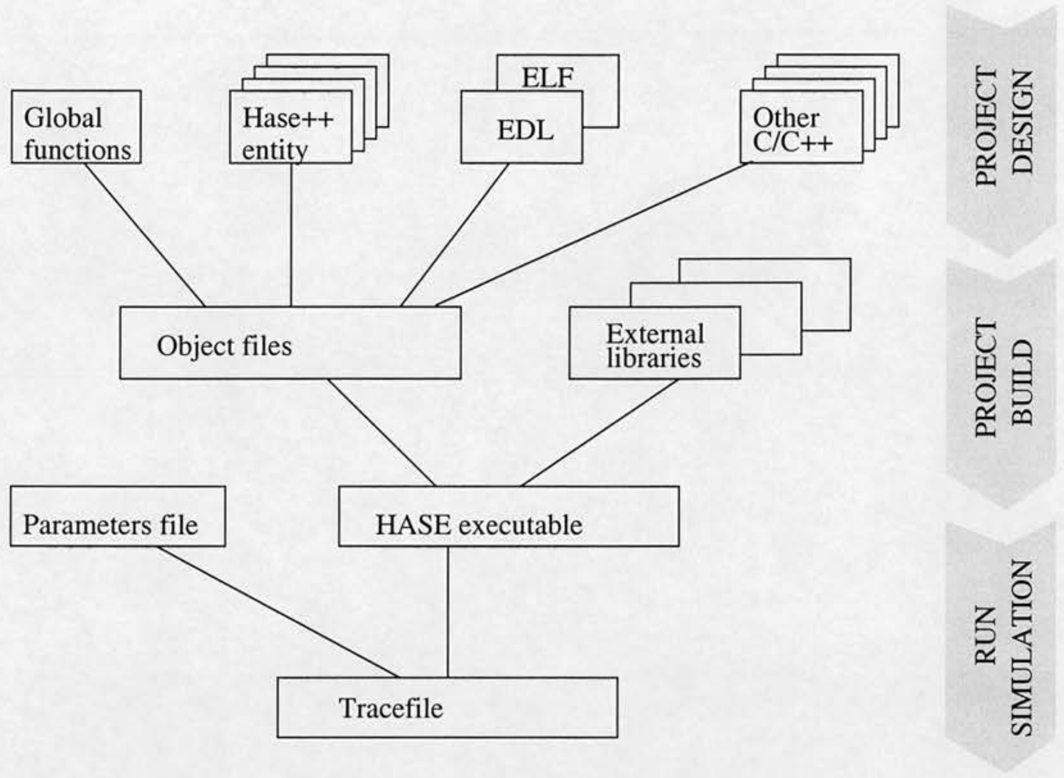


Figure 2.3: HASE simulation model — design steps

When the simulation executable is run, it uses user-supplied information as input for the user-defined parameters. If the experiment mode is being used, the range and increment values set up for selected parameters using the experiment control dialogue. The code generation options available in build mode allow the simulation to be run in different ways. If generate (trace) was selected, then during a simulation run, the HASE writes an event sequence into an event trace file and this trace can subsequently be played back to provide the user with a visual display of activity in the system. The trace is produced automatically from the simulation model without the need for the user to write explicit animation code.

2.2.1.2 HASE Design Templates

Software simulation are specifically useful for the study and exploration of parallel computers. Design replication is a key feature of tightly-integrated parallel systems, for instance, in an MPP system, processing nodes generally have identical physical properties (hardware design). The concept of replication has been exploited in the HASE by introducing templates for the common structures. The user can slot any component into the template (including hierarchical components), and the HASE does the rest. Current templates include general n-dimensional meshes and tori `MESHnD` and for shared and distributed-shared memory cluster computer design, HASE provides `BUSENTITY` and `NETWORKENTITY` templates. Templates allow the automatic creation of models by the designer, insertion of a user-defined entity into a predefined topology, and the provision of dimension parameters; this is a useful mechanism when modelling scalable systems.

The HASE `MESHnD` template can be used to connect any number of entities in a n-dimensional mesh format; figure 2.4 shows an application of a HASE `MESH2D` template.

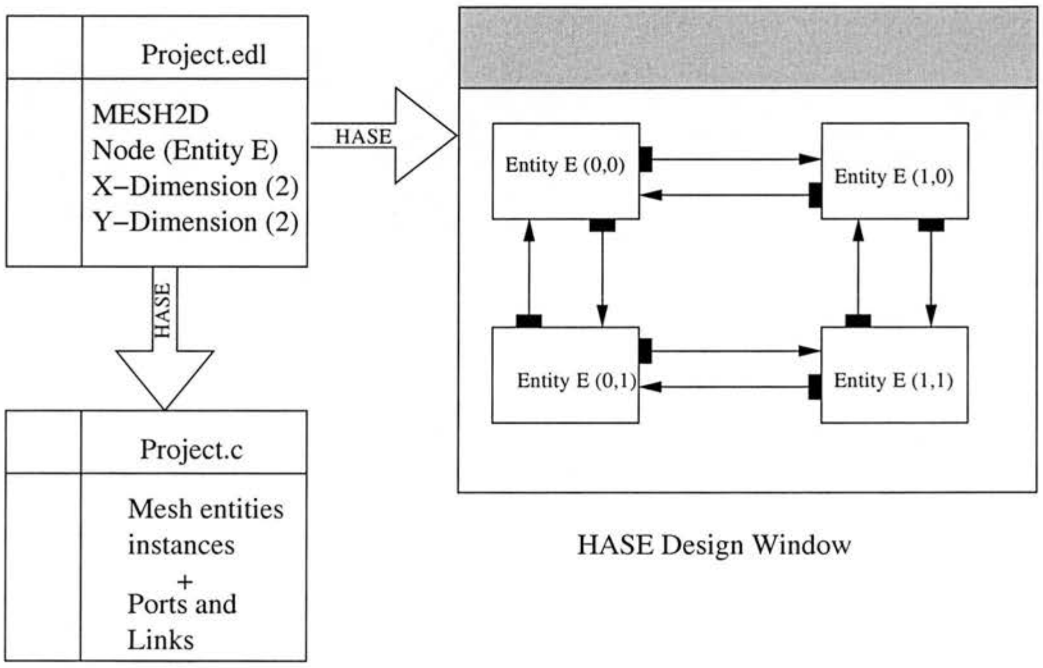


Figure 2.4: HASE template generation mechanism — MESH2D

In the EDL file, all a user has to do is to insert a basic HASE `ENTITY` or a high-level compound entity `COMPENTITY`, which is composed of pre-defined HASE entities, in a template definition. The number of *nodes* in each dimension is a parameter, which can be specified in the EDL file description. HASE then generates new instances of the *Node* entity along with ports and links (as required). Once a template has been created, a higher hierarchical level can be formed by linking the entity's instances; again templates could be used for this to link many sub-templates together. Furthermore, the HASE allows different hierarchical levels to be displayed simultaneously, enabling the level of on-screen simulation complexity to be controlled easily.

2.2.2 Parameterisation

In HASE, a large design space for a modelled system can be explored through the use of parameterised components and component libraries. These parameterised components enable changes to the architecture design configurations to be made quickly and easily. The ability to change and experiment with these parameters, and to monitor the effects whilst the simulation is running, provides the designer with instant feedback on these adjustments without the need to run the entire system simulation. Figure 2.5 represents a typical simulation cycle in HASE.

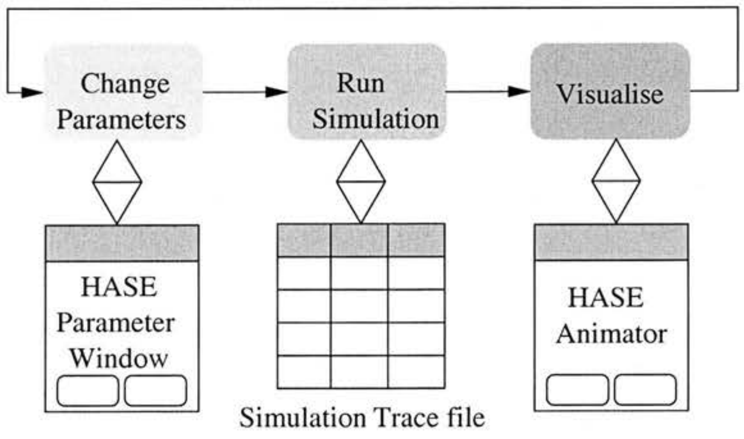


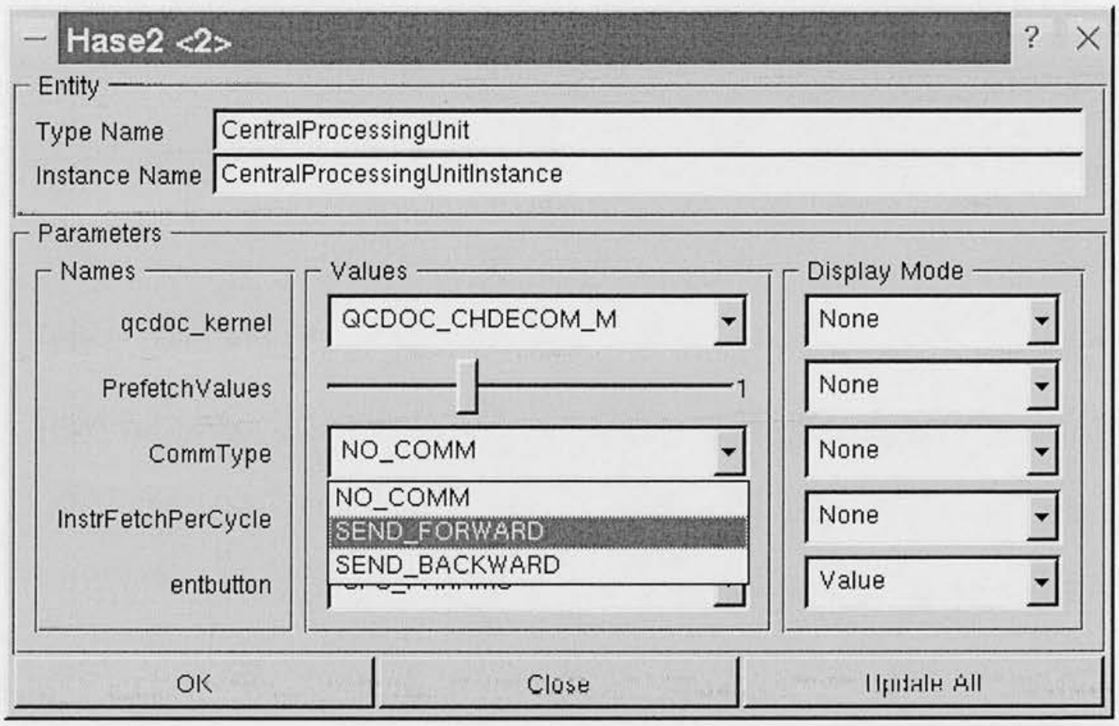
Figure 2.5: Steps involved in running a HASE simulation model

First, a user decides the configuration of the system or combination of parameter values through entities' parameter windows. Second, a trace file is generated as a result of a simulation run. Finally, the simulated system can be animated to observe the affect of parameter values on the simulated system entities during a simulation run. The results gathering capability in the experiment mode is facilitated with an ability to execute repeated simulation runs automatically. With an updated input parameter value in each simulation run, the overall results can be plotted as graphs of the output parameters as functions of the varying inputs.

A HASE entity EDL definition is shown in figure 2.6, and figure 2.7 shows the corresponding parameter window. These parameters can be changed via a graphical interface and after a parameter change, the simulation will run with a new value. For instance, changing `InstrFetchPerCycle` from 2 to 4, means that four instructions would be fetched by the entity instead of two in the following simulation run. Similarly, selecting an alternate QCD kernel function would instruct the entity to execute a new kernel.

```
ENTITY CentralProcessingUnit(
        EXTENDS(Biclocked)
        DESCRIPTION ("CPU")
        PARAMS (
        static RENUM(t_qcdoc_kernel,qcdoc_kernel,0)
        static RRANGE(PrefetchValues,0,3,0)
        static RENUM(t_CommType,CommType,0)
        static RRANGE(InstrFetchPerCycle,1,4,2)
............
```

Figure 2.6: EDL description of a HASE `ENTITY`


Hase2 <2>

Entity

Type Name: CentralProcessingUnit

Instance Name: CentralProcessingUnitInstance

Parameters

Names	Values	Display Mode
qcdoc_kernel	QCDOC_CHDECOM_M	None
PrefetchValues	1	None
CommType	NO_COMM	None
InstrFetchPerCycle		None
entbutton		Value

NO_COMM
SEND_FORWARD
SEND_BACKWARD

Figure 2.7: An `ENTITY` parameter window

The ability of HASE to control the complexity of the on-screen simulation, combined with animation features, aids in the understanding of the operation of the simulation as well as verification of its correctness. The graphical features range from simple colouring schemes to animating certain packet types for node synchronisation. Combining the multiple levels of on-screen complexity with the simulation visualisation tools enables the designer to perform an interactive visual verification of a model's behaviour.

2.2.3 Visualisation Capabilities

The visualisation mechanism allows the graphical display of an architectural model to be animated by reading in trace files generated by the simulation execution, thereby enabling the designer to inspect the model for correct operation. The hierarchical nature of HASE controls the displayed complexity of the simulation, according to the areas of the model being concentrated on.

The HASE animator control is shown in figure 2.8.

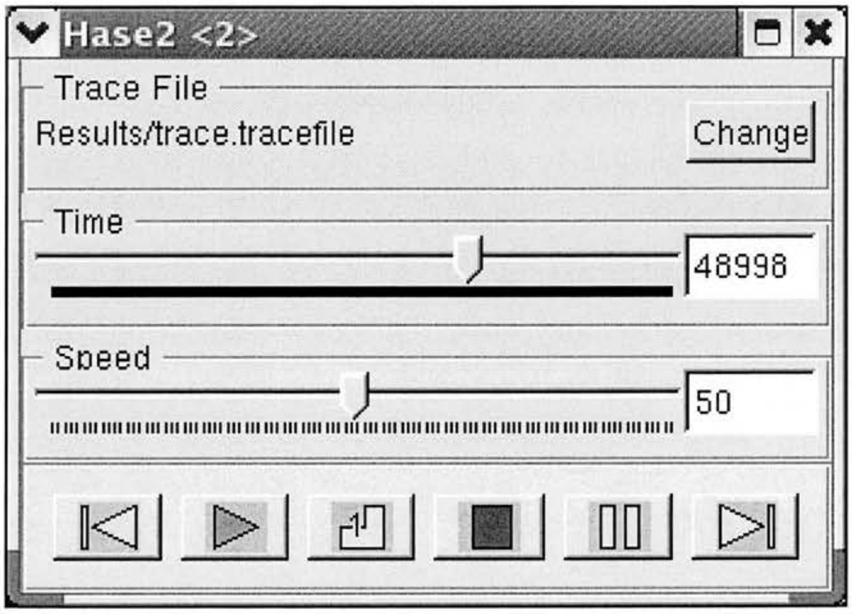


Figure 2.8: Animator control interface

The animator allows a user to observe and to validate the interactions between the components executing concurrently. There are six control buttons, from left to right: rewind, play, single step, stop, pause and fast forward. Activity in the simulation model can be visualised in a variety of ways; through moving icons showing data packets passing between components in the design window, by changing a component's icon to reflect its current state or by displaying the contents of registers and memories. A simulation user can establish the correctness of the model by tracing the animation steps back and forth.

The HASE also provides the facility to generate timing diagrams, allowing the states of the individual components and component parameters to be monitored for a given simulation run. A timing diagram viewer tool reads a HASE trace file and produces a timing diagram against the simulation time of:

- values of the entities' parameter; and
- communications amongst entities (send and receive events).

The animation facility, together with the timing diagrams, allows verification of system design at the earlier design stages and helps in understanding the details of dynamic behaviour of test software on the underlying hardware.

2.2.4 Hardware-Software Co-simulation

Hardware/software co-design techniques have been successfully employed and explored in computer architecture design for several decades. Co-design is a concurrent approach to systems design and there are several key issues to be addressed, these are specification, partitioning, synthesis, simulation and design-space exploration. Most systems are designed by applying these issues separately to the design of the hardware and the software. Co-design methodologies and environments attempt to merge these design paths so that expensive mistakes are not made when assigning components to hardware and software. Similarly, when modelling a computer system the hardware-software interactions have to studied to better inform and understand the system for a given workload. A detailed analysis of hardware-software behaviour is essential when understanding and establishing load-balancing and scalability of a parallel system.

The co-design and co-simulation concepts are slightly different. Co-design is normally associated with the system synthesis or gate level simulation designs. At a higher abstraction level, for instance, register transfer or block diagram level, co-simulation is the appropriate approach. Co-simulation, the ability to include hardware and software components of a system design in a simulation, is an important feature of HASE. The software components are usually included by loading the required code into a memory component, and then executing the code through a processor component. The level of detail at which the processor and memory components are simulated affects the balance between the accuracy and speed of the simulation, as well as the complexity of the code. The ability of the HASE to deal with these different levels of component abstraction is another useful feature of the environment. Like hardware abstraction levels, various software interpretation levels can be employed in a HASE model.

There are several features of the HASE that make it a suitable system to be used for the exploration of a variety of different trade-offs within a co-simulation methodology.

For example, before a simulation run, a user can select a separate memory file or update memory contents because test software components, in most cases, are not hard-coded in the entity's behaviour. Similarly, the workload of a system can be controlled via the HASE parameter control interface provided the designer specifies and implements the facility in the EDL and behaviour files. Figure 2.9 shows a clocked HASE entity with a memory file.

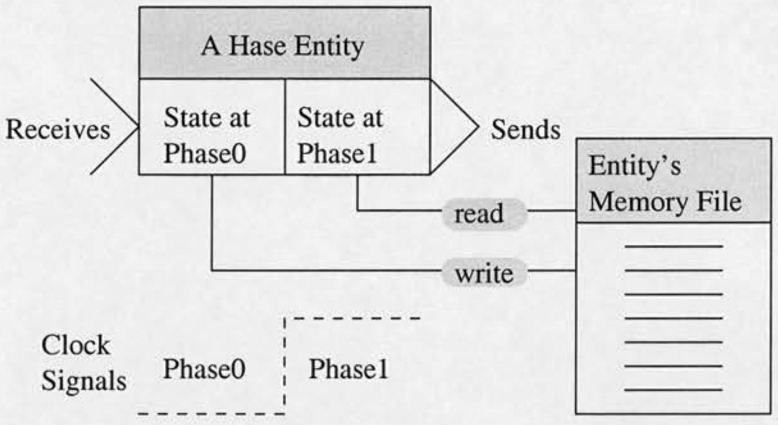


Figure 2.9: Example of a bi-clocked HASE `ENTITY` with a memory (`.mem` file)

At run-time the user can associate separate memory files with the entity, or select a memory file depending on the user input parameter.

2.2.5 HASE Projects: Past, Present and Future

There are a wide range of applications, from individual components design to multi-processor architecture research, for which the HASE has been proved to be a suitable design framework. The Stanford DASH architecture was designed to be a scalable high performance machine with multiple coherent caches and a single address space. The HASE simulation model of the DASH architecture by William and Ibbett [WI96] concentrated on implementing and illustrating the cache coherency protocols.

The Hierarchical PRAM model of parallel computation was simulated on a 2D mesh. The goals of the HPRAM project were to investigate the scalability and efficiency with which this model of computation might be implemented on realistic parallel architectures.

The HASE DLX simulation project [Ibb00] employed the HASE to simulate instruction-level parallelism concepts using the DLX architecture. Undergraduate students use the DLX simulation model to learn about the complicated control and data paths using the DLX scoreboarding and to investigate the performance of the architecture for several combination of pipeline latencies and input test code in the instruction memory.

The template generation mechanism of the HASE has been extended by Coe [Coe00] in a PhD project that investigated a large number of cache coherency protocols on a wide range of distributed-shared memory multiprocessor architectures.

The HASE is being used in a collaborative industry-academia research project called Storlite.[2] One of the aims of the Storlite project is to explore, through the HASE simulation, the design space of storage network architectures, to estimate the performance of different architectural configurations and thus to highlight areas where there is potential to improve performance by judicious use of new or emerging technologies and to explore the possibilities for new design ideas.

The UKQCD computer simulation project [HASb] uses the HASE to conduct performance studies of a parallel computer called QCDOC, which is specifically designed and optimised for QCD applications.

2.3 QCD and Parallel Computers

Gupta [Gup99] stated that simulations of lattice QCD are 'tailor made' for parallel computing, while Bowler and Hey [BH99] considered QCD calculations as 'ice-breaker' applications for research and advancement of the future generation parallel computers and their programming paradigms. Over the last two decades, one of the driving forces behind the development of the massively parallel computers has been the determination of theoretical physicists to solve the QCD equations. The 'Cosmic cube' supercomputer at the Caltech (1982), the Columbia University special purpose machine since 1982, the APE computers at Rome and Pisa (since 1986), the GF11 supercomputer at IBM (1986-1997), and the Japanese projects (QCDPAX and CP-

[2]For details visit <http://www.icsa.informatics.ed.ac.uk/research/groups/hase/projects/storlite/>

PACS) are examples of such projects that have successfully produced QCD results and were considered to be among the fastest supercomputers of their time. A number of these application-specific research initiatives, including the Cosmic cube and IBM GF-11 supercomputers, later spawned commercial parallel computer designs.

QCD experiments have not been limited to custom-built parallel computers. The lattice QCD community has also been amongst the first and most efficient users of commercial supercomputers of all kinds; vector supercomputers (CRAYs, ETA10) and massively parallel computers (Connection Machines, Intel Hypercube and Paragon, IBM SP2 and Silicon Graphics Origin 2000). Consecutively, computational lattice QCD researchers have contributed to the overall development of high performance computing by educating a generation of skilled computational scientists, by interacting with computer vendors to help to optimise and test new hardware and software, and by providing an environment and a successfully implemented application that motivated other communities to use novel parallel architectures. It is because of achievements made by the lattice QCD simulation on parallel architectures, Cabibbo *et. al.* [CIS99] asserted in the guest editorial, an issue of the Parallel Computing journal was devoted to convey an overview of the spectrum of activities in lattice QCD to non-specialists.

Beside the success stories of the parallel QCD calculations, the fundamental structure of matter, defined by the particle physics theories, poses great challenges to the high performance community. Bowler and Hey [BH99] wrote that the QCD study is a grand challenge computational problem, i.e., several QCD calculations need computational power not available from today's most powerful supercomputers. Bailey and Simon [BS92] defined a *grand challenge* as a fundamental problem in science and engineering, with broad applications, whose solution would be enabled by the application of high performance computing resources, which could become available in the near future. According to Bowler [Bow98], the enormous computational cost can only be addressed by improvements in computer power, better QCD algorithms and parallel software practices.

In order to understand the key characteristics of QCD calculations, which are considered as one of the highly parallelisable but compute-intensive applications, an overview of the following is presented:

- the underlying theory, which translates into a natural parallelism in the application;
- parallel QCD algorithms, software and hardware designs that realise the theory; and
- challenges, approaches and solutions to solve the grand challenge QCD calculations.

2.3.1 Theoretical Background

Presently, there are two major classes of physics: classical physics and quantum physics. In classical physics, gravity and electromagnetism are the two forces of nature and a force is conceptualised as ripples in the field or waves. A field or wave stretches out from an object and conveys a force. On the contrary, in quantum physics, particles are the carriers of force and force is exchanged between quantum entities. Quantum physics defines four forces of nature: gravity, electromagnetism, strong and weak nuclear force. The last two forces cannot be observed in everyday life because they operate at the nuclear level. These are called strong and weak nuclear forces. Electromagnetism and the strong force are treated in the theories of Quantum Electro-Dynamics (QED) and Quantum Chromo-Dynamics (QCD) respectively.

Gusken [Gus99] stated that at the current level of comprehension of the fundamental principles of nature, physical processes on an atomic or sub-atomic scale can be successfully described by the Quantum Field Theory (QFT). In QFT, particles, as well as their interactions, are represented by quantum fields, defined at each space-time point. QCD and QED are QFT theories.

Out of the four fundamental forces of nature, only three are included in the Standard Model of particle physics; the fourth, gravitation, has so far not been quantitised on the basis of standard gauge theories. Although gravitational force is important in the everyday world, it is weaker than all the other forces in the microscopic world and can be neglected in the calculations.

2.3.1.1 Standard Model

The Standard Model of particle physics describes the world in terms of fundamental particles and their interactions. Matter is made up of fundamental particles, quarks and leptons, and the force of interaction is provided by gauge bosons (the gluons). The complexity of these interactions comes from the fact that particles can take on different properties called quantum numbers, but more importantly they can be combined into larger units according to the rules set by their interactions. Three quarks combine to form a baryon while a meson is a pair of a quark and an anti-quark. Mesons and baryons are members of a larger class of particles called hadrons. The combination of quarks into more complex particles is not arbitrary, since they are intimately controlled by their interactions. Everyday particles, at the atomic level, are not hadrons or leptons, but atoms. An atom consists of protons, neutrons (baryons) and electrons (leptons) as shown in figure 2.10, and molecules that are made up by several atoms.

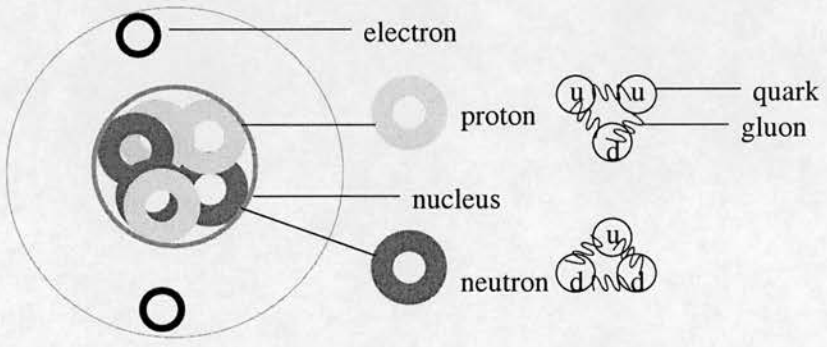


Figure 2.10: Structure within an atom

Quarks and gluons exist in several unified states: baryons, mesons and glueballs. The proton is the only baryon stable in isolation. In contrast, neutrons are not stable in isolation, though they can be stable inside certain nuclei. Mesons are particles with a basic structure of one quark and one anti-quark. There are no stable mesons. Only a few types of mesons live long enough to be seen directly by their tracks in a detector. Theoretically, there are additional types of hadrons called glueballs; made purely from gluons. Some observed particles are thought to be glueball-like. The classification of these combined states are represented in table 2.2.

Hadron	Description	Example
Baryon	Made of three quarks (q,q,q)	Proton
anti-Baryon	Made of three anti-quarks (q-,q-,q-)	anti-Proton
Meson	Made of a quark and anti-qurak (q, q-)	Pion
Glueballs	Made from gluons only	not known

Table 2.2: Known bound states of quarks and gluons

2.3.1.2 QCD: A Complicated Theory

The fundamental particles in QCD, quarks and gluons, have a property called colour. Like electric charge, colour charge is always conserved. But unlike electric charge, the colour charge (the chromo in chromodynamics) comes in six varieties, three colours and three anti-colours. The colours are usually called red, green, and blue. Protons and neutrons, collectively called hadrons are made up of quarks and have no colour charge, i.e., they are colourless or colour-neutral. The colour charge is used to describe the fact that, even though each quark in a hadron has a different colour, red+green+blue=white, the result is a colour neutral object. Another class of particles, called mesons, have a quark and an anti-quark (colour+anti-colour=white). Gluons carry colour/anti-colour pairs (not necessarily the same colour). There are 8 gluons since there are eight possible colour/anti-colour combinations. In total, there are four properties: "charge", "spin", "colour" and "mass". Spin is the name for the angular momentum carried by a particle. That data structure, in a typical parallel QCD calculation, for representing a quark is quite complex because of the four properties associated with each quark.

QCD is considered to be a complicated theory, so complicated in fact that extracting predictions from it is impossible by using pen and paper methods. In fact, extracting predictions from any QFT is a difficult thing to do. Before QCD came along people studying elementary particles typically resorted to an approximation method known as perturbation theory. For many applications, including QED, perturbation theory has proved to be a successful approach.

The perturbation theory is only partially useful for QCD calculations. At the high energies found in high-energy particle accelerators, the QCD perturbation theory experimental results match with the simulation results. However, at low energies, the perturbation approximation fails. It is extremely difficult to do a number of QCD calculations using the perturbation approximation, for example, prediction of proton mass. QCD physicists have therefore been rigorously exploring numerical and experimental methods. Figure 2.11 compares complexity of QCD against QED. Representation of QCD data structures and interactions in a parallel code are therefore far more complex and non-linear than QED.

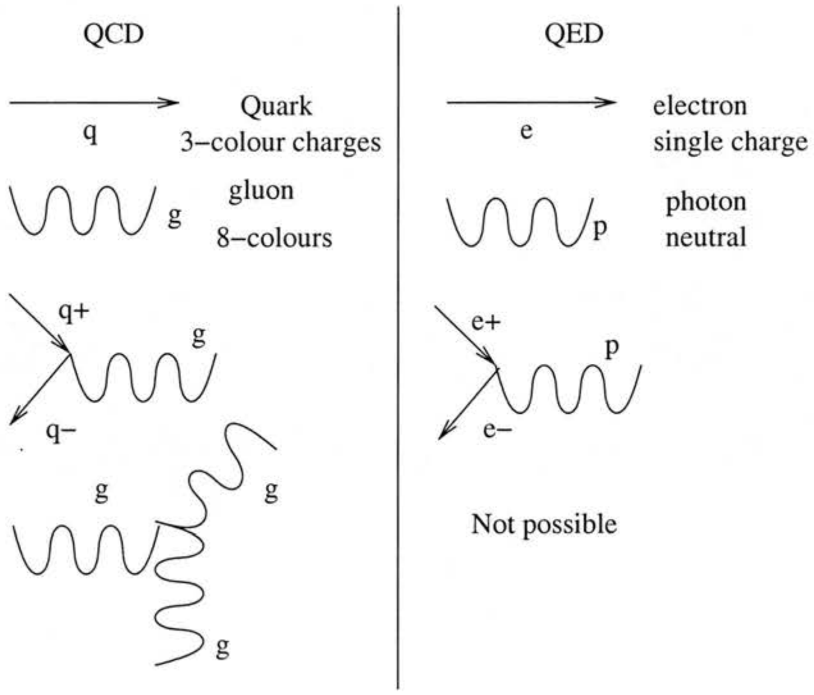


Figure 2.11: QCD complexity vs. QED

2.3.2 Lattice QCD

The experimental approach to QCD is hugely expensive; it involves building huge particle colliders to study, for example, what happens when protons and anti-protons collide at high energies. One innovation in the field is the realisation that modern computer power can be used to analyse interactions at low energies and reduce the need for such accelerators (or complement experimental studies).

In 1974 Nobel prize-winner Ken Wilson introduced a new technique to solve QCD, called lattice QCD. His idea was to break space-time up into discrete units. Instead of considering a continuum of positions and times, each space-time point can represent specific values. For instance, in the real world the position of an object could be anything from 1 metre away from another object to 10000 metres away, and anything in between. In Wilson's scheme there would only be a finite number of positions these two objects could have (e.g. some integer multiple of 1 metre).

Wilson's idea did not make QCD solvable by hand (using pencil and paper). To this day, nobody really knows how to get exact solutions out of QCD. However, Wilson's lattice idea made QCD amenable to computer solution. With the space time continuum replaced by a discrete lattice of points, powerful computer techniques could be brought to bear on the problem. Initial results in the late seventies and early eighties were considered promising. Of profound interest was the strong numerical evidence which predicted that quarks would be permanently confined, a result that could never be observed experimentally (Ukawa [Uka99]). Before that there was indirect evidence for their existence because they exist in confinement with other quarks, called hadrons, which can be detected by real experiments.

Such properties of quarks defined by QCD theory which are experimentally impossible to observe and verify are studied in computer experiments. Theoretical predictions can be deduced from the real experiments with hadrons, as the strong interaction can be unfolded by mapping the basic equations of QCD on a computer and matching the computed numbers to experimental data. These hadron experiments are central to QCD research [Gus99], since most of what one meets in everyday life is made of protons and neutrons (hadrons). Even with the lattice formulation of QCD, computer experiments without perturbative approximations are hugely expensive. It was realised that for non-perturbative QCD simulations, enormous computer power would be required. The simplest QCD calculations, in most cases, demand huge computing resources, and consequently much of lattice theorists' efforts in the first 20 years of lattice QCD were spent in accumulating computing time on the world's largest supercomputers, or in designing and building special purpose computers far larger than the largest commercially available computers.

2.3.2.1 Fundamental Particles on a Lattice

In a widely used lattice QCD textbook 'Quarks, Lattices and Gluons' [Cre83], Creutz explains lattice QCD as an approximation to QCD; the key simplification is the replacement of continuous space-time by a discrete grid — a standard methodology in numerical analysis. The nodes or "sites", of the grid are separated by lattice spacing a, and the length of a side of the grid is L. The QCD fields are specified only on the sites of the grid, or on the "links" joining adjacent sites; interpolation is used to find fields between the sites.

Figure 2.12 shows the mapping of a quark and gluon onto a three-dimensional lattice.

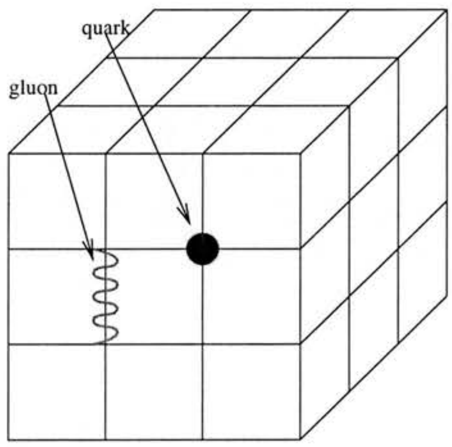


Figure 2.12: Quark and gluon mapping onto a lattice

In the lattice approximation, the path integral, from which all quantum mechanical properties of the theory can be extracted, becomes an ordinary multidimensional integral. The numerical integration is over a large number of variables and Monte Carlo methods are generally used in its evaluation. The Monte Carlo method is a method of approximately solving mathematical and physical problems by the simulation of random quantities.

2.3.2.2 Path Integral

In order to appreciate the complexity of QCD calculations it is essential to introduce the famous Feynman path integrals and QCD actions. The essence of the path integral approach is quite simple; to calculate the probability of a particle going from A to B. One should take account of every possible path from A to B, not just the most direct route, or the trajectory described by classical mechanics (figure 2.13(a)).

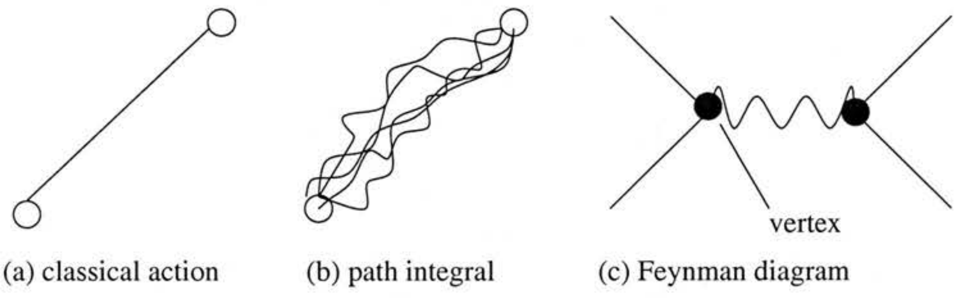


Figure 2.13: Feynman path integral formulation

Each path has a certain probability, which is related to the action associated with the path. Gribbon [Gri98] defined an action as a mathematical quantity. An action depends on the mass, velocity and distance travelled by a particle or the way energy is carried from one place to another by a wave. These actions are governed by laws like QCD. The overall probability is calculated by integrating the contributions from all paths (figure 2.13(b)). Although this means adding an infinite number of probabilities, most are negligible and cancel out, and the classical trajectory emerges as a consequence of the quantum probabilities (figure 2.13). Feynman, one of the most notable physicist of the last century, introduced the structure of the diagram, which is shown in figure 2.13(c). A Feynman diagram follows specific rules where at each vertex (point where two or more lines meet). A vertex is often represented as a series of complicated equations.

2.3.2.3 Lattice QCD Computation

The first step in a lattice QCD calculation is to prepare a vacuum or ground state, since the relevant physics information is enciphered in the QCD vacuum or ground state field configuration. To create an ensemble of vacuum configurations, 'Hybrid Monte-Carlo (HMC) algorithms' (Kennedy [Ken99] QCD algorithm for parallel computers) are generally applied. HMC is a Markov process[3] with global updating, based both on deterministic evolution through phase space and stochastic changes.

Gusken *et. al.* [GLS99], in explaining the goal of accumulating integrated performance of hundreds of sustained Teraflop-hours, identified the importance of the two distinct phases inherent in a QCD simulation:

1. the stochastic generation of field configurations and
2. the computation of the physics observables of interest to extract hadronic properties, in the form of stochastic averages over these configurations.

The two phases generate their own characteristic load requirements on the computer system and these phases can be organised into separate computing tasks. Phase (1), on the one hand, is compute intensive with little I/O, on the other hand, it is regular and therefore well suited for cost-effective and restricted compute engines, whereas phase (2) is much more data intensive but involves a much lower computational load, therefore lending itself to implementation on general purpose MIMD systems with adequate storage facilities.

2.3.2.4 Cost of QCD Simulation

Lepage [Lep99] deduced the cost of QCD simulation, which is governed by a formula of the form

$$cost \propto (\frac{L}{a})^4(\frac{1}{a})(\frac{1}{m_\pi^2 a})$$

The first factor is the number of lattice sites in the space-time grid, and the remaining factors account for the "critical slowing down" of the relaxation algorithm used in the numerical integration. The Lepage formula shows that the single most important determinant of cost is the lattice gauge spacing. The cost varies as the sixth power of $\frac{1}{a}$, implying that one ought to keep the lattice spacing as large as possible. L

[3]A process with a sequence of events where the probability of an event occurring depends upon the fact that a preceding event occurred.

is the length of the lattice site and m_π is called the meson trajectory. All quantities are expressed in consistent units, for example, in fermi (1 fermi is equal to 10^{-15} meter).

Figure 2.14 shows a path integral in two three-dimensional lattices; one with a large spacing a containing a small number of lattice points (called lattice sites) per volume, and the second, a *fine* lattice with small spacing and a considerably larger number of lattice points.

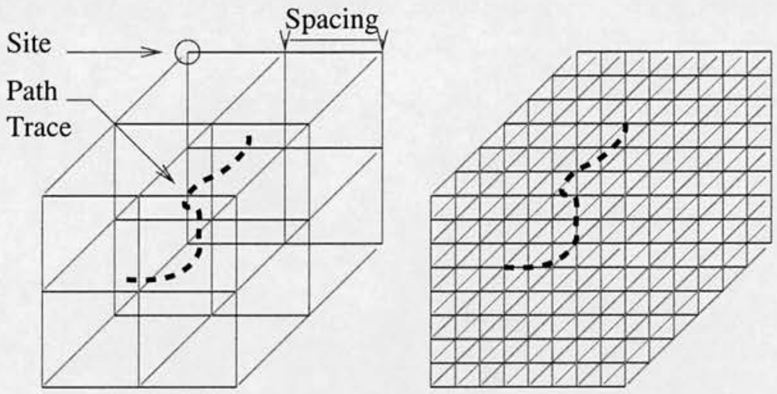


Figure 2.14: Path integral mapping onto a lattice

Finer lattices tend to approximate the path integral with minimum discretisation errors at an expense of increasing computation cost. Increases in computation cost are associated with the memory and execution overhead of a large number of lattice sites per compute node.

Using computers, it has proved extremely difficult to extract precise information from a lattice simulation. Computer generated results in most cases are less than 10% accurate when compared with experimental results. Lattice QCD is found to be enormously expensive computationally when attempts are made to improve accuracy of the lattice calculations. According to the current understanding and estimates of the field, a full solution will require a computer with a performance of at least hundreds of Teraflops. Research directions include development of advanced computer architectures and algorithms, software optimisation, low energy physics calculations to certify hardware and software systems, and application of the methodology in particle physics research areas such as the identification of new types of matter.

2.3.2.5 Limitations of Lattice Formulation

Monte Carlo algorithms are stochastic and there is an associated statistical error in estimates of expectation values. Lattice QCD calculations are also subject to systematic errors. Bowler [BH99] described two obvious sources: first, the finite box size per force used for simulations, $L = Na$, and second, the non-zero lattice spacing a. Making the physical lattice volume large for a given number of lattice points in each dimension may require too coarse computational grid and the results suffer from discretisation errors. Conversely, reducing the lattice spacing in the physical units means that the simulated system will be 'squashed', thereby introducing finite-size effects.

Ideally, the calculations should be repeated for a range of lattice spacing and volumes to confirm whether the answers are independent of spacing a and volume in a direction L. Due to the huge computation cost associated with a single lattice QCD calculation, the task of repeatedly performing a QCD calculation with varying input variables could have exponential cost. Keeping the physical box size fixed and halving the lattice spacing requires doubling the number of grid pints N in each dimension. This tradeoff is shown in figure 2.15.

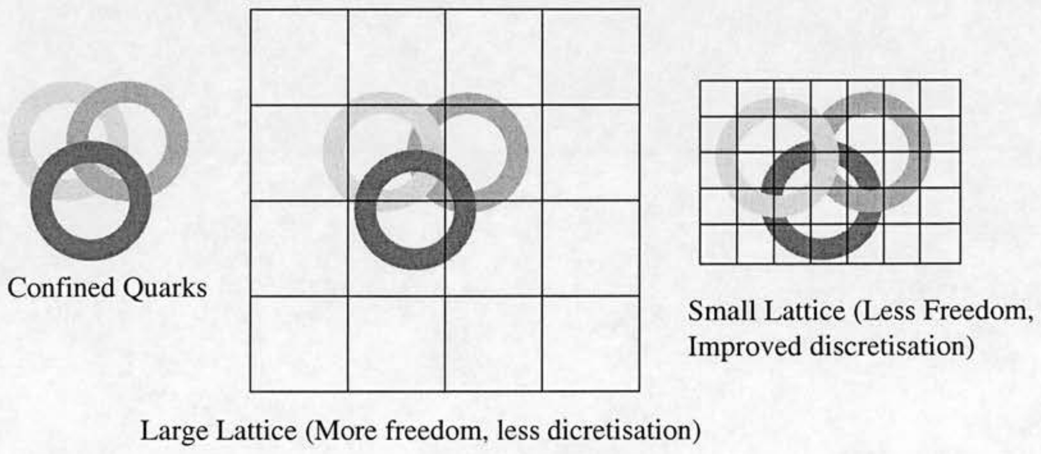

Confined Quarks
Small Lattice (Less Freedom, Improved discretisation)
Large Lattice (More freedom, less dicretisation)

Figure 2.15: Lattice QCD mapping — tradeoff between small and large lattice sizes

Unfortunately, the scaling properties of many QCD algorithms are not ideal. In full QCD, with least approximation, the most popular method for generating configurations incorporates the effects of virtual quark-antiquark pairs is the HMC algorithm. In HMC computations, the number of arithmetic operations scales roughly as N^{10}.

2.3.3 QCD Algorithms

In lattice QCD, large systems of linear equations have to be solved to compute physical quantities. An exact analytical treatment of the path integral function is not possible in most cases. The space-time continuum is approximated by a lattice with N_s^3 x N_t space-time points. The calculation of physical quantities is done in two steps. First, one generates a representative sample of quantum field configurations, where each configuration is represented according to its specific weight, by a Monte Carlo procedure. Secondly, one determines the "would be" value of the physical quantity in question on each of the quantum field configurations and takes the average.

The most time consuming part of the calculation, Sroczynski [Sro02] described, involves inverting a matrix for the quark field. QCD simulations are largely based on variants of the Monte Carlo algorithm. The numerical bottleneck

$$M\psi = (1 - \kappa D)\psi$$

is solved by various algorithms including the minimal residual or the Conjugate Gradient (CG) method. M is called fermion (quark) matrix of dimension $r = 3 \text{ x } 4 \text{ x } V$. Where V is the volume of the underlying four-dimensional space-time lattice, and D contains non-diagonal elements only. The size of the solution vector is of order O(10^7) to O(10^8) elements, κ is a scalar quantity, and ψ is a vector field.

Naively, the calculation of disconnected contributions requires the inversion of a complex matrix of size $(N_s^3 \text{ x } N_t \text{ x } 12)^2$ on each single quantum field configuration. For currently available lattice sizes, such a calculation would be prohibitively expensive. To circumvent this problem, stochastic estimator methods are applied. These methods converge to the true result in the stochastic limit. It turns out that even with such techniques, one still needs a parallel supercomputer to handle this problem.

Further improvement of the inversion algorithm can be obtained by using preconditioning techniques and novel discretisation techniques. A preconditioning algorithm should reduce the number of iterations and the computing time necessary to achieve a given accuracy. Lippert [Lip99] describes a widely used parallelisable preconditioning procedure in lattice gauge computation, which is based upon an odd-even decomposition of the matrix M. Other decomposition and precondition techniques used incomplete LU factorisation and a preconditioning scheme. Several discretisation techniques of QCD have been employed to solve a wide range of problems in lattice QCD on ordinary desktop workstations, while at the same time, substantially extending the physics reach of the supercomputers available to the field.

2.3.3.1 The Conjugate Gradient (CG) Algorithm

QCD computations are normally carried out with iterative solvers like CG and its variants. Ruede [Rue97] provided a comprehensive account of some iterative solvers including CG on parallel high performance architectures. CG is an approximate method, i.e. it results in an answer closer to the approximation limit required by a QCD simulation. Fox [FJL$^+$88] explained this procedure as follows:

Considering a task of solving M simultaneous QCD equations would be equivalent to finding the M-dimensional vector which minimises some residual error quantity defined on the M-dimensional QCD space. A gradient method makes use of trial values for the variable at step i to generate new values at step i+1 corresponding to a reduced value of the error function. By successively moving towards the error minimum, the method converges toward the desired solution (with desired approximation). Different gradient methods differ in their techniques for choosing the direction of the minimal trial vector. The CG method employs descent direction vectors, which are mutually orthogonal relative to an inner product weighted by the properties of the matrix.

For a given definite sparse matrix A and the vector b, a CG algorithm introduces three M-dimensional vectors and two scalars whose contents change in each iteration. The iteration number to which a given M-dimensional vector applies is expressed in the subscript in the CG description below. x_0 denotes the starting trial solution (which could be zero). The other two vectors required are r_k, the residual vector, and p_k, the CG search direction.

$$x_0 = 0 \;\; (\texttt{initial\_guess})$$

$$r_0 = b - Ax_0$$

$$p_0 = r_0$$

Then the following iterative loop labelled by k is performed

```
For k = 1 to MAX (until converged)
        $\alpha_k = \frac{(r_{k-1} * r_{k-1})}{(p_{k-1} * Ap_{k-1})}$
        $x_k = x_{k-1} + \alpha_k p_{k-1}$
        $r_k = r_{k-1} - \alpha_k A p_{k-1}$
  Stop when
        $r_k = 0$ or(converged)
  else
        $\beta_k = \frac{(r_k * r_k)}{(r_{k-1} * r_{k-1})}$
        $p_k = r_k + \beta_k p_{k-1}$ (Loop)
```

The iterations are terminated when x_k has converged to within some desired accuracy, as determined by the magnitude of the vector of residual r.

The CG algorithm does not depend upon any particular row or column structure, it only requires matrix A to be symmetric. Matrix A is used in the algorithm to form its product with M-dimensional vectors. It is possible to compute the products separately and sum them afterwards. Hence, matrix A can be distributed among parallel processors and a given processor may need access to parts of A residing in neighbouring processors.

2.3.4 Computational Characteristics of QCD Calculations

According to the Caltech Concurrent Computation Group classification of parallel applications [FWM94], QCD simulations are considered as *synchronous* applications. Synchronous applications have a regular structure, and in general, are the simplest to code and, in particular, the simplest to parallelise. These applications are characterised by a basic algorithm. A parallel algorithm, in most cases, consists of a set of operations that are applied identically at every point in a data set. The structure of the problem is typically very clear in such applications, and they can be considered as parallel in nature.

Trippiccoine [Tri99] identified certain characteristics of the lattice QCD calculations:

- The lattice QCD simulations are based on compute intensive kernels. Typically the ratio of operations to required operand is large (between 4-8). A limited memory bandwidth could sustain a huge floating-point performance.
- Physical lattices could be easily mapped onto several independent processors. A measurable performance loss due to parallelisation overheads is expected only for a number of processors well beyond any reasonable upper limit.
- All processors are only required to execute exactly the same program, on independent copies of the same data structure. Theoretically, one program flow has to be controlled.

2.3.4.1 Coding Lattice QCD on Parallel Computers

The basic dynamical variables of QCD are the gluon field and the quark field. On a four dimensional lattice of size N_x x N_y x N_z x N_t, the gluon field is represented by a set of complex 3 x 3 matrices $U(n,\mu)$, where n = (n_x, n_y, n_z, n_t) denotes lattice sites with $1 < n_{x,y,z,t} \leq N_{x,y,z,t}$ and $\mu = x, y, z, t$ the four dimensions. The quark field in the Wilson formulation is represented by a 12-component complex vector. The objective of lattice QCD simulations is to numerically evaluate, by a Monte Carlo method, the Feynman path integral. The action of lattice QCD describes the interaction of quarks and gluons. Typically the main simulation steps are:

1. update of the gluon configuration to generate distribution.
2. gauge fixing to reduce statistical noise in the measurement of the observables.

3. solver to compute the quark propagator for a number of given quark sources, where the quark matrix is a sparse $12V$ x $12V$ complex matrix depending on the gluon configuration and $V = N_x$ x N_y x N_z x N_t.

4. measurement of hadron observables by combining quark propagators.

The whole cycle is repeated several hundred or several thousand times. The algorithm for the update parts differ if full QCD simulations or quenched (an approximation) QCD simulations are performed. In both cases the computer time is mostly spent in the update and the solver part, with the update part weighted dominantly for full QCD simulations.

From a computational view point, numerical algorithms used to simulate lattice QCD have certain simplifying features. The calculations are homogeneous and the interactions are local. The first feature implies that exactly the same operations need to be done at each space-time point. It is therefore trivial to make synchronisation between points. The second feature implies that only a few neighbouring points are connected at best, thereby resulting in data dependence between small blocks (2^4 - 4^4), or none at all in the most time consuming part of the calculation (the matrix inversion). A point in each of these blocks can be processed simultaneously. In short, the problem is 'tailor made' for distributed-memory parallel computing.

The four dimensional space-time grid is divided into smaller hypercubes, and the data and calculations needed to process all the sites within each smaller hypercube is allocated to a separate processor. For example, a 64^4 lattice can be distributed as 16^4 sub-lattices on 256 processors setup as a 4^4 computational grid to maintain the spatial proximity of data. Points internal to this 16^4 volume do not need information from the neighbouring processors and can get the maximum throughput of the single processor's speed. On CRAY T3D QCD calculations, Berry *et. al.* [BGK94] explained the communication of lattice points on the boundary. These communications are said to be homogeneous on a three-dimensional torus topology. For example, each processor needs information from the processor on the right and simultaneously needs to send information to the processor on the left. Furthermore, the use of periodic boundary conditions makes the pattern of these communications cyclic on the four-dimensional grid of processors. Lastly, it is possible to overlap computation and communications.

The communication of the boundary points can be initiated while the points internal to the 16^4 volume are being processed. The only constraint is the amount of per processor memory to store this extra data. With the amounts of memory available per compute chip on todays computers, the extra storage requirement is not a major hurdle.

2.3.4.2 Communication Requirements

According to Gupta [Gup99] the Achilles heel of large distributed computers is inter-node communication speed and memory bandwidth if commodity processors are to be used. In this regard too, lattice QCD calculations are ideally suited for implementation on parallel computers. One can examine the worst scenario of the penalty imposed by communications in cases where overlap of computation and communication is not permitted by either the hardware or the software. The basic arithmetic unit in the lattice QCD calculations is the multiplication of a complex 3 x 3 matrix to a 3 x 1 complex vector, where the vector needs to be communicated to the neighbouring nodes. Analytically, three floating point operations are performed for every single byte of data communicated. In practice, the situation is even better as data for only the points on the boundary of the hypercube in each node is communicated. Lastly, global sums and global broadcasts, which are potentially slow, are not done often enough to be a significant overhead.

2.3.5 Parallel QCD Computers

After more than two decades of research, QCD still has not been solved using a non-perturbative analytical approach. A controlled numerical treatment of the theory on the lattice on extremely fast parallel supercomputers is widely considered to be the only viable scheme to extract quantitative physical results. The results from lattice gauge theory simulations are urgently needed as theoretical input for current and future accelerator experiments to facilitate the attempts to observe new physics beyond the Standard Model of elementary particle physics.

Christ [Chr99] identified that the QCD problem offers different possibilities for the design of optimised computer hardware. First, the floating-point operations that dominate QCD Monte Carlo computation can be performed with considerable speed and great cost-effectiveness by a class of specialised chips which have been manufactured with rapidly improving characteristics. Second, the Monte Carlo updates and sparse matrix inversions, the most expensive part of QCD calculations, are easily done in parallel. Consequently, machines combining many of these floating point chips can operate with great efficiency.

The enormous floating-point operation requirements of the stochastic simulation, in order to achieve statistically significant physical results, has led to a concentration of activities in this field of research. The construction of specialised QCD computers has now developed into a major activity in high-energy physics with a number of large and resourceful groups spending large sums of money to build a variety of machines each rivalling, or exceeding the capabilities of commercial supercomputers. Several collaborations in Japan (CP-PACS) [ABK$^+$99], the U.S. (MILC), Columbia, Caltech, IBM [KBD93], Fermilab, United Kingdom (UKQCD), and Italy APEs (APEmille [Tri99] and APE100 [ABDS95]) are investigating QCD systematically either on special-purpose or commercial high-end parallel supercomputer hardware performing with several hundreds of Gigaflops. In addition to special-purpose machine construction, QCD theory has been explored on commercial, general purpose computers including IBM SP2 [BDG$^+$95], cluster computers [BPE$^+$99], Connection Machine [BBJ91] and CRAYs (T3D [BGK94] and T3E [McN00]).

Parallel with the QCD hardware development, several software initiatives have been developing the software infrastructure for the QCD parallel application software. The Columbia Physics System[4] (CPS) of the Columbia University is a high performance QCDSP-specific software repository. The UKQCD collaboration currently possesses optimised Fortran kernels. In addition to the CPS and UKQCD code, a number of open-source repositories are available: MPI-base, multi-platform MILC code [MIL], macro-based data-parallel programming system SZIN [SZI] and C++ classes, functions and parallel algorithms for lattice QCD, based on Matrix Distributed Processing FermiQCD [Pie00].

[4]Information available at <http://phys.columbia.edu/~cqft>

2.3.5.1 Current Application-Specific QCD Computer Projects

Currently two major projects are under way to build the fastest, cost-effective, parallel computer for QCD calculations: QCDOC and apeNext. QCDOC is a 10-Teraflop MIMD supercomputer being developed through a collaboration between the IBM and Columbia University. A 5 Teraflop prototype is expected to be constructed by 2004. A second QCDOC system will be installed in the UK (UKQCD collaboration). apeNext is the next generation of APE computers, built by a collaboration of leading European QCD research institutes including DESY and INFN. It is a three dimensional grid of processing nodes and the whole system is a SPMD processor. apeNext is expected to achieve a substantial fraction of peak performance (50%) on a 15 Teraflops machine. It is expected to be operational after 2003. Earlier generations of APE machines are based on SIMD architecture. In addition to these application specific supercomputer developments, several cluster computers around the world have been exploited for QCD calculations. An up-to-date list of QCD clusters was presented by Lippert [Lip03] at the Lattice 2003 annual conference.

2.3.5.2 QCDOC

The QCDOC architecture has been designed to provide a highly cost-effective, massively parallel computer capable of focusing significant computing resources on relatively small but extremely demanding QCD calculations. It is not the first architecture of this type; in late 1990s two large QCD on Digital Signal Processors (QCDSP) machines provided an aggregate 1 Teraflops for the lattice QCD calculations. Mawhinney [Maw99] presented a detailed account of how the design and construction of cost-effective MPP QCDSP machines, which cost just a fraction of a commercially available supercomputer were realised. QCDSP machines have been providing supercomputer power to the QCD research community for the last five years.

Many in the QCDOC design team [CCC$^+$01] consider QCDOC as a natural evolution of the QCDSP. The expected increase in the performance of QCDOC is mainly attributed to the technological advances in processor and in communication network technology. The individual processing nodes in the QCDOC system are 500 MHz PowerPC-based processors interconnected in a six-dimensional, 12 Gigabit/sec mesh with the topology of a torus. A QCDSP processing node is 50 MHz DSP-based, connected in a four-dimensional, 0.64 Gigabit/sec torus. Out of the six dimensions, QCD calculations on the QCDOC system are expected to utilise a four-dimensional torus for the QCD calculations; the other two dimensions have been included for software partitioning of the machine to avoid re-cabling. A second Ethernet-based network provides booting and diagnostic capability as well as more general I/O. Over ten thousand compute nodes are expected to be packaged in a style that provides a small footprint.

An essential element of the QCDOC project is the collaboration with IBM, thus enabling the use of IBM's system-on-a-chip (SOC) technology. Nowadays it is possible to integrate most of a processing node's components on a single chip, creating an application-specific integrated circuit, or ASIC. Gara *et. al.* [GAB$^+$02] described QCDOC as the first SOC MPP supercomputer in an article on the IBM Bluegene/L supercomputer. Commodity components based SOC MPPs are presently considered the best design option for cost-effective scientific supercomputers.

2.3.5.3 Physics Goals of the UKQCD Computer

The lattice QCD research goal is to study experimentally accessible but theoretically difficult to calculate properties of nuclear matter. Kenway [Ken00] said the central aim of the UKQCD's physics programme is to quantify the effects of realistic light dynamical flavour quarks on phenomenologically important quantities. In addition to these demanding calculations of light quarks, a number of physics goals of the QCDOC computer were identified by Isgur and Negele [IN00] in a proposal submitted to the U.S. department of energy. The UKQCD and the US lattice community proposals suggested that progress in understanding QCD, verifying QCD, calculating strong corrections to weak matrix elements, and elucidating the behaviour of QCD at finite temperature requires a combination of four things:

1. Better theoretical understanding and formulation of quantities to calculate.

2. More efficient numerical algorithms for the generation of background configurations and quark propagators.

3. Improved ways of discretising QCD to reduce errors and computing requirements.

4. Specifically designed higher performance computers and their exploitation.

The success of the current UKQCD computer, QCDOC, will not only enable in achieving (1), (2) and (3) but it will set a milestone for (4).

2.4 Summary

High performance parallel computers are fundamental to the progress in the QCD research. QCD theories are generally defined as a set of equations, which in turn are solved by algorithms for solving complex set of equations. High performance supercomputers are essential for solving data- and compute-intensive QCD calculations. The UKQCD collaboration, together with leading QCD physics research groups and IBM, are in process of constructing the fastest computer for the QCD calculations. The success of a parallel computer lies in delivering a sustained performance for the application code. In order to establish the performance of the application code and to investigate the design of future generation parallel computers, extensive instrumentation is needed. Computer architecture simulation is an efficient, flexible and cost-effective way to explore performance and scalability of an application over underlying parallel hardware systems.

The current release version of HASE includes facilities for designing and creating parameterised hardware-software co-simulation models. Key HASE design features include the provision of simulation templates and component reuse, which provide an efficient and flexible mechanism for the rapid prototyping and exploration of scalable architectures incorporating system hardware and application software characteristics [IHH95]. The UKQCD computer simulation research, exploits the HASE framework to get an insight into the performance phenomenon of the latest UKQCD computer, QCDOC, for parallel QCD application. The next chapter details the internals of the QCDOC architecture and implementation of parameterised HASE QCDOC simulation model.

Chapter 3

Design and Implementation

This chapter presents the design and implementation of the QCDOC computer simulation model in the HASE. Firstly, a brief overview of the QCDOC machine architecture and the design details of the processor core and custom components are provided. Secondly, key considerations in modelling the complete system in the HASE and an account of the HASE design entities are presented. Thirdly, the notable extensions made to the HASE for the QCDOC architecture modelling are outlined. Lastly, the HASE tools constructed for the debugging, validation and experimentation of the QCDOC model are described.

3.1 QCDOC Architecture

QCDOC is defined as a custom-built, massively parallel computer optimised for lattice QCD calculations using system-on-a-chip (SOC) technology. A SOC processing node in the QCDOC system primarily contains commodity design components (known as IP blocks) including IBM standard macros. In addition to the commodity components, a number of hardware features are unique to the QCDOC design; these are mainly to assist scaling and to improve performance of the lattice QCD applications on parallel computing systems.

3.1.1 Design Philosophy

Presently, high-end parallel computing efforts for lattice QCD calculations include: commodity off-the-shelf-clusters and custom-built systems like QCDOC. It has long been debated which computing platforms are most suitable for the simulation of lattice QCD. Lüscher [Lus02] argued that recent off-the-shelf commodity PC processors by Intel or AMD are capable of delivering impressive floating-point performance on the order of 1 Gigaflops if vector processing units[1] are employed and cache management is optimised. Unfortunately, for parallel QCD applications, this single-node performance cannot be fully utilised in a PC cluster if the local lattice volume (problem size per processing node) becomes small. In this case, the surface-to-volume ratio (i.e., lattice sites per processing node) is large, and the communications latency inherent in standard network solutions such as Ethernet or Myrinet slows down the individual processors. Boyle *et. al.* [BCC$^+$01] concluded that in order to achieve reasonable efficiencies with a PC cluster, the local lattice volume must not be too small. This implies that for a given total lattice volume, the number of nodes working on a single problem is limited. To run many important QCD calculations on a moderately large lattice in a reasonable amount of time, it is essential to distribute the total volume onto as many nodes as possible, which implies very small local lattice volumes, as small as 2^4 (two lattice sites in each space-time direction). A small local volume requires communications between neighbouring nodes with extremely low latencies that cannot be achieved using off-the-shelf networking components. Massively parallel machines with custom-designed communications hardware, with considerably lower latencies than a commodity interconnect, appear to be the only viable alternative. In the field of lattice QCD calculations, with its very regular communications requirements, custom (low-latency, switchless) interconnects are especially feasible. The design of the QCDOC architecture revolves around supporting a small local volume design philosophy.

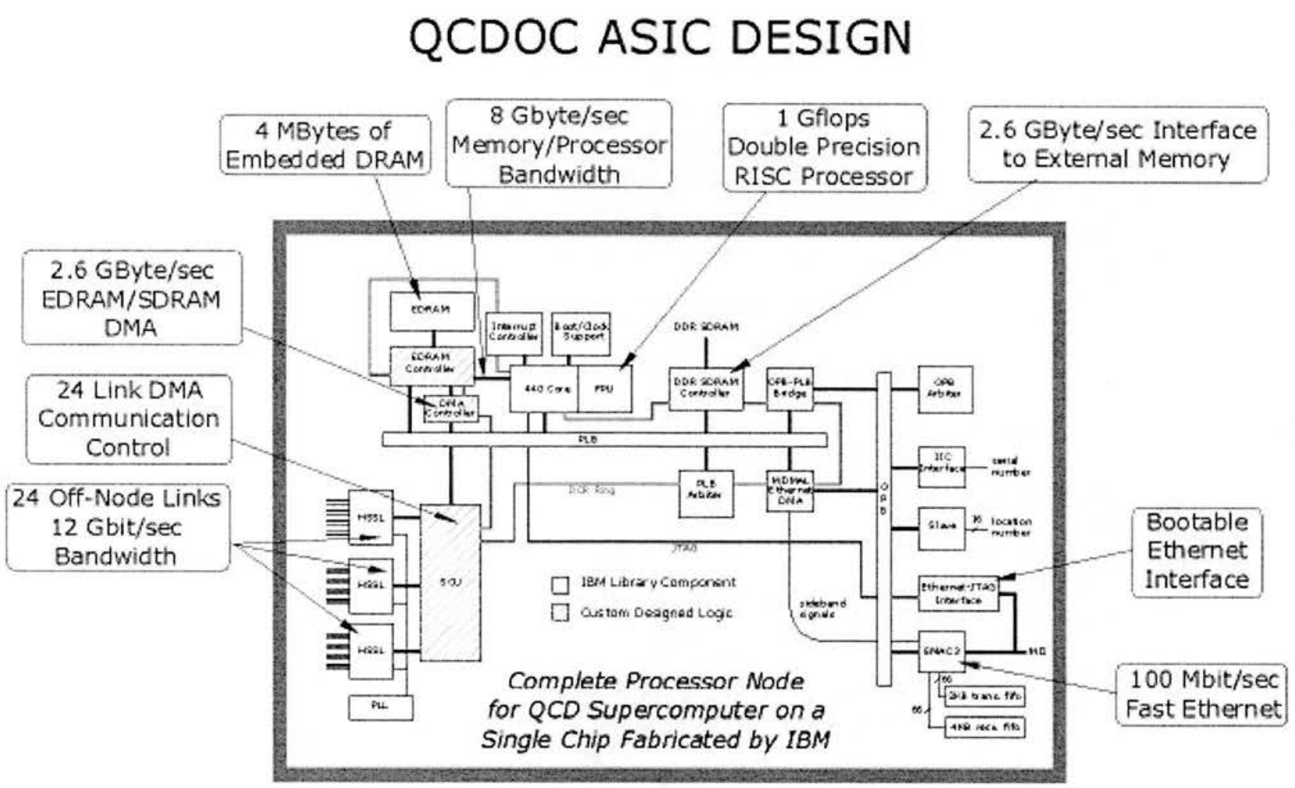
[1]Vector processing unit (VPU) handles vector-based, single-instruction multiple data (SIMD) instructions that accelerate graphics operations. Such vector-based instructions include Intel's multimedia extensions and Streaming SIMD Extensions (SSE). In some cases, there is no discrete VPU section; Intel and AMD incorporate those functions into the the FPU of their Pentium 4 and Athlon CPUs.

The rapid advances in silicon feature size, single-chip functional integration and communications technology offer tremendous opportunities to exploit the QCDOC design philosophy to build high-end, powerful and cost-effective machines. The QCDOC design is based on a high-performance, highly integrated Application Specific Integrated Circuit (ASIC) that combines all computation and communication logic on a single chip. By embedding computation logic on a single chip, considerably smaller communication latencies in terms of processor clock cycles are achieved for the off-node communication operations. The process of integrating all functions on a single chip is often referred to as SOC design technique, thereby introducing the architecture name QCDOC for "QCD-On-Chip".

The design blocks in the official QCDOC ASIC (figure 3.1) contribute to the start-up, loading, execution and file I/O operations of parallel QCD application code.

3.1.2 System Overview

A QCDOC node comprises a single ASIC chip and an industry standard Dual Inline Memory Module (DIMM) memory. The prominent features of the ASIC are IBM's embedded PowerPC 440 core, the double-precision Floating Point Unit (FPU) core, a 4 MByte on-chip memory called "Embedded DRAM" (EDRAM) and QCD specific design blocks. To a large extent, the ASIC is created from already existing IBM macros that are simply interconnected in the specified way to create the larger unit. Special to the QCDOC design is the Serial Communications Unit (SCU), the Prefetching EDRAM Controller (PEC) and the DMA controller permitting direct transfers between external and embedded DRAM. The QCDOC design report [BCC$^+$02b] details the design components shown in figure 3.1. A brief overview of the design blocks of the QCDOC processing node along with their key responsibilities in the system is outlined next.


QCDOC ASIC DESIGN
4 MBytes of Embedded DRAM
8 Gbyte/sec Memory/Processor Bandwidth
1 Gflops Double Precision RISC Processor
2.6 GByte/sec Interface to External Memory
2.6 GByte/sec EDRAM/SDRAM DMA
24 Link DMA Communication Control
24 Off-Node Links 12 Gbit/sec Bandwidth
Bootable Ethernet Interface
100 Mbit/sec Fast Ethernet
EDRAM
EDRAM Controller
DMA Controller
Interrupt Controller
Boot/Clock Support
440 Core
FPU
DDR SDRAM
DDR SDRAM Controller
OPB-PLB Bridge
PLB
PLB Arbiter
MDMAL Ethernet DMA
OPB Arbiter
IIC Interface
serial number
Slave
location number
Ethernet-JTAG Interface
EMAC2
OPB
JTAG
SCU
HSSL
PLL
sideband signals
IBM Library Component
Custom Designed Logic
Complete Processor Node for QCD Supercomputer on a Single Chip Fabricated by IBM

Misson-critical, custom logic (hatched) for high-performance memory access and fast, low-latency off-node communications is combined with standards-based, highly integrated commercial library components.

Figure 3.1: Official QCDOC processing node design [BJW03]

PowerPC 440 Core

As an embedded processor core, PowerPC 440 is designed and optimised for low power, high performance and custom embedded system designs, and therefore is an ideal candidate for the processor in a massively parallel system. The PowerPC 440 is considered as the first embedded system core to reach the one Gigaflops boundary. It has an efficient Memory Management Unit (MMU) with separate instruction and data cache, and an efficient memory control mechanism via a Translation Lookaside Buffer (TLB). The PowerPC provides standard interfaces to support on-chip and off-chip data transfers. A detailed description of the core is presented in section 3.1.3.

Floating Point Unit

PowerPC 440 is a 32-bit integer core. It does not contain floating-point execution units. Nevertheless, PowerPC 440 supports a seamless interface to external co-processors via its Auxiliary Processing Unit (APU). The one Gigaflop, superscalar, 64-bit IEEE Floating Point Unit (FPU) was especially designed by IBM as a co-processor to the PowerPC 440 core.

Embedded DRAM

EDRAM is a low-latency, high-bandwidth on-chip memory and is central to the QCDOC ASIC performance. This embedded 4 Mbyte DRAM provides code and data storage on-chip. In the most demanding parts of the code, this memory is large enough to hold the entire QCD kernel code and data.

Prefetch Edram Controller

The QCDOC status report [BCC$^+$02a] highlights the significance of the Prefetch Edram Controller (PEC). PEC is a memory controller that provides an independent, buffered interface for the EDRAM to the PLB, the DMA controller and the PowerPC core. The latter connection is provided through a dedicated 500 MHZ, 128-bit PLB bus. Prefetching and buffering are included in the PEC, allowing two independent streams of sequential data to be efficiently read from EDRAM from each of these three ports. In particular, this unit provides an 8 Gbyte/sec bandwidth between EDRAM and the PowerPC's data cache.

High Speed Serial Links

The High Speed Serial Links (HSSL) are the physical units responsible for managing the high-speed serial communication in the six-dimensional torus network (the physics network). Incoming data to the HSSL units are byte aligned and forwarded to the SCU. Similarly, outgoing 8-bit data are serialised and clocked out at eight times the processor frequency. HSSL is an IBM macro that provides high performance serialisation and deserialisation for the serial data transfers. An HSSL controls four independent serial inputs and outputs.

Phase Locked Loop

IBM HSSL units need a Phase Locked Loop (PLL) for serial data transfers clocking requirements.

Serial Communication Unit

Boyle *et. al.* [BCC$^+$02a] provide a detailed account of perhaps the most unique hardware component of the QCDOC ASIC, the SCU. Beside the PEC, the SCU is considered as a QCD-only design component on the ASIC. It is responsible for off-node data movements to and from the on-chip memory. Communication requirements specific to the QCD are supported by additional storage and control units within the SCU (details in section 3.1.5). SCU communications are governed by a custom protocol, which requires bad data to be re-transmitted. Furthermore, the SCU provides the store-and-forward function that supports low-latency global sums and broadcasts.

Direct Memory Access Controller

The Direct Memory Access (DMA) controller has a generic DMA functionality, i.e., once initialised by the processor, it takes responsibility for a large number of memory transfers. In the QCDOC ASIC, the DMA controller allows automatic transfers over the processor bus between external SDRAM and EDRAM. It is expected to use full 128-bit width of the PLB and exploit caching capability of the DDR SDRAM controller.

Processor Local Bus and Arbiter

The Processor Local Bus (PLB) is the main on-chip bus. It is 128 bits wide, runs at one-third of the processor clock speed and contains three independent sub-buses: address, read and write. The PLB supports eight bus masters in total. The PLB arbiter is a standard macro that manages the control and arbitration signals on the PLB.

On-chip Peripheral Bus and Arbiter

On-chip Peripheral Bus (OPB) is a part of the IBM SOC bus technology. It is included to remove nonnessential loads from the more time-critical PLB and to provide the standard interface required by the Ethernet controller. The OPB arbiter is a standard IBM macro; it manages the control and arbitration signals on the OPB.

OPB-PLB Bridge

Communication between the two buses, PLB and OPB, takes place via the OPB-PLB bridge. It is a standard IBM macro. In QCDOC, it appears as a slave on the PLB and a master on the OPB.

Bootable Ethernet Interface

The Ethernet connection permits hard-wired Ethernet control of the JTAG[2] interface to the PowerPC. The JTAG interface provides complete control of the processor to the host computer through the Ethernet interface allowing processor reset and boot code loading directly to the PowerPC instruction cache. This unit has been specifically built to provide an additional debugging support for the chip.

[2]JTAG or Joint Test Action Group is a protocol designed to do in-circuit testing and debugging of memory and CPU resources. Many CPUs provide JTAG port/connection to connect a serial or parallel port on the host to the target CPU board.

Boot/Clock Support

SOC components operate at a range of clock frequencies. At start-up or system boot, the boot support mechanism creates the 500MHz clock and sequences the power-up of the chip as the voltages appear.

Ethernet Media Access Controller

The Ethernet media access controller interfaces to the OPB and the DMA Ethernet controller.

DMA Ethernet Controller

Like any other DMA, the DMA Ethernet controller can be configured to load and unload packets to and from Ethernet network.

DDR SDRAM Controller

The SDRAM controller provides a standard interface to a larger off-chip or external memory. It supports PowerPC page mode transfers and can buffer read and write operations for efficient data handling over PLB.

Interrupt Controller

Interrupts generated by the components of the QCDOC ASIC and additional external interrupts generated elsewhere in the machine are processed by the interrupt controller.

Location Number Slave

QCDOC machines (at UKQCD, Columbia and RIKEN-BNL Research Centre) are expected to exceed 10K nodes. For this size of machine, it is essential to have a system in place to identify individual nodes, for example, at boot-up. The location number slave unit uniquely determines the location of the ASIC within the larger machine. This includes location on the motherboard, location of the motherboard within the crate and of the crate within the larger machine.

3.1.2.1 Inter-node Communications

In Massively Parallel Processing (MPP) systems, inter-node communication latencies are far higher than any other memory or compute latency. Direct networks, where one processing node is directly connected to another processing node, are preferred because of the low overhead of inter-node communications in these networks. Dally [Dal90] and Agarwal [Aga91] produced analytical models that proved a k-ary n-cube network topology optimal for inter-node communications in high-end MPP systems. Meshes and tori are considered as examples of k-ary n-cube networks; QCDOC processing nodes are connected in the topology of a six-dimensional torus.

A QCDOC processing node has the capability of sending and receiving data from each of its twelve nearest neighbours in six dimensions at a rate of 500 Mbits/sec. The aggregate off-node bandwidth is 1.5 GByte/sec. Each communication link has a phase locked receiver and single-bit error detection with automatic re-send. Associated with each of these twenty-four communication channels, a DMA capability allows autonomous read/write operations from either EDRAM or external SDRAM. Instructions controlling the DMA transfers are stored as 32 sequences of block-strided-moves[3] located in 24 separate, on-chip register blocks. Since two of these six dimensions are used to partition the machine, only two-thirds of this communications bandwidth or 1 Gbytes/sec is available for a typical QCD calculation. Low-latency, global functionality in the form of an automatic "store and forward" capability is provided for efficient collective communication operations, mainly global sums and broadcasts.

3.1.2.2 Booting, Diagnostics and I/O

In systems of the size and dimension of QCDOC, start-up and diagnostic checks, and file input-output (I/O) are important design issues, perhaps as important as achieving a high performance for the application code. The Ethernet network takes the non-application code communication burden off the physics (6-dimensional serial torus) network. The Ethernet connections of four processors are joined together with a Fast Ethernet switch, the output from which is fed to a higher level switch that

[3]memory blocks separated by a fixed stride (| blocksize | stridesize | numberblocks |).

includes a Gbit Ethernet link. This Ethernet tree can be used in broadcast mode to provide boot code to the processors, to allow individual processors to be interrogated for diagnostic purposes and to permit easy connection to industry standard RAID disks, providing a large aggregate I/O bandwidth. A fully-functional debugger is also provided, allowing a multi-node, window-per-processor, source-code-based graphical debugging interface.

3.1.2.3 Mechanical design

In addition to the performance, debugging and I/O considerations, other electrical and mechanical factors are critical to the design and smooth working of an MPP system. QCDOC exploits the homogeneity of MPP machines to achieve a high degree of mechanical modularity. Two individual processors are mounted per daughter card, 32 daughter cards on a mother board and then 8 mother boards in a rather large crate with a single backplane. Cable connections are provided on the backplane for the off-node communications of each motherboard.

3.1.3 PowerPC 440 Core Architecture

The PowerPC 440 embedded processor core is an IBM standard product designed for high performance, design flexibility and low power requirements — targeted to custom embedded system designs. It is based on the PowerPC BookE Architecture,[4] an enhanced PowerPC architecture conforming to the specification for a PowerPC family of RISC Processors [May94]. A PowerPC 440 core contains three integer execution pipelines, thirty-two 32-bit integer registers, an MMU and a branch unit with branch and control registers. In addition to these components, the PowerPC provides standard interfaces such as read and write bus masters, an Auxiliary Processing Unit (APU) for co-processors, an interrupt control mechanism and other necessary SOC interfaces.

The PowerPC 440 superscalar core incorporates a seven-stage execution pipeline with three execution units, as shown in figure 3.2.

[4]BookE specifies a standard version for the PowerPC architectures, such that several generations and versions (by IBM and Motorola) of PowerPC processors are consistent and compatible.

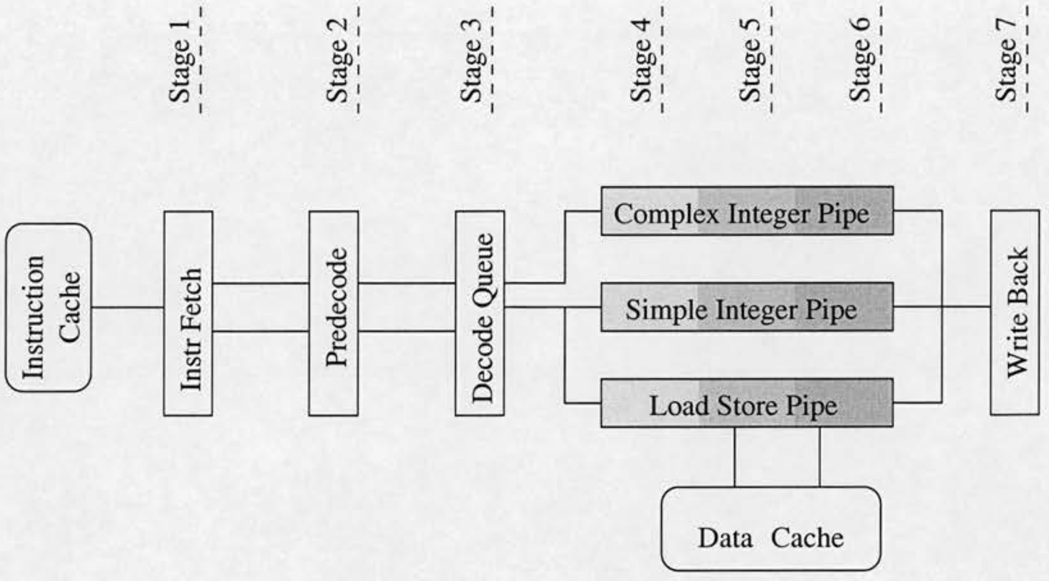


Figure 3.2: CPU execution pipeline

Up to two instructions can be issued per clock cycle. The three execution pipelines are:

1. **Load-Store Pipe** — This pipeline support all 32-bit BookE compatible integer and floating-point load/store instructions. For efficient execution, load requests can overtake store requests and there can be three outstanding loads in the load queue.

2. **Simple-Integer Pipe** — Arithmetic instructions that do not update one of the control registers are issued to the simple integer execution pipe. The control registers include the link register, the counter register and the condition register.

3. **Complex-Integer Pipe** — Branch instructions and instructions utilising one of the condition registers are issued to the complex integer pipeline. It has a single-cycle throughput but some versions of multiply and divide instruction can take multiple clock cycles.

Instructions in a clock cycle can be issued as a pair using (1) and (3), or (2) and (3). For added system performance, the PowerPC 440 includes dynamic branch prediction and 24 Digital Signal Processing (DSP) instructions. The PowerPC processor has a

Multiply and Accumulate (MAC) unit that supports single-cycle, DSP-style, multiply-and-add instructions. The PowerPC 440 core is combined with the floating point unit core via the APU for the custom-designed QCDOC ASIC.

Conditional and unconditional branches are handled by the Branch Unit (BU). It contains a 16-entry Branch Target Address Cache (BTAC) and a 4K-entry Branch History Table (BHT). The PowerPC 440 uses the BHT to maintain dynamic branch prediction of conditional branches. To perform dynamic branch prediction, a 2-bit counter in the BHT is used. Four valid states exist: "Strongly agree", "Agree", "Disagree", and "Strongly disagree". The BTAC is used to predict branches and deliver their target addresses before the instruction cache can deliver the same data. It is accessed during the instruction fetch stage, whereas normal branch prediction would not occur until the pre-decode stage, and therefore avoids a one cycle penalty for unconditional branches and one-step decrement branch `bdnz`.

Another important feature of the processor is its interrupt control mechanism. The processor internally recognises interrupts such as TLB miss, arithmetic overflow and QCDOC specific communication interrupts. All interrupts are mapped into interrupt handling routines using 16 vector registers to improve response time.

3.1.3.1 Memory Management

The PowerPC MMU has three separate caches: data cache, instruction cache and TLB. The data cache is connected to the PLB with 128-bit read and write bus masters while the instruction cache is connected with the instruction read master.

The 32 KByte instruction and data caches in PowerPC 440 are 64-ways set associative. The cache line size is fixed at 32 bytes (8 words). The PowerPC MMU supports non-blocking load operations and can have up to three outstanding load requests. For efficient load handling, the PowerPC MMU can serve load operations in advance of store operations.

A PowerPC cache memory array can be viewed as a two-dimensional array of sets and ways as shown in figure 3.3.

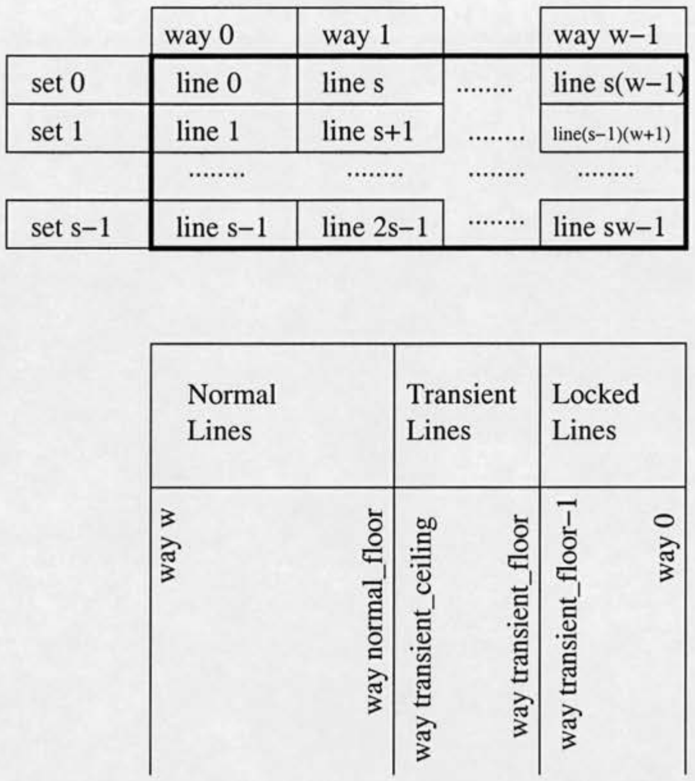


Figure 3.3: L1 cache layout

The PowerPC cache architecture has been extended to define three areas in the instruction and data caches: normal, transient, and locked. A cache line in the locked area is one that will not be replaced during normal read and write operations.

The definition of normal and transient areas is an innovative concept that provides the opportunity for significant performance improvements. The TLB (table 3.1) specifies memory pages as normal, which is the default, and transient (or locked).

1. Locked regions can be used for low-latency code or interrupt service routines.

2. Transient regions handle use-once data without disturbing the whole cache.

3. Normal regions are used for the rest of the cacheable data.

Data or instructions from normal pages only replace cache lines in the normal area of the cache. Likewise, data or instructions from transient pages only replace cache lines in the transient area. Specifying a transient cache area permits the PowerPC to perform repetitive operations on data arrays that stream through the cache, such as in packet processing, without overwriting data or instructions in normal areas that have long-term utility.

Normal and transient lines are replaced using a round-robin replacement policy. The store latencies of non-cacheable data can be compensated by a "store gathering" attribute, where instead of writing back each data block, the MMU gathers 128-bit data before sending it to a lower level. Store gathering is particularly useful for non-cacheable writes, where the cost of writing individual data packets to the lower level memories is high.

The TLB is used to control memory translation and protection. Each one of its 64 entries specifies a page translation. It is fully associative and can simultaneously hold translations for any combination of page sizes. To prevent TLB contention between data and instruction accesses, a 4-entry instruction and an 8-entry data shadow TLB are maintained by the processor transparently to the software. Software manages the initialisation and replacement of TLB entries.

A TLB entry specifies memory attributes on a per page basis. Table 3.1 shows the page size and memory attributes fields of a TLB entry.

TLB field	Description
Page number	Used in address translation
Valid bit	Indicates that the TLB entry is valid
Page size	4^{size} in KBytes
Cacheability	Whether the page is cacheable or not
Write policy	Write through or copy back
Region	Normal or transient memory/cache regions
Allocate policy	Whether to allocate a line on a write miss

Table 3.1: TLB fields

In addition to cache access control fields, there is provision for including custom-defined memory attributes. The MMU provides address translation, flexible memory protection, and storage-attribute control. It supports multiple page sizes and a variety of storage-protection attributes and access-control options. Multiple page sizes in the

TLB can improve memory efficiency and minimise the number of misses. A TLB that supports fixed-size pages has disadvantages; small pages result in more entries in the TLB and can contribute to frequent TLB misses while large pages do not utilise memory effectively.

The PowerPC 440 includes instructions for managing TLB entries by running software in privileged mode. This capability gives significant control to system software over the implementation of a page replacement strategy. Storage attributes are provided to control access to memory regions. When memory translation is enabled, storage attributes are maintained on a page basis and read from the TLB when a memory access occurs. When memory translation is disabled, storage attributes are maintained in storage attribute control registers. A detailed account of the PowerPC MMU can be found in the PowerPC 440 user manual [SA102]; it devotes two chapters to its innovative MMU and cache operations. QCDOC operating system and the optimised QCD kernels for the QCDOC machine — benchmark software for the HASE QCDOC simulation models — exploit the PowerPC cache and TLB features by distributing critical computation and communication data in separate memory regions.

3.1.3.2 Floating Point Unit (FPU)

Dockser [Doc01] describes the seamless integration of a 64-bit high performance FPU to the PowerPC embedded core. The PPC440 FPU is a superscalar core with two five-stage execution pipelines. It has separate load-store and arithmetic execution pipelines and is super-pipelined giving single cycle throughput for most instructions. The FPU pipelines use out-of-order issue and completion.

The FPU receives instructions from the PowerPC CPU via the APU. It can issue two instructions in one clock cycle, one load-store and one arithmetic. It has thirty-two 64-bit floating-point registers. APU load and store instructions directly access the PowerPC core data cache, with operands of up to 128 bits (quad-word). Figure 3.4 shows the PPC440 FPU pipeline.

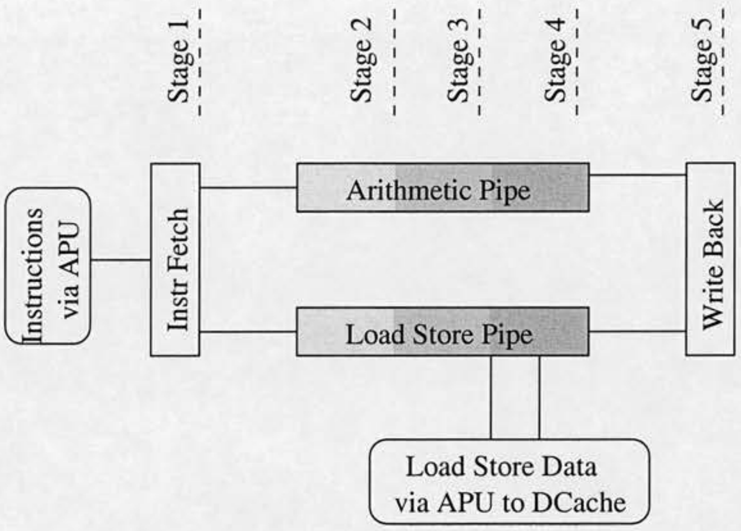


Figure 3.4: Floating-point execution pipeline

3.1.3.3 CoreConnect Bus Architecture

IBM CoreConnect is an open standard bus architecture designed to ease the integration and reuse of processor, system and peripheral cores within standard and custom SOC designs. This bus architecture is necessary for the on-chip data transfers required as part of the QCD code execution. It is composed of three buses: Processor Local Bus (PLB), On-chip Peripheral Bus (OPB) and the Device Control Register (DCR) bus. PLB is the high-performance system bus; it has separate read and write data buses and it connects the performance intensive devices on the chip. In the QCDOC ASIC, the processor has two read and one write master connections to the PLB and accesses memory-mapped devices via the PLB. For example, the processor write master transfers QCD communication instructions to the SCU DMA registers via the PLB. The PLB sequential burst protocol allows byte, half-word, word and double-word burst transfers; in *burst* mode it can transmit up to 128-bit data. Moreover, the PLB is block-transfer oriented in the sense that a single command and address can result in multiple data transfers. The PLB is designed to operate at one-third of the processor clock speed for a 500 MHz processor.

The second data-transfer bus, the on-chip peripheral bus (OPB), is used for devices that have lower data-rate demands. The OPB is also a block-transfer bus with separate control, address (36 bits) and data (32 bits) sub-buses. It is clocked at half the rate of the PLB, resulting in a bandwidth of around 266 MBps.

A third on-chip bus that connects the cores is the device control register (DCR) bus. Having the DCR bus means PLB and OPB cycles are not used to transfer control and status information. Physically, the DCR bus is a ring that loops through the various cores with the processor acting as master.

3.1.3.4 DDR SDRAM Controller Core

An important feature of the QCDOC ASIC is that a processor can interface with a relatively large off-chip memory via the system bus, PLB. In QCDOC, the off-chip memory is 256 MByte DDR SDRAM. The DDR SDRAM controller acts as an interface for large off-chip memories; a maximum of 2 GB can be attached. It provides efficient access to a standard off-chip memory. The memory side of the core can transfer either 32- or 64-bit data packets with an optional 8-bit Error Correction Code (ECC). Page-mode access and PowerPC variable size paging is supported.

The controller contains 256 bytes of read buffer, 512 bytes of write buffer, and a six-deep request queue. These facilities smooth out irregularities in the memory request patterns and facilitate the high data rates. It makes use of the PLB and data can be transferred on both clock edges (double data rate — DDR). Thus, the memory controller can stream data onto the PLB as fast as the memory can deliver it.

3.1.4 Prefetch EDRAM Controller (PEC)

Even though the PowerPC core is central to the execution of the QCD code, it is not a QCD-specific piece of hardware. The presence of an on-chip memory, the EDRAM, opens up a range of significant performance improvement possibilities as reported by Panda *et. al.* [PCD$^+$01]. For on-chip data movement, the PowerPC core features are enhanced by a custom-designed block called the Prefetch EDRAM Controller (PEC) [Tea01b]. The idea central to the PEC design is quite simple: high-bandwidth, low-latency data transfers between performance critical on-chip devices and the EDRAM. PEC is composed of three design blocks that serve three main sources of data transfers:

1. Data for the execution pipelines — The processor core interface to the EDRAM is provided via the Processor Direct Bus (PDB).

2. Communication data transfers — Communication data movements take place between the SCU and EDRAM, the processor accesses communication data from EDRAM. The second PEC design block, the PLB slave interface called PLBDBLK, assists data movement between SCU and EDRAM via the PLB.

3. File I/O — EDRAM to DDR SDRAM transfers via the DDR SDRAM controller and PEC DMA. Data moving between EDRAM and DDR SDRAM is communicated through the PEC DMA controller, which is initialised by the PowerPC processor.

The above three PEC design blocks, the PDB, the PLBDBLK and the DMA controller, are shown in figure 3.5.

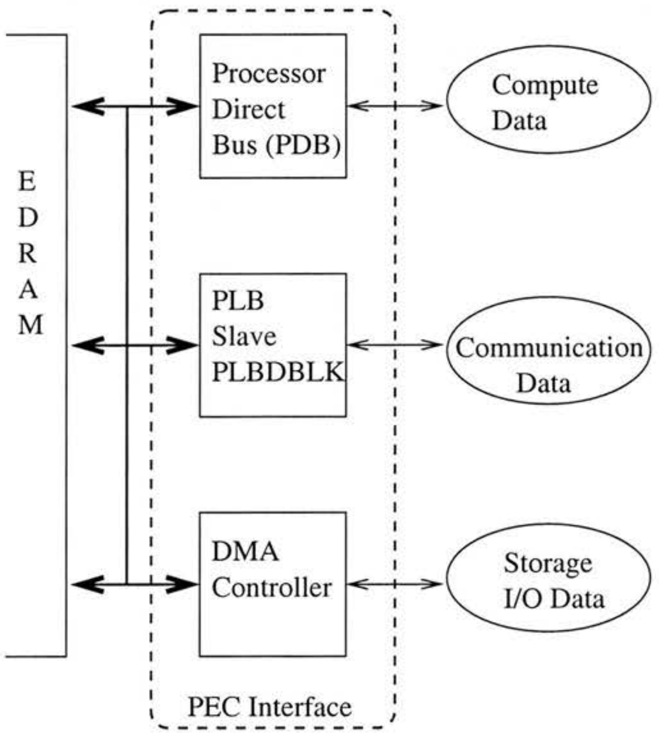


Figure 3.5: PEC design blocks

The three PEC design blocks have 512-bit read and write connections to the EDRAM. Data coherency between these design blocks is maintained via a snoop-style protocol.

Processor Direct Bus (PDB)

The PDB module interfaces to the processor read and write PLB buses. The normal processor to Processor Local Bus (PLB) clock speed ratio is 1:3, i.e. PLB operates at one-third of the processor clock speed. The custom-designed PEC bus, the PDB, however operates at 1:1 clock ratio. In other words, it is capable of handling processor read and write data at the processor clock speed. Moreover, it supports the PowerPC load and store data transfer modes. Memory-mapped requests from the processor are decoded by the PDB and handled via the PLB at one-third of the processor clock speed.

PLB Slave (PLBDBLK)

The PEC interface also has a PLB slave interface that allows for read and write operations from any PLB master on the 3:1 (one-third of the processor clock frequency) interface. In the QCDOC ASIC, the PLB masters are expected to utilise the PLB slave interface to the EDRAM, which includes the SCU. On the PLB side of the interface, full 128-bit PLB burst mode transfers are supported at 133MHz.

DMA for PLB

The PEC also has a DMA engine which is a PLB master interface that operates at 3:1. This is intended to interface to the 3:1 PLB and allows the user to transfer data utilising DMA between the external DDR memory, which resides on the PLB bus, and the EDRAM. This is the third port to the EDRAM.

3.1.4.1 Embedded DRAM (EDRAM)

Recent advances in SOC and memory technology have permitted the integration of large on-chip memories with controlled size and power requirements. The QCDOC on-chip memory, the EDRAM, provides an opportunity to enhance data movement speed for the compute-intensive part of the QCD calculations. The EDRAM is considerably faster than the external SDRAM. The EDRAM can deliver data at up to half of the processor clock speed via a 512-bit link. In contrast, external memory access can only take place via the PLB, which operates at one-third of the processor clock speed. Furthermore, in burst mode, the PLB can only support up to 128-bit data transfers.

3.1.4.2 Data Prefetching

QCD calculations, like many other scientific calculations, operate on a large number of arrays, and spatial locality of array data can be exploited by using a prefetching scheme. The PowerPC instruction set offers a data cache line prefetch instruction called *data cache touch* that brings in a cache line in the first level cache. PEC employs a similar scheme for prefetching data from EDRAM and contains a set of registers, called the prefetch read registers, to store the incoming prefetched data.

According to the PEC prefetching scheme, data is always fetched in 1024-bit lines named flines. If the data from any fline is accessed by the PDB, the rest of its fline is fetched in sequence and put into a prefetch register. Each of the PLB data read and PDB data read interfaces has two sets of prefetch read registers. These registers can prefetch four different 1024-bit regions of the memory without invalidating the already fetched data. Read requests to sequential addresses benefit most from this scheme. Data transfers between the EDRAM and the PEC modules implement logic in hardware for read and write operations and data coherency.

3.1.4.3 Prefetch Read Register Logic

There are four 1024-bit data registers, paired as two sets, associated with the two read ports: the PLB data read port and the PDB data read port. These ports are independent but arbitrate for a common EDRAM. Figure 3.6 shows the arrangement of the prefetch read registers of PEC modules.

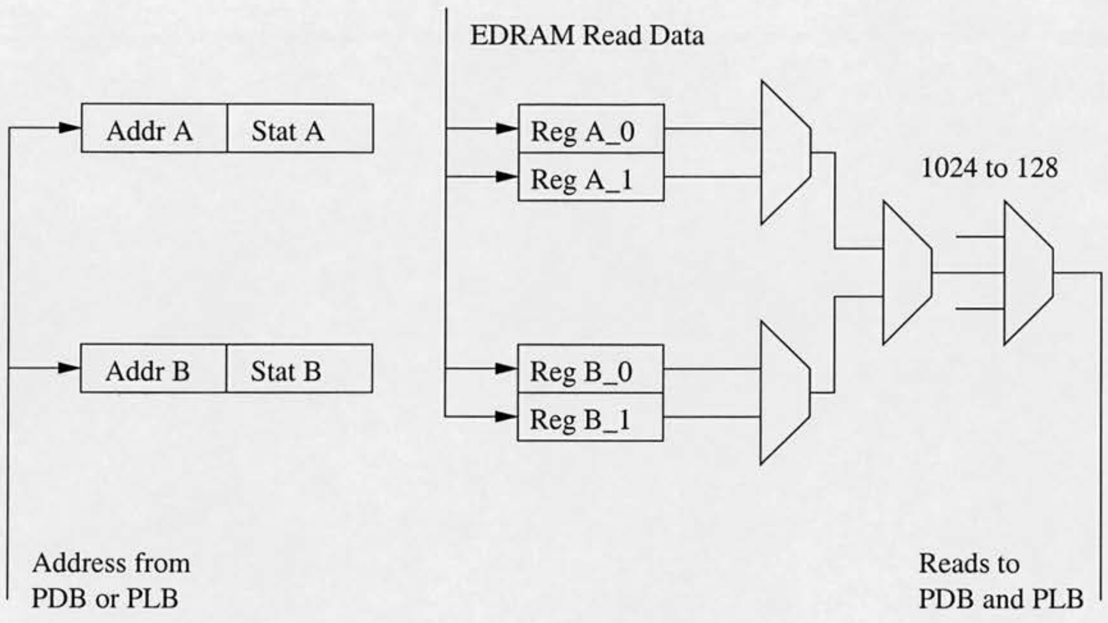


Figure 3.6: PEC prefetch read registers [Tea01b]

There is a status register (e.g. Stat A) and an address register (e.g. Addr A) associated with each prefetch data register set. Registers are loaded from the EDRAM according to logic that maintains a full 1024 bits beyond the furthest PLB or PDB request corresponding to each register set. When a PLB or PDB read is requested that does not reside in the prefetch buffer (and is not currently being prefetched), one set of registers is invalidated using a Least Recently Used (LRU) replacement protocol and a fetch from EDRAM is initiated to obtain the data. If the address space of a read from EDRAM overlaps address space with any write data that is pending in the write buffer registers, the write data is flushed to EDRAM before the read is executed.

The status bits associated with the read registers are:

1. Valid — Indicates that current data is valid.

2. Pending — Indicates that a read request to update the register has been accepted but that valid data has not returned.

3. Invalid — Indicates that the current contents are not valid and are not being prefetched.

3.1.4.4 Write Buffer Register Logic

In addition to prefetch read registers, the PEC modules maintain two 1024-bit write buffer registers. These write buffer registers are used in a circular fashion to avoid delays associated with writing to EDRAM while bus writes are in progress. Each write buffer register has a corresponding address and status register.

All write requests remain in the write buffer registers until one of the following conditions occurs:

- The register contains a full 1024 bits of valid data.
- There is a write flush initiated by an EDRAM read request to data in an overlapping address.
- There are no available registers to accept random write data.

3.1.4.5 Acknowledge Logic

Together with the read and write register logic, logic for efficient acknowledgements of data transfers is incorporated in the design. The PEC module allows for different acknowledge sources.

1. Write bus requests are single cycle acknowledgements which occur when there is a register available in the corresponding port write buffers.
2. Read bus requests are single cycle acknowledgements which occur when there is a hit with one of the two sets of registers in the corresponding port read buffer registers. This requires an address comparison with each set of address registers as well as additional condition flag checking for validity of data.

3.1.4.6 Coherency Logic

Data coherency is maintained within the PEC hardware interface in a very simple manner. All accesses to the PEC module are assumed to have a time association equal to when the access was initiated such that the responses are generated in an orderly manner. Coherency is strictly enforced on the PDB bus for processor read and write requests while coherency between the PDB and PLB is weakly enforced. Therefore, data coherency cannot be guaranteed between PLB and PDB accesses for requests issued within 1-2 processor clocks. Practically, the software is responsible for ensuring coherency when dealing with back-to-back data transfer requests.

Coherency of the system is maintained through the following conditions:

- All writes to write buffer registers invalidate read buffer data that has been prefetched. This is done across all ports.
- All writes that overlap with presently valid buffered data (any port) result in the earlier data being invalidated. Therefore, all writes result in the write address being broadcast (aligned on a 256-bit boundary).
- The loading of any read prefetch register which overlaps in address space with any write buffer register results in the write buffer register being flushed to the EDRAM. The read data is not accepted by the requesting port thereby resulting in a subsequent read, which reads in the newly written data. The flush write command has higher priority with respect to the EDRAM.

Coherency is maintained independently by the read and write sides of the port. The architecture naturally partitions in this way due to the independent PLB read and write bus structure. The read and write sides obey certain rules:

Read Side

The new write addresses (256-bit lines) broadcast from the 3 ports (PDB, PLBDBLK and DMA) are all simultaneously monitored for conflict. If a conflict is detected, the entire 1024 bit read register is invalidated. The read data accepted by a PEC module is conditional on there not being any conflict signals issued by PDB, PLBDBLK and DMA. If any of these are true at the time of the reception of the read data, the data is ignored and another read request is issued.

Write Side

The read address for all data being read from the EDRAM must be monitored. If there is an overlap with the 1024 bit write buffer, a conflict signal is generated (from PDB, PLBDBLK and DMA) depending on the conflicting port. If a conflict is generated, the write buffer that conflicts is immediately written to EDRAM with high priority. All new write addresses from the other two ports are monitored for overlap with local write buffered data. For example, the PDB monitors writes from the PLBDBLK and the DMA. If there is an address overlap, all bytes in the local data which overlap are invalidated.

To maintain coherency for read-after-write and write-after-read within a single port, all address acknowledgements are delayed until the data is either accepted (for writes), or the read data has been fetched and is in the pipeline (for reads).

3.1.4.7 Arbitration

In the presence of a number of read and write ports from the PDB, PLBDBLK and DMA, an arbitration policy is in place for EDRAM. Typically, a write flush has top priority if it is generated as a result of a read request overlap. The EDRAM arbitrates according to the following priorities:

1. Writes from PDB (flush priority) - *the highest*
2. Writes from PLBDBLK (flush priority)
3. Reads to PDB
4. Writes from PDB (non-flush)
5. Reads to PLBDBLK
6. Writes from PLBDBLK (non-flush)

3.1.5 Serial Communication Unit (SCU)

The Serial Communication Unit (SCU) is specifically designed for the QCD communication paradigm. Figure 3.7 provides an overview of the functions and organisation of the SCU.

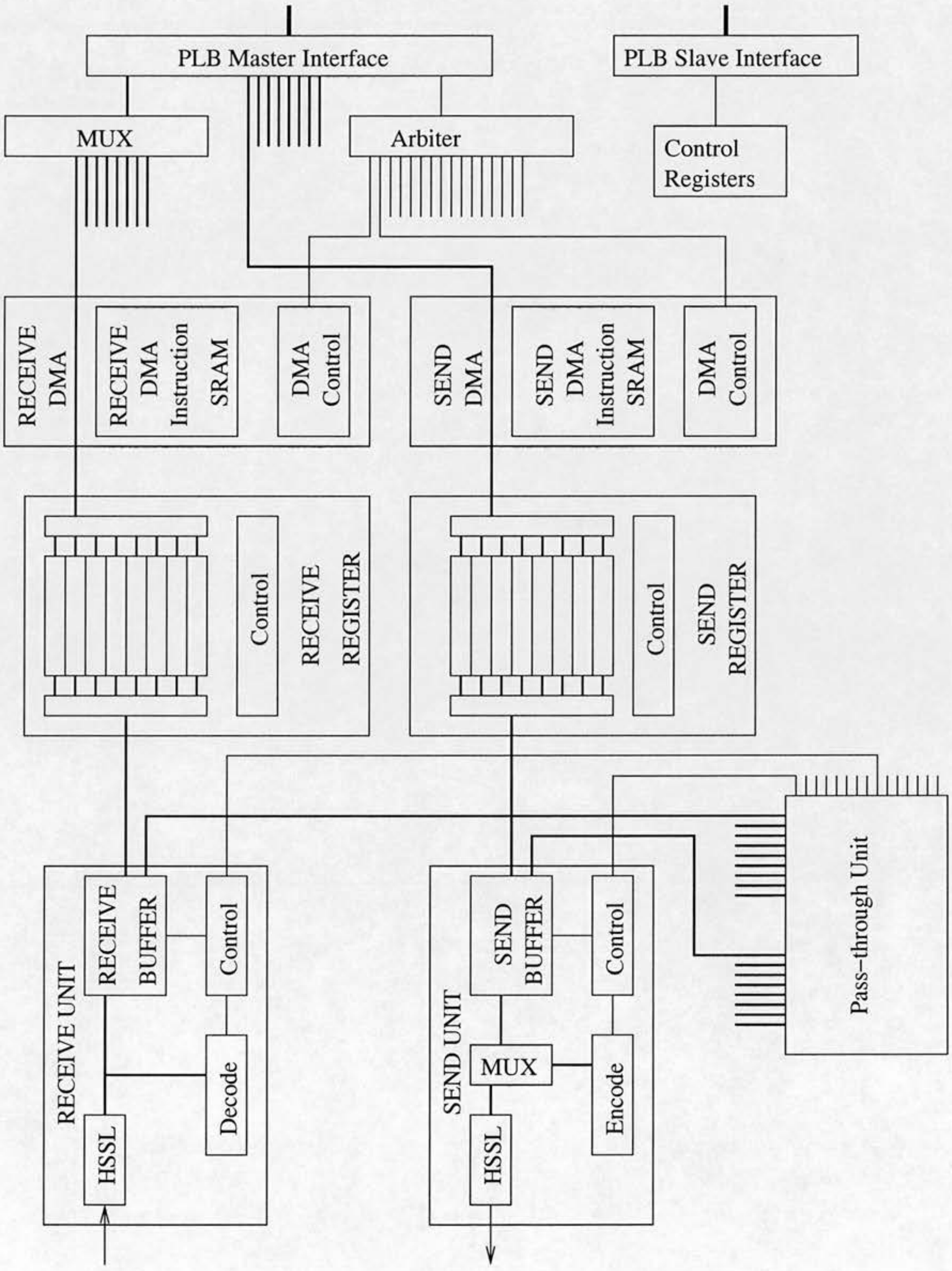


Figure 3.7: Serial Communication Unit (SCU) [BCC+02b]

Topologically, the QCDOC architecture is a six-dimensional torus, i.e., each processing node has pairs of twelve incoming and out-going links with twelve nearest neighbours. Figure 3.7 shows only one of the twelve SEND-RECEIVE units. The stubs required by the other eleven units are shown on the pass-through module, arbiter and the PLB interface. The RECEIVE unit buffers the bytes provided by the HSSL and assembles them into full 64-bit words after interpreting and stripping away various control and parity bits. It can store up to three 64-bit words so that initially the sending unit can transmit 3 words before an acknowledgement is received. Simultaneous sending to and receiving from a particular neighbour is possible because of the decoupled send and receive capabilities to each individual neighbour.

The three incoming words are stored in the RECEIVE BUFFER. Although incoming received words are destined for the memory, they are first transferred to the RECEIVE REGISTER where 32 received words can be accumulated. The presence of the RECEIVE REGISTERs allows efficient transfer to the page-oriented on-chip EDRAM or external DDR SDRAM. In the SCU, received word transfers may come from up to eight incoming receive ports; the final burst transfers are sequenced by the arbiter. A word that has been stored in a RECEIVE BUFFER is acknowledged (by an acknowledgement packet sent to the sender) after it has been copied into a RECEIVE REGISTER location.

The pattern of data writing or reading is controlled by a separate DMA engine for each of the twelve incoming and twelve outgoing directions. The instructions for each DMA engine are stored in a dedicated on-chip SRAM. They are specified by a chain of block-strided-move commands with enough space in each of these 24 SRAMs to permit chains of up to 16 such moves. The SRAM is addressed by the communication start-up instruction allowing frequently used patterns to be kept resident in SRAM and reused. This same DMA control stores the incoming data and extracts the data to be sent out, placing it in the corresponding SEND REGISTERs. Extracted data is then moved to the SEND BUFFER from which it is parsed into bytes, the appropriate control and parity bits added and the resulting bytes sent out to the HSSL. Up to three double-words can be stored in the SEND BUFFER before they are successfully acknowledged by the receiver node.

3.1.5.1 Pass-through Unit

Although the dominant communication pattern in QCD calculations is nearest-neighbour communication, infrequent but regular collective communication operations are also required. For a machine with over 10K processing nodes, a system is necessary to facilitate efficient implementation of these collective communication operations. In the QCD kernel, the regular collective communication is the floating-point global sum of 64-bit variables. A pass-through unit is introduced that accelerates these global sums and minimises processor intervention. The pass-through unit provides low-latency global operations. A sequence of words coming in on one of the twelve input wires can be stored locally and re-routed out to any combination of the output wires with minimal latency. When instructed by the processor to perform global sums, the SCU is said to be in "store-and-forward" mode. In this mode, the pass-through unit not only stores the incoming data packets but also forwards it to the send unit where it can be forwarded to the designated neighbouring node. Figure 3.7 shows 8-bit data links between the pass-through unit and the send and receive buffers.

3.2 Hierarchical Model Design

One of the key characteristics of the HASE framework is that it permits simulation designers to model systems at various levels of abstraction. Having the concept of modelling a system at different levels of detail, a classical bottom-up design approach has been exploited in creating the HASE QCDOC simulation models, similar to the physical QCDOC ASIC design approach. Central to this bottom-up design approach was the main aim of this research, which is to gain an understanding of the factors which influence the performance of QCD computers.

Thus, the components of the QCD system that contribute to the execution of the QCD application code and that can directly affect the achievable and sustained performance have been modelled and parameterised. Figure 3.8 shows the components of the HASE QCDOC node.

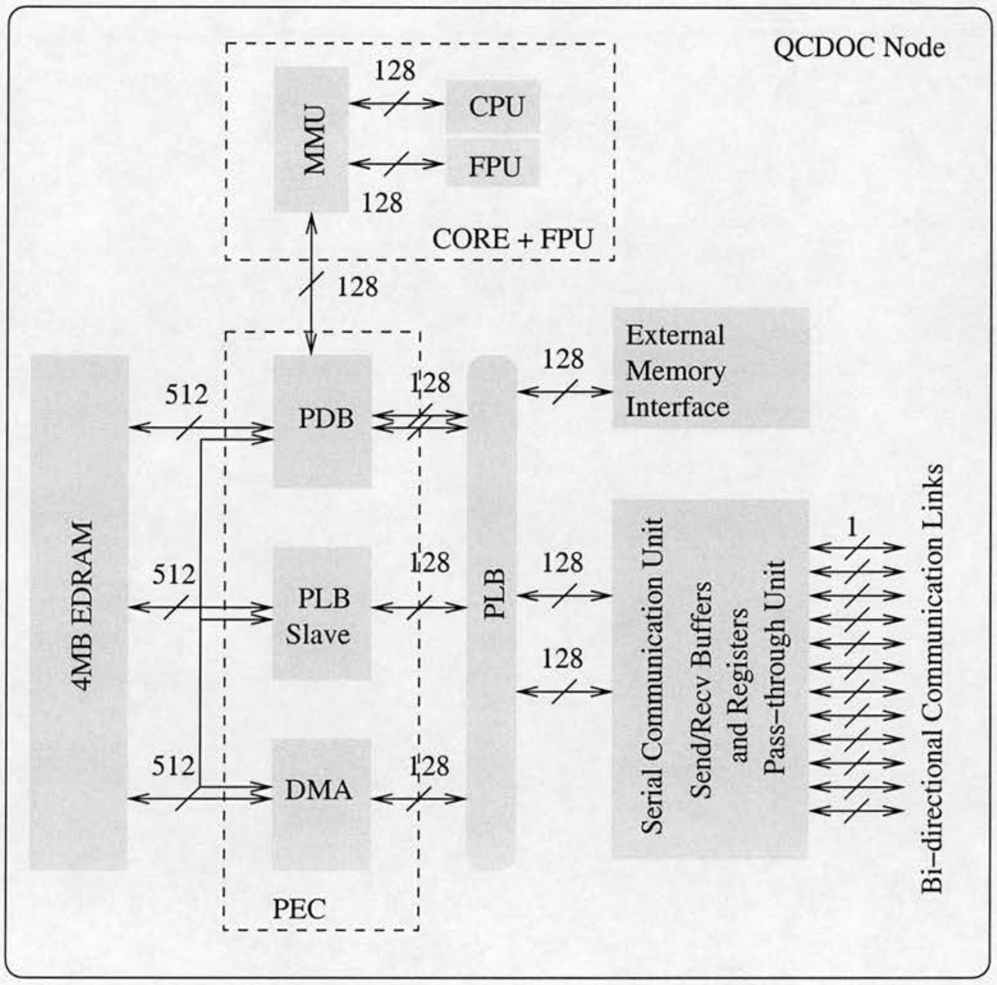


Figure 3.8: HASE QCDOC processing node

The Ethernet network and associated components do not participate in processing the computation and communication routines of parallel QCD code; therefore these are omitted from the simulation design. The Ethernet network is required only at system start-up and later for diagnostic purposes. Similarly, the file I/O and the external memory are only required after several executions of the QCD application code — to move the simulation data to disk storage. The file I/O does not influence the kernel code performance and has also been omitted. The external memory is implemented to furnish the functionality of the PEC DMA.

Having identified the HASE design components and the design strategy, the modelling of the QCDOC architecture in HASE posed several challenges. The HASE QCDOC simulation model is based on an existing system (albeit one under construction), hence the model accuracy is the primary design target. A simulation model that captures low-level details of the hardware is likely to produce high fidelity results. However, larger configurations of a precise model would take perhaps too long to run; due to the host processing power and operating system as well as the simulation environment's limitations and restrictions. Hence, in building such a large complex model it is essential to use more abstract models of some of the components. This allows meaningful results to be obtained provided that the more abstract models have been individually checked against the corresponding detailed models. A two-phase implementation strategy was therefore adopted: simulating the sub-systems first, and then building abstract models that can be verified against the detailed sub-system models.

For the two-phase implementation strategy, a three-level design scheme was adopted. The overall structure of the hierarchical simulation model design of the QCDOC system in HASE is shown in figure 3.9.

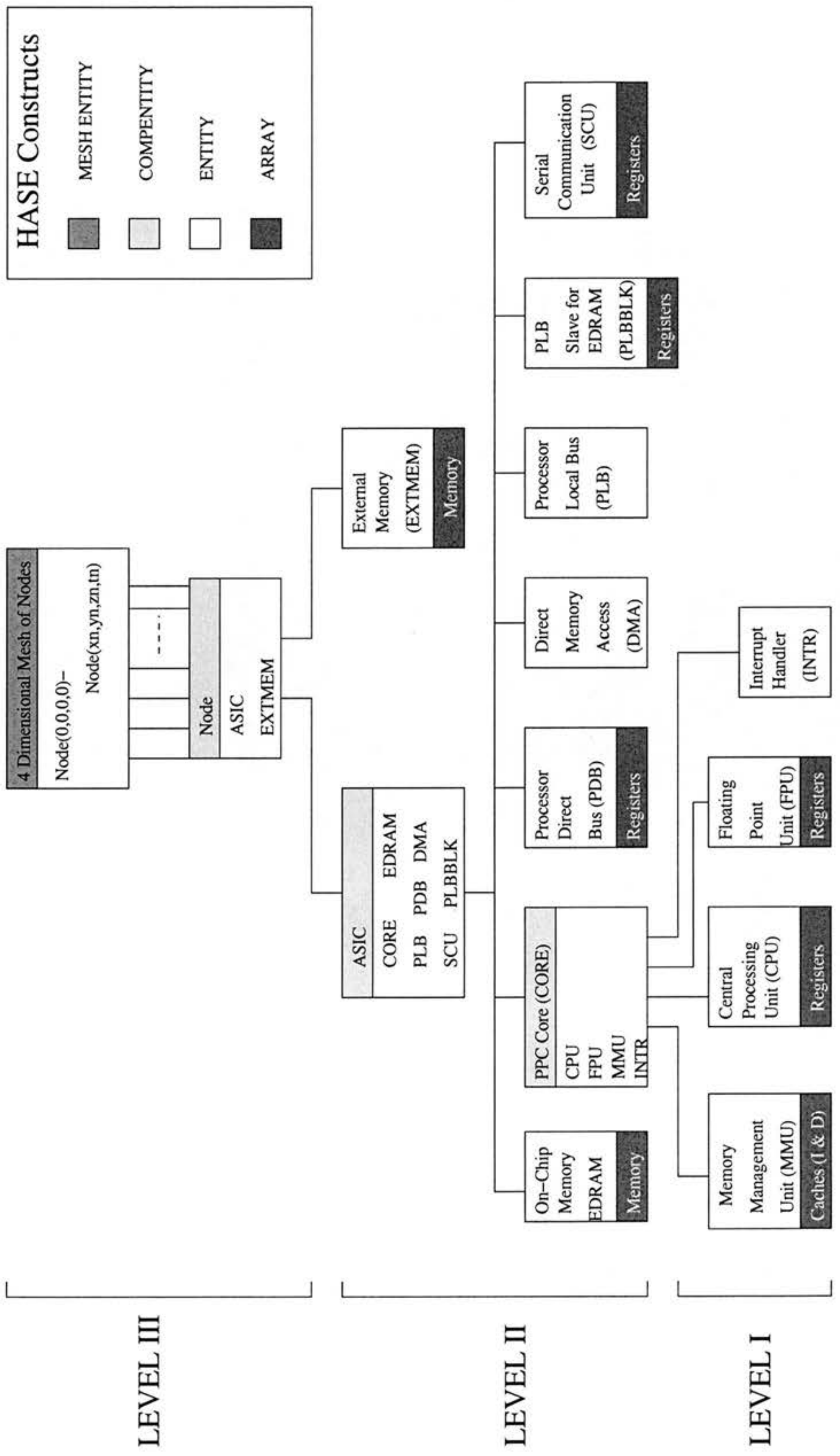


Figure 3.9: HASE QCDOC model design hierarchy

First, the processor core design entities were developed, simulated and their functional and timing vaildation was conducted. After the successful implementation and testing of this level, the next level was to simulate the on-chip data movements; these are essential to capture the QCD computation code behaviour. Level II entities are included in the design while Level I entities are aggregated and included as a single high-level entity in the model using the HASE `COMPENTITY` mechanism. Finally, larger machine configuration in which off-chip QCD communication patterns have been simulated and benchmarked is created by abstracting Level II entities' detail, where a processing node is represented as a single HASE `COMPENTITY`. At each design level, there were limitations imposed not only by the HASE platform but also by the host operating system. A number of extensions have been made to the HASE platform as a result of these requirements (explained in section 3.3). Larger system configurations, i.e., simulation models with thousands of processing nodes are constructed by designing a version of the model with reduced memory and processing capabilities of a full HASE QCDOC node.

3.2.1 Level I : PowerPC Processor Core

The PowerPC core components together with the FPU form the lowest-level entities in the model. These entities are: Central Processing Unit (CPU) `ENTITY`, MMU `ENTITY`, FPU `ENTITY` and interrupt controller (INTR) `ENTITY`. All entities at this level are built using the `ENTITY` construct in HASE. Memories (data cache, instruction cache and TLB) and registers (CPU and FPU) are declared as `ARRAY` constructs and all Level I entities operate at processor clock speed.

3.2.1.1 Central Processing Unit (CPU) `ENTITY`

The CPU `ENTITY` communicates with the MMU, the FPU and the interrupt controller `ENTITY`s. Figure 3.10 shows CPU `ENTITY`'s communication interfaces.

The CPU has 32 integer registers, a branch and control unit and three execution pipelines in which instructions can operate in out-of-order issue, execution and completion. Floating-point load-store instructions are issued in parallel to the FPU and to the CPU load-store pipe for the effective address generation. The CPU `ENTITY` is

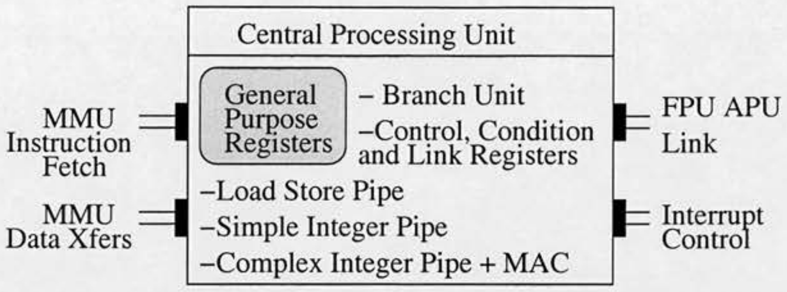


Figure 3.10: HASE CPU `ENTITY`

responsible for handling interrupts generated by all other entities in a processing node. It has two communication links with the MMU: one for instruction fetches and another for data read and write operations.

The CPU hardware and software control parameters in the HASE QCDOC model include QCD subroutines, nearest-neighbour communication operations, prefetching values of function registers, enabling and disabling of cache touch instructions and the number of instructions issued per processor clock cycle.

3.2.1.2 Floating Point Unit (FPU) `ENTITY`

FPU `ENTITY` communication interfaces are shown in figure 3.11.

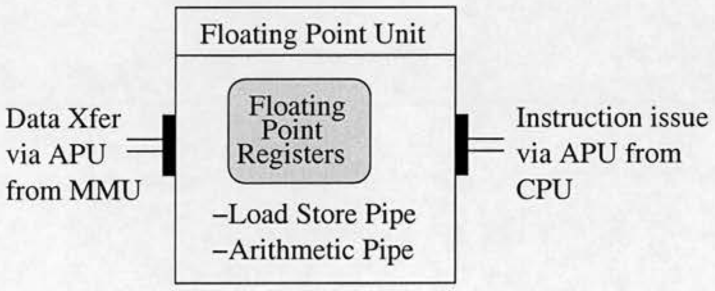


Figure 3.11: HASE FPU `ENTITY`

The FPU waits for instructions from the CPU. It has two out-of-order execution and completion pipelines. Data reads and writes to the data cache in the MMU `ENTITY` are supported by the APU interface. Two double-words can be read and written per processor clock cycle. The FPU operates on double-precision floating-point values (double-word) and has 32 double-precision registers.

Performance studies of the floating-point computation intensive QCD code require a detailed analysis of the FPU `ENTITY` behaviour. The parameters of the FPU `ENTITY` include the instruction issue rate and an ideal mode selection to establish the upper bound on the performance. The intermediate values generated during the simulation are recorded in text files for off-line detailed inspections. The number of floating-point instructions issued and completed per clock cycle, wait cycles for load instructions and trace of all FPU instructions are recorded in separate files.

3.2.1.3 Memory Management Unit (MMU) `ENTITY`

The MMU has three memory arrays: data cache, instruction cache and TLB. The data cache is set-associative. The instruction cache contains instructions in PowerPC assembly code format. TLB translations are performed in parallel with cache searches. The MMU serves load and store data to the CPU and the FPU. The processor read and write bus masters are interfaced via the MMU data cache. A cache miss issues addresses and data (for writes) on these buses which communicate with the PDB. The read and write buses can transfer 128-bit data, half the size of a cache line. Figure 3.12 shows the MMU `ENTITY` interfaces.

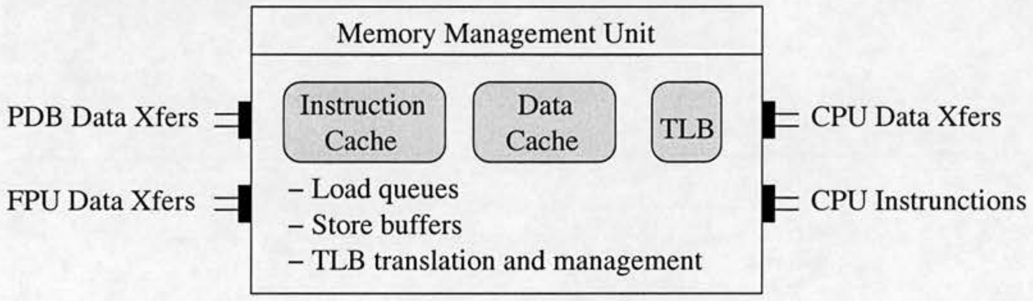


Figure 3.12: HASE MMU `ENTITY`

The PowerPC is a load-store processor; hence, performance of the execution pipelines relies on the first level cache configuration management in the MMU `ENTITY`. The MMU `ENTITY` parameters allow for experiments to be run with a combination of parameters including the data cache size, set-associativity, bandwidth to lower level memory, write policy and line allocate policy.

3.2.1.4 Interrupt Controller ENTITY

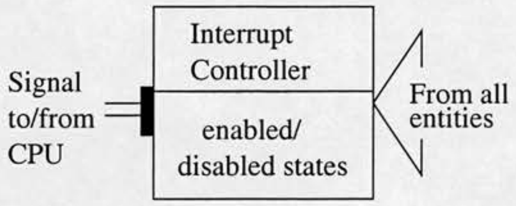


Figure 3.13: HASE Interrupt Controller ENTITY

Figure 3.13 shows the interrupt controller ENTITY's interfaces. This ENTITY has only one physical connection or HASE port link, however, all entities can raise interrupts via the Hase++ sim_schedule function, which does not require a physical link in the model.

3.2.2 Level II : ASIC and External Memory

After implementing and testing the PowerPC core and the FPU ENTITY in HASE, the next task was to focus on the ASIC components and on-chip control and data paths. The MMU ENTITY data transfer ports from Level I interface with the Level II PDB ENTITY. As part of the Level II design, the PDB, together with the PEC design blocks, the EDRAM, the system bus, the external memory controller and the communication unit are modelled in HASE. Details of the Level I entities were abstracted by combining the Level I entities into a COMPENTITY such that only the MMU-PDB interface is visible to the Level II design components.

3.2.2.1 Processor Direct Bus (PDB) ENTITY

The MMU read and write buses are linked with the PDB ENTITY. The PDB is responsible for address translation, i.e., which memory-mapped device it needs to access. Hence, the PDB can communicate with the system bus PLB, on-chip memory EDRAM and the other two PDB devices, DMA and PLBDBLK, to implement the custom data coherency protocol. It has two memory ARRAYs: prefetch read registers and write buffer registers. Figure 3.14 shows the HASE PDB entity.

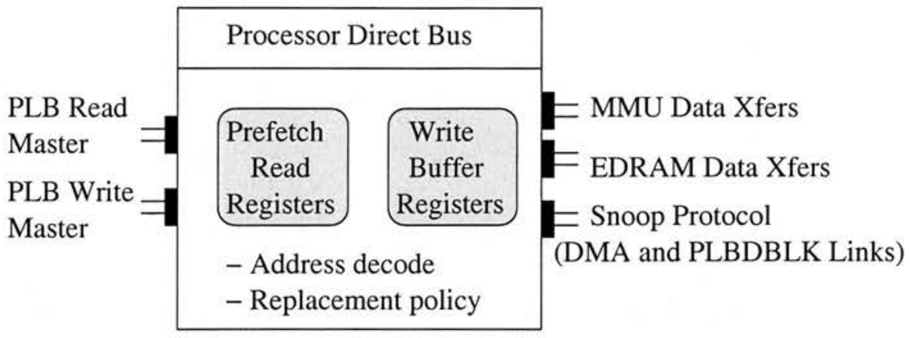


Figure 3.14: HASE PDB `ENTITY`

In order to analyse the dynamics of the PDB `ENTITY`, which is a custom-designed component in the model, alternate PDB design configurations are studied through the PDB `ENTITY` parameters. These parameters include the size of the PDB registers, number of prefetch read registers, number of write buffer registers and the read buffer register replacement policy.

3.2.2.2 Processor Local Bus (PLB) Slave (PLBDBLK) `ENTITY`

The two PEC interfaces, PDB and DMA, are bus masters. The PLB slave provides a slave interface to the EDRAM for other PLB master devices. It is therefore named as PLB Data Block (PLBDLBK) `ENTITY`. It has one PLB slave interface, one EDRAM data interface and one PEC protocol interface. The PLBDBLK `ENTITY` has an identical set of parameters to the PDB `ENTITY`.

3.2.2.3 Direct Memory Access (DMA) `ENTITY`

The PEC DMA is similar to the PLBDBLK except that it has a PLB master interface to the PLB. Once initialised, it can transfer a page between EDRAM and the external memory without CPU intervention.

3.2.2.4 Embedded DRAM (EDRAM) `ENTITY`

The on-chip memory (EDRAM) can be accessed by the three PEC devices: the PDB, the PLBDBLK and the DMA. It has 512-bit wide data connections for data read and write to the PEC components. It has a 4 MByte memory array. The EDRAM interfaces are shown in figure 3.15. The EDRAM `ENTITY` has one parameter; data bandwidth to the PEC devices.

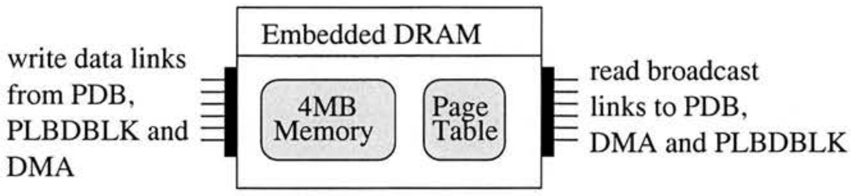

Embedded DRAM
write data links from PDB, PLBDBLK and DMA
4MB Memory
Page Table
read broadcast links to PDB, DMA and PLBDBLK

Figure 3.15: HASE EDRAM `ENTITY`

3.2.2.5 Processor Local Bus (PLB) `ENTITY`

The PLB `ENTITY` supports multiple bus masters and has separate address, read and write buses. It operates in split transfer mode, i.e., the PLB split bus transaction capability allows the address and data buses to have different masters at the same time. Additionally, a second master may request ownership of the PLB, via address pipelining, in parallel with the data cycle of another master's bus transfer. Overlapped read and write data transfers and split-bus transactions allow the PLB to operate at a very high bandwidth by fully utilising the read and write data buses. The PLB master and slave interfaces in the HASE processing node are shown in figure 3.16.

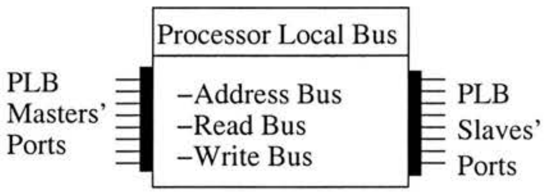


Figure 3.16: HASE PLB (System Bus) `ENTITY`

3.2.2.6 DDR SDRAM `ENTITY`

The DDR SDRAM (external memory) `ENTITY` is a bus slave to the PLB `ENTITY`. It has only one parameter: the external memory size.

3.2.2.7 Serial Communication Unit (SCU) ENTITY

Figure 3.17 shows the SCU ENTITY, which has bi-directional off-node links to communicate with the neighbouring nodes and has two on-chip PLB interfaces: one PLB master and another PLB slave.

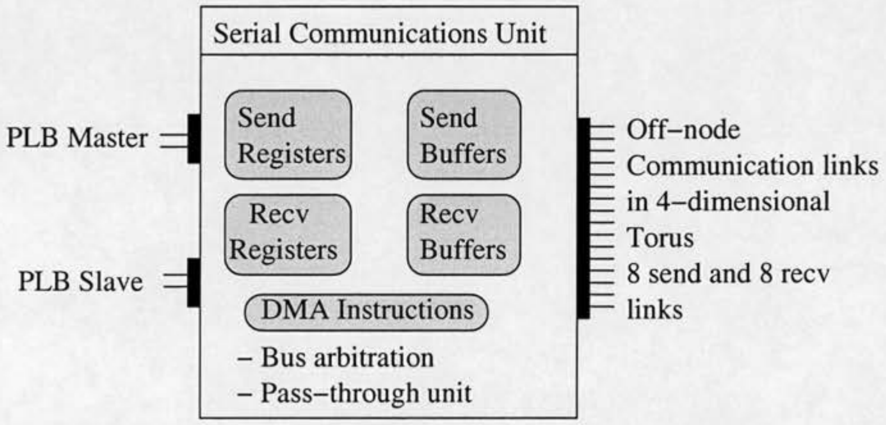


Figure 3.17: HASE SCU ENTITY

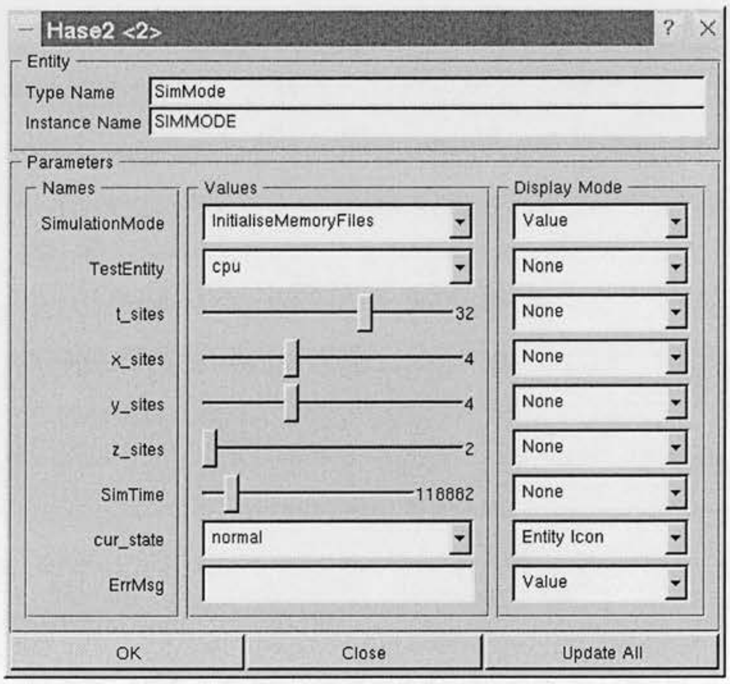
The QCD communication performance studies have been conducted through the HASE SCU ENTITY parameters. The effect of the communication buffer and register sizes on nearest-neighbour performance are studied through the send and receive buffer register and buffer size and number parameters. Moreover, the alternate communication channel properties can be simulated by using the send, receive and data acknowledge latencies, which are parameters of the SCU ENTITY.

3.2.3 Level III : A Four-Dimensional QCDOC Machine

In a similar manner to the abstraction of Level I entities to Level II in the form of a COMPENTITY, Level II entities are abstracted to a higher level as a processing node, the Node COMPENTITY. At this abstraction level, the HASE QCDOC processing nodes communicate with each other through the SCU off-node interfaces; their on-node program executions are not visible to each other. A simulation user, however, can zoom into and out of a processing node's details as required. The HASE MESH4D construct has been used to generate a four-dimensional QCDOC machine based on the Node COMPENTITY.

3.2.3.1 SimMode ENTITY

A HASE QCDOC machine Node COMPENTITY, as explained earlier, is composed of a number of HASE ENTITYs. When replicated using the HASE MESH4D template facility, multiple Node COMPENTITYs, and subsequently low-level ENTITYs, are generated by HASE. In a physical, tightly-integrated multiprocessor machine, managing individual entities within a system of processing nodes requires a front-end host server. Likewise, a HASE ENTITY is developed, which behaves as a front-end server for the HASE QCDOC model entities, similar to the front-end host machines in MPP systems. This ENTITY is called the SimMode ENTITY. Along with other features (explained in the next chapter) it controls the mode in which a simulation is running. Figure 3.18 shows the SimMode ENTITY parameter window in which the SimulationMode is currently set to InitialiseMemoryFiles.



Type Name	SimMode
Instance Name	SIMMODE

Names	Values	Display Mode
SimulationMode	InitialiseMemoryFiles	Value
TestEntity	cpu	None
t_sites	32	None
x_sites	4	None
y_sites	4	None
z_sites	2	None
SimTime	118862	None
cur_state	normal	Entity Icon
ErrMsg		Value

Figure 3.18: Simulation control — SimMode ENTITY

The currently active mode instructs all entities in the HASE QCDOC model that they should not execute their normal behaviour, instead they are in a special mode in which only the memory files initialisation processes should take place.

A detailed description of the HASE QCDOC design entites is presented in Appendix B. It contains the input parameter list for each `ENTITY` as well as the values that are recorded during and reported after a simulation run. Furthermore, this Appendix presents the contents of text files that contain cycle-by-cycle simulation information.

3.3 HASE Extensions

During the construction phases of the HASE QCDOC model, a number of limitations of the HASE platform were identified and subsequently addressed by extending the existing capabilities and by introducing new ones within the HASE platform. First, at the QCDOC ASIC modelling stage, the HASE multi-frequency clock mechanism was implemented. Second, at the four-dimensional torus simulation stage, new memory array representations and multi-dimensional meshes were incorporated in the HASE platform. Third, static control provision was included to facilitate the debugging and testing of the multi-node QCDOC model.

An insight into the following notable modifications is presented in the next sections, which were made as a direct result of the HASE QCDOC modelling requirements:

- Multi-frequency clock mechanism;
- Memory `ARRAY` options;
- `MESH`n`D` template facility; and
- Static parameters.

3.3.1 Multi-frequency Clock Library

An ASIC's components operate on a range of clock frequencies. Processing components tend to run faster while memory module and communication units are slower. Likewise, QCDOC ASIC blocks have four different frequencies, relative to the processor clock frequency. For instance, the PowerPC core is the fastest while the PLB operates at one-third of the processor clock speed.

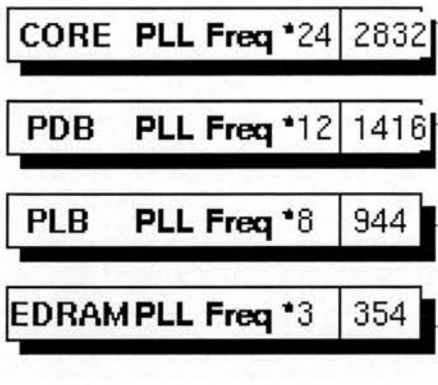
It was recognised that the ratio of clock frequencies should be a parameter for the performance experiments, i.e., it can be altered before a simulation run. Previous HASE simulations employed a barrier synchronisation mechanism using a clock `ENTITY` which had to be re-defined in the `project.edl` and its behaviour re-coded in the `clock.hase` file. Moreover, all clocked HASE entities were linked to the clock by hand-coding the `clock.hase` file. The `clock.hase` file had to be updated whenever a new instance of a clocked `ENTITY` was added to the project.

The old clock mechanism was not practical for the QCDOC simulation for the following reasons:

- The clock `ENTITY` description `clock.hase` needed to be modified as the size and dimension of the QCDOC model was changed.
- For a range of frequencies, multiple frequency clocks, or entities were required.
- Experiments were error-prone using a variable clock frequency ratio as it would have required several steps of manual intervention.

Mallet *et. al.* [MIA02] introduced a flexible and extensible clock mechanism in the HASE. A clock library was created, which results in code re-use, and a clocked `ENTITY` was implemented using the object-oriented inheritance mechanism. This inheritance mechanism is called *registration*, where an entity registers with a Clock or a Pll `ENTITY`. Bi-phased clocks are also supported. Another library entity called Pll multiplies the frequency of a master clock by an integer to give a faster clock frequency. The HASE entities that need a faster clock are registered to a Pll. In a project there can be several instances of a Pll `ENTITY`, where each instance has a separate ratio. A Pll's ratio can be altered, using the standard HASE parameter box, for simulation runs (e.g., figure 3.19). Asim [EAB$^+$01] is the only other simulation framework that allows insertion of a clock method in microarchitecture designs. The HASE and Asim clock mechanisms share a notion of logical activity of an entity, like sending and receiving information from other entities through ports, within a clock cycle. A further extension in HASE is the bi-phased clock mechanism where logical activities can be associated with the rising or falling clock edges.

3.3.1.1 Clock Frequency Control



CORE	PLL Freq *24	2832
PDB	PLL Freq *12	1416
PLB	PLL Freq *8	944
EDRAM	PLL Freq *3	354

Figure 3.19: PLL control

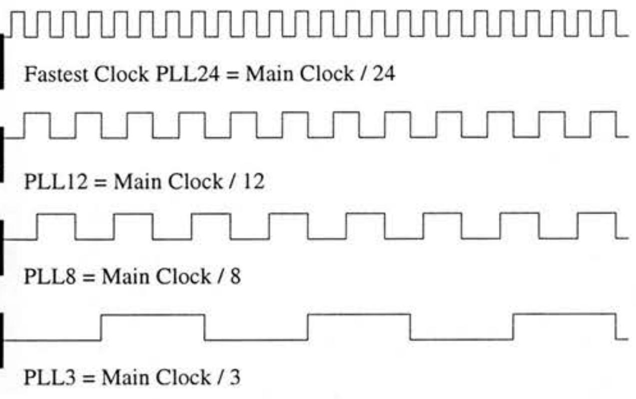


Figure 3.20: Relative clock speeds

The HASE multi-clock mechanism was employed to provide a range of clock frequencies for the QCDOC processing node. A PLL's ratio determines the speed of entities driven by it; a large ratio results in a high clock frequency as illustrated in figure 3.20. Clock frequency control (the ratio) is a PLL parameter for a slower or faster clock speed. Alternatively, an entity can change its PLL instance. In both cases however, the process of altering individual clock frequencies can be error prone and tedious, therefore, a control window has been provided (figure 3.19) for Pll frequency alterations.

3.3.2 ARRAY Construct in HASE

The ARRAY construct provides a mechanism for defining lists of objects of the same type, similar to the array construct in C and C++. In HASE, the ARRAY construct is mainly used to represent the contents of memory systems, like caches and main memory.[5] The contents of a memory file are typically resident in a physical file in the project directory with an extension .mem.

3.3.2.1 Memory Requirements for the QCDOC Machine

In order to explain the limitations of the HASE memory ARRAY construct, a brief account of the HASE QCDOC memory array requirements is first presented. QCDOC is a distributed-memory system and a processing node in the model has the following memory modules:

- Instruction Cache and Data Cache
- On-chip Memory
- External Memory
- Set of prefetch registers
- Set of communication registers

A four-dimensional 2x4x4x2 QCDOC machine needs a multiple of 2x4x4x2 physical memory .mem files in the project directory. A slight increase in the number of nodes would result in a large increase in the number of .mem files. The host operating system, Linux, imposes a restriction on the number of files a directory can contain.[6] To allow for large number of nodes, all having their own .mem file contents, the HASE memory array mechanism was modified and array initialisation scheme options were introduced.

[5]Refer <http://www.icsa.inf.ed.ac.uk/research/groups/hase/manuals/edl/arrays.html> for details.
[6]The kernel code (namely /usr/include/linux/ext2_fs.h) defines EXT2_LINK_MAX as 32000.

3.3.2.2 Memory Array Initialisation Process

The initialisation process depends on the context of the array (entity type name, entity instance name, parent type and parent instance name) and the name of the `RARRAY` instance. Typically, each array instance has a `.mem` file associated with it. The large number of memory modules required by the QCDOC model were handled by implementing the following memory initialisation schemes:

1. A shared `.mem` file.
2. A sectioned `.mem` file (contains `//$next_section` tags to identify separation).
3. Separate `.mem` files.

Selecting any of the above options has no effect on the display, read or write mechanism of a memory file during a simulation. In other words, users of the simulation will not experience any effect from the choice of the mechanism with which a `.mem` file is defined.

Table 3.2 lists the alternative ways of defining HASE memory `.mem` files.

Mechanism	Number of `.mem` Files	Initialisation
Shared	One	Single `.mem` file initialisation.
Sectioned	One	Single `.mem` file contains Sections separated by `$next_section` tags. A Section maps to an `ARRAY` instance.
Separate	Equal to the number of `ARRAY` instances	`.mem` files in project directory. Have to initialise as many files as there are `ARRAY` instances

Table 3.2: HASE memory array (`.mem` files) options

The HASE memory ARRAY options are discussed below for the HASE QCDOC model:

- **Shared Memory**

  QCDOC is a Single Program Multiple Data (SPMD) architecture. Therefore, a shared .mem file is a straightforward choice for the instruction memory (instruction cache). Each entity loads its own copy of the simulation file from a single, physical .mem file which contains the QCD simulation benchmark code.

- **Sectioned Memories**

  The data memories in the QCDOC model are physically distributed memories. Unlike the instruction memories, the data memories contain non-identical data. Thus, the above-defined shared memory approach was not appropriate for the data memories. Furthermore, due to a restriction on the number of files a directory can have (mentioned in section 3.3.2.1) it was decided to contain separate memory modules in relatively fewer files. A new scheme was introduced called sectioned memories. In sectioned memory files, which are relatively large as compared to shared and separate memory files, each section (separated by `$//next_section` tag) corresponds to an instance of a memory array. This scheme poses a challenge to the simulation designer, as it is the responsibility of the simulation designer to ensure that the sections are correctly mapped to the memory instances (according to order in which they are generated by the HASE system).

- **Separate Memories**

  This is the conventional way to define memories in HASE, i.e., one .mem file per instance of a memory module. For the QCDOC machine, it is not a practical choice because this method requires a large number of .mem files in the project directory. Existing HASE models do not require a large number of memory files. Hence, their memory files are separate and are kept in the project directory.

3.3.3 MESHnD Template

HASE templates provide an ideal mechanism for the rapid prototyping and exploration of scalable architectures. These templates allow the automatic creation of models by inserting a user-defined HASE entity into a predefined topology and the provision of dimension parameters. For instance, mesh networks are architectures consisting of the aggregation of several basic components communicating through a specific scheme; the mesh topology. Earlier versions of the HASE mesh templates supported one-, two- and three-dimensional meshes and tori. This HASE mesh generation mechanism was extended to an n-dimensional scheme to allow for the four-dimensional QCDOC torus interconnection network topology. In the HASE QCDOC model, the entity that is replicated using the MESH4D template is the Node COMPENTITY.

Figure 3.21 shows a 4-dimensional mesh description in the ENTITYLIB section of a project Entity Description Language (EDL) file.

```
MESH4D QCDMachine (          -- 4-dimensional Mesh QCDMachine
  ENTITY_TYPE (Node)         -- Node is a COMPENTITY defined in ENTITYLIB
  SIZE1 (2)                  -- 2 Nodes in first dimension
  SIZE2 (4)                  -- 4 Nodes in second dimension
  SIZE3 (4)                  -- 4 Nodes in third dimension
  SIZE4 (2)                  -- 2 Nodes in fourth dimension
  NO_LINKS(2)                -- Bi-directional Links (send/recv)
  WRAP(1)                    -- Wrap-around links at edges
  DESCRIPTION("4D QCD Machine") -- Textual description
  PARAMS()                      -- Additional Parameters if required
)
```

Figure 3.21: A 4-dimensional mesh template (MESH4D) definition in EDL

The ENTITY_TYPE attribute identifies the type of the entity to be replicated in the mesh. The SIZEn attributes specify the size of the mesh for each dimension. The NO_LINKS attribute specifies communications either as unidirectional or bidirectional, and the WRAP attribute specifies whether the mesh has to be wrapped at the border.

Figure 3.22 show the contents of the HASE QCDOC project EDL file where off-node communication links are generated for Node _0_0_0_0_ using the `MESH4D` template. The SCU `ENTITY` contains all off-chip communication ports.

```
sim.link_ports("QCD._0_0_0_0_.SCU","to_first",
               "QCD._1_0_0_0_.SCU","from_first");
sim.link_ports("QCD._0_0_0_0_.SCU","back_to_first",
               "QCD._1_0_0_0_.SCU","back_from_first");
sim.link_ports("QCD._0_0_0_0_.SCU","to_second",
               "QCD._0_1_0_0_.SCU","from_second");
sim.link_ports("QCD._0_0_0_0_.SCU","back_to_second",
               "QCD._0_3_0_0_.SCU","back_from_second");
sim.link_ports("QCD._0_0_0_0_.SCU","to_third",
               "QCD._0_0_1_0_.SCU","from_third");
sim.link_ports("QCD._0_0_0_0_.SCU","back_to_third",
               "QCD._0_0_3_0_.SCU","back_from_third");
sim.link_ports("QCD._0_0_0_0_.SCU","to_fourth",
               "QCD._0_0_0_1_.SCU","from_fourth");
sim.link_ports("QCD._0_0_0_0_.SCU","back_to_fourth",
               "QCD._0_0_0_1_.SCU","back_from_fourth");
```

Figure 3.22: 4-Dimensional mesh links generated by the HASE

From the MESH4D description (figure 3.21), the HASE instantiates enough components (2x4x4x2 = 64) and creates communication links among these components in all four dimensions. When the HASE project containing the above mesh description is built, the HASE automatically generates 64 entity instances such that the first Node is recognised with _0_0_0_0_ prefix and the 64th Node has a _1_3_3_1_ prefix. Node _0_0_0_0_ is connected to Node _1_0_0_0_ in the first dimension, to Node _0_1_0_0_ in the second dimension, to Node _0_0_1_0_ in the third and to Node _0_0_0_1_ in the fourth. Wrap around connections are generated likewise.

3.3.4 Static Parameters

In addition to the hardware prototyping, the HASE allows parameterised entities to be created, i.e., a simulation designer can introduce hardware parameters within an entity that can be altered for simulation runs. For instance, the data cache line size can be a parameter and a user can change this parameter for successive experiments. The HASE parameterisation mechanism provides a flexible and convenient mechanism to run experiments to explore the hardware design space of a system.

Considering the previous `MESH4D` example, a HASE QCDOC model with 64 processing nodes, it is not practical for a simulation user to alter a parameter in 64 individual instances of an entity through 64 separate parameter windows. For simulation experiments, it is often necessary to alter parameters within entities several times. It was considered that another property would be necessary for a HASE parameter such that one value of that parameter is visible to all instances of an entity. The C++ `static` identifier serves this purpose, as a result, static parameters were introduced in the HASE.

After introducing a static parameter, a second problem arose because parameters are visible in a HASE parameter window. The on-screen position of a parameter in the HASE design window is controlled via project's Entity Layout File (ELF) description. According to the ELF description, a parameter position is relative to the position of an entity. `static` parameters have one single value, hence, it was desired that there should be one on-screen appearance of these variables. A modification in the HASE allowed the ELF description of a parameter declared as `static` in the HASE EDL file to be placed at an absolute position on the HASE design window. The modifications in the EDL and the ELF description for the static parameters were particularly helpful for the debugging, testing and experimenting of the template-based HASE model.

3.3.4.1 Central Control in the HASE QCDOC Model

In a multi-level abstraction and template-based HASE QCDOC model, the flexible parameter alteration facility via a graphical window imposed some overheads. For the HASE QCDOC simulation experiments, altering a parameter in all instances of an entity through parameter window GUI is likely to be error prone as the simulation user repeatedly opens, closes and edits parameter boxes in multiple entities for a single simulation run. Moreover, it is not useful in the QCDOC simulation model since all processing elements have identical configurations. In order to simplify the MESH-based entities parameter alterations, a central control interface to the parameterised entities is provided as shown in figure 3.23. By double-clicking one of the ENTITY_PARAMS, a corresponding static parameter window would pop-up. For example, by double-clicking PDB_PARAMS in figure 3.23, the PDB static parameter window, figure 3.24 appears. A parameter selection via figure 3.24 is applicable to all instances of the PDB ENTITY.

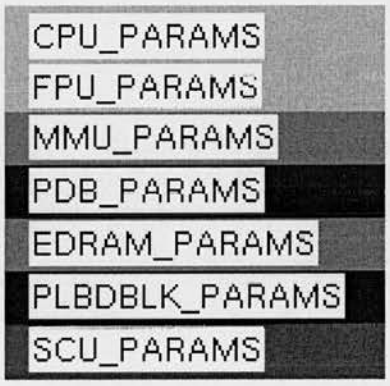
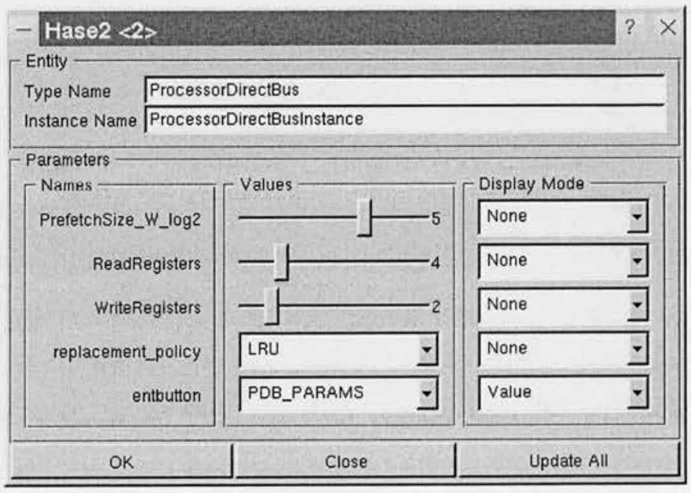

CPU_PARAMS
FPU_PARAMS
MMU_PARAMS
PDB_PARAMS
EDRAM_PARAMS
PLBDBLK_PARAMS
SCU_PARAMS

Figure 3.23: PARAMs Interface



Entity

Type Name: ProcessorDirectBus

Instance Name: ProcessorDirectBusInstance

Parameters

Names	Values	Display Mode
PrefetchSize_W_log2	5	None
ReadRegisters	4	None
WriteRegisters	2	None
replacement_policy	LRU	None
entbutton	PDB_PARAMS	Value

Figure 3.24: PDB static parameter window

3.4 Model Debugging, Validation and Experimentation

A model design will not be significant and meaningful if the behaviour of the model does not correspond to what was originally required, or it does not work correctly. In multiprocessor simulations, the debugging requirements are compounded by the fact that a considerably large number of entities have to be inspected not only for their own behaviour but also for their concurrent interactions. The HASE animations are normally used as a mechanism to debug and to test the design and implementation of a small-scale, simulation model. Large-scale multiprocessor simulation models, in contrast, are generally validated by inspecting the end results of a benchmark code in the memory arrays [Coe00].

In the QCDOC simulation model, the validation requirements were slightly different. Firstly, complete pipeline stages were modelled within an entity, therefore HASE animation which has been used in DLX pipeline modelling could not be employed for validating the PowerPC pipelines execution. Secondly, a tool was needed so that the results gathered by individual instances of an entity within the four-dimensional mesh could be viewed after the simulation. Finally, for a comprehensive understanding of the behaviour of application code on the QCDOC system, overall code execution information was considered together with the simulation end results. In view of these requirements, the following mechanisms and tools are explained:

1. Animation of simulation trace file;
2. Timing diagrams;
3. EDL tracing tool; and
4. Interactive plots.

Except for (4), the others use a trace file generated by the HASE system as a result of a simulation run. This trace file contains updated parameter values of an all instances of an entity, send events and memory read and write events with respect to the simulation execution time. HASE template-based simulation models, particularly large scale models with a number of entities like QCDOC, tend to have enormous amount of information within a single project trace file (order of MBytes).

3.4.1 HASE Animator

HASE animations are useful for

- showing packet movements between entities along with packet content values or icons;
- highlighting current memory read and write contents in a memory array;

- changing an entity's icon colour, showing its current state; and
- displaying useful parameter values as they are updated by the simulation.

In the initial design verification, for instance, interactions between two entities to identify an unusual or race condition, HASE animations were quite useful. Moreover, in the communication protocol implementation, animations can show the order in which data and acknowledge packets are exchanged. Finally, memory read and write operations are validated as the simulation proceeds by opening a memory array window. Nonetheless, the HASE animation mechanism has its limitations. For example, a memory file contents can only be viewed if it contains a few tens of lines of entries. Extremely large memory files cannot be viewed during animation, and are almost impossible to read through by a simulation user. The entity icon changes, on the other hand, make a simulation run slower as it involves writing additional information to a trace file. In multiprocessor simulations, it is not preferred as the trace files can be enormous in size. However, implementation, debugging and validation of the PDB coherency protocol and the nearest-neighbour off-chip transfer protocol exploited the HASE animation facilities.

3.4.2 Timing Diagrams

The first QCDOC CPU prototype was based on the HASE DLX [Ibb00] simulation model. In this model, execution pipelines and pipeline stages are represented as separate HASE entities. Thus the first QCDOC CPU prototype contained about 19 entities; three (seven-stage) execution pipelines where each stage was represented as a HASE `ENTITY`. When considering the implementation of the FPU and the additional components of the ASIC, it was calculated that about 25-30 entities would be required for one processing node. This would have incurred a big overhead in execution time and memory on the host if these entities were replicated using the HASE mesh templates.

At the same time, it was decided that the cycle-by-cycle dynamics of instruction level parallelism (ILP) of the PowerPC processor and the FPU should be not be abstracted by a single trace value or by employing execution-driven or distribution-driven simulation techniques. QCD is a floating-point computation intensive code; by employing techniques like execution-driven simulation and distribution-driven simulation, accuracy of the performance result is affected (the affect is quantified in the next chapter).

The solution to the two conflicting requirements, i.e., reducing number of entities while not sacrificing essential details, were met by making each pipeline stage a parameter within a single CPU `ENTITY`. Validating the three CPU pipelines could not be performed via the HASE animation so timing diagrams (as in figure 3.25) were used to verify the out-of-order issue, execution and completion pipelines.

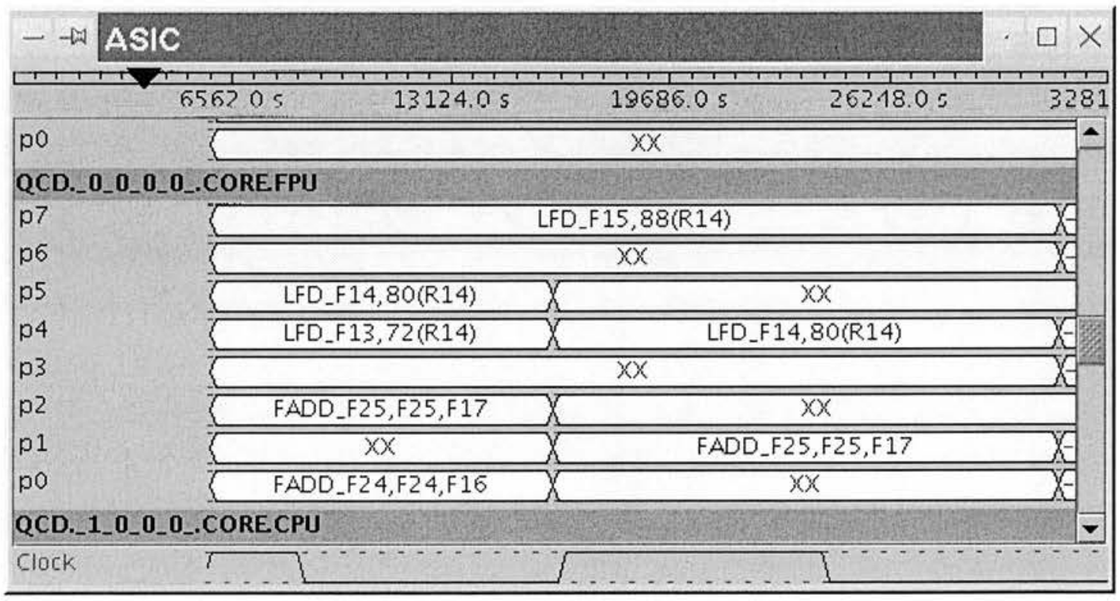


Figure 3.25: FPU pipeline stages in a timing diagram

The parameters p0 to p7 represent the parameters defined for a HASE `ENTITY` in an EDL file. Figure 3.26 represents the code listing for the FPU `ENTITY` in the EDL file and its timing diagram output is shown in figure 3.25. This timing diagram shows the movement of PowerPC floating-point instructions from one pipeline stage to the next for two clock cycles. `XX` indicates that the pipeline stage is currently not being used.

```
ENTITY FloatingPointUnit(
   EXTENDS(Biclocked)
   DESCRIPTION ("FPU")
   PARAMS (
     RINSTR(t_CoreISA,fexe_WBACK)        -- p0 (exe pipe write back)
     RINSTR(t_CoreISA,fexe_2)            -- p1 (exe pipe 2)
     RINSTR(t_CoreISA,fexe_1)            -- p2 (exe pipe 1)
     RINSTR(t_CoreISA,fexe_RACC)         -- p3 (exe pipe register access)
     RINSTR(t_CoreISA,ls_WBACK)          -- p4 (load/store write back)
     RINSTR(t_CoreISA,ls_CRD)            -- p5 (load/store cache access)
     RINSTR(t_CoreISA,ls_AGEN)           -- p6 (load/store normalise)
     RINSTR(t_CoreISA,ls_RACC)           -- p7 (load/store register access)
```

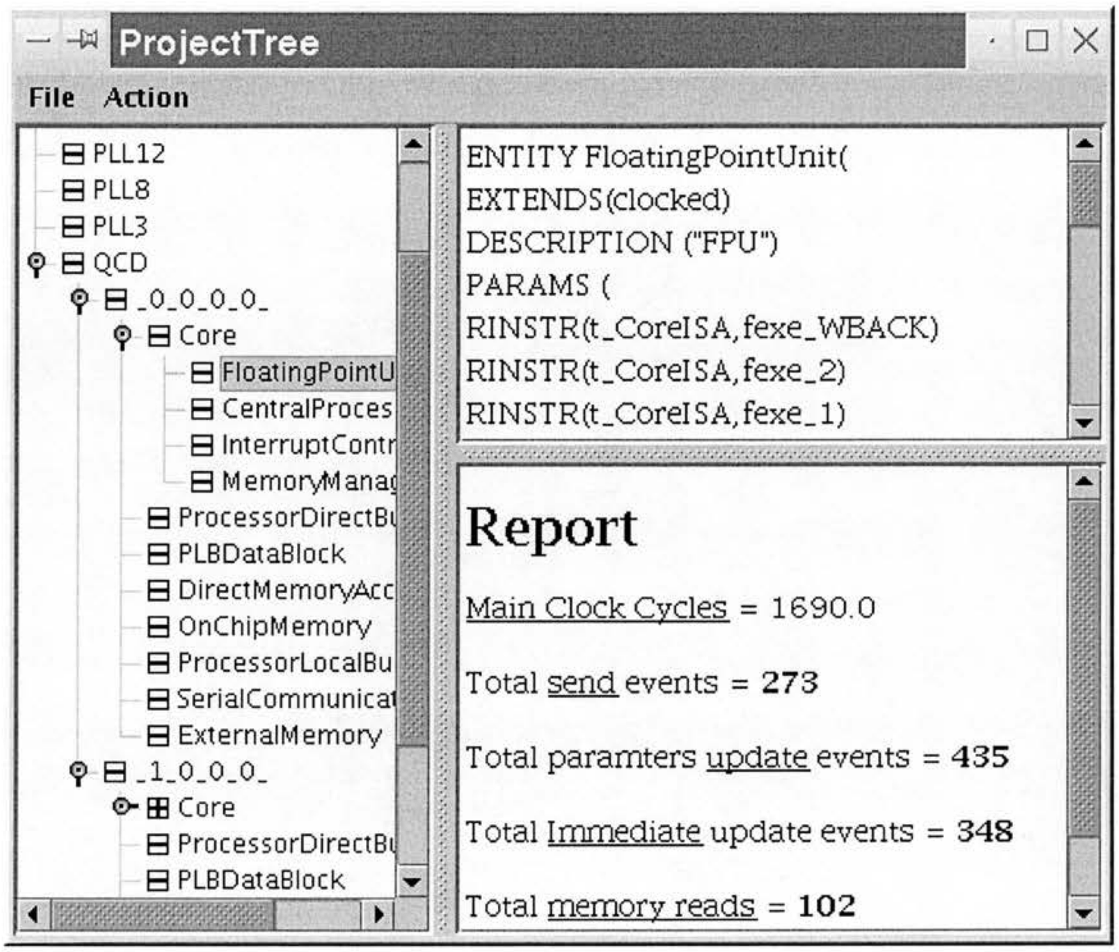
Figure 3.26: FPU parameters in the project EDL file

3.4.3 Trace file Viewer

The HASE animation and timing diagram tools are effective to inspect a simulation trace file as long as the project trace file is not very large and the graphical representation of the model allows all entities to be viewed on-screen. For detailed test runs on the QCDOC model, neither of the conditions is valid; the trace file is huge and an on-screen view of simulation entities is not possible.[7] An alternate mechanism was introduced whereby simulation end results can be viewed by selecting instances of an entity.

A tool was created that reads the EDL description (from `project.edl` file) and the simulation trace file to generate a summary of what happened to an entity in a simulation run. Figure 3.27 shows an example; the left window shows all instances of a HASE entity, the top right window prints the description of the selected entity provided by the simulation designer in the EDL file and the bottom right window report event changes that belong to a particular instance of an entity.

[7] Even in the smallest configuration, a 2x2x2x2 HASE QCDOC model, all entities cannot be viewed on-screen.



```
ENTITY FloatingPointUnit(
EXTENDS(clocked)
DESCRIPTION ("FPU")
PARAMS (
RINSTR(t_CoreISA,fexe_WBACK)
RINSTR(t_CoreISA,fexe_2)
RINSTR(t_CoreISA,fexe_1)
```

Report

Main Clock Cycles = 1690.0

Total send events = **273**

Total paramters update events = **435**

Total Immediate update events = **348**

Total memory reads = **102**

Figure 3.27: Entity instances and trace file viewer

For large simulation runs, for example, it is sometimes necessary to compare estimated sends and receives or memory reads and writes. The EDL viewer provides a quick way to read these estimates.

3.4.4 Off-line Interactive Plots

A HASE simulation trace file contains send, memory and parameter update events of all instances of entities in the model. In template-based multiprocessor systems, these trace files can be tens of MBytes, from which filtering useful information efficiently is not possible. Consequently, multiprocessor simulations often use C++ standard file input/output within a HASE entity behaviour description to store selective information. These text files can later be inspected off-line.

In the QCDOC model, simulations run for several thousand clock cycles. Tracing back and locating an error to an entity, and regenerating the sequence of events that led to the error was often a tedious and time consuming process. Graphs or plots showing unexplained spikes or curves plotted against the simulation time can reveal errors or anomalies. Moreover, they helps in identifying and understanding the overall behaviour of system variables during a simulation run.

An interactive plotting tools was employed to zoom into and out of the graphs of interesting values plotted over time. It also gives an opportunity to validate the overall behaviour of an entity. Figure 3.28 shows FPU performance (MFlops/s) with respect to clock cycles while figure 3.29 shows the diagram after zooming in to the selected area in figure 3.28.

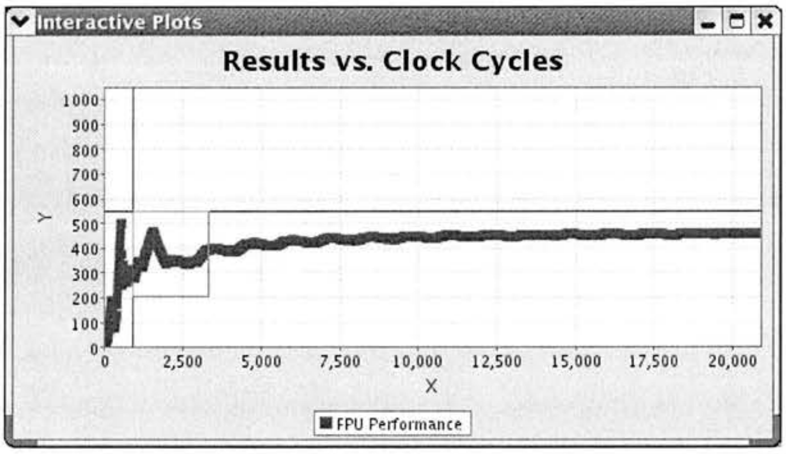


Figure 3.28: Before zoom

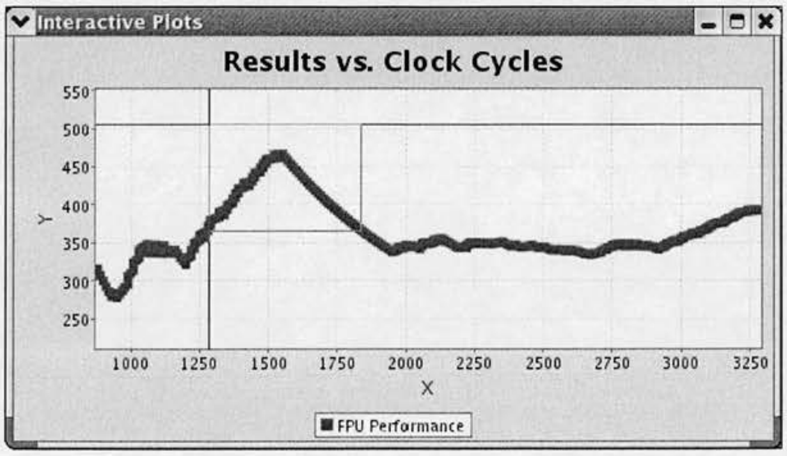


Figure 3.29: After zoom

By identifying the precise clock cycle value or set of values, it is possible to establish what happened during those cycles from the result files. For instance, in the FPU `ENTITY`, latencies for load instructions are stored in a text file with a clock cycle value. A load wait of 13 clock cycles was observed from the plot. The precise clock cycle values is then inspected in the instruction issue and commit files and traced back to the MMU `ENTITY`, the PDB `ENITY` and the EDRAM `ENTITY` to investigate the cause of this load wait. A prefetch load request that was subsequently invalidated as a result of an incoming write in the PDB `ENITY` was found to be the cause of this load wait.

3.5 Summary

The aim of this chapter was to introduce the background to the HASE QCDOC simulation model. An overview of QCDOC ASIC was presented. This was followed by the design and implementation and implementation details of the HASE QCDOC simulation model. Extensions and limitations to the HASE platform were identified and notable simulation model validation issues were addressed. The dynamics of the HASE simulation model, benchmark software, experimental setup and simulation results are presented in the next chapter.

Chapter 4

Design Space Exploration and Performance Analysis

The design details of the QCDOC architecture and implementation of its simulation model in the HASE were described in the previous chapter. This chapter is concerned with exploring the design space of the QCDOC architecture through its parameterised HASE simulation model. First, the computation and communication data paths, logic and control are explained. Second, the benchmark QCD code and the simulation parameters are introduced as part of the experimental setup. The performance studies on the HASE QCDOC model were undertaken with an optimised version of the QCDOC kernel routines. An understanding of the benchmark software structure and dynamics of the underlying system enabled identification of hardware parameters that are most likely to influence performance of the QCD code on the QCDOC system. Third, results of experiments conducted with various combination of HASE model parameters to explore design space of the QCDOC processing node are presented. Fourth, the interconnection network performance experiments and results are provided. Lastly, requirements for a larger (10K node) HASE QCDOC model are evaluated and collective communication schemes and their performance results are presented.

4.1 Data Paths, Logic and Control

The previous chapter introduced the custom components of the QCDOC design: the Prefetch EDRAM Controller (PEC) and the Serial Communication Unit (SCU). The PEC was introduced primarily to reduce the memory latency and to enhance memory bandwidth to the on-chip memory. The PowerPC read and write buses are of quad-word (128-bit) width and a 512-bit wide data bus between lower on-chip cache levels provides for efficient handling of read and write requests coming in and going out from the processor read and write master ports. The Processor Data Bus (PDB) interface of the PEC can be represented as a Level 2 (L2) cache because of its read data prefetching and write data buffering capabilities for L1 or primary data cache.

The L2 prefetching and buffering cache, together with another PEC interface called PLBDBLK, the Processor Local Bus (PLB) slave interface, create unique System-On-a-Chip (SOC) data paths for compute and communication data. Similarly, the QCD-specific communication component, the SCU, results in controls and logic for local as well as collective communication operations that are considered special to the QCD communication routines. In order to learn the behaviour of the QCD code execution on the underlying QCDOC system for performance studies on the simulation model, it was essential to establish the precise details of the computation and communication data movements over a QCDOC processing node.

4.1.1 Read and Write Logic

The advantages of a PEC with L2 prefetching and buffering cache are:

- L2 cache offers higher spatial locality (1024-bit lines) as compared to L1 (256-bit lines).
- It is a small memory and it can operate at the processor clock speed. Read data can be returned with only an additional 1-2 clock cycles latency to the CPU and FPU load pipes.
- A high bandwidth (512 data bit transfers at half of the processor clock frequency) offers low latency access to a large amount of on-chip data.

- Write buffering accepts L1 store data and reduces latencies associated with writes to the on-chip memory.

The on-chip memory, EDRAM, has three data ports connected to three PEC components: the L2 cache, the PLBDBLK and the DMA (for on-chip and off-chip memory transfers). A custom coherency protocol governs read and write operations across the three PEC ports to EDRAM, which snoops read and write addresses generated from the three ports. This snoop-style protocol, together with a small (4 KByte) prefetching and buffering L2 cache, can have potential short-comings:

- Thrashing may occur as the pattern of read and write requests from L1 causes repeated write flushes and read invalidates.
- As the write policy for most pages in the TLB is set to write back, to avoid latencies associated with stores in a load-store intensive code, it is the responsibility of the software to flush communication data from the L1 cache to avoid the problem of stale data.
- Load-intensive applications with reads from scattered memory locations, or with a sequence of reads beyond the L2 prefetch read register line boundary, may cause earlier read data to be replaced by incoming read data frequently; L2 prefetch read registers use a Least Recently Used (LRU) replacement policy.

The above mentioned scenarios can have an adverse effect on the performance of the overall code as read and write data transfers from EDRAM can take multiple clock cycles, in particular, the write flushes.

4.1.1.1 Read Data Path

Figure 4.1 shows the read data path of the QCDOC SOC. When a load request is at the cache read pipeline stage, the CPU initiates a request for a read to the MMU specifying the type of data and whether it is a CPU or an FPU read. If the address is found in the data cache, it can be made available without any blocking in the CPU load store pipe. In the case of a read miss, the TLB entries determine the storage attributes of the page to which the address belongs:

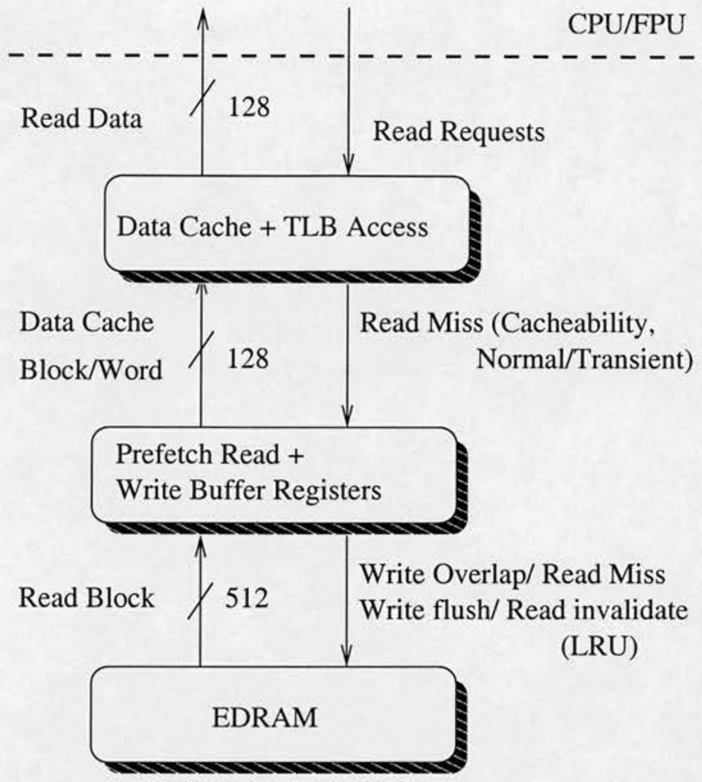


Figure 4.1: Read data path on the QCDOC ASIC

1. A cacheable page; a memory region can be either 'Transient', 'Normal' or 'Locked'.

2. A non-cacheable page.

If a line is cacheable, a normal or a transient line is allocated depending on the TLB entry, and if required, a line is replaced according to the round robin replacement policy. A cache line in the locked area will not be replaced during normal read and write operations. The critical word is delivered first to the operand register before the line fill. If a page is non-cacheable, no line will be allocated. For a read miss, and for a non-cacheable page, a read request is then made to the next memory level, the PDB.

A miss address request to the PDB first causes read addresses to be snooped by the write buffer registers. If a read-write overlap is found, the write buffer must be flushed before a read register is allocated according to the LRU policy and a prefetch read request is initiated. On the other hand, if the read address is found in a prefetch register with a "Valid" status, then the data is returned to the MMU. However, if the read address is found with "Pending" status, no new request will be made and data will be returned when that data arrives from EDRAM. When the address is not found in the read registers, a register will be allocated according to the LRU replacement policy, and a prefetch read request will be made to EDRAM to fill the 1024-bit read register.

4.1.1.2 Write Data Path

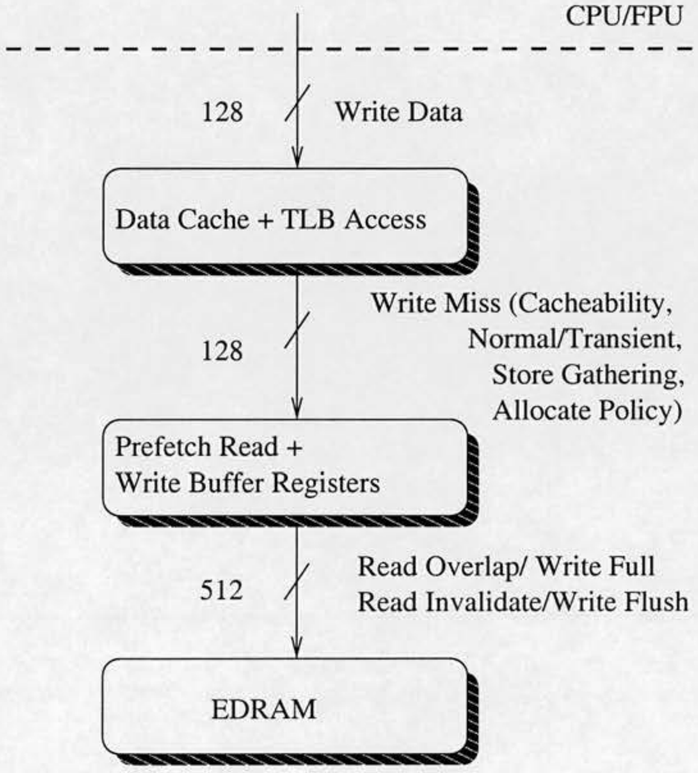

CPU/FPU
128
Write Data
Data Cache + TLB Access
128
Write Miss (Cacheability, Normal/Transient, Store Gathering, Allocate Policy)
Prefetch Read + Write Buffer Registers
512
Read Overlap/ Write Full
Read Invalidate/Write Flush
EDRAM

Figure 4.2: Write data path on the QCDOC ASIC

The write data path for the QCDOC SOC is shown in figure 4.2. When a store instruction is at the cache access stage of the pipeline, the data cache and TLB accesses are made. The data cache is accessed to determine whether there is a write hit or a write miss. At the same time, TLB storage attributes of the page are checked for:

1. A cacheable page that could:
   - be a copy-back or a write-through page.
   - have a write allocate policy or have no allocation for a write miss.
2. A non-cacheable page: this can have its store gathering attribute set or unset.

If an address is not found in the cache, and the page is cacheable, then the allocate policy is checked to determine whether a line needs to be allocated. Then the data packet is sent to the PDB. On the other hand, if the page is non-cacheable, the store-gathering attribute is checked. Non-cacheable data is forwarded to the PDB if store gathering is set to false or 128-bit gathering is complete.

Once a data packet (a word or quad words) arrives at the PDB, the read registers are checked to find an overlap of addresses. The status bit of a prefetch read register is set as "Invalid" if an overlap is detected. The write data is then written to a write buffer register and broadcast to other entities connected to the EDRAM read/write ports. If there is no space available for incoming writes or the write buffers are full, then a write register must be flushed.

4.1.2 Off-node Communication Operations

QCD communications requirements are regular and deterministic. Likewise, the QCD-specific communication unit, SCU, permits simple and straightforward communication data initialisation and off-node data transfer mechanisms. Off-node communication is handled via a custom protocol while on-chip communication instructions and data movement are controlled via memory-mapped device addresses.

The SCU supports simultaneous bi-directional communication in four space-time (t, x, y and z) directions.[1] QCD-specific communication ports include:

- A pair of four [send,receive] ports for plus direction communication ([TP,TM], [XP,XM], [YP,YM] and [ZP,ZM]).

[1]In fact, the QCDOC machine supports six-dimensional communications; however, at any given time, QCD calculations exploits four-dimensional communication support.

- A pair of four [send,receive] ports for minus direction communication ([TM,TP], [XM,XP], [YM,YP] and [ZM,ZP]).

Associated with each of the above 16 ports, there are buffers and registers. A send buffer stores double-word send data until a successful delivery to the receiver is acknowledged, while the receive buffers store the incoming receive data to perform error detection. Upon a successful receipt, an acknowledge message is sent to the sender. These send and receive buffers primarily deal with the communication data until it is successfully sent and received. Send and receive registers in the SCU, in contrast, serve as intermediate sources and sinks for the communication data in the send and receive buffers respectively. On the QCDOC ASIC, the source of the send buffer data is the EDRAM and receive buffer data must be written back to EDRAM for CPU processing. The processor clock cycles required to send and receive a 64-bit data packet are listed in table 4.1.

Communication Step	Clock Cycles
Send Overhead	14
Receive Latency	36
64 (data) + 8 (information) bits packet transfer	72
8-bit ACK packet + overhead	16

Table 4.1: Clock cycles required for 64-bit data transfers [BCC+02b]

The QCDOC SCU synchronises communication packets in double-word (64 bits) sizes. Data integrity checking and the custom communication protocol only support double-word data transfers between twelve nearest neighbouring nodes. A high speed serial link, HSSL, provides the point-to-point communication channel between the QCDOC processing nodes.

4.1.2.1 Bi-directional Communication Protocol

For reliable communications over a point-to-point link, each successful message transfer is expected to be followed by an acknowledgement message. A packet must be re-sent if the receiver sends a retry message or if there is a time-out. It is straightforward to devise a buffering scheme and a communication protocol if it is assumed that there are no packet losses over the communication channel. In practice, this is not the case. There can be communication errors: information loss, sender failure to deliver messages and loss of acknowledgement packets. Therefore, the communication sender is responsible for keeping a data packet in its send buffers until it is successfully acknowledged. On the other hand, the receiver should keep the message in a receive buffer to perform error checking before sending an acknowledge packet to the sender and flushing it to memory.

A conservative approach would be to send one packet at a time and wait for an acknowledgement. A time-out causes the sender to re-send the data packet. Although this scheme guarantees data integrity and is simple to implement in hardware, it results in poor utilisation of the communication channel due to idle time (as shown in figure 4.3).

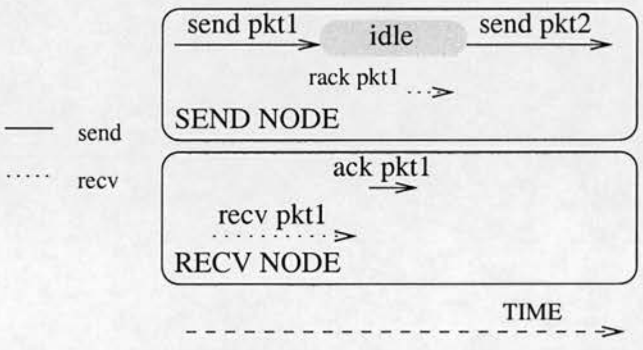


Figure 4.3: Data transfer between neighbouring nodes without communication buffers

With the availability of send and receive buffers in the QCDOC's SCU, it is possible to send more than one double-word before an acknowledgement is received. A custom communication protocol and a single-bit Error Correcting Code (ECC) are implemented in the SCU ensure successful data transfers. The custom SCU communication protocol allows up to three double-words to be transferred before an acknowledgement of the first is received. A send port keeps a record of acknowledged data packets sent and re-sends a data packet if it either fails to receive an acknowledge message or it receives an error/retry message. The send buffers are updated in a round robin fashion and the receive buffer flushes successful packets to the corresponding receive register. Figure 4.4 shows that using the communication send and receive buffers, a second packet can be sent without first being acknowledged.

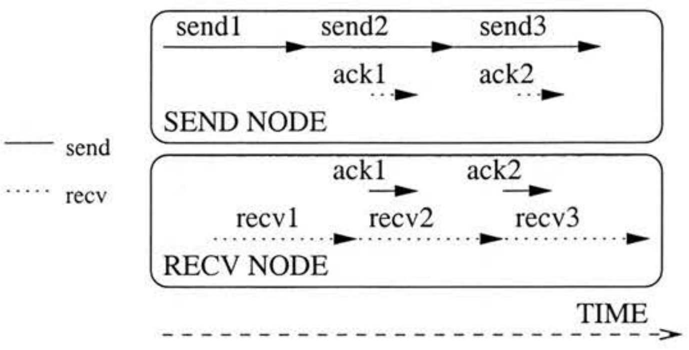


Figure 4.4: Data transfers between neighbouring nodes with communication buffers

As a result, the channel utilisation improves; figure 4.4 shows no idle clock cycles. In order to send more than one message without an acknowledgement, the sender must be able to identify messages in its communication buffer. This is necessary because an error message from receiver or a time-out should result in a re-send of the erroneous message (not all messages in the communication buffer). Similarly, the receiver should be able to inform the sender which packet has an error. The cost of implementing this protocol increases, both at sender and receiver, with an increase in the communication buffer sizes.

For QCD communication requirements, the SCU supports bi-directional communication such that it is possible to send and receive data in both positive and negative directions simultaneously. The bi-directional communication with custom protocol is shown in figure 4.5.

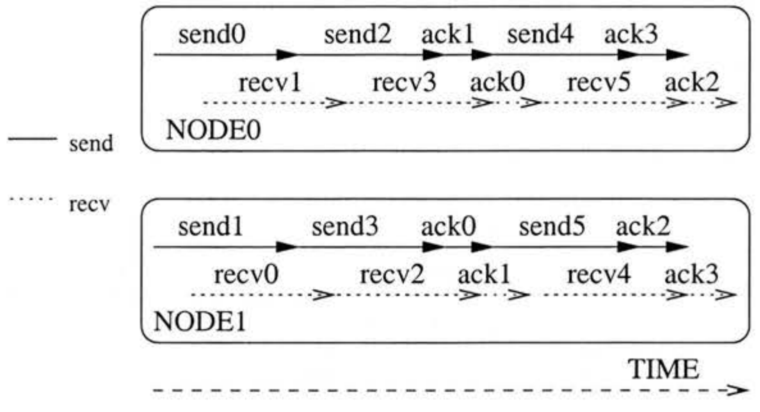


Figure 4.5: Bi-directional SCU protocol implementation

The send and receive communication buffers, and the communication data integrity checking mechanism in the form of a custom communication protocol are essential for bi-directional simultaneous communication between neighbouring nodes. Associated with each send port is a send buffer and with each receive port is a receive buffer. Up to 24 (12 simultaneous send and receive) operations between QCDOC neighbouring nodes in a six-dimensional torus are permissible in the QCDOC machine using the custom communication protocol.

4.1.2.2 On-chip Communication Control and Data Paths

The two nearest-neighbour communication steps in a QCD kernel routine follow similar sequences of operations in opposite directions. A "Send forward" call performs a data send in the positive direction and a data receive in the negative direction. Similarly, the "send backward" process involves sending data in the negative direction and receiving from the positive direction. Figure 4.6 shows four of the 12 send registers and four of the 12 receive registers performing nearest-neighbour communication. The step-by-step process of a QCD nearest-neighbour communication is as follows:

1. The CPU sends block-strided send instructions to TP, XP, YP and ZP DMA registers and block-strided receive instructions to TM, XM, YM and ZM DMA registers. A block-strided instruction contains the start address in the EDRAM,

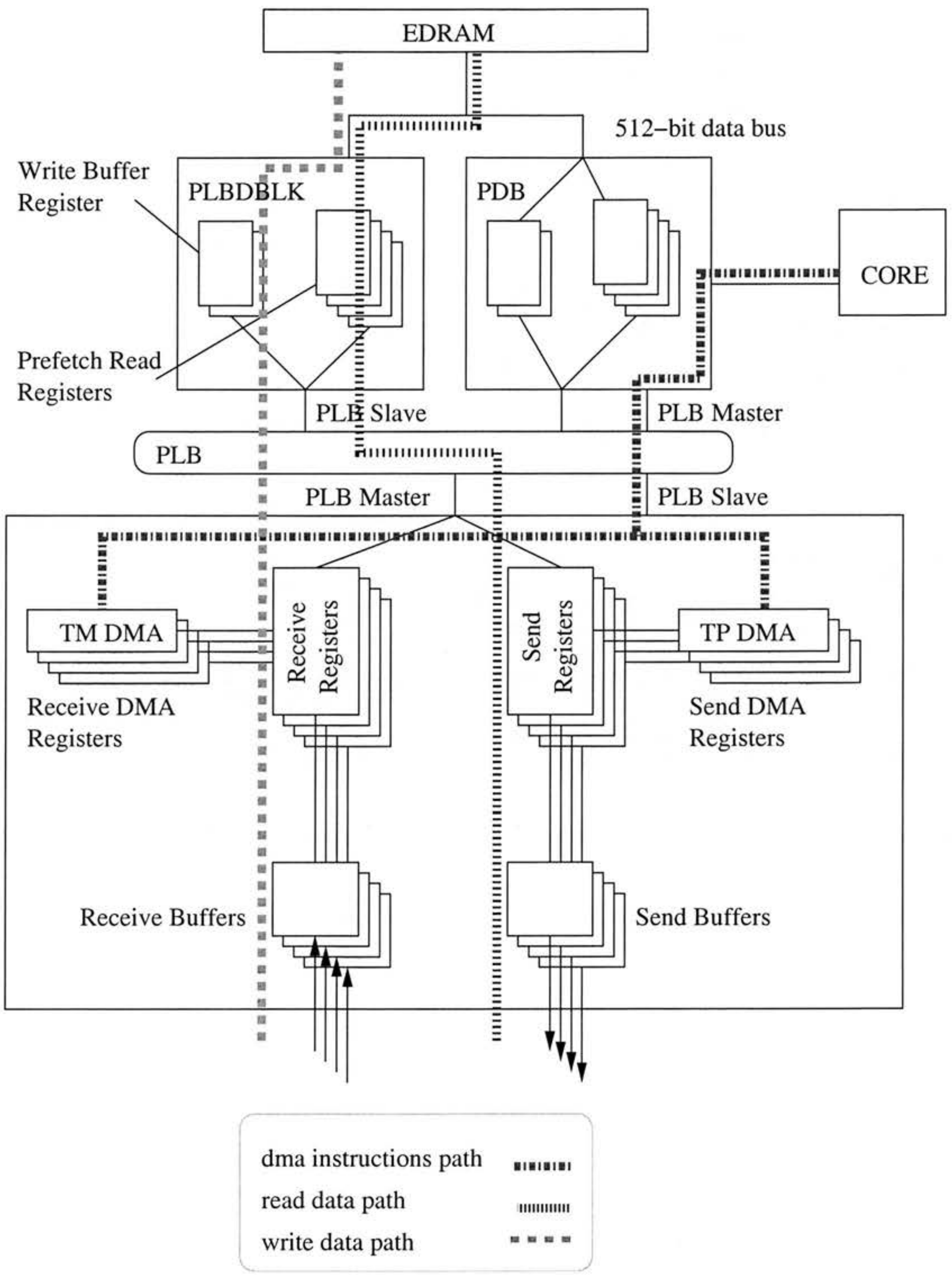


Figure 4.6: Communication data and control paths on the QCDOC ASIC

number of blocks, size of blocks and stride.[2] An instruction from the CPU uses a PLB master interface via PDB to write to the SCU SRAM through the SCU slave port.

2. Once the DMA instructions are set up, the send DMAs initiate reads from EDRAM via the PLB master port of the SCU. Since more than one register is ready to read at one time, a First In First Out (FIFO) arbitration policy is employed for the PLB accesses. A read request via PLB goes to a PLB slave called PLBDBLK. PLBDBLK contains a set of 1KByte prefetch read registers and can read 512 bits from EDRAM at half of the processor clock frequency. The required data, once ready in a PLBDBLK read register, is transmitted to SCU via PLB. The read data received is copied into the corresponding send register and then transferred to the send buffer for off-node communication.

3. Data received at the neighbouring node is acknowledged and the successfully transmitted data can be removed from the send buffers. Receive buffers flush data to receive registers. The frequency at which the read registers are filled is much higher than the frequency of off-node transmission. Hence at times, no read via PLB is performed because the read registers are full. The number of send and receive registers and buffers have been made parameters in the HASE QCDOC model to investigate their effect on communication performance and channel utilisation. At the receiving node, the data received is stored in the receive register and, according to the DMA sequence, transferred to the EDRAM via PLB. The PLBDBLK writes data in its write buffer registers and flushes to the EDRAM if the buffer is full or in response to a CPU request (read-write overlap).

The above steps are repeated until all reads and writes, according to the send and receive DMA specifications, are complete.

[2]A DMA instruction represents a specified number of blocks of contiguous double-words, with each block separated from the preceding by a fixed stride.

4.1.3 Collective Communication Operations

Two double-precision floating-point global reduction operations are required per QCD kernel execution. Although only a few global communication steps are required per kernel iteration, as compared to the nearest-neighbour communication steps, efficient collective communication implementation on a QCD machine is critical to the overall performance. This is because the global sum communication times cannot be compensated by the parallel execution of local computation steps. The QCDOC SCU contains a "pass through" unit for automating the global communication processes, thereby maximising the efficiency of the global sum operations.

In the QCDOC machine, potentially, there are three mechanisms to perform global sums over a torus network [BCC$^+$02b]: hardware shift-and-add, enhanced hardware shift-and-add and QCDSP-style. The HASE QCDOC model contains a binary tree network in addition to the torus network, which opens up the possibility of three more global sum modes. Details of the HASE QCDOC global sum options over a four-dimensional torus and a binary tree network are presented in section 4.5.

4.1.3.1 Global Sum on the QCDOC Network

One way to compute a sum over a four-dimensional torus requires adding data simultaneously along each axis. For example, to find a global sum over a (Nt * Nx * Ny * Nz) machine, the first step is to collect data in the `t` direction from Nt-1 nodes and then from Nx-1, Ny-1 and Nz-1 nodes for `x`, `y` and `z` directions respectively. The number of steps required will be (Nt-1)+(Nx-1)+(Ny-1)+(Nz-1). Since the SCU supports bi-directional communication, these steps can be halved. Figure 4.7 shows the detail of this communication pattern for a 2D torus.

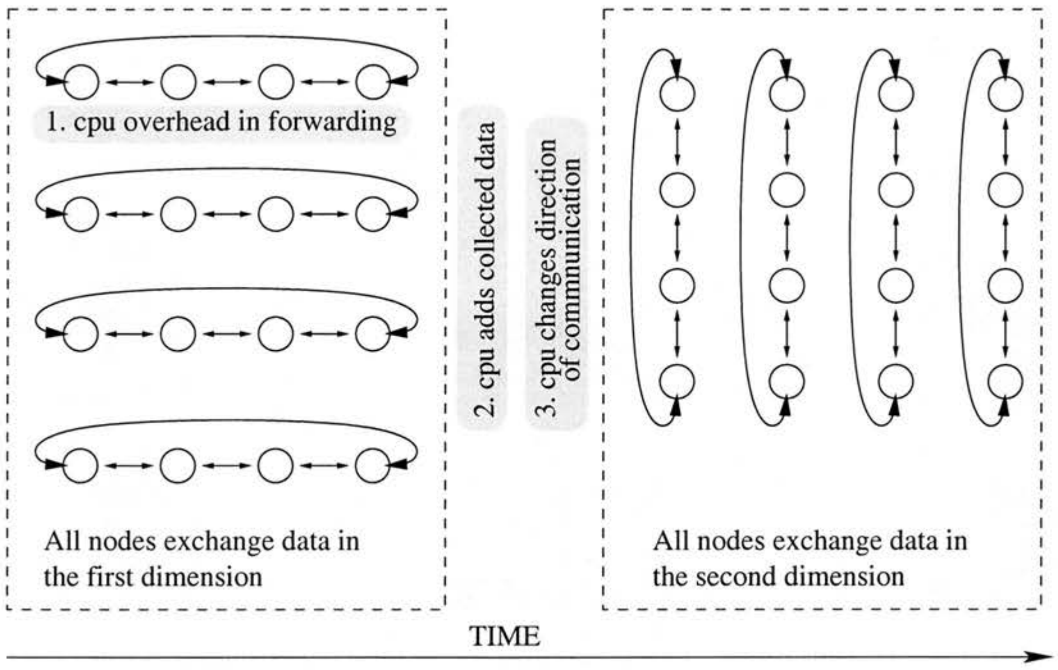

1. cpu overhead in forwarding
2. cpu adds collected data
3. cpu changes direction of communication
All nodes exchange data in the first dimension
All nodes exchange data in the second dimension
TIME

Figure 4.7: Global sum operation on a torus network

CPU/FPU overheads include:

1. CPU intervention may be required to forward data from an incoming node to an outgoing node.
2. At the end of (N-1) steps in each direction, data is sent to the FPU for floating-point addition.
3. The sum is received from the FPU and the direction for both sends and receives changes.

Without the pass-through unit, shown in figure 4.8, each packet received will be copied into memory, then read again and copied to a send register to be transferred to the neighbouring direction.

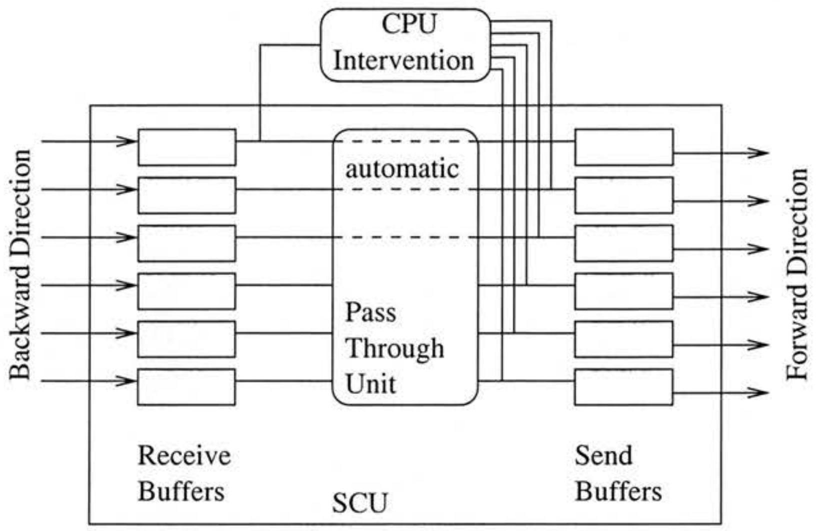


Figure 4.8: CPU overheads in a QCDOC SCU pass-through unit

Once (N-1) transfers are complete along an axis (direction), the N data packets (in an identical order on each node, ensured by assigning a pointer to values collected with respect to node identifiers) are sent to the CPU to compute the sum of N floating-point numbers. Then the data transfers along the next direction start.

It is also important to identify the role of the CPU i.e., to determine how automatic (or otherwise) the pass-through mechanism can be. The CPU's contribution in calculating floating-point sums after communication in each direction is essential. It cannot be avoided without a bit manipulation component in the SCU. However, individual receives followed by sends can be automated; this is classified as *hardware enhancement*. One more enhancement is possible by allowing the incoming bits to be re-directed to the send port, rather than waiting for a successful transfer to complete. Although such a scheme was successfully adopted in the QCDSP machine, the HSSL protocol in the QCDOC machine restrict the bit-serial re-direction.

4.2 Experimental Setup

Potentially, a large number of factors can influence the performance achieved by a parallel code over a high-end parallel system including the application workload software characteristics, underlying hardware configurations and the influence of operating system interactions. Thus a full-scale system analysis, theoretically, needs very large number of experiments, each with a certain combination of above-mentioned factors. In order to identify parameters that are likely to have a significant impact on an application code performance, a thorough knowledge of the three factors is necessary: the parallel workload characteristics, data and control paths of the underlying hardware, and the operating system behaviour.

- Operating system interactions and interventions do not occur during the execution of the QCD kernel over the QCDOC machine. QCDOC operating system is a custom-designed operating system described as a subset of the unix operating system. In addition, the QCDOC operating system contain highly optimised, QCD-specific communication libraries. During the QCD kernel code execution on the QCDOC processing node, the user program runs in a single thread,[3] or has uninterrupted access to a QCDOC processing node's resources.

- On-chip computation and communication data movement, their logic and control have been described earlier in section 4.1.

- The application workload for benchmarking comprises a parallel QCD kernel developed and optimised for the QCDOC system. The computation and communication characteristics of this kernel are explained in the following sections.

4.2.1 Characteristics of the QCD Benchmark Software

The QCD sparse matrix kernel shares characteristics with many other scientific application codes. Vetter and Mueller [VM02] examined the communication characteristics of a number of scientific codes and found that sparse matrix computations are ubiquitous in computational science since they arise whenever differential

[3]B. Joo (2002) QCDOC operating system design team, private communication.

equations are solved using numerical techniques such as finite element and finite difference methods. A large number of data-parallel matrix-vector multiplication iterative solvers, Ruede [Rue97] reported, need to share intermediate results frequently with neighbouring processing nodes. In distributed memory systems, according to Cypher *et. al.* [CHKM93], these message passing communication calls are explicit, unlike shared memory communication. Conjugate Gradient (CG) is an example of one of the most popular iterative algorithms for solving sparse matrix-vector equations. Since the research presented here has been based on the QCDOC architecture, the variant of CG for QCD calculations is presented. At each step of a CG iteration, the neighbouring processing nodes exchange data. Figure 4.9 shows this communication in a periodic two-dimensional torus.

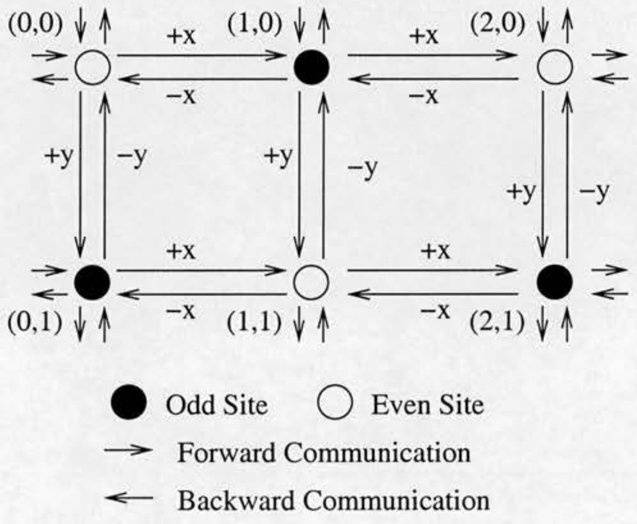


Figure 4.9: QCD Nearest-neighbour communication on a 2-D mesh

Several QCD software repositories: MILC [MIL], FermiQCD [Pie00] and Columbia Physics Systems (CPS) offer a range of application codes developed according to the particular interests of those research communities. These repositories are in high level languages like C and C++, but contain optimised sparse vector matrix multiplication routines. MILC and FermiQCD have routines optimised for Pentium SSE[4] instruction.

[4]A vector type instruction. The SSE data type specifies that four single precision floating-point numbers can be stored in each register and that the instructions that operate on these registers can do four floating-point operations in parallel as they operate on the whole register as a single register.

CPS contains QCDSP optimised kernels for the QCDSP platform, the predecessor of the QCDOC. These optimised routines are selected because they form a large fraction of code execution times, namely solving a system of linear equations. The parallel CG algorithm and its variants govern implementations of parallel QCD calculations.

Bernard *et. al.* [BDG$^+$95] described the sparse matrix computations as the "rate-limiting" step in the QCD code. The high-dimensional quark matrix to link vector multiplication requires gathers from widely separated locations in memory.[5] In the formulation

$$Mx = b$$

x and b are complex vectors containing three colour indices and four dimensional lattice coordinates. The matrix M is given as

$$M = 2maI + D$$

where I is the identity matrix, $2ma$ is a constant, and matrix D is called "Dslash". The computation requires summation over four lattice directions where each site has eight nearest neighbours. The linear system using CG method involves solving

$$M^{\dagger}M = (2ma)^2 I + D^{\dagger}D$$

where $D^{\dagger}D$ represents the kernel test code; $M^{\dagger}$ represents the conjugate transpose of matrix M. Lattice computations are simplified using "even odd preconditioning" techniques, thereby applying inversion of the matrix at even sites only. The sum of the coordinates of a lattice site determines whether it is an even site or an odd site. Using the preconditioning scheme, the problem size can be reduced by half.

[5]Mapping of QCD particles onto a lattice is explained in detail in section 2.3.

4.2.1.1 Parallel CG Multiplication

The CG inversion requires repeated multiplications by the "Dslash" matrix D. During these solutions, the solution vector evolves from the trial vector. Link matrices are stored with sites at the processing nodes (section 2.3). A typical application involves:

1. Sending intermediate vectors in forward (positive) directions.
2. Multiplying intermediate vectors by link matrices in positive directions.
3. Sending matrix vector products in backward (negative) directions.
4. Multiplying intermediate vectors from neighbouring sites by the link matrices of the receiver.
5. Subtracting matrix vector products from neighbouring matrix vector products.

A number of the above computation and communication steps have no data and control dependencies, hence they can be overlapped. For instance, the multiplication of intermediate vectors and their communication in the forward direction can be performed in parallel.

The above steps written for the QCDOC assembler kernels by Boyle [Boy01] are performed in a parallel manner using the subroutines listed in table 4.2.

Subroutine	Description
QCDOC_ChDecomp	This routine decomposes and shifts information in separate lattice directions.
Communicate Backward	Previously decomposed values are sent to the backward and negative direction neighbours.
QCDOC_ChDecomp_hsu3	Decomposition and multiplication of the Wilson matrix. This computation step can be performed in parallel with the previously initiated communication operation.
Communicate Forward	Communicate the result of the above multiplication to the four neighbours in the forward directions.
Backward Complete	Wait or gather result from the first local communication step, the nearest-neighbour backward communication.
QCDOC_ChRecon_su3	Reconstruct the matrices and multiply by the incoming values from the four neighbours collected earlier. In parallel with this computation step, the nearest neighbour forward communication is performed.
Forward Complete	Wait or gather result from the first local communication step, the nearest-neighbour forward communication.
QCDOC_ChRecon_add	Final vector reconstruction.

Table 4.2: QCD assembler kernel subroutines

The kernel subroutines (in PowerPC assembly) are developed by the QCDOC design team, which are optimised for the QCDOC machine. Appendix A contains the code listing of the smallest subroutine QCDOC_ChDecomp, and describes the input code format for the HASE QCDOC model. In addition to the Dslash multiplication routine, nearest neighbour communications and double-precision floating-point global sum (collective communication) operations are computed. The results of these global sums determine whether a CG iterations has converged or not within the approximation limits specified at the beginning of a QCD calculation.

For the above subroutines execution, the data layout is defined in the TLB using PowerPC memory management instructions such that send data resides in non-cacheable pages, receive data in transient pages and other compute data in normal cacheable pages. Receive data is kept in the transient pages to avoid unnecessary flushing of useful data. A flush routine can selectively flush the transient pages.

4.2.1.2 Computational Requirements

Lattice QCD kernel code execution is deterministic; its computation and communication requirements do not rely on dynamically computed or run time values. The number of kernel iterations however depends on the residual vector, which is computed after each kernel iteration. Yet, QCD is classified as a grand challenge problem primarily due to the enormous floating-point data- and computation-intensive requirments.

Hence, first the floating-point instructions mix of the computation subroutines at run-time is quantified for the smallest lattice volume 2x2x2x2. This is shown as a fraction of 64-bit floating-point load, store and arithmetic (ALU) instructions for the optimised QCD routines, QCDOC_ChDecomp, QCDOC_ChDecomp_hsu3, QCDOC_ChRecon_su3 and QCDOC_ChRecon_add in figure 4.10, figure 4.11, figure 4.12 and figure 4.13 respectively.

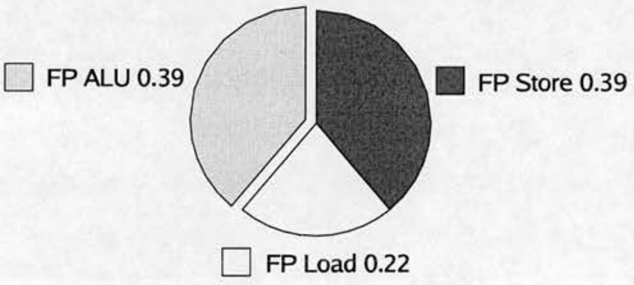


Figure 4.10: QCDOC_ChDecomp floating-point instructions

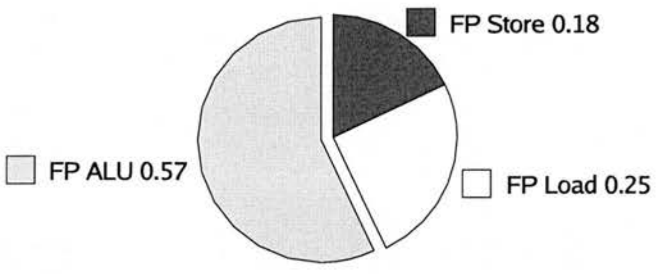


Figure 4.11: QCDOC_ChDecomp_hsu3 floating-point instructions

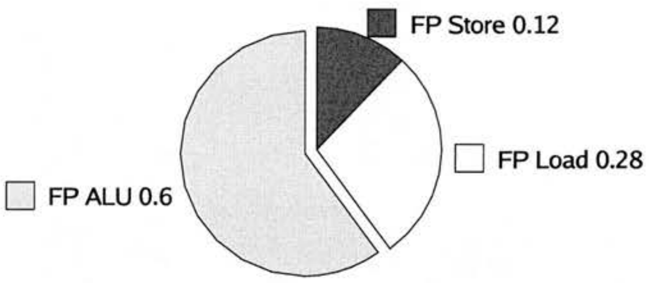


Figure 4.12: QCDOC_ChRecon_su3 floating-point instructions

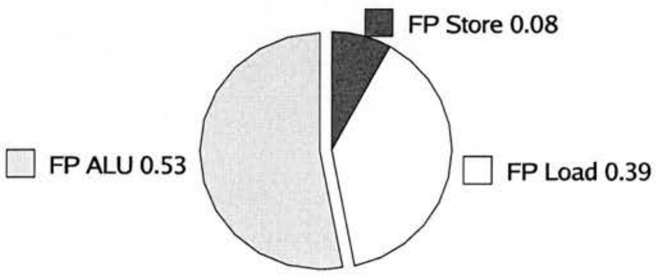


Figure 4.13: QCDOC_ChRecon_add floating-point instructions

These figures not only show the fraction of load-store instructions in the test kernels but also the large fraction of arithmetic instructions, except for the decomposition subroutine in figure 4.10. Typically, a number of load operations are followed by an even larger series of arithmetic operations. The PEC prefetching scheme addresses this load requirement of the QCD calculations to hide the data cache fill latencies by bringing in four data cache lines in the prefetch read buffers.

4.2.1.3 Communication Requirements

The SCU is designed to support QCD communication subroutines. The QCD CG computation steps require:

- nearest neighbour communications in four space-time directions: communicate backward and communicate forward operations; and
- 64-bit floating-point global sums to compute the norms of two intermediate vectors.

Thus, during each iteration of a Dslash routine (four in a CG step) two nearest-neighbour communication steps are performed. In a CG step, two global double-precision floating-point sums are needed. Hence, neighbouring communications are frequent as compared to the collective communication operations.

In order to optimise the QCD-specific communications, the SCU has custom-designed components:

- For nearest-neighbour communication
  - DMA registers to store block-strided instructions;
  - Send and receive registers to store incoming and outgoing data to and from on-chip EDRAM memory;
  - Buffers associated with send and receive ports.
- A design block called the Pass-through unit that allows simultaneous storing and forwarding of neighbouring data to enhance the efficiency of collective communications.

The nearest-neighbour communications are governed by the custom protocol described in section 4.1.2.1. This protocol allows up to three double-words to be transmitted before an acknowledgement is received. A three double-word buffer is therefore associated with each send and receive port; a packet must be re-transmitted in case of an error or absence of an acknowledge packet.

Dally [Dal90] proved analytically that the six-dimensional torus network (the "physics" network) is a k-ary n-cube topology which outperforms high-dimensional networks of the same bisection width and is considered best suited for high performance MPP interconnection networks. k-ary n-cube interconnection networks contain n as the dimensions of the cube and k as the radix. The dimension n, radix k and number of nodes N are related by the equations:

$$N = k^n, \qquad k = \sqrt[n]{N}, \quad \textit{and} \qquad n = \log_k N$$

The full size QCDOC machine will have k = 10, n = 4 and N = 10K, the number of processing nodes. Bi-directional communication in the QCDOC interconnection network can take place between neighbouring nodes along n axes simultaneously.

Since off-node communication latencies are considerably larger than on-node latencies, a communication buffering scheme in the communication unit is essential. The SOC design permits communication data transfers from on-chip memory to the communication unit at a high clock rate. Likewise data received in the communication buffers from neighbouring nodes can be forwarded to the on-chip memory as shown in figure 4.14.

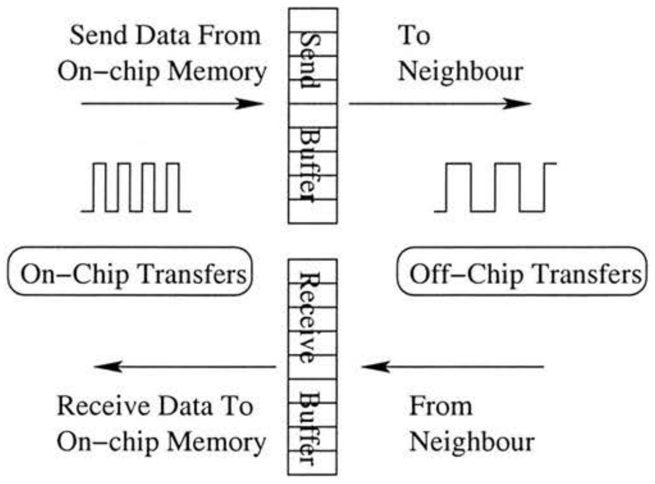


Figure 4.14: Clock frequencies for on-chip and off-chip data transfers via the communication send and receive buffers

High connectivity (24 send and receive links per processing node) was realised with the high performance serial link technologies. Traditionally, parallel links have provided high performance for multiprocessor interconnection networks. Presently however, these parallel data buses are being replaced by serial links, especially in high-end parallel networks [Kon99]. Parallel links, at a high clock speed, suffer from clock skews. Serial links operating at up to 1 Gigabit per second are available and Galloway *et. al.* [GNA02] demonstrated that they are now an ideal choice for inter-cabinet links in large systems.

4.2.2 Parameters of the HASE QCDOC Model

An in-depth knowledge of the communication and computation requirements of the QCD kernel, together with an understanding of the computation and communication data paths, logic and control, enabled identification of the components of the QCDOC model that may have a significant contribution to performance improvement. The custom-designed components, the PEC and the SCU, are the key candidates for performance exploration studies. Also, the memory hierarchy and the interconnection network parameters can also affect the overall code execution times. Performance studies of a complex system do not depend on a single parameter: a combination of parameters and different parts of systems can contribute to the performance results. Therefore, the custom-designed entities in the HASE QCDOC model are parameterised along with the entities that are involved in the on-chip and off-chip communication data movements. A range of experiments on the parameterised HASE model was conducted in order to explore the memory hierarchy design space and to investigate the off-node QCD communication performance. Modelling the QCDOC machine in the HASE allows simulation experiments by varying:

Machine size and dimensions: The HASE MESH*n*D templates provide a flexible and efficient mechanism for altering the number of processing nodes in a mesh in a chosen dimension. Also, using n, the number of dimensions, i.e. a three- or a four-dimensional mesh can be specified.

On-chip operation frequencies: An advantage of modelling the QCDOC node in the HASE was a cycle-accurate simulation of the custom components: the memory hierarchy and the communication interface. A further advantage of cycle-accurate simulation came from the HASE library clock mechanism such that it allows the clock frequencies of individual HASE entities to be parameters. Before a simulation run, the clock frequency of an entity can be altered with respect to the processor clock frequency through a parameter window. This was found to be extremely useful in experiments when the processing rate or data bandwidth of an entity has to be altered.

Memory configurations: The HASE memory `ARRAY` parameter mechanism, which allows the size of a memory to be altered at run time, was exploited whenever a change in array configuration would result in an increase in the physical memory size for the host machine. Altering the memory module configurations, even though highly desirable in a system experiments, posed great challenges in the simulation design. An increase in the size and number of memory registers can exhibit enormous memory and execution constraints on the host machine. The HASE `ARRAY` options that were introduced in HASE for light-weight physical memory requirements were quite useful in template-based HASE QCDOC model with a custom-built memory hierarchy. Most multiprocessor simulators, for example, SimOS [RBDH97] and RSIM [PRA97] are restricted to a conventional memory subsystem. Network simulator SMART [PV97], on the other hand, represents a static configuration of a processing node.

In addition to the above features, the HASE permits simulation models to incorporate different bus widths between entities and a range of data transfer protocols including coherency and communication protocols. A combination of parameters results in a different machine configuration and the performance results allow the simulation user to compare and contrast various design features of a parameterised system.

Appendix B presents the details of parameters of HASE QCDOC model design entities. Along with the parameter description, values filtered by an entity during a simulation run are listed. These time stamped values (against execution clock cycles) helped in validation, performance exploration and analysis of the simulation runs.

For a range of simulation experiments, the parameterised HASE entities are essential to allowing alternative configurations of the QCDOC processing node design components to be analysed and investigated. Central to the execution of the HASE QCDOC model is the host interface entity, the SimMode `ENTITY`. The SimMode `ENTITY` controls and directs a simulation run for the HASE QCDOC processing node entities.

4.2.2.1 The SimMode `ENTITY`

This entity provides a central control mechanism over the HASE QCDOC model between successive simulation runs. SimMode parameters include `SimulationMode`, `TestEntity`, `t_sites`, `x_sites`, `y_sites`, `z_sites` and `SimTime`.

`SimulationMode`

> The HASE QCDOC simulation behaves differently in each of the five simulation modes, as stated in table 4.3. These modes are included in order to facilitate design, debugging and experimentation with the HASE QCDOC simulation models. For instance, in the HASE QCDOC model, often when a memory parameter value is altered, it results in writing to a physical memory file. The file write option is extremely slow, especially if multiple entities and files are involved, thereby making it extremely difficult to synchronise in a *clocked* system. Moreover, during the development of the HASE QCDOC model, testing and debugging of different design entities require switching on/off selective entities. Thus, a `SimulationMode` parameter is introduced in the SimMode `ENTITY` such that all HASE QCDOC model entities read a notice informing them of the current simulation mode. In test mode, only the selected entity is considered active.

Mode	Description
`WarmUp`	Start up check. A failure in this mode means that no other mode will work. It indicates a problem in the HASE platform or an erroneous system configuration.
`InitialiseMemoryFiles`	When the configuration of a HASE memory file changes, the physical memory contents must be updated. For instance, if the number of sets in the set-associative data cache is altered through the MMU parameter window, the data cache `.mem` file must be re-written. The `InitialiseMemoryFiles` mode allows these contents to be updated selectively (through the `TestEntity` selection box), such that only the memory contents of the `TestEntity` are updated. No other `.mem` file in the system is written, since only the `TestEntity` is active in the simulation.
`SetupCommunicationNetwork`	HASE permits updating of the machine dimensions via the project EDL file. The `SetupCommunicationNetwork` mode prints the unique identity of each processing node in the complete machine and its neighbours' identities in x, y, z and t directions.
`RunProgram`	The `RunProgram` mode causes the QCD code in the instruction cache to be executed.
`TestMode`	During the project development phase, it was necessary to design and test individual properties of complex entities. The `TestMode` mode was included in the SimMode `ENTITY` so that individual entities can be tested and debugged while remaining entities do not participate in the simulation. The individual components are selected via the `TestEntity` parameter.

Table 4.3: Simulation modes (`SimulationMode`) in the HASE QCDOC model

`TestEntity`

This parameter box shows all entities in a QCDOC processing node. The `TestEntity` parameter selects an entity for the InilitialiseMemoryFiles and TestMode simulation modes.

`t_sites, x_sites, y_sites` and `z_sites`

The computation and communication requirements of a QCD code depend on the number of lattice sites per node. The values of the `t_sites, x_sites, y_sites` and `z_sites` parameters are included to allow the scalability of the QCD hardware to be studied as the computation and communication demands of the system workload vary.

Varying the workload properties as well as the architectural parameters allows hardware and software co-simulation in the HASE system. This hardware software co-simulation approach, along with the flexible parameter change mechanism in HASE, resulted in a wide range of experiments, which are otherwise impractical to conduct within a unified framework.

`SimTime`

`SimTime` represents the maximum number of clock cycles for a simulation. There are two reasons for this parameter: firstly, for the debugging and analysis of the simulation, it is sometimes necessary to allow a simulation to stop after a couple of cycles. Secondly, in case of a deadlock situation, to prevent a simulation from running indefinitely.

4.3 Processing Node Performance Search Space

Exploration of the design space of the QCDOC architecture is one of the main aims of the research presented in this thesis. The task begins with the processing node design space. Experiments with a combination of HASE QCDOC entities' parameters have been conducted to investigate the performance bottlenecks in the QCDOC processing node design. Furthermore, ideal or upper limits to the achievable performance are quantified. In a simulation model, unlike real systems, it is possible to emulate an ideal case scenario, for instance, 100% cache hit or a zero latency.

Firstly, experiments were performed to quantify the difference in the simulation execution times with and without taking advantage of the complete Instruction Level Parallelism (ILP) in the CPU and FPU. HASE QCDOC processing node simulations interpret each PowerPC instruction in an execution pipeline, as in a real system. The drawback of this approach is that one target machine clock cycle requires many host processing clock cycles for its execution, thus resulting in long simulation times. A number of execution-driven multiprocessor simulators, for efficient simulation runs, do not completely interpret ALU instructions. Instead, these are executed on a host machine and replaced by a fixed cycle count. Experiments are conducted on the model to identify whether complete interpretation of a large number of floating-point ALU instructions is necessary for high fidelity execution times for the QCD code experiments.

Secondly, the benchmark code for these experiments is the hand-coded QCD kernels in the PowerPC assembly produced by the QCDOC design team. The maximum achievable performance of the instruction mix was not documented. Even though the QCD kernel is floating-point ALU intensive, on a RISC or load-store processor, performance loss is mainly attributed to the unavailability of data in registers or L1 data cache. In order to determine how much improvement is possible with a given instruction mix in a QCD routine, a set of experiments was performed so that the overheads of the load store misses could be quantified. Thus, experiments were conducted in an "Ideal Mode" to determine performance loss due to the given instruction mix and to determine the ideal performance that could be achieved with the given code, rather than a setting a target of 1 Gigaflops.

4.3.1 Execution-Driven CPU

Execution-driven simulations have been a popular choice because of their relatively small simulation run times compared to instruction-driven simulations. Sivasubramaniam [Siv97] described an execution-driven simulator as one which tries to execute the bulk of instructions at the native machine speed. In a multiprocessor simulation model, an execution-driven model only simulates events like load/store and off-node communication operations. In doing so, i.e., by not completely executing an instruction and data path, the end results may not be accurate. Different host and target machine characteristics can result in misleading performance results. For example, the target system for QCDOC model is an embedded PowerPC processor with no operating system overhead for the application code execution, while the host machine is an Intel Xeon processor with Linux operating system.

The effects of not executing bulk register-to-register instructions are shown in figure 4.15, in which execution-driven simulation times (without ALU instructions interpretation) in clock cycles are compared with instruction-driven simulation times (with ALU instruction interpretation).

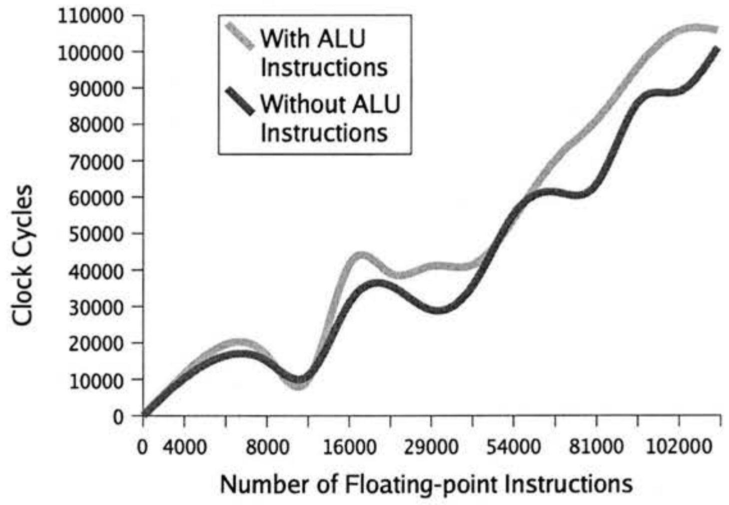


Figure 4.15: Execution-driven simulations vs. instruction-driven simulations

A parameter was introduced in the CPU `ENTITY` which selectively interpreted the load, store and branch instructions. From figure 4.15, it was identified that there is no correlation between the total number of instructions executed and the number of load and store instructions clock cycles. Hence, it is not possible to replace execution of an ALU instruction by a fixed cycle count. In ILP processors, instruction execution depends on a range of factors, including dynamic data and control dependencies within the execution pipelines. It was therefore deemed necessary to interpret and to execute complete instruction and data paths on the model.

4.3.2 Maximum Achievable Performance

Since the QCD test software was used on as *as is* basis, the maximum peak performance the code could achieve in an ideal situation (with two floating-point and three integer execution pipelines) was explored. As the code is floating-point operation intensive, a parameter called `Ideal` mode was introduced in the FPU `ENTITY`. If the `Ideal` mode is selected, all floating-point load and store instructions assume an L1 cache hit. As a result, no performance loss is incurred from load and store misses in the code execution.

Figure 4.16 shows percentage of theoretical peak performance achieved by QCD test routines on the model with the smallest local volume (2x2x2x2 sites per node) and with the default QCDOC machine configurations. Peak performance of a QCDOC processing node is 1 GFlops; with two floating-point execution units operating at a frequency of 500 MHz.

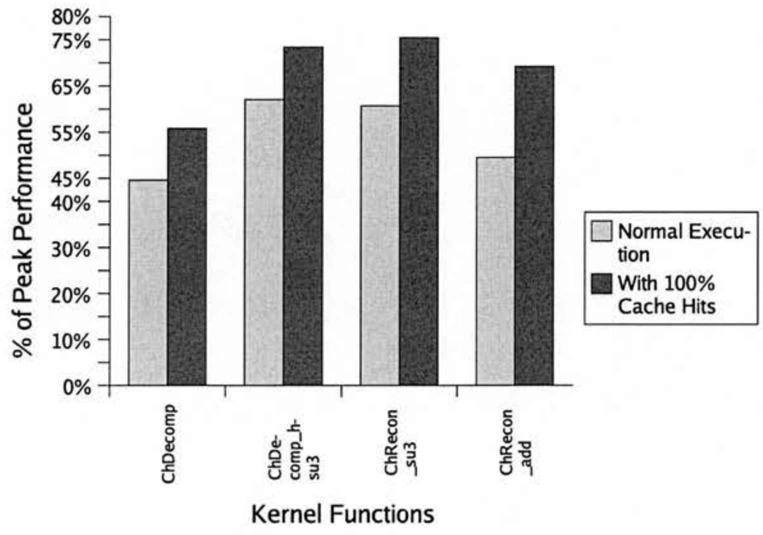


Figure 4.16: Maximum achievable performance of kernel subroutines

With 100% Level 1 cache hits, the performance loss is solely due to the instruction mix of the test code. In other words, it can be assumed to be the upper bound on performance. The FPU `ENTITY` has two execution units; the percentage of peak performance is therefore calculated by dividing the number of instructions executed by twice the total execution time.

```
              Floating-Point Instructions Executed
%performance = ---------------------------------- * 100
              2 *  Execution Cycles
```

The four subroutines shown in figure 4.16 are executed once during each iteration; there are four such iterations per QCD kernel. Thus, potentially, on average, a 14% performance gain is possible in computation subroutines provided the execution pipelines processor configuration remain the same. This gain can be achieved by minimising L1 cache misses or maxmising cache hits.

4.3.3 Memory Hierarchy Design Space

The custom SOC memory hierarchy was explored by varying the on-chip memory latencies and bandwidths. These values were controlled via the parameters in the MMU `ENTITY`, the PDB `ENTITY` and the EDRAM `ENTITY`. The default configuration of memory modules is as follows:

- L1 Data Cache: 32 KBytes, 64-way set associative, 32 bytes (8 words) cache lines. Out of 64 ways, 2 ways are configured as 'transient' for communication data and the rest are set as 'normal'. A TLB storage attribute specifies whether a cacheable page belongs to a transient way or to a normal way.
- L2 Prefetch/Buffer Cache: 4 x 1024-bit read registers, 2 x 1024-bit write registers. Able to return 128-bit data in case of an L2 read hit at the processor clock speed.
- L3 On-chip Memory: 4 MByte EDRAM, multiport, with a separate address space. Data path width to L2 is 512 bytes; it operates at half of the processor clock speed.

Although the size of each level of memory is a parameter, i.e. it can be altered, the results presented in the following sections do not change L1 cache and on-chip memory sizes for experiments. This is because in practice, changing the size of memory modules on an SOC has many implications, including power and chip physical size requirements. The run time for each experiment is measured in clock cycles, where the PowerPC clock cycle rate is 500 MHz.

Table 4.4 and table 4.5 list the L1 and L2 cache parameter values in the HASE QCDOC model that are altered for simulation experiments.

L1 Line Size	128 bits	256 bits	512 bits	1024 bits
Number of sets	64	32	16	8
L1 Data Bus Width	128 bits	256 bits	512 bits	1024 bits

Table 4.4: L1 cache parameter values

L2 Line Size	256 bits	512 bits	1024 bits	2048 bits
Prefetch Registers	2	4	8	16
Buffer Registers	1	2	4	8
L2 Bus Width	128 bits	256 bits	512 bits	1024 bits
Replacement Policy	LRU	Random	Round Robin	

Table 4.5: L2 cache parameter values

Combinations of the L1 and L2 caches' parameters in table 4.4 and table 4.5 are explored in the simulation experiments. Operations that access these caches as well as the EDRAM are the load and store operations and the "data cache block touch" (`dcbt`) instruction and prefetch operations initiated in the L2 cache. All floating-point operations including load-store are performed on 64-bit values for double precision QCD calculations, while integer load/store are 32 bits.

4.3.3.1 L1 and L2 Cache Line Sizes

The L1 cache line size can be altered by changing the number of sets or the number of cache lines per set (keeping the overall cache size constant). Prefetch read register sizes, the L2 cache line size, can be altered by increasing or decreasing the number and size of prefetch read registers and write buffer registers, such that the L2 read register size remains constant at 4-Kbit. In exploring the effect of L1 and L2 line sizes, the bandwidth and latency of the three levels of on-chip memories were kept constant.

Figure 4.17 shows the floating-point unit performance for the optimised QCD routines with varying L1 and L2 line sizes.

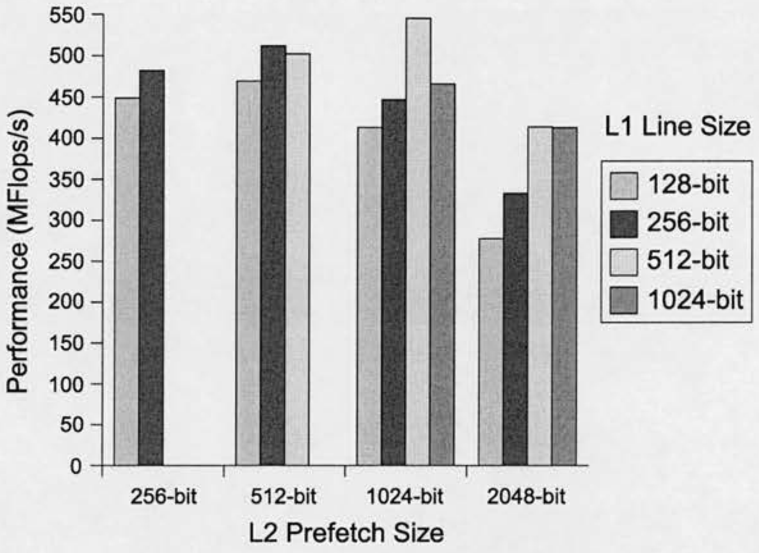


Figure 4.17: Performance variations as a result of L1 cache line size and L2 cache prefetch size

With a small prefetch size, 256-bit and 512-bit, the effect of increasing L1 line size is small. A large prefetch size, 1024-bit and 2048-bit, means fewer prefetch registers in the L2 cache. These results demonstrate that the QCD kernel code does exploit the spatial locality offered by a large L1 and L2 cache lines, but needs frequent memory accesses beyond the prefetch boundary lines. The best performance is achieved with the default L2 configuration and a 512-bit L1 cache line (default is 256-bit). This confirms that a large L2 line is essential for the QCD computation and the a relatively large L1 line size can enhance the achievable performance.

The results in figure 4.17 show that floating-point performance depends on the L1 line size. For a small L1 line size, for instance, 128-bit line and 256-bit line, the execution times are high. At the same time, a very large L1 line does not always give a better performance. There are two reasons for this behaviour: firstly, with a large L1 line size, the load latencies are high as the processor has 128-bit read lines. The second reason is linked to the write latencies. A large L1 line takes longer to be written back to L2 and it needs more memory for write flush buffers. Performance is best when L1 and L2 sizes differ by a factor of 1 or 2. A small L2 line compared to large L1 line is not useful because for each L1 line fill, a large number of L2 lines (prefetch read registers) have to be invalidated and prefetched. Moreover, a small L2 compared to the L3 bus width will not utilise the available bandwidth to L3 and possibly result in a large number of small prefetch read registers. Due to the LRU replacement policy and the coherency protocol requirements, a large number of prefetch registers would add to the complexity of the L2 cache design, which must run at the processor speed.

The performance results also show the affect of L2 line sizes. The size of prefetch register increases at the cost of the number of available associative prefetch read registers. For example, for the default L2 configuration of four 1024-bit read registers, a read request from the L1 cache brings a whole 1024-bit line from the L3 cache into L2. With four available registers, there can be 4 memory sections present in the L2 cache. If the configuration changes to 2 x 2048 registers, an L2 miss brings a whole 2048-bit line into the L2 cache. However, in this instance, it can only access two memory regions and with an LRU replacement policy, this configuration does not perform well for scattered data and data beyond the 2048-bit boundaries.

Finally, comparing the effect of L1 line size and L2 prefetch line size, it was found that the former has the greater impact on performance. Similar results were obtained by Bernard *et. al.* [BDG$^+$95]; they explored QCD code performance on POWER processor based IBM SP2 supercomputers. They suggested that the performance of QCD application code depends on the width of the primary cache line. From the simulation experiments on the QCDOC model, it was found that a large L1 cache line does not (always) help in performance improvement on the QCDOC SOC design. This is because a large L1 cache line will result in transfers of a large numbers of read data blocks from L2 cache and L3 cache, which may cause a number of write flushes across L3 write ports at a slower clock rate. Particularly, for the store intensive code, a large L1 line may not bring extra benefits as stores have to be flushed immediately to the L3 cache due to the incoming read requests.

4.3.3.2 Bus Widths

In a pipelined load-store processor, maximum processing power can be achieved only with zero latency and an infinite bandwidth between levels in the memory hierarchy. A wide 512-bit on-chip data bus, operating at half of the processor clock speed cannot achieve the optimal target, but it can assist in lowering access latencies and keeping up the bandwidth. Wide on-chip memory buses that can move sequential words between memory levels can increase the memory bandwidth and, by being closer to the processor clock rate, can reduce the latencies associated with a cache line fill. There is a cost associated with wide on-chip buses however, in terms of on-chip area and power consumption. In order to establish the effect of data bus widths (128-bit between MMU and FPU/CPU and 512-bit between PDB and EDRAM in the QCDOC ASIC), experiments were performed with different bus widths in the model. For these experiments, the L1 and L2 line sizes were fixed at their default values: 256-bit and 1024-bit respectively.

The effect on the floating-point execution unit performance with respect to changes in the L1-L2 and L2-L3 access bus widths, and consequently bandwidths, is shown in figure 4.18.

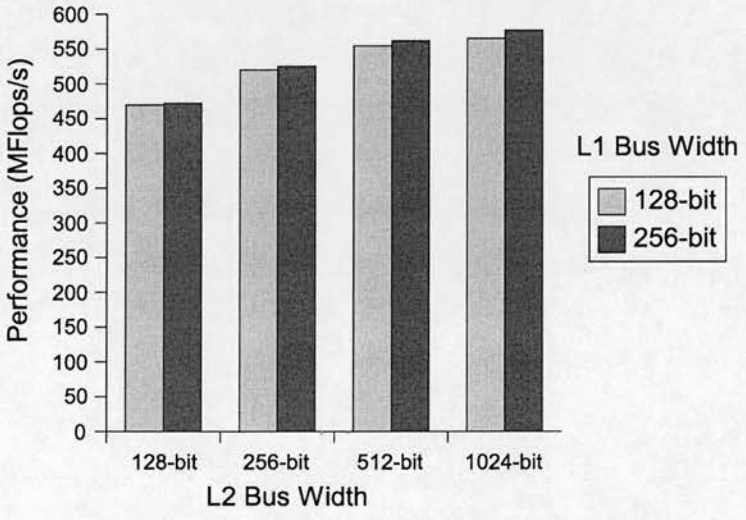


Figure 4.18: Effect of L1–L2 and L2–L3 cache bus widths

For the first two levels of memory hierarchy on the QCDOC ASIC, the clock rate is the same as that of the processor clock. Therefore, potential increases in the bandwidth were effected by changes to the bus width parameters. An increase in L2 bus width has a steady effect on the performance, while an increase in L1 bus width only has a marginal effect. A quick L2 line fill, with a wide data bus between L2 and L3 caches, results in a quick response to successive L1 line fill requests. Moreover, the latency of L2 cache accesses to L3 is higher as compared to L1 cache to L2 cache accesses, and reducing the former latency has a greater impact on the performance than the latter does.

4.3.3.3 Data Cache Configurations

Although the QCDOC operating system assigns most memory regions as cacheable, the data received from the neighbouring nodes are kept in the non-cacheable pages. Communication data to be sent to the neighbouring processors are kept in the *transient* pages; PowerPC offers special instructions to flush these pages. Information about the classification of memory pages as normal, cache-inhibited and transient pages is stored in the TLB, which in turn relies on the distribution of cache lines into normal, transient and locked regions. The data cache configuration is critical to processor performance, particularly the data cache line sizes and their access latencies. As identified earlier, in the QCD kernel code, a series of load instructions is followed by an even larger sequence of floating-point arithmetic operations, which in turn is followed by store operations. Floating-point data is double-precision, 64-bit data, and a default cache line can store up to two floating-point values.

Theoretically, QCD data structures with a high degree of spatial locality, should benefit from large cache line sizes. However, the disadvantage of a large L1 line size can restrict the overall performance gains. The potential disadvantages include loading of large segments of unused data and reduced set associativity for constant cache size. Using data cache prefetch instructions, this problem can be increased many-fold. The overall effects of varying the cache line and data read and write bus widths are shown in figure 4.19.

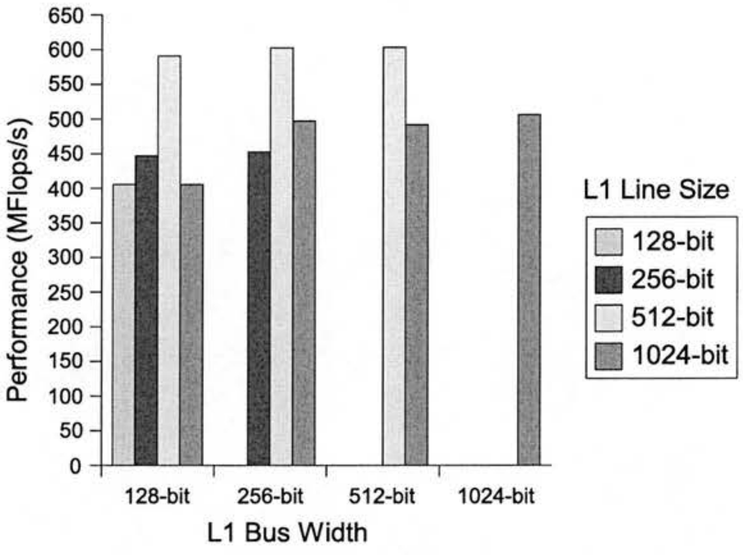


Figure 4.19: Performance variation due to L1 cache configurations

The results demonstrate that a wide L1 cache line, 512 bits (8 floating-point words), is best suited for the QCD application code. Large cache lines increase spatial locality at the cost of reduced associativity, reducing the cache read hit rate and causing frequent line replacements. Moreover, a large cache line with a relatively small read and write bus width, results in an increase in bus traffic. At the same time, the effect on the outstanding load and store buffers of a very large cache line is to reduce the effectiveness of the spatial locality they offer. Another observation is that the bus width has very little effect on performance. A more efficient line fill can only make a difference if load requests to very large memory blocks (a few tens to hundreds of cache lines) are initiated within a short interval, which is not the case with the QCD calculations. Adjacent memory words are needed at regular intervals in the calculations.

4.3.3.4 Cache Prefetch Instructions

The cache touch (prefetch) instruction in the PowerPC ISA plays a significant role in improving the performance of load-store operations. For the results presented in section 4.3.3.1 and section 4.3.3.3 with varying L1 line sizes, the cache touch (`dcbt`) instructions were not executed as part of the hand-coded assembler kernels. These optimised assembler kernels were written for 32-byte cache lines. Figure 4.20 shows the benefits of using cache prefetch instructions for the default QCDOC machine configuration.

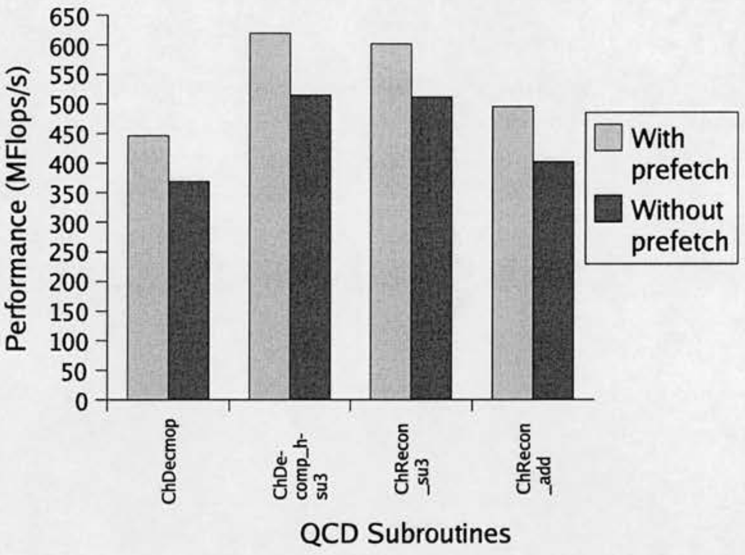


Figure 4.20: `dcbt` — cache touch instructions

A significant performance gain can be achieved by using the prefetch instructions: the QCD memory access pattern does not benefit from standard cache optimisation techniques that rely on the locality of data. The optimised QCD assembler kernels therefore make frequent use of cache touch instructions.

For portability of these assembler kernels across systems with different cache line size, the Motorola AltiVec [Alt] technology Data Stream Touch instructions (`dst`) would be quite useful, in particular, when a series of prefetches needs to be performed with fixed block strides. The AltiVec `dst` permits a program to indicate that a sequence of units of memory (described by an effective address, size of words, number of units and a fixed stride) are likely to be accessed soon by memory access instructions. Although the `dst` instruction has a benefit over the `dcbt` instruction, `dst` instructions must be used carefully to avoid poor utilisation of the available memory bandwidth due to increased memory traffic.

4.3.4 Prefetch Engine Configurations

A set of experiments was performed by varying the prefetch size with three L2 sizes: 4K-bit, 8K-bit and 16-Kbit. For a fixed L2 cache size, increasing the prefetch size reduces the total number of prefetch registers, thereby limiting access to discrete regions of memory. A small prefetch size allows more registers in the L2 cache, potentially permitting access to discrete memory regions. Figure 4.21 shows results of experiments performed using the QCD test code with varying prefetch sizes.

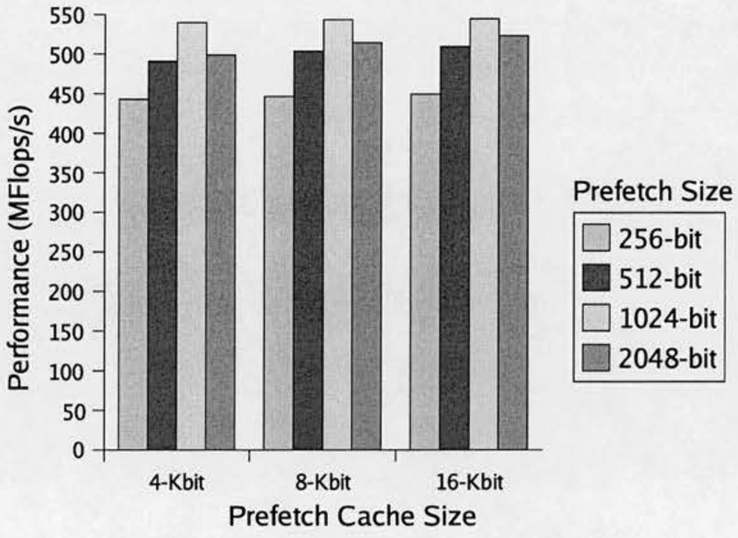


Figure 4.21: L2 prefetch cache configurations

With no prefetch, *i.e.*, the L2 fetch size is equal to the L1 line size, an L1 read miss has a minimum latency of 6 clock cycles compared to a 1-2 clock cycle latency when the data is available in L2. It was found through HASE QCDOC simulation experiments that a very large prefetch size does not improve performance because of frequent invalidations of a small number of L2 prefetch registers. Similarly, overall L2 size has very little impact on code performance. The main advantage of a large L2 is infrequent prefetch register invalidations; incoming data invalidates the existing data less frequently because of the availability of a larger pool of registers.

A second set of experiments was performed by varying the relative clock frequency of L2 with respect to the processor clock frequency. In QCDOC, the small, on-chip L2 cache operates at the processor clock frequency. In practice, if larger L2 sizes were to be used, a lower frequency would be inevitable. For instance, the Bluegene/L has a 32-KByte L2 which operates at half the processor clock frequency [Tea02]. Figure 4.22 shows the importance of having a low latency L2 cache when running QCD code. As in the prefetch size experiments, a bigger L2 size has very little impact on QCD code performance.

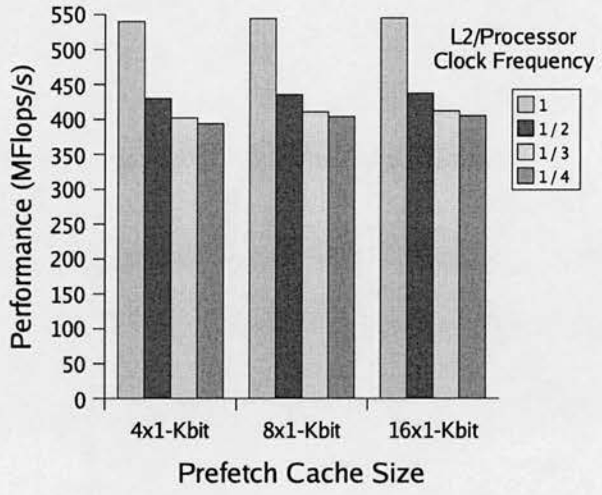


Figure 4.22: L2 prefetch cache access latency

Finally, the replacement policy for the prefetch read registers was evaluated. The default replacement policy is LRU for 4x1024-bit prefetch read registers. This LRU policy is compared with round robin and random replacement policies.

In figure 4.23, the number of prefetch registers is increased while the prefetch size is kept constant (1024-bit). The results shows that the QCD load request pattern, especially when a large number of prefetch registers are involved, does not significantly benefit from the LRU policy. Particularly, the round robin policy has an effect similar to the LRU in all performance experiments. Likewise, increasing the prefetch size with four prefetch read registers has very little effect across the different replacement policies, as shown in figure 4.24.

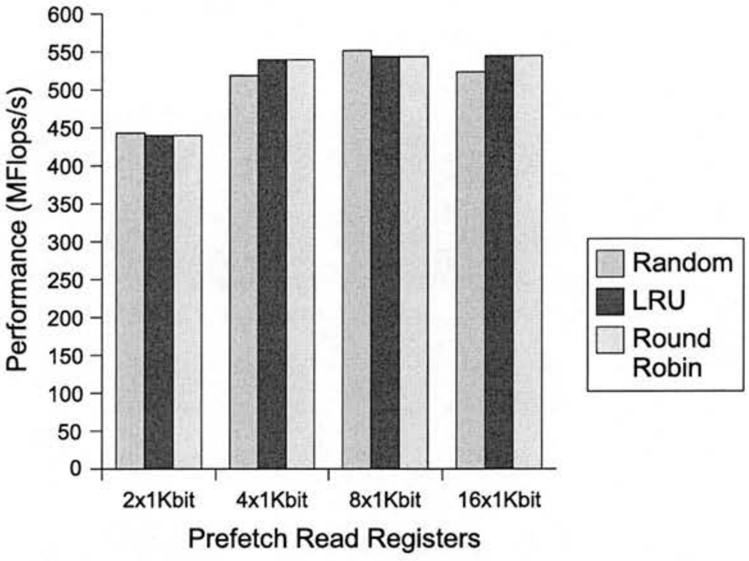


Figure 4.23: Prefetch read registers replacement policies (L2 Cache) — with constant prefetch size

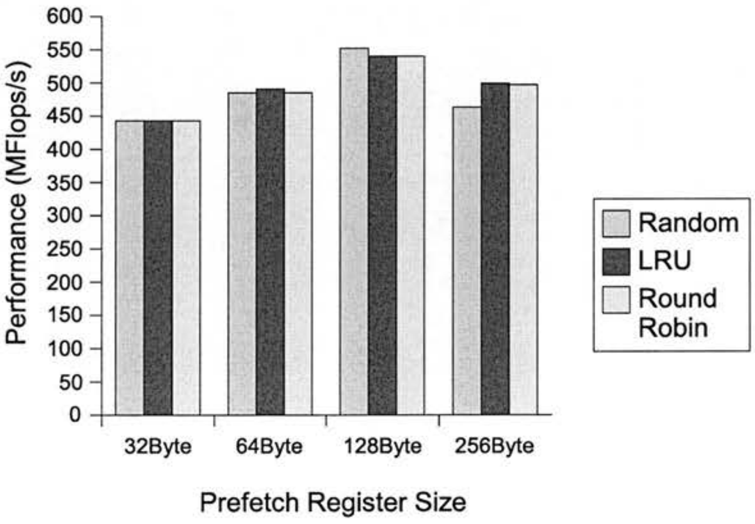


Figure 4.24: Prefetch read registers replacement policies (L2 Cache) — with fix number of registers

In the case of a very large sequential line, 128 Bytes, the random replacement policy replaces lines that may still be used, while the LRU and the round robin policy keep a line which is sequentially transferred to the L1 cache. Overall, QCD code performance is not improved by employing an LRU policy with the default L2 cache configuration. A round robin policy, which is far simpler to implement in hardware, can have similar effect on the QCD code performance.

In summary, the prefetch engine configuration experiments demonstrate that the QCD kernel, like a number of other compute- and data-intensive scientific kernels, gains little from conventional data cache locality enhancements. Therefore, a prefetch engine assisting low latency and high bandwidth L1 line fills improves code performance. The spatial locality of the QCD code takes advantage of the prefetch engine which is capable of returning data within 1-2 clock cycles. Increasing the size of L2 cache does not compensate for increased L2 to L1 data fetch latency. Typically, a QCD subroutine begins with a series of 64-bit floating-point data fetch (load) instructions. By prefetching a number of cache lines in advance, subsequent cache requests can be served relatively quickly. Similarly, for a small prefetch size, back-to-back L1 read misses follow L2 read misses. For a very large L2 prefetch, the data coherency protocol and a small number of available registers induce frequent invalidations of the prefetch read registers.

4.4 Custom Interconnection Network Performance

The custom-designed communication unit SCU in the QCDOC ASIC supports the QCD-specific communication patterns. The SCU ensures, with minimum CPU intervention, concurrent bi-directional communication of data packets in the four-dimensional torus network. The communication unit contains buffers and registers associated with each send and receive port. Communication data integrity and efficient performance are governed by a custom-defined network protocol [BCC+02a]. Parameters of the SCU `ENTITY` and the computation-communication overlap characteristics of the parallel QCD code have been explored in the simulation experiments.

4.4.1 On-chip and Off-chip Communication Latencies

Table 4.1 lists off-node communication time for a 64-bit data transfer over the three-dimensional torus network between nearest neighbours in a given dimension. 138 processor clock cycles in total are required for sending 64 bits over a serial link including the acknowledgement receipt time. By comparison, transferring on-chip 128-bit communication data between the EDRAM and the communication unit (SCU) takes about 10 clock cycles over the system bus, PLB (operating in burst transfer mode). The PLB is not dedicated to the EDRAM and SCU data transfers; it is shared by a number of devices, including the off-chip memory SDRAM controller. Likewise, in a non-ideal situation, off-node communications suffer from network errors: communication packet loss and/or corruption. In order to establish the effect on performance of on-chip and off-chip communication latencies, experiments were performed by varying the on-chip and off-chip data transfer latencies. These latencies are increased by 10 clock cycles for each set of experiments. Figure 4.25 shows the results of these experiments.

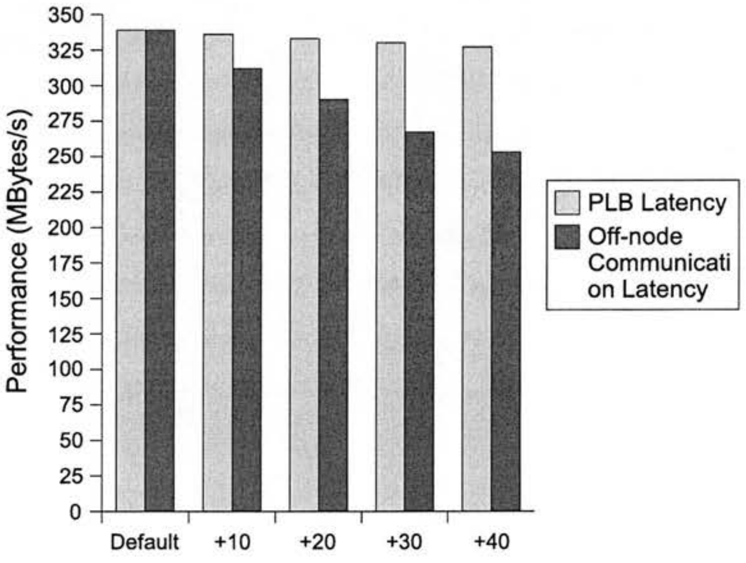


Figure 4.25: Effect of on-chip and off-chip communication latencies

PLB (on-chip) transfer latencies do not affect performance at all. This is because in 10 clock cycles two 64-bit packets are delivered to the SCU send buffers, which in turn takes more than 200 clock cycles to be delivered to the neighbouring nodes' receive buffers. An increase in the off-node communication latency on the other hand has a major impact on performance, as shown in figure 4.25. Most main-stream interconnection networks have a sophisticated switching and routing scheme. These benefit large packet transfers but have initial startup latencies. QCD communication packets, on a fine-grain system like QCDOC, are much smaller in size and their communication performance significantly suffer from large startup latencies. That is why QCDOC has a switchless, direct communication network and a special, low overhead communication protocol.

4.4.2 Local Computation and Communication Load Balance

In parallel QCD code execution, performance is enhanced by overlapping the local floating-point code execution with the nearest-neighbour communication along the four dimensions. QCD code is considered as compute-intensive, i.e., more time is spent in execution pipelines than off-node communications. In order to observe this phenomenon, experiments were conducted with varying problem size on a QCD node. These experiments do not take into account any kind of communication errors: lost or corrupted packets. In the presence of network errors, communication times may exceed computation times, as the off-chip transmit latencies are far bigger than the on-chip latencies. The effects of unreliable links on the communication times are presented in section 4.5.

Figure 4.26 shows the scaling behaviour of computation and nearest-neighbour communication functions of parallel QCD workload.

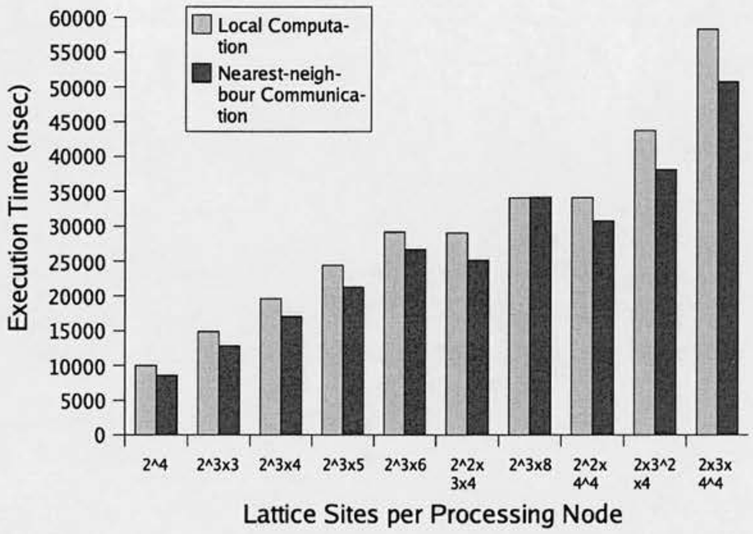


Figure 4.26: Computation and local communication scaling

Although these results confirm that the QCD kernel code execution is dominated by the computation routine execution, an interesting observation was made. When the numbers of sites per node are evenly distributed per lattice site per lattice dimension, for example, 2x2x2x2 lattice sites per node, the computation time far exceeds the nearest-neighbour communication time. On the other hand, if the number of sites mapped onto a processing node varies in the number of sites in a dimension the communication volume becomes unbalanced. This results in an unbalanced and significantly high communication requirement along the neighbouring nodes, for example, with 2x2x2x8 sites per node, where the communication volume along three dimensions is larger than the fourth one. Thus, a mapping of 2x2x4x4=64 sites per node is more efficient in the nearest-neighbour communication times than a 2x2x2x8=64 sites per node. The QCD code, once the overlapping computation part is finished, keeps on polling for the nearest-neighbouring data to arrive before the next computation and communication step. If, for some reasons, the nearest-neighbour communication in a single direction out of the possible four space-time directions does not complete, it can have an adverse effect on the overall performance of the code.

4.5 Large Machine Configuration Modelling

Characteristics features of parallel QCD kernel calculations include the intensive, double-precision floating-point computation, nearest-neighbour communications and global sums. The distinguishing feature of global communications, also known as collective communications, is that all the processors make the call to synchronise the communication routines, whereas in the case of a point-to-point communication, the sending and the receiving neighbouring pairs communicate independently of other inter-node communication operations. Precisely for this reason, a small machine cannot establish the global sum performance; a full-size HASE model is necessary to run these simulation experiments. Single node performance experiments and local communication protocol tests have been run on a 16-node HASE QCDOC model. A 16-node model allows execution of SPMD style programs and four-dimensional nearest neighbour communications. QCDOC will achieve the 10-Teraflop peak performance with over 10K processing nodes operating at a 400-500 MHz clock. The HASE QCDOC model was expected to scale to up to 10K processing nodes. This target proved to be infeasible with the current Linux operating system limits.

The Hase++ library employs multithreading to exhibit parallel execution of HASE entities and among several instances of an entity. The maximum thread limit of the host operating system Linux at the time of designing and implementing the simulation models was 7168 (/proc/sys/kernel/thread-max).[6] Additionally, the simulation engine, Hase++, enforces mutual exclusion zones for the execution of entities' threads. Hence, in the presence of very large number of threads, the simulation run-time in terms of wall clock time is extremely slow on a uniprocessor host machine. A single HASE QCDOC processing node requires several threads to be in execution concurrently. Furthermore, the required memory overhead per processing node for a 10K processing node on the host machine was considered impractical. Hence, the first design decision was the removal of all memory ARRAYs from the HASE QCDOC entities. This did not solve the other crucial problem — the thread limitation of the host.[7] It was therefore

[6]Linux kernel versions 2.4 and above allows the maximum thread limit to be altered at run time.

[7]A minimum of 10K threads are required if the HASE MESH4D template generates a full scale QCDOC simulation.

decided that the behaviour of multiple processing nodes should be simulated within a single HASE entity, or a high level abstraction of the HASE QCDOC processing node. Furthermore, the QCDOC four-dimensional torus topology is mapped onto a conventional four-dimensional array construct to simulate a 10K processing node system. This process involved extending the current SCU `ENTITY` (no other entities were effected because their behaviour was represented as a trace). The SCU `ENTITY` is the only entity in the model that has off-node links. The parameters added to the SCU `ENTITY` are presented in table 4.6.

Parameter	Description
size_4dim	Number of nodes in each dimension of a four dimensional physics network
glbsum_policy	This parameter determines the scheme used to add the double-precision floating point value over the inter-node communication network (section 4.5.1)
error_mode	If set to 1, a random probability is applied over the communication channel to test performance of the nearest-neighbour communications

Table 4.6: Additional parameters in the SCU `ENTITY`

By creating a high-level abstraction model and incorporating the parameters shown in table 4.6 additional sets of experiments are possible with a HASE QCDOC simulation model containing a large number of processing nodes. The extended SCU scheme allowed two experiments in particular to be conducted on the model: one was the custom-protocol evaluation in presence of unreliable links and the other the global sum performance over a large machine. The global sum (collective communication) operations are implemented in the QCDOC and other high ends systems using a variety of techniques.

4.5.1 Global Sums Schemes

In order to avoid rounding errors of floating-point operations carried out on 10K processing nodes, the global floating-point addition has to be performed in an identical order across all nodes. The QCDOC SCU contains a pass-through unit that not only ensures that these sums are performed in an identical order but also maximises the efficiency of these collective communication operations. The HASE QCDOC model simulates complete functionality of the pass-through unit over the torus network. In addition, it simulates a binary tree network, thereby allowing a user to investigate the impact of several global sum schemes over the two networks on the QCD collective communication operations.

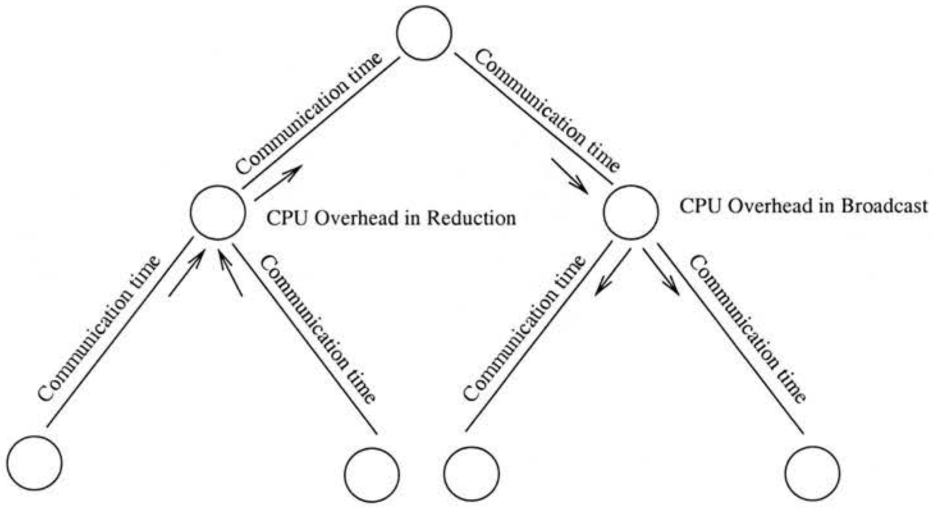


Figure 4.27: A binary tree network

In a binary tree network (figure 4.27), a global sum can be performed in one round-trip to the tree:

1. Reduction: Each leaf node sends its double-word value to the parent node. The parent node collects the double-word values from the left and right nodes and the FPU adds the values with the local data. The result is loaded into the send register of its parent node. This process is repeated until the root node contains the sum of its left and right nodes' values and its own value.

2. Broadcast: The second step is the broadcast of the floating-point sum value to all nodes. This step may involve CPU intervention if re-direction is not made automatic. Each node sends the value it receives from its parent to its left and right nodes (except the leaf nodes).

Three latencies are involved in a tree network sum: communication latency, computation of floating-point sum and CPU overhead in re-direction. The overall complexity of a global operation on a tree network is a function of the height of the tree network (log_2(N)).

QCDOC global sum communications do not follow the same bi-directional custom protocol as local communications: a sent packet is not acknowledged. It is reported[8] that unreliable links are more likely to be identified in the frequent nearest-neighbour communications. Moreover, the authenticity of the global sum results, indeed the whole QCD calculation, is established by rotating lattice points on processing nodes. In practice, a QCD calculation with a given lattice configuration is performed several times to validate and verify simulation results.

On the simulation model, it was considered useful to quantify the overheads with and without the nearest-neighbour protocol latencies. The following sections provide details of the four global sum options in the HASE QCDOC model.

4.5.1.1 Shift and Add

This mechanism follows the default global four-dimensional torus communication pattern,[9] i.e., it sums along one direction at a time in (N-1)/2 steps, then sums the nodes and communicates data in the next direction. This process is repeated for the four directions. The simple shift and add mechanism assumes that two CPU overheads exist: the pass-through overhead and direction change overhead.

[8] A. D. Kennedy (2003) UKQCD collaboration, private communication.

[9] Explained in section 4.1.3.1

4.5.1.2 Enhanced Shift and Add

This mechanism is quite similar to the shift and add in terms of the pattern of communication. The main differences are in the CPU overhead and communication latencies. It assumes that the pass-through process is automatic and only a direction change overhead is involved. For the communication, incoming data can be redirected before a successful double-word transfer is complete. Enhanced shift and add performs far better than simple shift and add because of low CPU overheads, but with added hardware complexity.

By making the pass-through mechanism automatic and allowing the incoming bits to be re-directed without waiting for the whole double-word packet, the message transfers and redirection times are significantly reduced, as shown in figure 4.28.

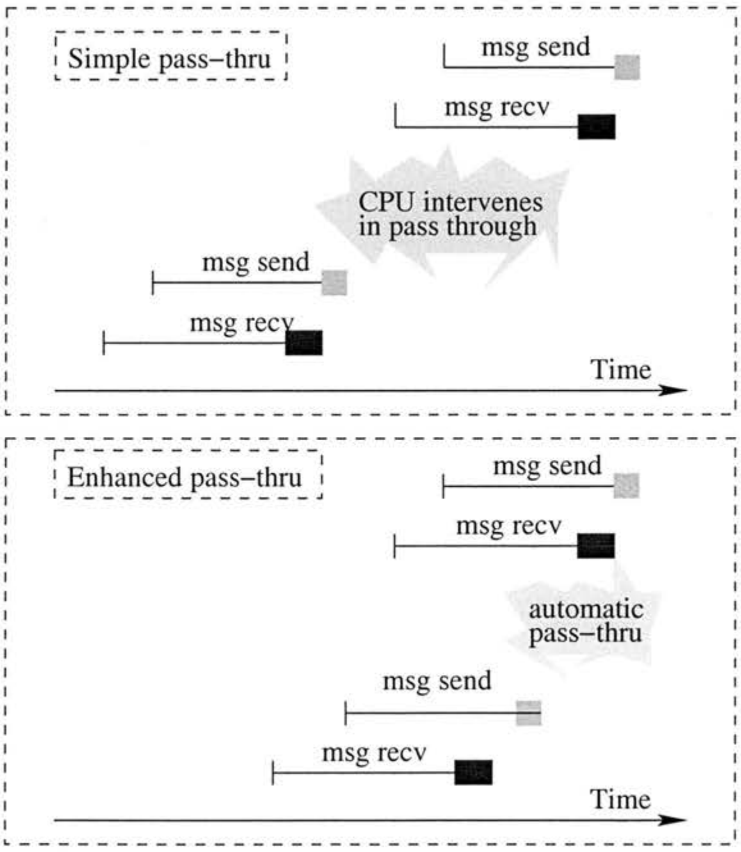


Figure 4.28: Simple and enhanced shift-and-add pass-through global sums

In the simple pass-through mode, since CPU intervention is needed for data forwarding to the sender, the double-word must be received first. When data forwarding is made automatic, as in the Enhanced pass-thru mode, the incoming data can be forwarded to the sender port without waiting for the whole word. This significantly reduces the message transfer times by overlapping incoming and outgoing message transfers.

4.5.1.3 QCDSP-style Global Sums

The communication unit in the QCDSP (predecessor of QCDOC) machine contains a bit adder in addition to a pass-through unit and allows hardware-controlled communication and bit manipulation between a subset of incoming and outgoing ports. This flexibility permits a tree network to be formed from a periodic four-dimensional mesh where an assigned node can behave as a tree node, sending and receiving from a subset of connected nodes as shown in figure 4.29.

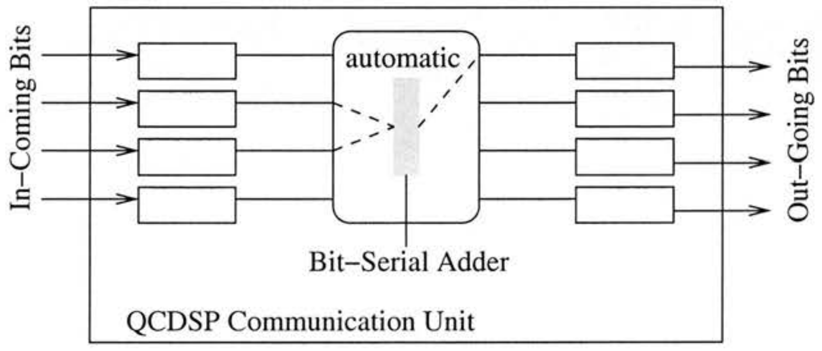


Figure 4.29: A QCDSP SCU pass-through unit with a serial adder

A root node can be designated as a source of broadcasts and a destination for reduction operations. The QCDOC binary tree exchanges double-word values and requires double-precision floating-point addition. In other words, CPU intervention occurs at each step of the reduction operation. It can however be replaced by arbitrary-sized packet transfers (exponent and mantissa) which involve two round-trips of the tree network:

1. Trip 1:
   (a) Reduction: Find maximum of 11-bit exponents. Result in root node.
   (b) Broadcast: Broadcast maximum exponent to leaf nodes.
2. Trip 2: Each node normalises its 52-bit mantissa according to max exponent.
   (a) Reduction: Integer sum of 52-bit mantissae. Result in root node.
   (b) Broadcast: Root broadcasts the sum of the mantissae.

Since this option assumes no data manipulation in an SCU, a CPU overhead is included in each step of the reduction operations. Another component of the CPU overhead will be added if re-direction is not automatic in broadcast operations.

4.5.1.4 Binary Tree QCDSP-style with a Serial Adder

The previous option (tree plus QCDSP-style) identifies the importance of a QCDSP serial adder in a tree network. A serial adder with a capability of operating on incoming bits as well as locally stored data eliminates the CPU overhead in reduction operations. Assuming the redirection overhead can be eliminated or controlled in hardware, the resulting global sum is extremely efficient. This option involves two round-trips of the tree but with considerably less CPU overhead compared to the above two tree network options.

Figure 4.30 shows the pass-through mechanism operating between three processing nodes.

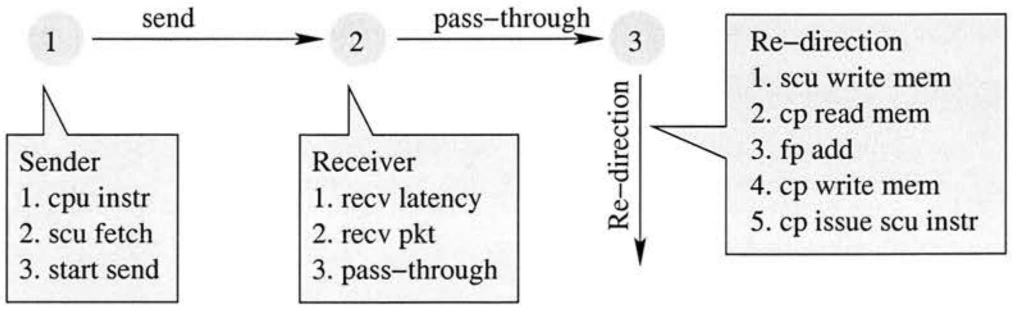


Figure 4.30: Steps for computing global sums

Firstly, a sender, node 1, sends a data packet to its nearest neighbour, node 2. This involves:

1. CPU writes instructions to SCU SRAM (DMA) to initiate a send.
2. SCU fetches data element from the on-chip memory via PLB.
3. SCU starts sending; this incurs a send overhead plus message transfer costs.

When the CPU instructs the sender node to send a data packet, it simultaneously instructs the receiver (a sender itself is also a receiver for some other node) that the receiver is expecting a packet. At the receiver side this involves:

1. Startup time: latency associated with a packet receive.
2. Transfer time: time to receive a complete packet.
3. Pass-through time: time to forward receive data to the designated sender port.

Once (N/2) sends and receives complete in one direction, the communication has to be re-directed along a new axis. This involves first sending the collected data to the on-chip memory. The collected data must be tagged i.e., each packet has a pointer so that the sum will be performed in an identical order across all processing nodes on an axis. This mechanism ensure no rounding errors. In the re-direction process:

1. SCU writes collected data to the memory.
2. CPU reads data to L1 cache.
3. FPU adds the collected double values.
4. CPU writes sum to the memory.
5. CPU gets ready to send next pair of instructions.

Analytically, the total communication cost, ignoring on-chip data transfers and manipulation overheads, for a four-dimensional torus network is:

$$T_{comm} = 2 * (N_x + N_y + N_z + N_t)$$

and for a tree network is

$$T_{comm} = 2 * log_2(N_x * N_y * N_z * N_t)$$

where N_x, N_y, N_z and N_t are the number of nodes in x, y, z and t directions respectively. Communication costs place an upper bound on overall global sum costs. In particular, when CPU overheads are low, the scheme with a small number of sends and receives performs better. With simultaneous, bi-directional communication of the QCDOC network, the task is to minimise the CPU and on-chip bit transfer and manipulation overheads.

4.5.2 Global Sum Performance

Table 4.7 lists the configurations used for the CG global sum experiments.

Option	Mechanism
1	Simple Shift and Add
2	Hardware Enhanced Shift and Add
3	Hardware Enhanced Shift and Add with Nearest-Neighbour Protocol
4	QCDSP-style global sum (MAX-BCAST-ADD-BCAST) with bit-serial adder in SCU

Table 4.7: Global sum mechanisms — for 4D torus and binary tree networks

Each global sum mechanism is implemented in both the torus network and the binary tree network topology. Machine dimensions vary from 2^4 processing nodes to 12^4 processing nodes. Figure 4.31 shows the results of the global sum simulations.

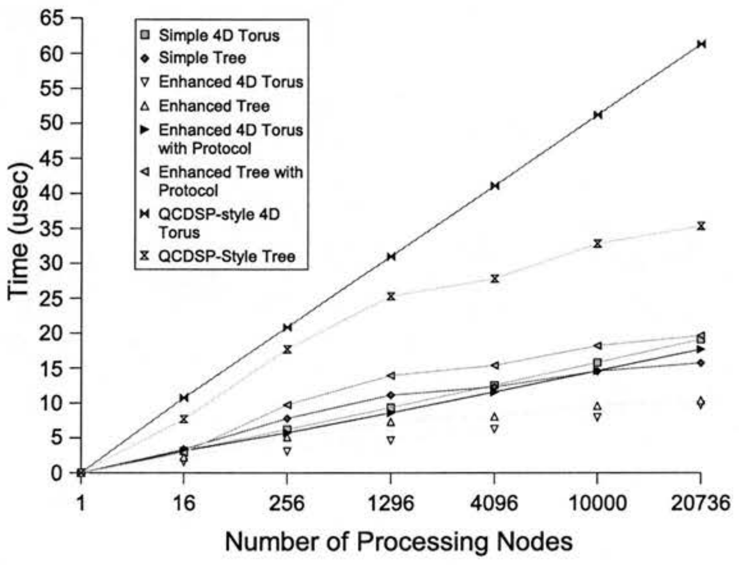
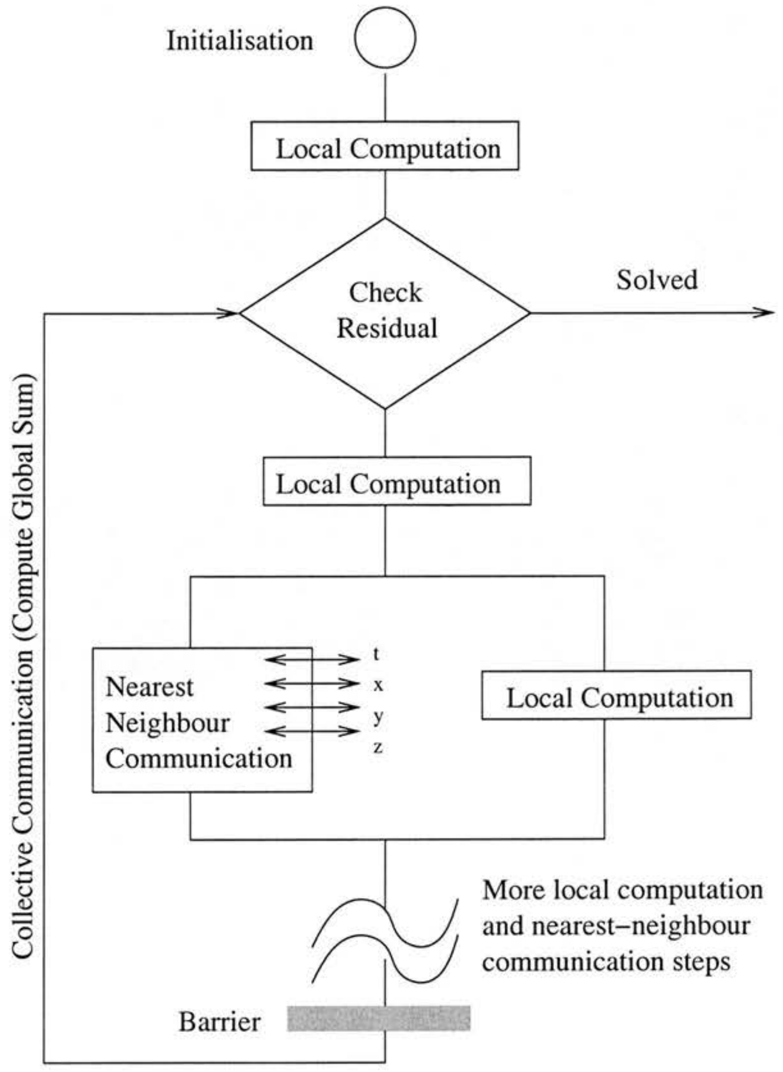


Figure 4.31: Global sum performance

When the system is configured as a hardware enhanced torus, the most efficient results are obtained. The results are worst when the system is configured as a QCDSP-style four-dimensional torus. In all other modes, except for the QCDSP-style mode, the torus network has fewer communications than the binary tree networks. That is why, when the CPU overheads are low, the torus network outperforms the binary tree network. On the other hand, the tree network outperforms the torus network for QCDSP-style global sum mechanisms.

4.5.3 Communication Protocol Efficiency

Figure 4.32 shows key steps of a QCD CG solver.



Initialisation
Local Computation
Check Residual
Solved
Local Computation
Nearest Neighbour Communication
t
x
y
z
Local Computation
More local computation and nearest–neighbour communication steps
Barrier
Collective Communication (Compute Global Sum)

Figure 4.32: Conjugate Gradient (CG) solver

The QCD CG algorithm permits local computations to be overlapped with nearest-neighbour (NN) communications. Although these NN communications involve only the adjacent nodes in each of the four directions, in some cases these communication steps are explicitly synchronised by a barrier synchronisation mechanism, as implemented on the Cray T3E by Berry *et. al.* [BGK94].

In the absence of an explicit synchronisation step immediately after each NN communication, a global reduction (a collective communication) operation that determines the iteration condition variable implicitly requires local computation and communication steps to synchronise before the global reduction operation proceeds, as shown in figure 4.32. Thus poor performance by a single node can potentially affect the overall performance of a CG application.

Generally, the performance metrics reported for interconnection networks are the peak throughput and average latency, according to Chien and Konstantinidou [CK94]. For deterministic and realistic estimates of off-node communication performance of an application code, not only is the throughput at saturation important, but also the other throughput measures, for example, throughput in the presence of network errors, should be considered. Hence this research has focussed on investigating the effects of the communication buffer size with varying channel reliability and communication volumes. The overall reliability of communication channels is determined by taking a ratio of the number of packets transferred to the number of erroneous messages. Since the error distribution is random, the channel reliability across communication channels varies; some nodes have no packet loss while other may have a higher than average error rate.

Experiments were performed with three different combinations of local volumes. For the lattice QCD calculations, the number of lattice points (sites) determines the local volume. The communication and computation requirements of a QCD code are directly proportional to the local volume (number of sites per node). Three local volumes were used for the experiments: 2^4 sites per node, 2^2 x 4^2 sites per node and 4^4 sites per node. The results are measured in processor clock cycles (assuming a 500MHz clock).

4.5.3.1 Communication Workload Properties

Chien and Konstantinidou [CK94] identified that the space of workloads for parallel interconnects has three dimensions: traffic pattern, message size distribution and temporal distribution. For QCD simulations, the traffic pattern is quite regular, i.e., a node communicates with its nearest neighbours along four (T, X, Y and Z) directions. The number of messages varies with local volume, and local communication takes place in a burst mode: the exchange of a large amount of data at regular intervals.

Agarwal [Aga91] found that the *k*-ary *n*-cube topology favours small message sizes. In the QCDOC network, a double-word (8 bytes) was considered as a basic unit of communication. Small packets sizes are thus stored in double-word receive registers and buffered in double-word send registers. In addition to message size, two key characteristics of a parallel QCD application code are the scaling of communication data with respect to local volume and load balancing. In the case of 2^4 (2x2x2x2 sites per node) and 4^4 (4x4x4x4 sites per node) there is no problem of load balancing as each NN communication involves exchanging the same amount of data. However, with the other distribution, 2^2x4^2 (4x2x2x4 sites per node), there is an imbalance in the NN communications; two communication ports transfer more messages than the other two. Moreover, the communication requirements (size of data transfers) increases four-fold as the local volume increases from 2^4 to 2^2x4^2. For a 4^4 local configuration, nearly eight times more data transfers are required as compared to a 2^4 configuration.

4.5.3.2 Impact of Communication Buffer Sizes

Figure 4.33 shows the average throughput and figure 4.34 shows the percent utilisation of the communication channels for an NN communication for the smallest local volume, figure 4.35 and figure 4.36 show the performance and channel utilisation respectively for the unbalanced workloads. The average throughput and communication link usage are shown in figure 4.37 and figure 4.38 respectively for communication message transfers with 4x4x4x4 sites per processing node. For the experimental results represented in figures 4.33, 4.35, 4.37 and figures 4.34, 4.36, 4.38, channel error rates vary from an ideal case with no packet loss to a situation where on average 20% of packets are lost. Furthermore, the size of the communication buffers varies from a

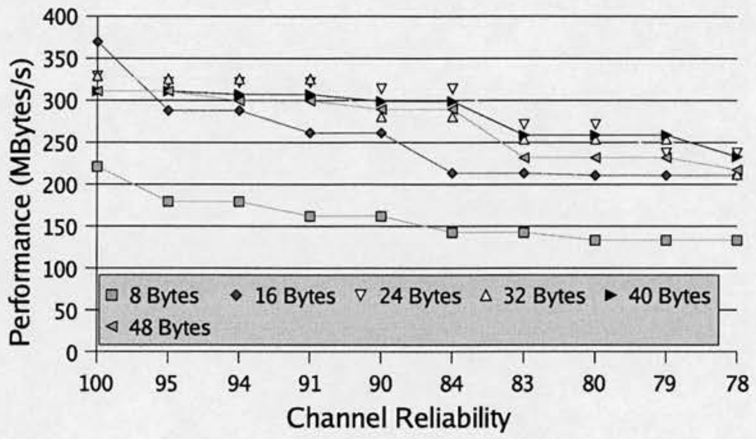


Figure 4.33: Communication performance over unreliable channels (2^4 sites/node)

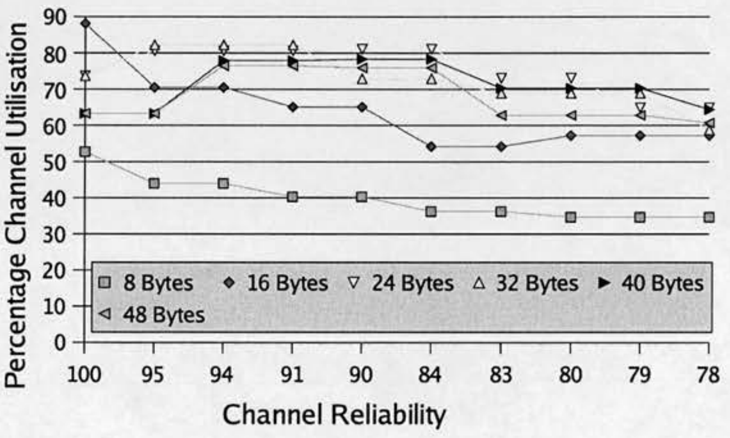


Figure 4.34: Percentage channel utilisation over unreliable channels (2^4 sites/node)

single double-word buffer to up to six double-word buffers. Without an extra buffering capability, i.e., sending a single double-word at a time with a 8 Bytes buffer, both the performance and the channel utilisation is poor. Although an increase in buffer sizes adds extra information processing time for message communications, the overall NN communication performance improves with an increase in communication buffer sizes except for the ideal case (no packet loss). This performance gain is a result of

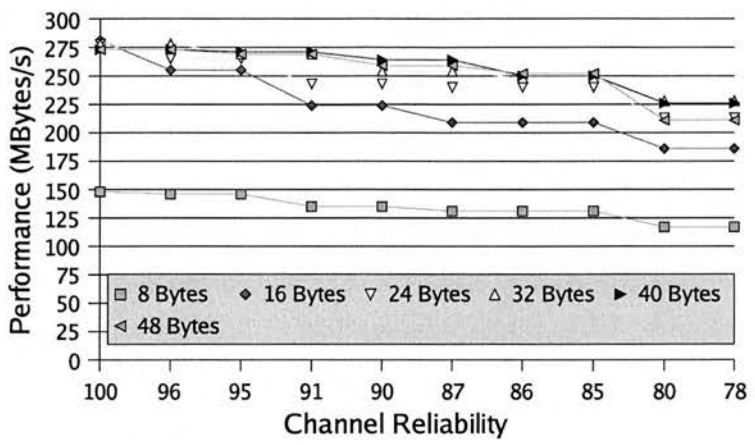


Figure 4.35: Communication performance over unreliable channels (2^2x4^2 sites/node)

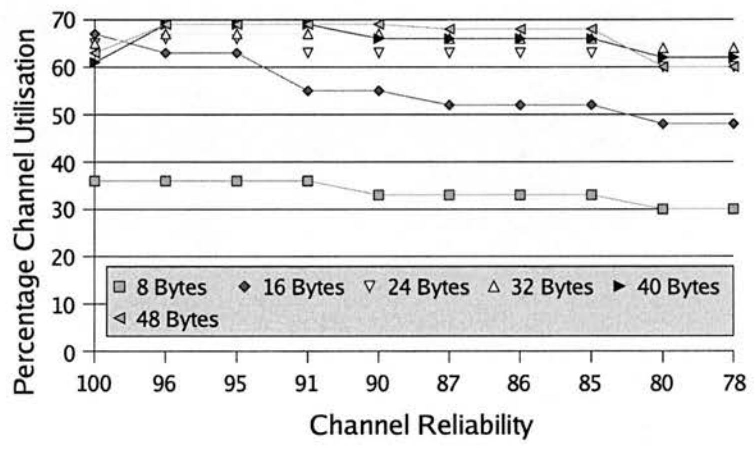


Figure 4.36: Percentage channel utilisation over unreliable channels (2^2x4^2 sites/node)

an increase in the communication channel utilisation. The difference in throughput from a single double-word buffer to the bigger buffer sizes is quite significant. At the same time, however, there is no, or only a marginal difference in the communication throughput as buffer sizes increase from a 2-double-word buffers to 6-double-word buffers, particularly for the smallest local volume. For a large local volume this difference becomes greater as the channel reliability decreases. Due to the message

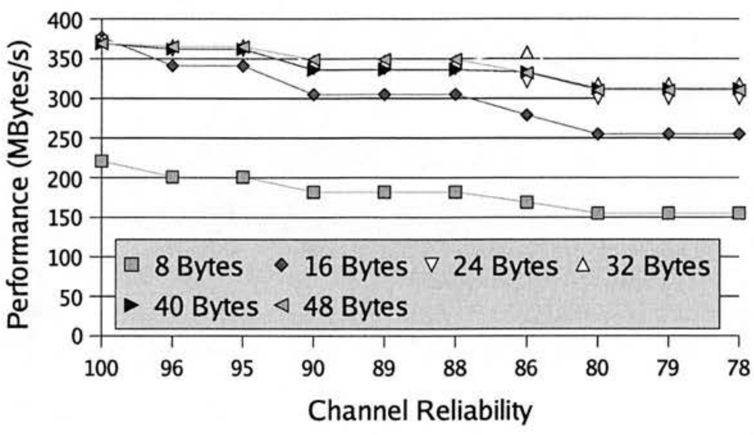


Figure 4.37: Communication performance over unreliable channels (4^4 sites/node)

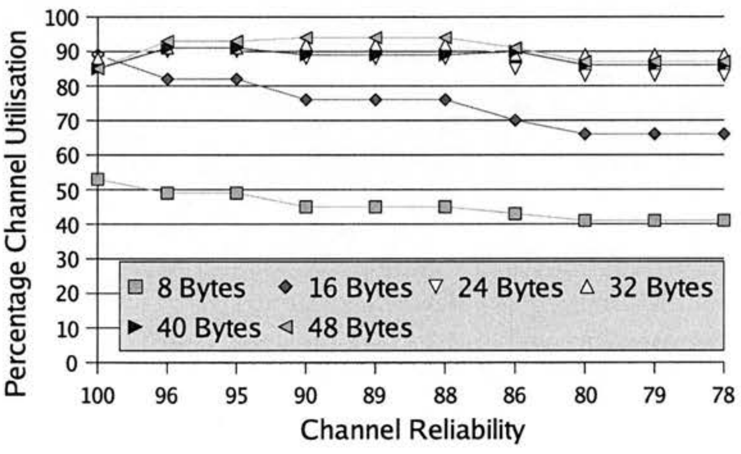


Figure 4.38: Percentage channel utilisation over unreliable channels (4^4 sites/node)

processing and time-out interval requirements, buffer sizes over 3 double-words offer a diminishing return on the throughput improvement.

Furthermore, an imbalance of communication data results in poor utilisation (figure 4.36) of communication channels and lowers the average throughput (figure 4.35). The local volume configuration 2^2x4^2 is an example of this behaviour. The size of communication data in one direction is determined by the number of lattice sites mapped per processing node in the other three dimensions. Hence, an increase in the local lattice volume due to an uneven increase in the number of lattice sites in one space-time direction results in an unbalanced communication volume for individual space-time directions.

4.6 Summary

This chapter has shown the HASE QCDOC model at work. Computation and communication characteristics of the QCDOC machine have been explored and the key software characteristics of the QCD test kernel have been identified. Performance evaluation experiments have been performed by altering the configuration of various design parameters. Results have been analysed and discussed. Furthermore, the high level abstraction model has been introduced and the performance results over the large scale model with thousands of processing nodes have been presented.

The parameterised hardware and software co-simulation scheme employed in designing the HASE QCDOC simulation models allows a range of experiments to be conducted: (a) with alternate QCDOC design configurations and (b) with several QCD workload distribution and mapping schemes. The next chapter outlines how the application-driven QCDOC simulation design approach facilitate in predicting and comparing parallel QCD performance on a future-generation supercomputer, Bluegene/L.

Chapter 5

Simulation Metamodelling

The previous chapter explained how the flexibility and ease with which parameters can be included and altered within the HASE framework have been exploited in creating parameterised simulation models encompassing QCDOC hardware characteristics and parallel QCD software properties. The design space of the QCDOC architecture has been explored and performance characteristics of the QCD kernel investigated through experiments with alternative system configurations, which is the central research aim. A related aim of the UKQCD computer simulation research is to predict achievable performance of parallel QCD code on a next-generation supercomputer architecture using parameterised HASE simulation models. The future-generation massively-parallel processing (MPP) machine that has been considered is the Bluegene/L supercomputer. This chapter explains the design and implementation issues for generating HASE Bluegene/L model design configurations. Furthermore, the application-driven simulation metamodelling concept is introduced; metamodelling enables efficient generation of alternate simulation models by employing maximum component reuse and minimum design overhead.

The layout of the chapter is as follows: firstly, an overview of the Bluegene/L supercomputer is presented. Secondly, the similarities and differences between QC-DOC and Bluegene/L systems are identified. Thirdly, an analytical description of the QCD application workload mapping and scaling behaviour is provided. Fourthly, an introduction to the application-driven metamodelling of scientific supercomputers using hardware-software co-simulation in the HASE is given. Finally, the results of experiments comparing parallel QCD code performance on the HASE models simulating QCDOC and Bluegene/L design configurations are presented.

5.1 QCDOC $\longrightarrow$ Bluegene/L Machine

Like a number of previous QCD-specific computing research efforts that influenced main-stream parallel architectures, the 10 Teraflops QCDOC machine design has been extended for a 180/360 Teraflops supercomputer called Bluegene/L. In 2001, IBM announced construction of the Bluegene/L supercomputer for protein research [Tea01a]. "Blue Gene" is an IBM supercomputing project — it aims to build a new family of supercomputers optimised for bandwidth, scalability and the ability to handle large amounts of data. Among the first applications IBM is exploring to exploit Blue Gene's massive computing power is modelling of the folding of human proteins. Learning more about how proteins fold is expected to give medical researchers a better understanding of diseases, as well as potential cures.

Unlike QCDOC, Bluegene/L is a general-purpose scientific supercomputer targeted towards a number of scientific workloads, according to IBM [GAB+02]. The Bluegene/L architecture is said to be optimised for a range of matrix-vector multiplication based scientific applications, most notably large-scale molecular dynamics calculations. The processing node and internode communication architecture are believed to be readily adaptable to a range of high-end scientific applications. Another design merit is cost-effectiveness; by exploiting off-the-shelf, high performance embedded System-On-Chip (SOC) and interconnect design technology, Bluegene/L and QCDOC machines are more cost-effective than current supercomputing resources due to: smaller physical sizes, low power and cost per processing node and power efficiency. QCD calculations are considered among the scientific applications that are expected to benefit from the Bluegene/L system architecture; hence, QCD simulations are planned to be executed on the Bluegene/L machine.

5.1.1 Bluegene/L Architecture

Bhanot *et. al.* [BCGV02] stated that the Bluegene/L is the first step by IBM towards their Petaflops initiative. The Petaflops computer design initiatives for protein research are collectively named "Blue Gene" [Tea01a]. Bluegene/L is one version in the "Blue Gene" series. Bluegene/L generalises QCDOC, wrote Gara *et. al.* [GAB$^+$02]; they declared QCDOC as the *first SOC supercomputer*. It is believed that the current SOC design integration techniques and the ongoing advancements can result in high-end, cost-effective supercomputers. The conventional approach of building tightly-coupled clusters of large symmetric multiprocessing (SMP) nodes suffers lower application efficiencies due to the increasing difference between processing power and memory access latencies. Many scientific applications gain little with cache locality schemes and access gaps between the processor and the memory levels degrades the achievable performance. In contrast to this, the SOC approach offers low latencies to all levels of memories, thereby compensating the cache misses and flushes overheads.

Supercomputer designs based on the SOC approach are likely to have tens of thousands of processing nodes, because a SOC processing node's performance is significantly lower as compared to, for instance, an SMP node. Thus a theoretical peak performance in tens or hundreds of Teraflops requires a significantly large number of 1-2 Gigaflops-scale processing nodes. A direct result of a huge number of processing nodes in a system is an enormous degree of hardware parallelism: instruction level parallelism and fine-grain internode parallelism. This will pose a great challenge on software practices to utilise the available parallelism to achieve a better fraction of peak-to sustainable performance for the application code. Simon *et. al.* [SMK$^+$03] cast doubt over performance sustainability of this trend of supercomputer design for the very same reasons. Parallel QCD application code however is capable of taking maximum advantage from fine-grain parallel systems. The likely performance of QCD code on the Bluegene/L machine is therefore worth investigating. The research presented in this thesis addresses and investigates the characteristics of the Bluegene/L system that can influence the execution of QCD code and subsequently its performance on the Bluegene/L supercomputer.

Exploiting fine-grain parallelism is the central idea of the Bluegene/L supercomputer design. The system is built out of a very large number of processing nodes (64K). A Bluegene/L processing node operates at a relatively modest clock rate (700 MHz) compared to an SMP processing node (a couple of GHz) resulting in 1.4 Gigaflops peak node performance — two processors per processing node. A direct benefit of the scheme is low power consumption and low cost per processing node. The Bluegene/L design utilises IBM's commodity design components including: IBM PowerPC embedded processor cores, embedded DRAM (EDRAM), and SOC techniques that allow for the integration of all system functions onto a single Application Specific Integrated Circuit (ASIC). Because of the relatively low speed processing clock, the memory latencies in terms of processor cycles are not high. The on-chip memory modules are therefore close to the processing speed and the penalties of accessing on-chip memory level are not as high as those in parallel Non-Uniform Memory Access (NUMA) architectures.

5.1.2 Bluegene/L ASIC

The Bluegene/L ASIC includes two standard PowerPC 440 embedded processing cores; one is referred as a *compute* processor and another as a *communication* processor. Both processors can perform computation in parallel; however, the main purpose of the communication processor is to off-load communication overhead and bookkeeping from the compute processor. A PowerPC 440 core contains 32-bit integer processor, so a 64-bit floating-point processors is interfaced with each PowerPC core via the Auxiliary Processor Unit (APU) for the floating-point computations. The PowerPC cores have separate data and instruction caches. In addition to the L1 data cache, each has a 2-KByte Level 2 (L2) prefetching cache. The prefetching caches are connected to the 4 MByte EDRAM (on-chip memory called Level 3 (L3) cache) with a high bandwidth data bus. Access latencies to the on-chip cache levels are limited to a few tens of processor clock cycles in the case of an L2 miss. The EDRAM interface provides a connection to external off-chip memory, which is expected to be 256 MBytes. A Bluegene/L ASIC is shown in figure 5.1.

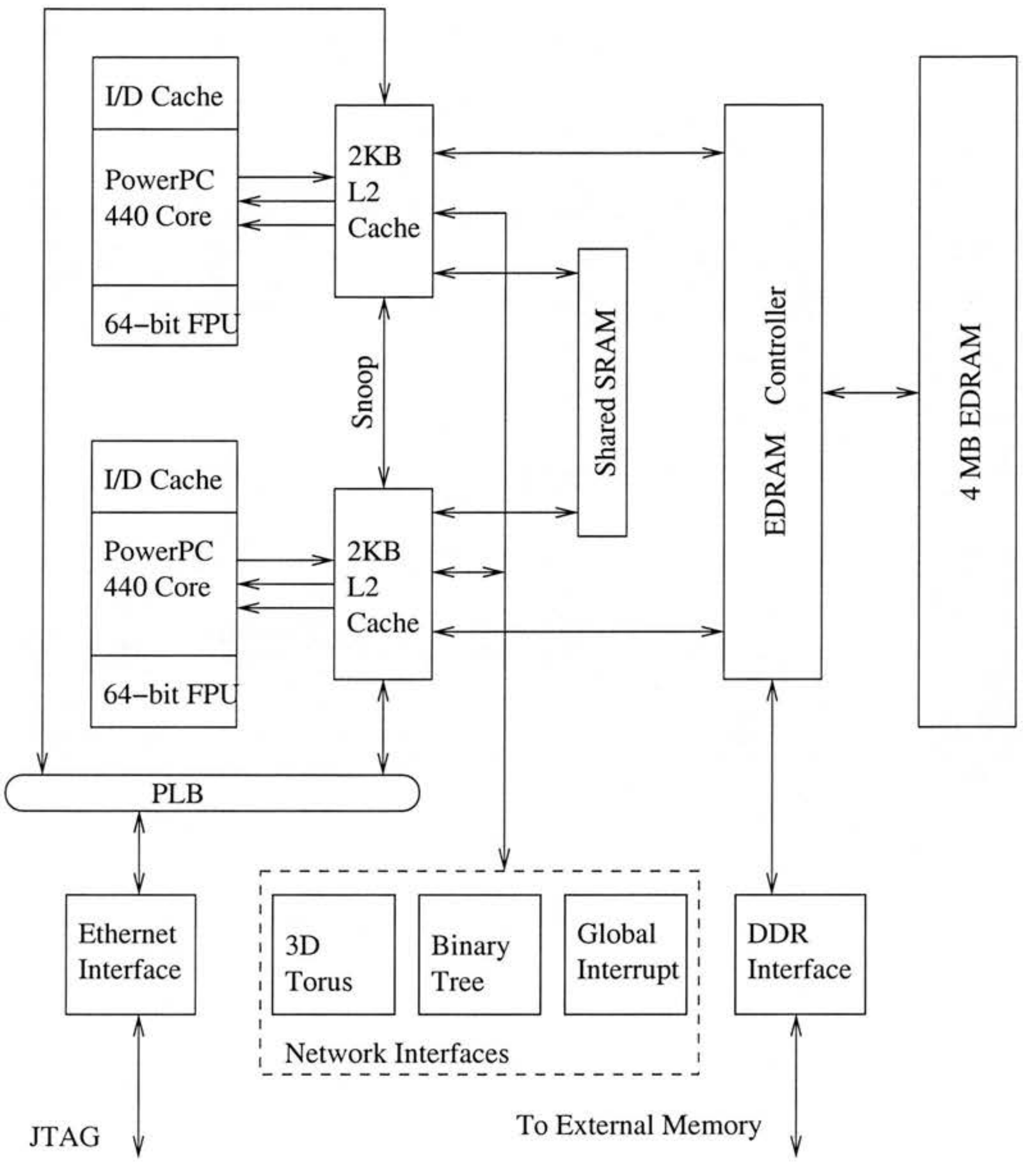


Figure 5.1: Bluegene/L ASIC (adopted from [Tea02])

Most computational instructions in the Floating-Point Unit (FPU) execute with a five cycle latency and single cycle throughput. The PowerPC 440 Central Processing Unit (CPU) and PPC440 FPU are superscalar with a quad-word (128-bit) access to lines in the Level 1 (L1) data caches. Up to three outstanding loads are permitted at a time. The Bluegene/L memory system is being designed for high bandwidth, low latency cache and memory accesses. An L2 cache hit returns in 6 to 10 processor cycles, an L3 hit is about 25 cycles, and an L3 miss is about 75 cycles. L3 misses are served by external memory.

The three on-chip memory levels require a coherency mechanism. L1 data cache coherency is provided in form of a lockbox. The lockbox allows processor-to-processor communications. L2 and L3 are coherent between the two cores. L2 caches are controlled by data prefetch engines and a fast on-chip SRAM array is used for communication between the two cores. Register files in the two cores can access data on either side, preventing swapping of data between registers. A special set of instructions is included in the existing PowerPC instruction set architecture that operate on data elements residing in both register sets.

Off-node communication data (data destined for and coming from neighbouring nodes) is stored temporarily before send and after receive operations in the link buffers within the network interface unit. The link buffers are connected to the L2 caches to facilitate efficient data transfers between the PowerPC processor and the network interface unit. Thus data from neighbouring nodes can be made available to the processor as soon as it arrives in the network unit. The presence of a dedicated communication processor and the availability of the communication data in L2 cache rather than a low level slow memory facilitate efficient off-node communication handling.

5.1.3 Communication Networks

Bluegene/L nodes are interconnected through five independents networks (c.f. figure 5.2):

1. **Three-dimensional Torus Network:** A three-dimensional torus interconnect between compute nodes for internode, point-to-point communication.

2. **Global Tree Network:** The tree interconnection network is organised as a tree of nodes. It is intended mainly for global reduction (for instance, QCD global sums) operations.

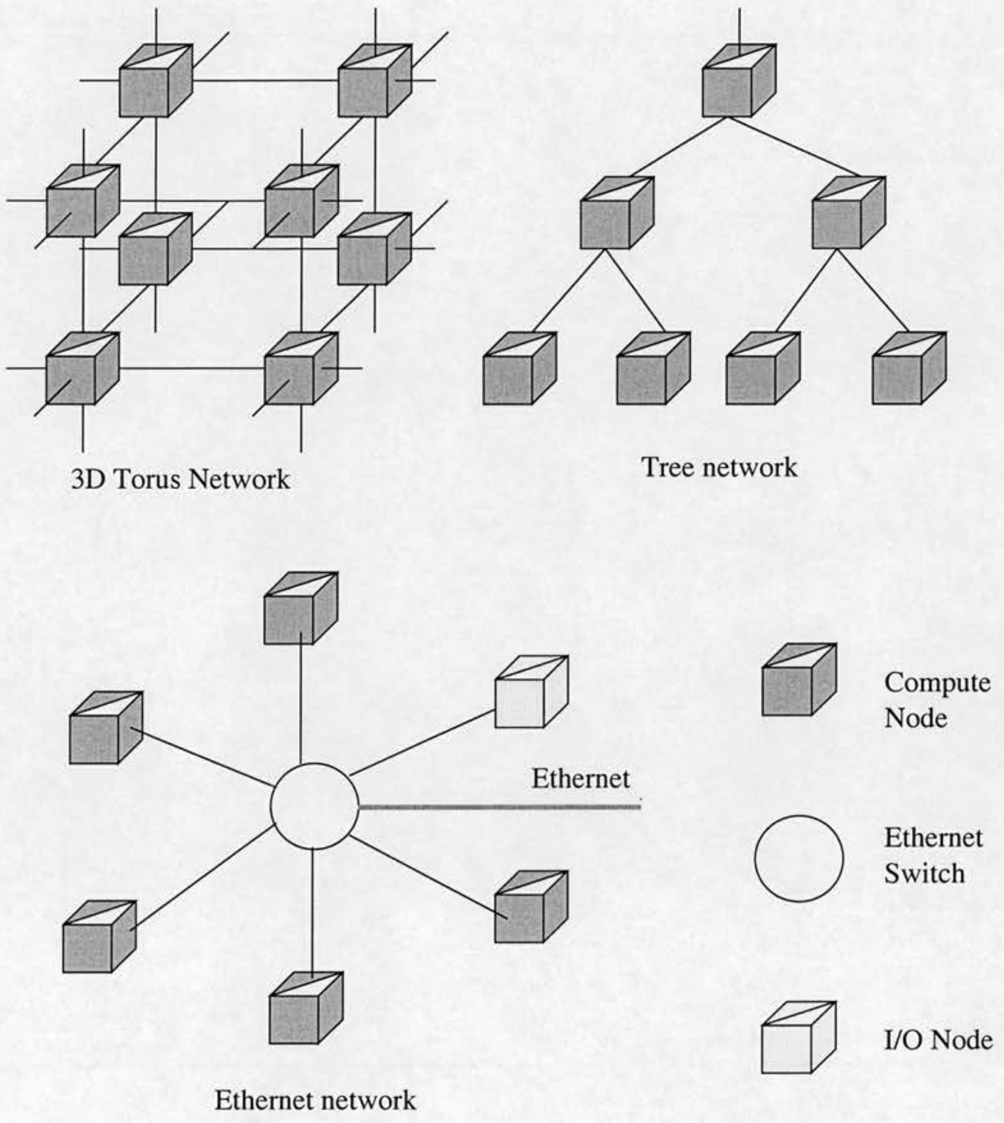


Figure 5.2: Bluegene/L networks [GAB+02]

3. **Global Barriers and Interrupt Network:** Also organised as a tree network, the global barrier and interrupt network provides global AND/OR operations. A low latency global interrupt mechanism is often necessary in MPP systems, for system synchronisations.

4. **Control Network:** It is an Ethernet network that supports machine booting, monitoring and diagnostic operations. It provides communication between the control system and individual nodes. The kinds of message this network

transports are low-level control messages. For example, accessing the PowerPC instruction cache of a particular node (each processing node has a unique identifier).

5. **System Support Network:** All I/O nodes[1] are interconnected with a Gigabit Ethernet network, for connection to other systems, such as hosts and file systems. Tera- or Peta-bytes of results generated from MPP simulations need an efficient file I/O facility in conjunction with a fast processing power.

Networks (1) and (2) are essential for the parallel QCD code execution: the point-to-point three-dimensional torus (1) for the frequent nearest-neighbour communications and the tree network (2) for collective communications, which are the conjugate gradient global sums. System support, diagnostics and file input/output are supported by the (3), (4) and (5) networks. These last three networks do not directly contribute to the parallel QCD code execution on the Bluegene/L machine.

5.1.3.1 Three-Dimensional Torus Network

The three-dimensional torus interconnection network topology supports the regular communication pattern of a number of parallel scientific applications that are governed by molecular dynamics algorithms. This three-dimensional torus provides a direct network for the frequent nearest-neighbour QCD communications. Unlike the QCDOC's dedicated nearest-neighbour torus network, the Bluegene/L machine's three-dimensional torus network is intended to be used for general-purpose, point-to-point message passing and multicast operations to a selected subset or class of processing nodes. The topology is constructed with point-to-point serial links between routers embedded within the Bluegene/L ASICs. Hence, a Bluegene/L ASIC has six nearest-neighbour bi-directional serial links, or twelve send and receive ports with a target bandwidth of 175 MBytes per second.

Figure 5.3 shows the structure of the torus connections within a processing node.

[1] In a 64K processing (compute) nodes system, there is 1 I/O node per 64 compute nodes. I/O nodes' architecture is identical to the compute nodes but these have a full Linux operating system to perform file I/O operations (compute nodes have a light-weight Linux kernel).

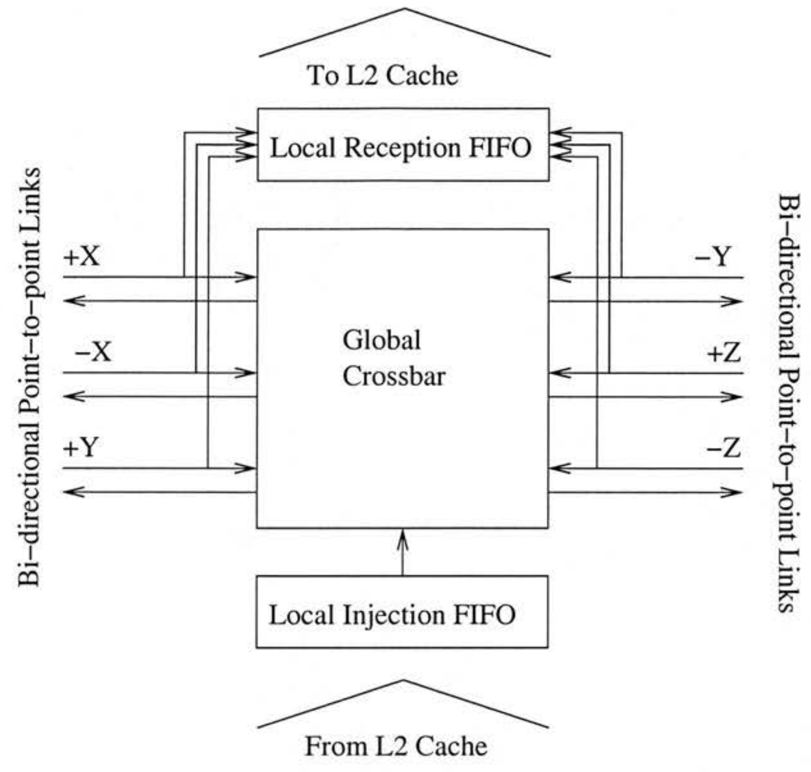


Figure 5.3: Torus connections and routing

Packets are injected into the network at one of the local injection First In First Out (FIFO), and are deposited into the local reception FIFO upon reaching their destinations. The messaging co-processor is responsible for injecting and removing packets from these FIFOs. A co-processor can be conceptualised as the simplified Serial Communication Unit (SCU) of the QCDOC machine, which also supports the routing and buffering schemes. Throughput and hardware latency are optimised through the use of virtual cut-through routing. Packets can be from 32 bytes to 256 bytes.

5.1.3.2 The Global Tree Network

A global combining tree is designed for collective communication operations. The tree module is equipped with an integer Arithmetic and Logic Unit (ALU) for combining incoming packets and forwarding the resulting packet. This is shown in figure 5.4.

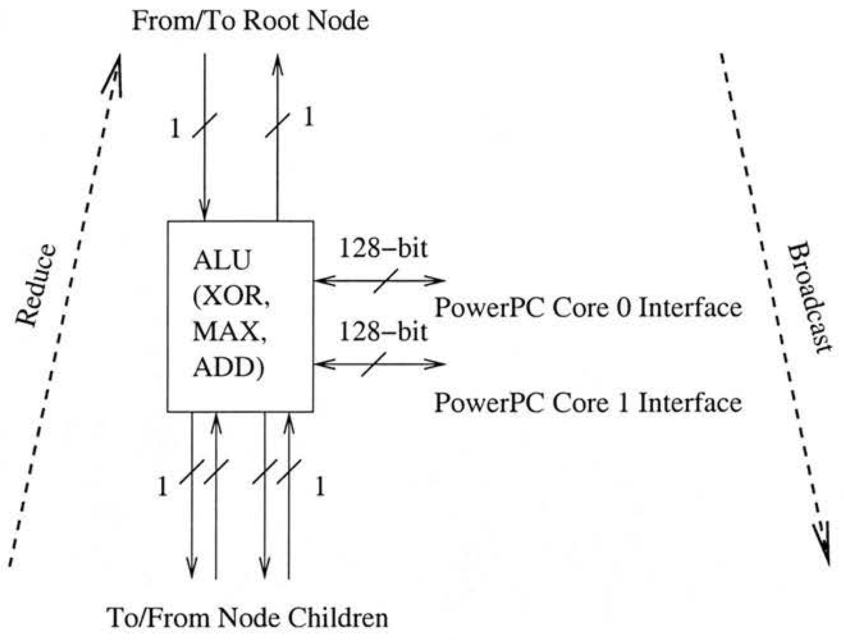


Figure 5.4: Tree module with a serial ALU

Packets can be combined using bit-wise operations such as XOR or integer operations like ADD or MAX for a variety of data widths. All packets coming down the tree are broadcast further down the tree according to the control registers and received upon reaching their destination. For collective operations packets are received at each node.

To perform a floating-point sum reduction (the QCD global sum) on the tree requires potentially two round-trips to the tree. In the first trip each processor submits the exponent for a Max reduction. In the second trip, the matissae are appropriately shifted to correspond to the common exponent as computed on the first trip and fed into the tree for an integer sum reduction.

5.2 Comparison of QCDOC and Bluegene/L Hardware Features

From a QCD application code execution view point, a block diagram of a Bluegene/L processing node is as shown in figure 5.5.

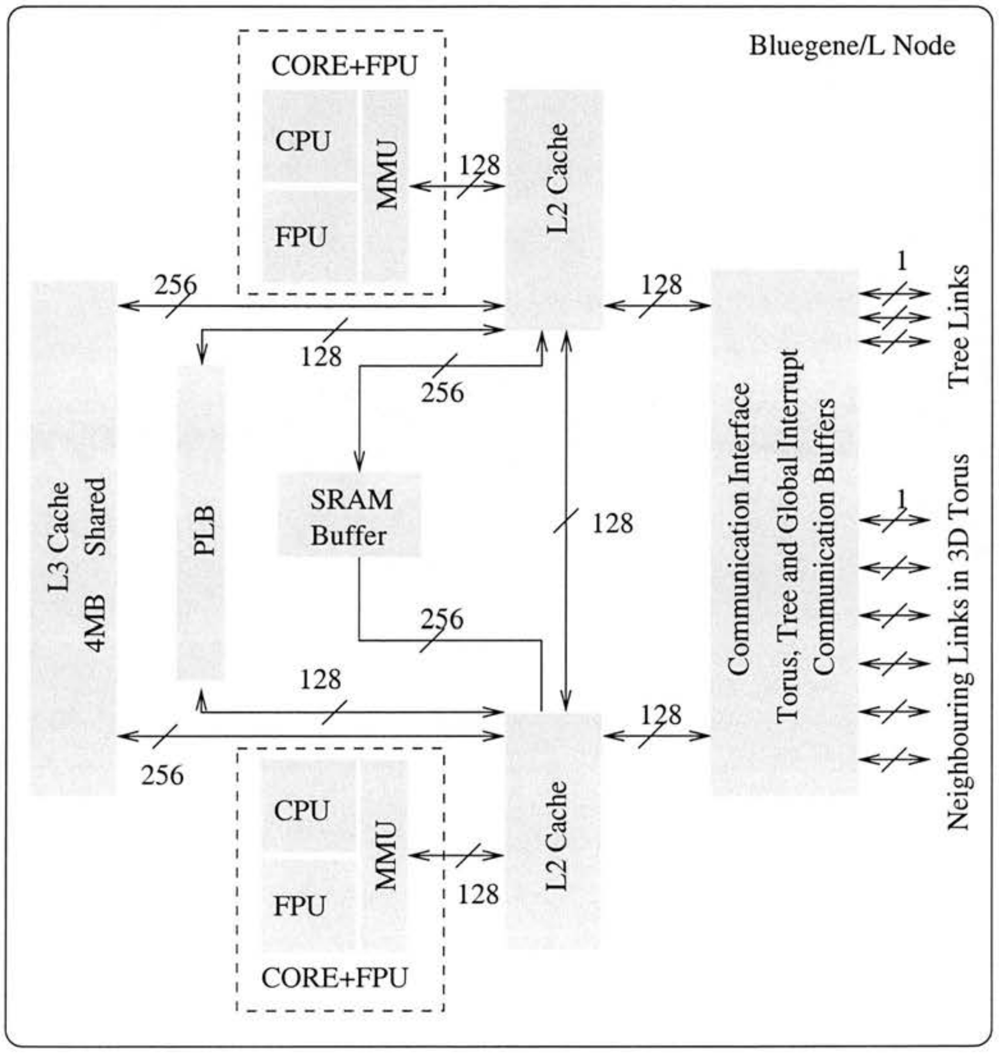


Figure 5.5: Simplified Bluegene/L processing node

Comparing figure 5.5 to the block diagram of the HASE QCDOC processing node (in figure 5.6), a number of similarities can easily be identified, including the processor core, system bus and memory levels. The most obvious differences are the presence of two PowerPC processing cores in Bluegene/L instead of one in the QCDOC and the communication unit designs of the two systems.

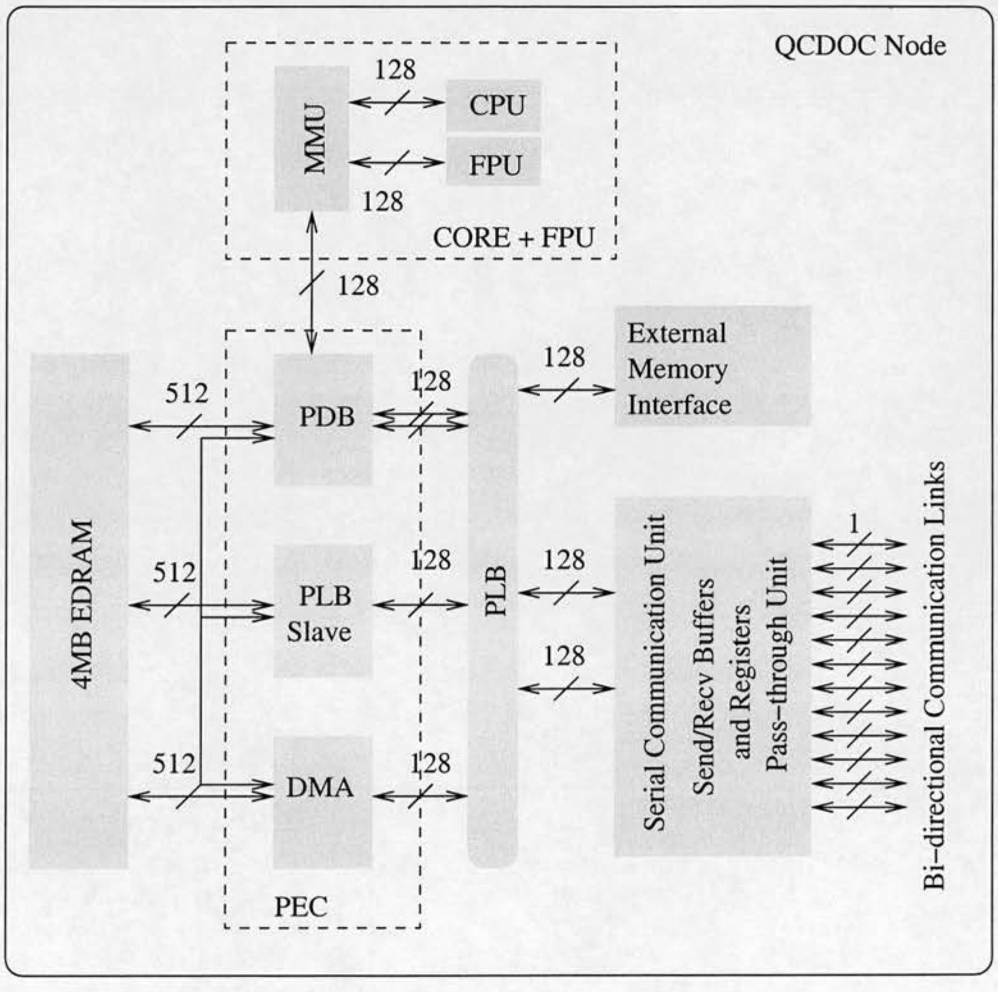

QCDOC Node
MMU
CPU
FPU
128
128
CORE + FPU
128
4MB EDRAM
512
512
512
PDB
PLB Slave
DMA
PEC
128
128
128
PLB
128
External Memory Interface
128
128
Serial Communication Unit
Send/Recv Buffers and Registers
Pass-through Unit
1
Bi-directional Communication Links

Figure 5.6: HASE QCDOC node

The HASE QCDOC simulation models were benchmarked with an optimised version of parallel QCD code written in PowerPC assembly. The Bluegene/L ASIC contains an advanced version of the PowerPC 440 core with the same Instruction Set Architecture (ISA). In addition, the memory hierarchy and the on-chip control and data paths of two systems have similar design features, because the parallel

QCD calculations share a number of characteristics with other scientific, message passing kernels. Thus, to facilitate low-latency and high-bandwidth data access to the processor, the Bluegene/L on-chip memory hierarchy levels are analogous to the QCDOC on-chip memory hierarchy levels.

Hence, as an immediate successor to the QCDOC machine, the Bluegene/L machine has a number of hardware features similar to the QCDOC machine. Nevertheless, there are some subtle differences. Table 5.1 summarises the differences between Bluegene/L and QCDOC machines that will affect execution of parallel QCD code on these systems.

The fundamentally distinct features of the Bluegene/L machine are the interconnection network and associated components. In QCDOC, since the QCD communication routines are well understood through the QCDSP design, the custom-built SCU design and the network topology are precisely targeted towards the communication requirements of the QCD code. Other parallel applications are unlikely to get benefit from the QCDOC communication network topology and SCU design. Even QCD software repositories other than Columbia Physics Systems (CPS) require extensive code tuning. Bluegene/L, in contrast, can execute a range of message-passing parallel applications including QCD; but, its communication network is not *optimised* for the QCD communication patterns.

	QCDOC	Bluegene/L
Target applications	Optimised for QCD calculations.	Can serve a broad range of scientific applications.
Peak performance	10 Teraflops scale. A 400-500 MHz floating-point execution unit with two execution pipelines per processing node.	180/360 Teraflops scale. 180 Teraflops if the processing power of compute node is considered only. 360 Teraflops for combined compute and communication node performance.
Processing node	One PowerPC 440 core plus one 64-bit FPU per node.	Two PowerPC 440 cores plus 64-bit FPUs per node.
Per node performance	1 GF/s per ASIC, 1 GF/s per node.	2.8 GF/s per ASIC, 5.6 GF/s per node.
Prefetch L2 Cache	One 4K-bit per node.	Two 2-KByte per node.
Communication networks	A reconfigurable 6-D torus and an Ethernet network.	A 3D torus, a combining tree, Ethernet networks.
Communication protocol	Point-to-point communication using a custom protocol.	Point-to-point and multicast communication employing virtual cut-through routing.
Communication overheads	Small communication packet size with no routing information overhead.	There may be routing information and overhead.
Off-chip memory	256 MB DDR per daughter board, two processing nodes per daughter board	512 MB DDR per compute card (two nodes).
Collective communications	Pass-through unit for global operations.	Tree network with an ALU for collective communication operations.

Table 5.1: Comparison of QCDOC and Bluegene/L systems

5.2.1 On-chip Memory Hierarchy

Theoretically the two on-chip processors can participate in computation simultaneously in a Bluegene/L processing node. Presently, however, it is envisaged that the complexity of code generation is likely to restrict one processor, the compute processor, to be solely responsible for handling computation [MCS+03]. Another disadvantage of using the two on-chip processors would be the data coherency overheads and operating system complexity. By considering the compute processor only and its relationship to the three levels of on-chip caches, the Bluegene/L on-chip data paths are similar to the QCDOC on-chip data paths (shown in figure 5.7). The difference in processing speed and ratio of memory module clock frequencies to the processor clock result in slightly different on-chip memory latencies and bandwidths.

Figure 5.7 compares the data movement between levels of memories on a Bluegene/L node and a QCDOC node. The L2 caches on the Bluegene/L operate at half of the processor clock frequency while an L2 cache in QCDOC operates at the processor clock frequency resulting in a single clock cycle latency in case of an L2 cache hit. Bluegene/L has faster processors than the QCDOC, hence the bandwidth to L2 caches is not exactly half of the QCDOC L2 bandwidth. Another difference is the relationship between on-chip memory EDRAM and the external memory. In QCDOC, these memories have separate physical addresses and the Processor Direct Bus (PDB) is able to identify which addresses are destined for the EDRAM, and which are destined for the external SDRAM. In contrast, the L3 cache controller in Bluegene/L will identify a L3 cache miss, which is expected to take more than 75 processor clock cycles. The data bus width between the L2 and L3 caches is different in the two systems, i.e., QCDOC has twice the bus width to the L3 cache as compared to the Bluegene/L L2 cache.

The Bluegene/L processing node enforces coherency between its two L2 caches in hardware through a snoop protocol. This mechanism is similar to the Prefetch EDRAM Controller (PEC) of the QCDOC processing node; PEC maintains coherency between its design blocks including the L2 cache. A write address from the other processor causes invalidation of earlier read data in the L2 prefetch read registers. Data coherency between Bluegene/L's L1 caches is maintained through the fast on-chip SRAM. It is

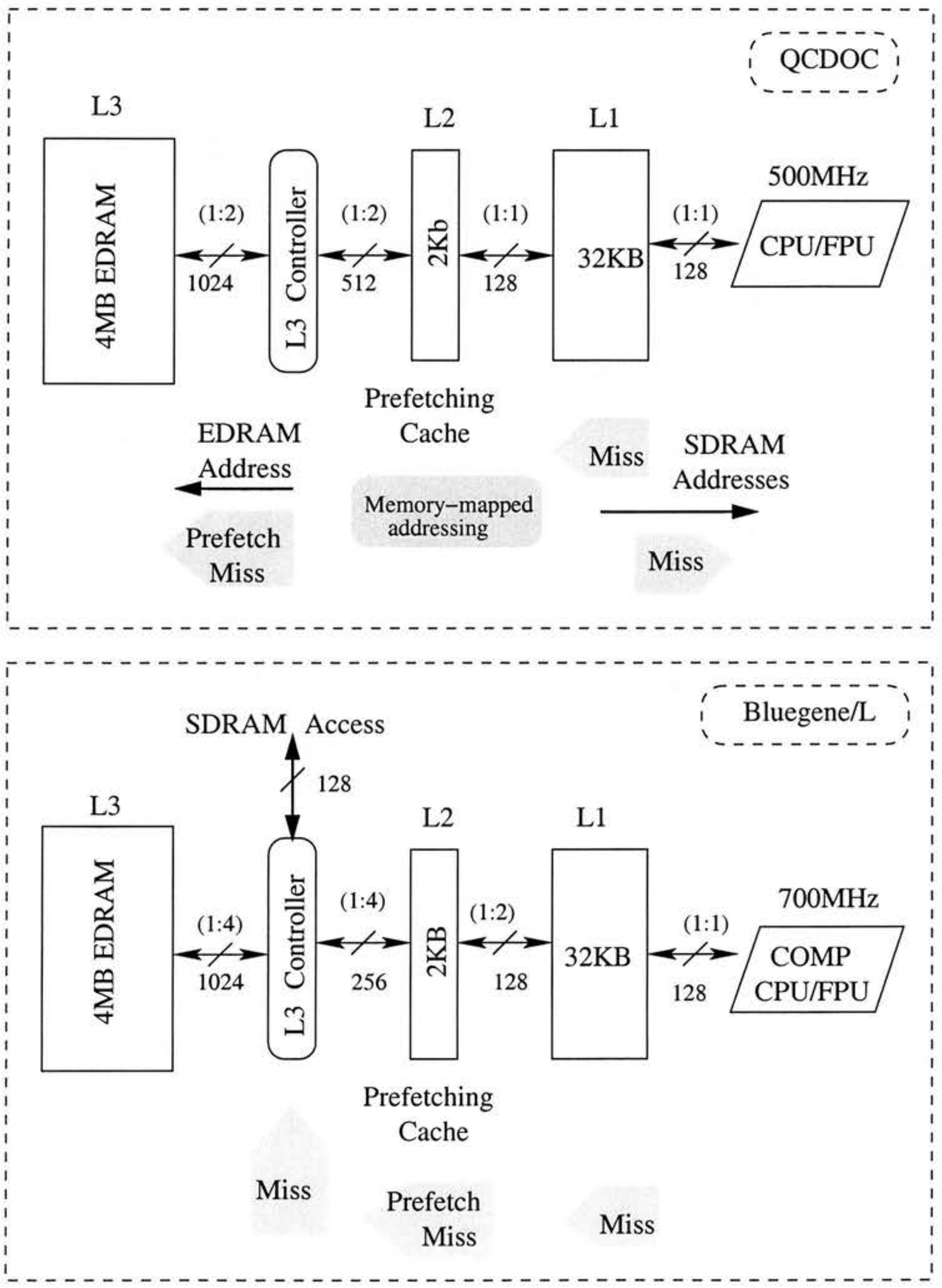


Figure 5.7: Data paths — Bluegene/L and QCDOC on-chip memory hierarchy

not clear from the Bluegene/L design documents whether it is strictly enforced via hardware or whether it would be the responsibility of the software. QCDOC does not have this L1 cache coherency overhead because it contains a single processor core.

Apart from on-chip data movements, accesses to and from the off-chip or external memory follow different patterns in the two ASICs. In QCDOC, external memory accesses are memory-mapped and movement between on-chip and off-chip memory is invoked explicitly by the processor. It sets up a Direct Memory Access (DMA) controller which takes care of transferring data between on-chip memory and the external memory. Off-chip memory transfer latencies depend on the page sizes. Bluegene/L off-node transfers are invoked as a result of L3 cache misses.

5.2.2 Off-chip Communication Data Transfers

The on-chip communication logic and control in Bluegene/L are substantially different from communication data handling in the QCDOC ASIC. In QCDOC, the processor initialises the communication unit like a DMA device. The SCU DMAs in turn initiate and control transfers of a given number of data packets to and from the on-chip memory. A set of DMA instructions then make use of the system bus, the Processor Local Bus (PLB), and a PLB slave to the on-chip memory, the EDRAM, to read and write incoming and outgoing data.

QCDOC and Bluegene/L 's off-chip communication control and data transfers are compared in figure 5.8. The Bluegene/L design is not restricted to a fixed subset of off-node communication patterns, thus it does not have a QCDOC-style, customised and dedicated, serial communication unit. Instead, the L2 caches are connected to the three communication interfaces: the three-dimensional torus, the global tree and the global interrupt. Instructions and data are transmitted via the L2 caches directly to these network interfaces without the intervention of the PLB. The PowerPC communication core is expected to oversee the Bluegene/L communications. The associated L2 cache is likely to be the source and destination of the torus and tree communication interfaces.

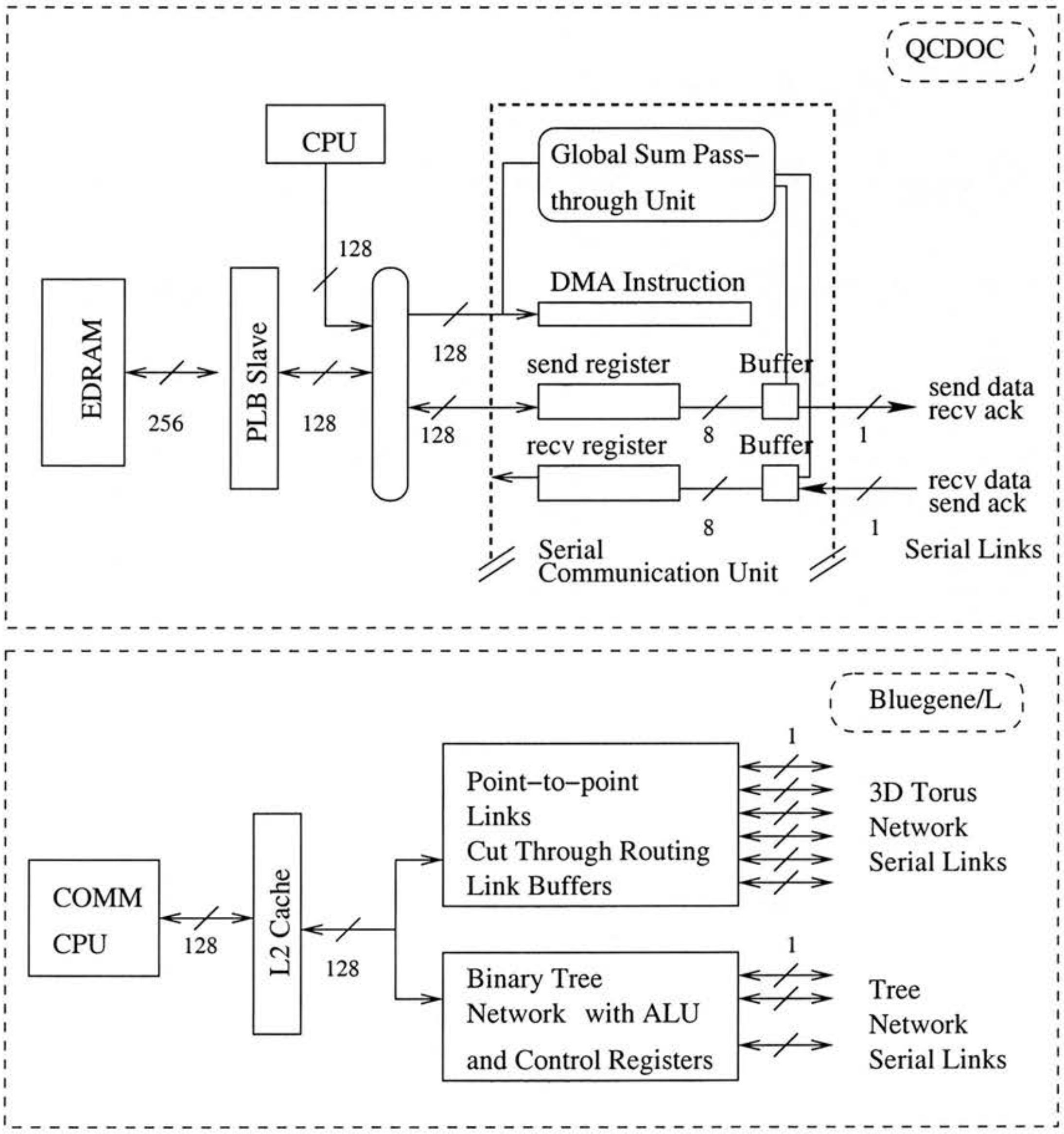


Figure 5.8: On-chip data and control paths for point-to-point communication

In Bluegene/L, quad-word data packets from the processor can be decoded and transferred to the memory-mapped devices through the L2 caches, which operates at half the processor clock frequency. These memory-mapped devices include the torus and tree network interfaces; their initialisation and control are managed by the communication core. Off-chip communication initialisation and control overheads and contention between two L2 caches can be minimised by assigning one processor core to take responsibility for the communication related events, thereby avoiding unnecessary flushes of caches.

5.3 Lattice QCD Performance on MPP Systems

Performance exploration and analysis of a future generation MPP systems for parallel QCD application requires, as precisely as possible, understanding of the interactions and dynamics of system hardware and application software. Ideally, the simulation studies should be conducted using accurate models of the MPP hardware and complete executions of the parallel QCD application code software. In practice:

1. detailed, cycle-accurate MPP hardware simulations are not possible. Emulating physical properties including the processor power and memory capacity of a high-end system need enormous computing resources. An appropriate level of abstraction is needed, such that it should not compromise on the details of parallel hardware and software interactions, which is the essence of software simulations;

2. the application code execution is in most cases represented by a kernel code. Benchmarking using the complete execution code in cycle-accurate simulations is not feasible because of prohibitively long simulation times associated with executing them with a cycle-accurate simulator. In scientific applications, many calculations share some frequently executing code patterns and the optimisation of these core methods guarantees the overall optimisation.

A high-level abstraction approach is therefore necessary, both for the application code software and the underlying parallel system. Because the UKQCD computer simulation design has followed a hardware-software co-simulation approach in which the underlying hardware components and the application workload properties have been parameterised, it is possible to alter the QCDOC and its application code characteristics to explore the Bluegene/L supercomputer. The parameterised software modelling provides a seamless transition for experiments from a QCDOC model to a Bluegene/L model by allowing parallel QCD computation and communication workload to be mapped onto the Bluegene/L design configurations. Performance prediction and comparison experiments with the HASE QCDOC and Bluegene/L models require a detailed understanding of the mapping of the QCD code onto different parallel machines along with their hardware features.

5.3.1 Parallel QCD Code Configurations

As described in Chapter 2, the computation and communication cost of a parallel QCD application code directly depends on the number of lattice sites mapped onto a node. The number of sites mapped onto a node, in turn, depend on the lattice volumes. It is necessary to identify the difference between the two: the lattice volume and the number of lattice sites per processing node. Lattice volume is the four-dimensional lattice configuration, which a physicist decides to simulate on a parallel hardware. The number of sites on the node depends on the hardware properties of the system as well as the mapping strategy adopted by the user. In most cases however, a uniform mapping is preferred for lattice QCD calculations.

An (incomplete) data structure declaration of a variable central to the lattice QCD calculations called `Wilson`, a C/C++ `struct`, is given in figure 5.9.

```
#define ND                    4       /* Space time dimension*/

/* The Wilson structure typedef */
typedef struct{

   /*pointer to an array with addressing offsets */
   unsigned long *face_table[2][2][ND];

   /* Sites Mapped per node */
   int local_latt[ND];

   /* Local lattice Volume */
   int   vol[2];

   /* Send and Receive Arrays in Forward and Backward Directions */
   double *send_f[ND];
   double *recv_f[ND];
   double *send_b[ND];
   double *recv_b[ND];

   /* Boundary Values */
   int nbound[4];

   /* Local Communication requirements */
   int local_comm[4];

   ...........................

} Wilson;
```

Figure 5.9: C++ data structure for lattice QCD calculations

`Wilson struct` fields are initialised right at the beginning of the calculations, and most values and assignments depend on the numbers of lattice sites mapped per node. The communication layout in the four space-time directions depends on the number of sites in the individual directions. In summary, lattice QCD calculations are driven by the knowledge of the total number of lattice sites on a processing node, together with the number of sites along the separate space-time directions.

An increase in the number of lattice sites mapped onto a processing node results in an increase in the memory, execution and off-node requirements of that node. Ideally, a very small local configuration, 2x2x2x2 should be used. A small local configuration can only be supported either by a very large number of processing nodes or small QCD lattice volumes. The drawback of the former approach would be the collective communications bottleneck, while the latter would not yield results in the range of approximations required by the physicists.

Table 5.2 compares the computation cost of an example lattice size on the QCDOC (4D torus) and Bluegene/L (3D torus) machines.

	4-D torus	3-D torus
Lattice Volume	8x8x8x8	8x8x8x8
Machine dimensions	4x4x4x4	4x4x4
Sites per node	2x2x2x2	2x2x2x8

Table 5.2: Example: lattice mapping onto a 3-D and 4-D torus networks

For a small lattice volume, 8x8x8x8, a large number of Bluegene/L nodes are not an advantage. Instead, the extra dimension offered by the QCDOC reduces the computation requirements, by removing the need for local computation in that particular direction. An upper bound on the local computation cost is estimated as the minimum amount of extra work per lattice site simulation. Figure 5.10 illustrates this effect on computation cost over a two-dimensional mesh and a one-dimensional mesh system.

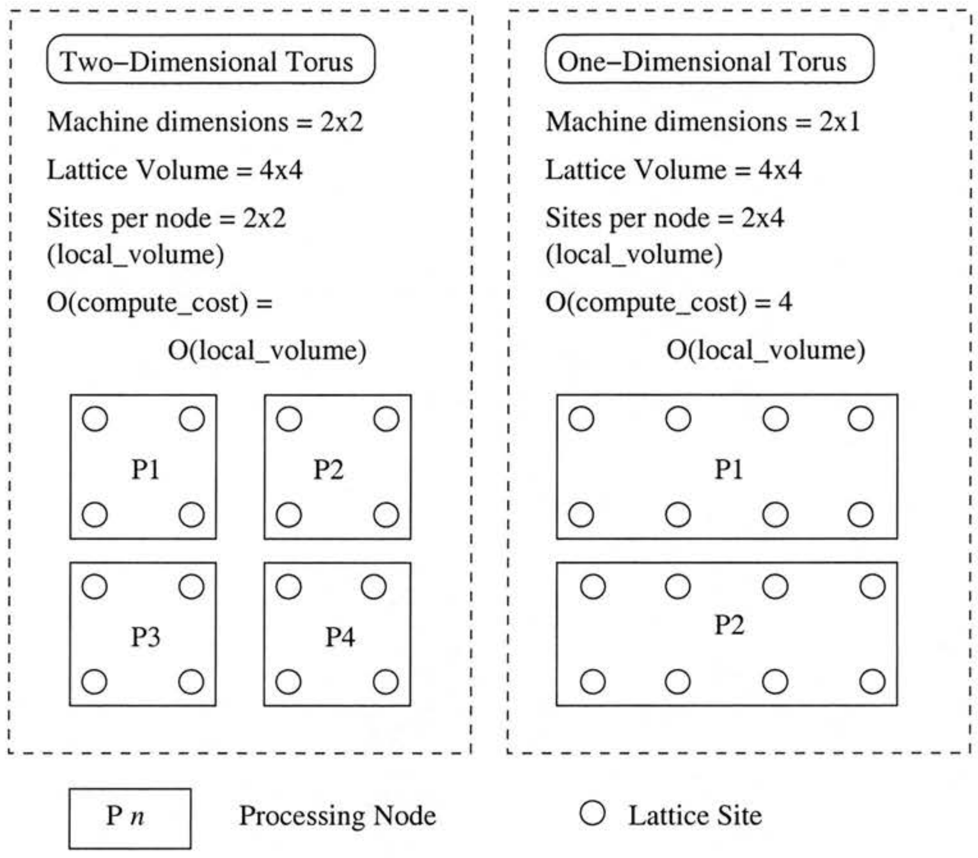


Figure 5.10: QCD computation requirements scaling with respect to the underlying physical network

A characteristic feature of the parallel QCD application code is frequent and regular communications in the four space-time directions. QCDOC supports the four-dimensional communication requirements by providing a four-dimensional periodical torus network. Bluegene/L has a three-dimensional torus network. Figure 5.11 shows the effect of one less dimension in the network topology on QCD communication requirements.

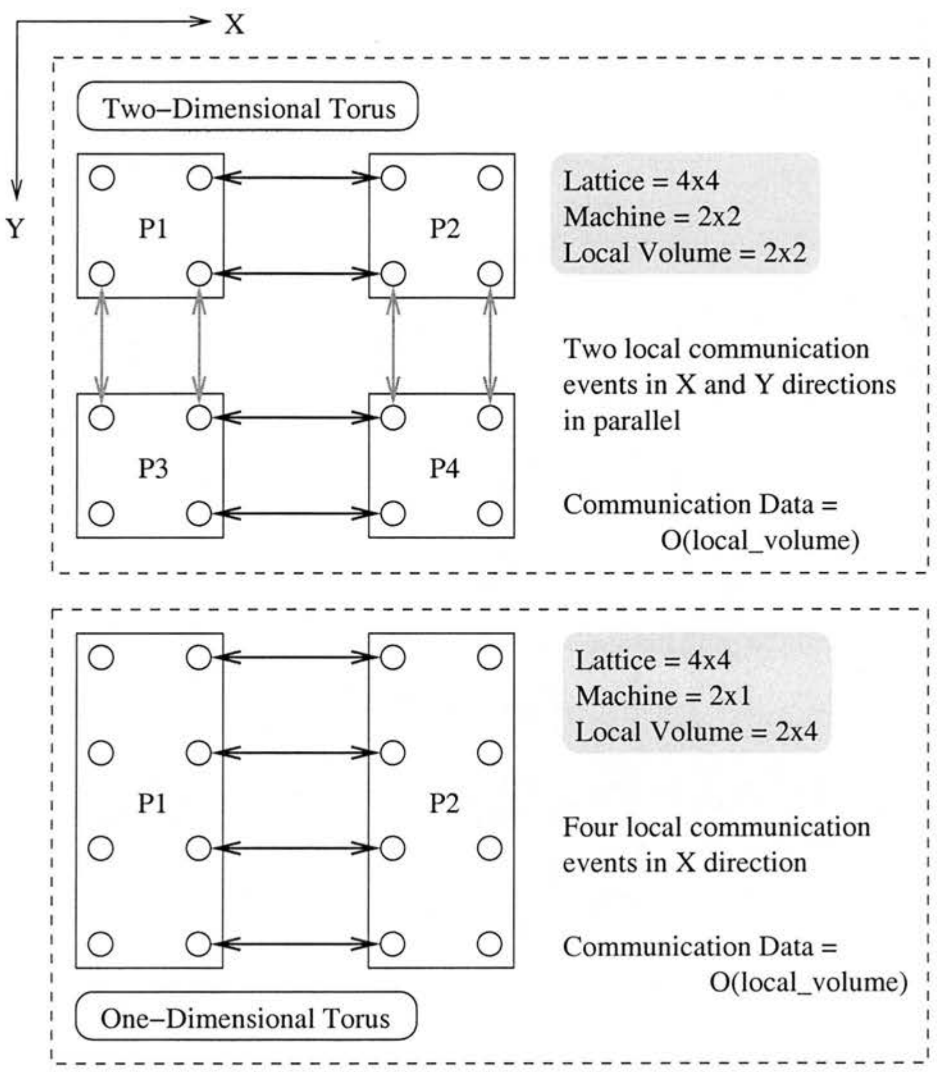

X
Y
Two–Dimensional Torus
P1
P2
P3
P4
Lattice = 4x4
Machine = 2x2
Local Volume = 2x2
Two local communication events in X and Y directions in parallel
Communication Data = O(local_volume)
P1
P2
Lattice = 4x4
Machine = 2x1
Local Volume = 2x4
Four local communication events in X direction
Communication Data = O(local_volume)
One–Dimensional Torus

Figure 5.11: QCD nearest-neighbour communication volume scaling with respect to the underlying physical network

An upper bound on the communication data requirements is estimated, which depends on the number of sites that need off-node communications as well as the size of the communication data. Two concurrent nearest-neighbour communication costs in two separate directions are comparatively smaller than the four nearest-neighbour communication cost in any one neighbouring dimension.

In an evenly-distributed lattice scheme, one less dimension means that all lattice sites in one dimension have to be mapped onto each processing node; this increases the memory requirement. A consequence of assigning all sites in a dimension to a processing node would be increased local interactions and on-node communications. Local communications result in increased processing time as well as temporary memory requirements of the local interaction computations. In summary, ignoring the overheads of infrequent collective operations, a system with a very large number of processing nodes with fast floating-point units and capable of supporting concurrent off-node communications with very little or no overhead between four neighbouring nodes, can be considered as an ideal system for parallel QCD calculations.

5.3.2 Practical Lattice Volumes

So far a bottom-up approach has been adopted in explaining the lattice QCD workload mapping onto a processing node, i.e., number of sites/node rather than the size for a QCD calculation. While this approach is nearly always employed for prototyping and benchmarking, practical QCD calculations take a rather different point of view. End results with a range of precisions are the target for the QCD calculations. Generally, physicists estimate what lattice size would yield a result within a given range or provide better approximation for the continuum space-time over a discrete lattice. Bowler [Bow98] summarised the cost of discretising the continuum QCD calculations on a lattice: either the lattice spacing should be close to zero or the continuum limit for the best precision QCD calculations and lattice volume should be as large as possible. As the lattice spacing becomes small, or alternatively, when large lattice volumes are considered, the computation and communication requirements per processing node increase. These effects are not always linear and depend on the memory capacity and floating-point execution capability of the underlying hardware system. For QCDOC and Bluegene/L comparison experiments, a top-down approach has been taken, i.e., practical lattice volumes have been mapped onto the known configuration of the QCDOC and Bluegene/L systems.

The choice of lattice size depends on a number of factors: available parallel system characteristics and calculation requirements. Presently, the lattice volumes (number of lattice sites) considered for various QCD calculations on high-end parallel supercomputers include:

- 8x8x8x8 (*Low precision, minimum accuracy and low computing requirements*)
- 16x16x16x16
- 32x32x32x23
- 48x32x32x32
- 96x32x32x32
- 96x48x48x48
- 32x64x64x64 (*High precision, high accuracy and enormous computing requirements*)

5.3.3 Communication Workload Mapping

A regular three-dimensional mesh has been a popular choice in many MPP architectures including Cray T3D, T3E, Blue Mountain and IBM's Bluegene/L. QCDOC implements a reconfigurable, bi-directional, four-dimensional torus topology in order to evenly map the four-dimensional lattice calculation evenly on the underlying communication network. If the underlying physical system does not have a four-dimensional torus topology, lattice sites along the smallest dimensions are stored locally at all processing nodes. This not only increases the storage and processing requirements but, most importantly, multiplies the communication volume.

When executing QCD code on a parallel machine, the first step is to decide the volume of the lattice simulation. The bigger the volume, the better approximation to the continuum lattice. However, large lattice volumes impose enormous computation and communication requirements on the parallel system. As mentioned earlier, an even mapping is the most common lattice distribution scheme over a parallel machine with a regular torus topology. Table 5.3 shows practical lattice volumes used presently by QCD physicists on high-end parallel machines, and the lattice volume per node distributions on a 3-D and 4-D torus.

Lattice Volume	Volume per Node	
	3-D Torus	4-D Torus
8x8x8x8	64 (2x2x2x8)	16 (2x2x2x2)
16x16x16x16	128 (2x2x2x16)	16 (2x2x2x2)
32x32x32x32	256 (2x2x2x32)	128 (4x4x4x2)
48x32x32x32	256 (2x2x2x32)	192 (3x4x4x4)
96x32x32x32	384 (3x2x2x32)	384 (6x4x4x4)
96x48x48x48	864 (3x3x2x48)	1024 (8x8x4x4)
64x64x64x32	1024 (2x4x4x32)	1024 (8x8x4x4)

Table 5.3: QCD lattice volume mapping per processing node

For example, for a 48x32x32x32 QCD lattice, 2x2x2x32 sites would be mapped on to a 24x16x16, three-dimensional torus machine. A four-dimensional 16x8x8x8 system, on the other hand, would be capable of distributing the same lattice as 3x4x4x4 lattice sites per node. The first two volumes in the table are normally considered too small for high precision calculations.

5.3.3.1 Experiments

Experiments have been carried out, using the lattice volumes shown in table 5.3, to explore communication traffic scaling on 3-D and 4-D torus topology networks with identical numbers of processing nodes. Figure 5.12 shows the communication traffic per processing node for a 3-D and a 4-D torus topology machines.

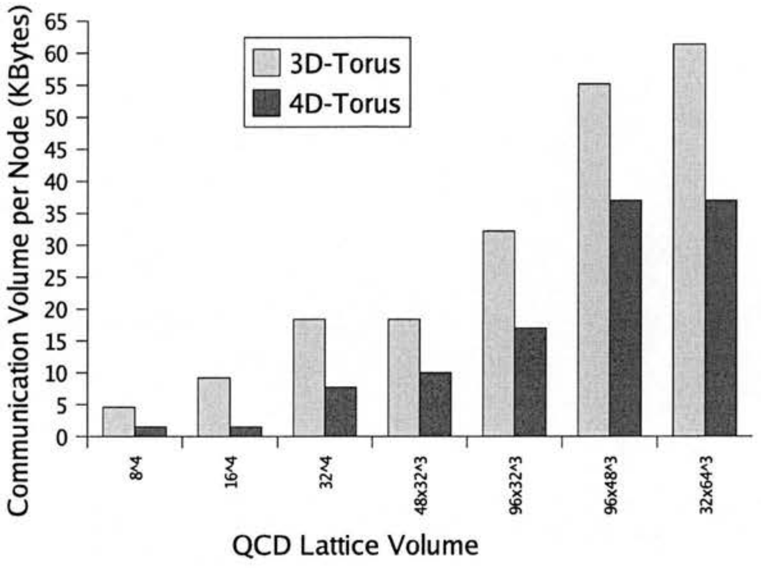


Figure 5.12: Communication volume scaling per processing node

The total number of processors is fixed, but because the underlying networks are different, the local volumes are different on the two machines (table 5.3). Figure 5.12 shows that the volume of communication traffic does not scale linearly with the number of sites mapped per processing node. Instead, the number of data packets transferred in each direction depends on the number of sites in the remaining three space-time directions.

Also, the implications of large communication traffic for nearest-neighbour communication times have been quantified in experiments. Figure 5.13 shows the ratio of a single iteration of nearest-neighbour communication time in a QCD calculation against the lattice QCD volume.

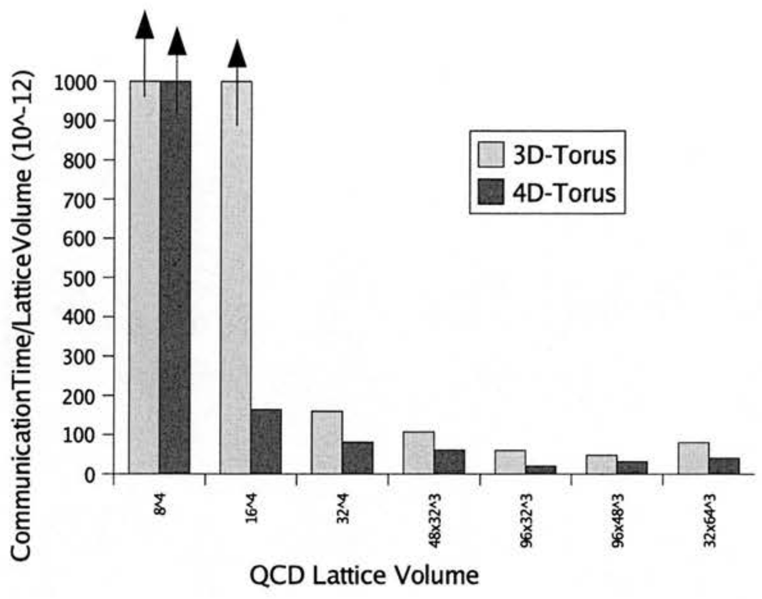


Figure 5.13: Nearest-neighbour communication efficiency with varying lattice volume over a 3-D and a 4-D torus network

For small lattice volumes, time to perform off-node communication for an individual lattice site is significantly greater than the communication times of larger lattice volumes. Moreover, a 3-D torus with bigger communication traffic (as shown in figure 5.13) is more affected by the communication times per QCD lattice site than a 4-D torus.

Although the overall QCD code performance is dominated by the intensive floating-point computation performance, a reduction in nearest-neighbour communication performance can affect the overall performance. In the QCD Conjugate Gradient kernel code, nearest-neighbour communication is overlapped with local computation. However, a number of local computation subroutines depend on the availability of the communicated data from all neighbouring directions. A substantial performance gain for the four-dimensional torus topology, as shown in figures 5.12 and 5.13, confirms the association of QCD code performance with the underlying network topology.

5.3.4 Collective Communication Operations

Analytically, Bluegene/L will have an advantage over QCDOC because it has a separate tree network. However, since collective communications are few and far between, and the QCD software does not overlap them with on-going local communications and on-chip computations, the overall difference is likely to be small. In section 4.5.1, it was shown that for up to 10K+ processing node systems, the performance of hardware-enhanced "pass-through" global sums compares well with binary tree-based floating-point reduction. Considering the cost-effectiveness of the system, a separate network for global communications is not feasible for a 10 K processing node QCDOC system. The Bluegene/L system is a 64K node supercomputer and a global tree is included for enhancing potentially overlapping point-to-point and collective communications. Table 5.4 compares the characteristics of the two systems for a floating-point global reduction operation.

Features	QCDOC	Bluegene/L
Network	6D Torus	Binary tree
Bit manipulation	In CPU	Bit-serial adder
Mechanism	4D Ring	2 Round-trip to Root
CPU Overhead	Must involve in FP addition	Potentially completely automatic
Communications	$4\ (N_x+N_y+N_z+N_z)\ /\ 2$	$4 * log_2(N_x*N_y*N_z*N_z)$

Table 5.4: QCd global sum computation mechanism on QCDOC and Bluegene/L machines

In short, the Bluegene/L and QCDOC processing node architectures and the overall design logic have a number of similar features. However, there are a number of differences, mainly the design component configurations, their operating clock frequencies and the interconnection network topologies. One of the design targets for the research presented in this thesis was to emulate these two machines using parameterised HASE simulation models. HASE QCDOC model's parameterised entities are capable of running the Bluegene/L configurations. But, for parallel code, running performance experiment also involve modifying the way in which a parallel software is distributed and mapped onto a hardware. Software mapping and scaling is not straightforward when the two architectures have different network topologies, especially for an explicit message passing software like QCD. The parameterised hardware-software co-simulation scheme used for this research provided a means to design and implement a simulation metamodel for HASE QCDOC and Bluegene/L machines for parallel QCD code performance experiments.

5.4 Simulation Metamodelling in HASE

Metamodelling concepts in computer simulations revolve around reproducing computer-generated simulation models by reusing their common characteristics and domains. A metamodel is the one that provides a basis for creating, describing or instantiating other simulation models. The complexity of the simulation studies increases with the complexity of the real systems, and there is an increasing need for tools and methods that facilitate flexible and efficient generation of simulation models. Like complex real and embedded system designs, component reuse is the key to this process. Syntactically and semantically correct models can be created by the metamodelling approach, by utilising and creating generic metamodels for existing and new systems. Within the HASE framework, a number of component reuse practices exist. HASE design templates and the multi-frequency extensible clock mechanism [MIA02] are examples. The metamodelling method employed in the HASE for creating the QCDOC and Bluegene/L simulation models is largely influenced by the simulation and modelling practices in SOC based designs. Techniques used in the embedded and SOC design communities are driven primarily by the fast time to market pressure and ever-increasing complexities of SOC design architectures.

Mathur and Prasanna [MP01] introduced SOC metamodelling by claiming that current state-of-the-art design tools and methodologies are not adequate to manage the design complexity of SOCs. Current design processes are based on an independent design flow for each architecture component and have not co-evolved with changing system designs and requirements. Programming models and design tools for each component are independently utilised to map an application, and system integration is performed later. System-wide performance analysis is typically a manual process. It involves the use of component specific simulators in isolation, and is tedious since each simulator has a different input/output interface. This approach results in sub-optimal design because multi-objective optimisation requires exhaustive traversal of a large design space. A hierarchical simulation framework for SOC architectures, similar to the hierarchical abstraction levels concept in HASE, was proposed by Mathur and Prasanna [MP01]. They argued that a unified simulation environment that provides performance estimates for a given application-to-architecture mapping is necessary. In the QCDOC simulation models, application workload configurations are made HASE parameters for in-depth and extensive performance evaluation studies within unified co-simulation models. During the simulation design stages, the hardware and software parameter space co-evolved to facilitate application-to-architecture mapping of parallel QCD code over alternate hardware design configurations.

The metamodelling approach has been employed in order to compare QCD application code performance over two different HASE supercomputer simulation models: QCDOC and Bluegene/L. In order to provide a unified framework for the efficient models generation and experimentation a HASE metamodel is created that generates a HASE QCDOC model as well as a HASE Bluegene/L model for the QCD application code simulations.

5.4.1 Component Reuse

Central to the SOC metamodelling concept is the notion of component reuse. By reusing an existing HASE model, it is possible to avoid re-modelling and re-designing overheads of new simulation models for the performance studies of the QCD code. Design provisions in the form of HASE parameters are essential in the existing models so that alternate system configurations can be simulated. This includes both the hardware parameters and the application software parameters.

Parallel QCD workload parameters are included in the HASE QCDOC models for load balancing and scalability experiments together with the hardware parameters for a wider design exploration. A combination of hardware and software parameters result in alternative QCDOC machine designs as well as QCD workload configurations. Likewise, generation of the Bluegene/L design configurations is attributed to these parameters, without which performance experiments would require tedious, off-line simulation test code generation. Another important factor in the performance evaluation experiments involving the QCDOC and Bluegene/L models is an understanding of the QCDOC workload mapping onto different interconnection network topologies. Bluegene/L has a distinct interconnection network topology, thereby requiring a different mapping technique for parallel QCD workload. The application-to-architecture mapping concept is exploited in creating a metamodel for the QCDOC simulation models and the Bluegene/L simulation models. Figure 5.14 illustrates the metamodel design approach for the SOC-based MPP architectures executing parallel QCD code.

HASE QCDOC and Bluegene/L simulation models are instances of the metamodel; further instances representing alternate design configurations of each machine can in turn be generated by altering the design parameters. The HASE QCDOC models were originally designed as simulation models that allow a wider design space to be explored. It was envisaged that new HASE simulation models would have to be developed by reusing the existing components to experiment with future generation MPP architecture. But the design flexibility of the HASE platform and its hierarchical abstraction levels allow the simulation designer to generate a QCDOC and a Bluegene/L simulation model by inputting and configuring a set of parameters in the HASE metamodel without redesigning a new HASE Bluegene/L simulation model. One of the most useful feature is the common instruction set architecture: the PowerPC BookE enhanced architecture specifications [May94]. Thus, the contents of instruction memory, the optimised parallel QCD benchmark kernel (in PowerPC assembly code), is used in original form in comparison experiments without any off-line code modification.

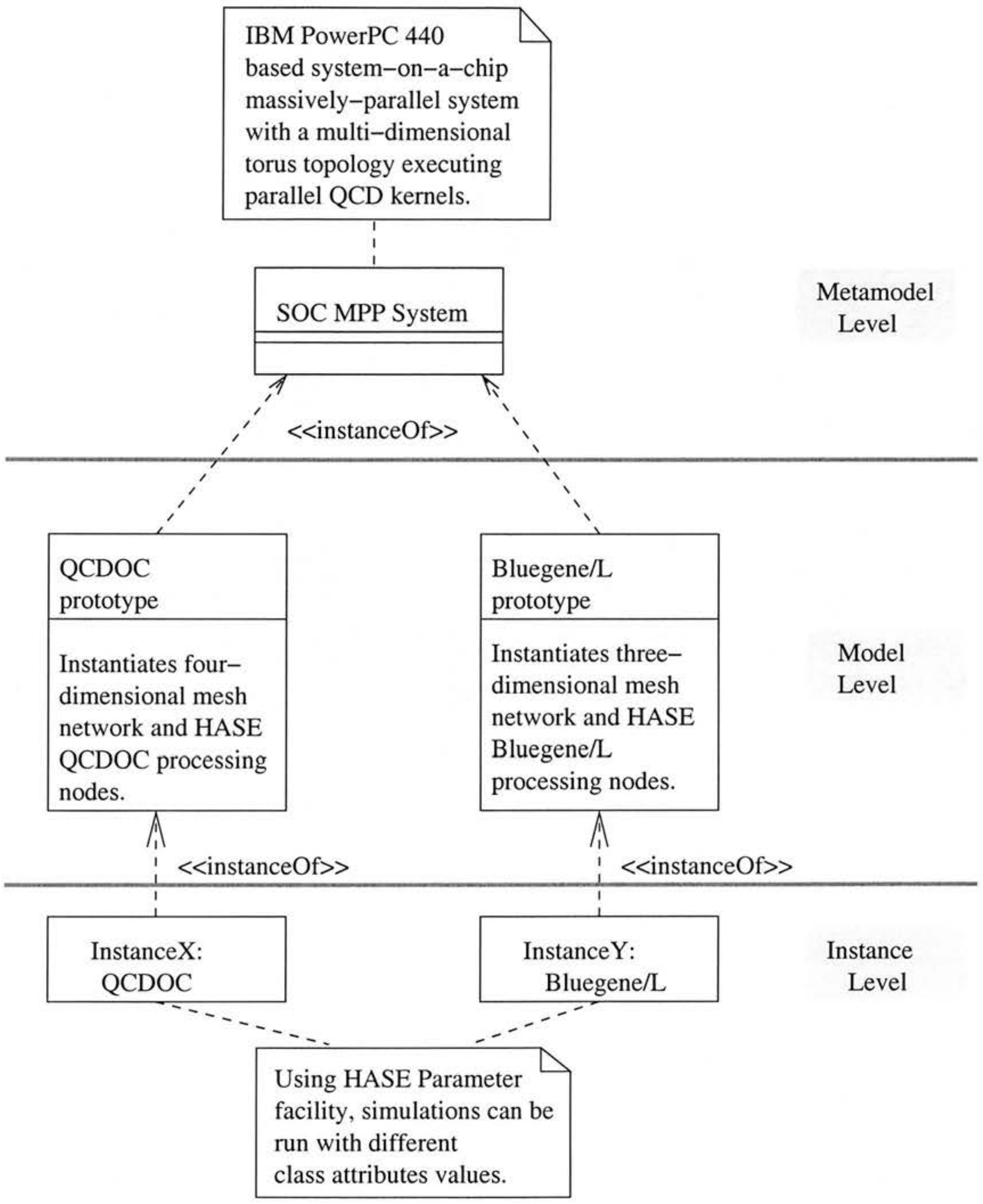

IBM PowerPC 440 based system–on–a–chip massively–parallel system with a multi–dimensional torus topology executing parallel QCD kernels.
SOC MPP System
Metamodel Level
<<instanceOf>>
QCDOC prototype
Instantiates four–dimensional mesh network and HASE QCDOC processing nodes.
Bluegene/L prototype
Instantiates three–dimensional mesh network and HASE Bluegene/L processing nodes.
Model Level
<<instanceOf>>
<<instanceOf>>
InstanceX: QCDOC
InstanceY: Bluegene/L
Instance Level
Using HASE Parameter facility, simulations can be run with different class attributes values.

Figure 5.14: Application-driven metamodelling for simulating scientific MPP systems in HASE

5.4.2 HASE Metamodel Design Features

In addition to a common PowerPC instruction set architecture, the on-chip memory hierarchies of the QCDOC and Bluegene/L processing nodes are quite similar. Yet, there are a number of differences in the two machines' architectures, most notably, the network topologies, control and data paths, relative on-chip clock frequencies, memory sizes and the prefetch logic. The flexible design entity parameterisation facility, the clock library and the template generation mechanism of the HASE platform eliminate the need to incorporate customised design solutions. Model-specific design solutions compromise component reuse and inevitably introduce overheads for incorporating them in other HASE simulation models. Together with the HASE facilities, a hardware-software co-simulation approach that permits alteration of QCD workload characteristics are important contributing factors for the QCDOC and Bluegene/L HASE simulation metamodel.

Figure 5.14 shows that the parameterised HASE simulation metamodel not only allows simulating two SOC MPP systems, but it also enables a user to run a range of simulation experiments with varying hardware configurations of the two simulated systems with the optimised parallel QCD test code.

5.4.2.1 The Host `ENTITY` Interface

The parallel QCD software or system workload parameters can be altered in the HASE model via the model's host entity, a design entity in the metamodel that serves as a front-end host interface of an MPP machine. This central control entity in the HASE metamodel, the SimMode `ENTITY`, provides an interface to change the number of sites mapped per processing node. A user can configure `t_sites`, `x_sites`, `y_sites` and `z_sites`, which represent the number of sites on a processing node in the `t`, `x`, `y` and `z` directions respectively. The QCD kernel in PowerPC assembly is stored in the instruction cache of the MMU `ENTITY`. This code has not been fixed for a single lattice configuration. According to the number of sites mapped per node, the code initialises and inputs several QCD subroutines depending on the local lattice configuration. Moreover, the PowerPC Entity Link Format (PPC ELF) [ppcb] and Application Binary Interface (ABI) [ppca] specifications have been exploited in conjunction with an understanding of the QCD code to simulate the behaviour of a range of lattice configurations per processing node. PPC ELF and ABI details are presented in Appendix C.

The application software initialisation process is emulated by the CPU `ENTITY` and the EDRAM `ENTITY`; it is equivalent to the initialisation process that takes place at the beginning of the lattice QCD code execution. By making lattice configurations as a generic HASE parameter, a single simulation model can be used not only to vary the hardware configuration of the system but also to change the workload properties. In the comparison experiments on the QCDOC and Bluegene/L simulation models, this easy-to-use and flexible mechanism removed the need for off-line tweaking and instrumentation of the application code, thereby constituting a significant feature of the HASE metamodel design.

5.4.2.2 PowerPC Core and Relative Clock Frequencies

A flexible and efficient mechanism for altering the clock frequencies as HASE parameters via a GUI interface is an important factor in designing of the metamodel. This is because a number of relative on-chip clock frequencies in the QCDOC and Bluegene/L ASICs are different. The HASE library clock mechanism explained in section 3.3.1 enables the generation of multiple clock frequency ratios such that no redesign effort for individual entities is required.

Considering the most straightforward scenario, i.e., generating the HASE QCDOC and Bluegene/L processing node default parameters' configuration, this requires (a) fastest clock frequencies for the PowerPC `COMPENTITY` components: the CPU, the Memory Management Unit (MMU) and the FPU and (b) a change in all on-chip memory modules clock frequency ratios with respect to the fastest clock. These on-chip clock frequency ratios are listed in table 5.5.

Design ENTITY	QCDOC	Bluegene/L
PowerPC Core	1:1	1:1
L2 Cache	1:1	1:2
L3 Cache	1:2	1:4
PLB (system bus)	1:3	1:4

Table 5.5: QCDOC and Bluegene/L ASIC components clock frequencies

The HASE library clock mechanism provides a mechanism to alter an ENTITY clock frequency through its reference PLL ENTITY parameter called ratio. For instance, before a simulation run, the L2 cache's reference PLL name has to be altered through its parameter window from 1:1 for the QCDOC ASIC to 1:2 for the Bluegene/L ASIC. Similarly, the PLB to processor clock frequency rate was changed from one-third to one-fourth.

5.4.2.3 On-chip Cache Configurations

In QCDOC, the L2 prefetching cache is 4x1-Kbit with two sets of two 1-Kbit prefetching registers. Bluegene/L has bigger prefetching caches, 2 K Bytes and 16 prefetching registers, each prefetching four times the size of a L1 cache line. Another difference is the bus width between L2 cache and the EDRAM. The bus width in the HASE model is a parameter which can be set to 256 bits for the Bluegene/L and 512 bits for the QCDOC experiments. Similarly, L2 memory size and configuration are also HASE parameters.

The QCDOC processor operates at a frequency of 500MHz. With this frequency and a relatively small L2 cache with a limited associativity, hits in L2 result in the availability of the data to the L1 cache in the next clock cycle. This low latency is a result of the L2 prefetching cache operating at the processor clock. This situation is slightly different in Bluegene/L. L2 caches have a large number of registers as compared to the QCDOC L2 cache, but the L2 latency is considerably higher in terms of processor clock cycles. It is estimated that data will take up to 7 processor clock cycles before it is available in the L1 data cache. The processor clock frequency is higher, 700MHz as compared to 500MHz in QCDOC.

Finally, the QCDOC L3 cache (EDRAM) has a high bandwidth as compared to the Bluegene/L on-chip EDRAM. There are two contributing factors to a small bandwidth: first, the bus width to the Bluegene/L EDRAM from L2 is 256-bit while it is 512-bit in the QCDOC ASIC. The second factor is the ratio with respect to the processor clock frequency; the EDRAM operates at one-fourth of the processor clock frequency in a Bluegene/L ASIC and at one-half of the processor clock frequency in the QCDOC ASIC. Together with the memory size and bus width parameters, the metamodel generates the QCDOC and Bluegene/L on-chip default and experimental cache configurations.

5.4.2.4 Communication Interface

Perhaps the most distinctive feature of the Bluegene/L processing node is the presence of an extra communication processor. The simplicity and regularity in the QCD communication processor does not need a sophisticated processor like a PowerPC core. In fact, something as simple as the SCU-type communication processor containing DMA registers serves as a special purpose communication processor. Hence, without modelling a communication PowerPC processor, the SCU to EDRAM interface is used for on-chip communication handling. In this case, the PLB slave can serve as the second L2 cache as it contains the prefetch logic and the prefetch registers. QCD communication data handling logic involves processor intervention at two stages: one for flushing send data packets to EDRAM and second to read data received from neighbouring nodes from EDRAM. Therefore, a separate communication L2 cache is not required to read data from the communication processor's L1 cache. Send data will be read from the EDRAM in exactly the same way as it is read on a QCDOC chip. All that remains is the PLB overhead, since Bluegene/L nodes are directly connected to the network unit. It was found through the HASE QCDOC simulation experiments that apart from the initial start-up latencies, the overall communication latencies are largely dominated by the off-chip data communication times.

A detailed analysis of the HASE metamodel communication interface is not presented here. This is because the internal details of the Bluegene/L torus and tree network are not available so precise comparisons with the QCDOC communication characteristics cannot be made. Since the communication latencies and buffer sizes are model design parameters, it is possible that additional changes may not need to be incorporated in the metamodel. The existing model, in that instance, would be sufficient to simulate the network interface behaviour of the two systems.

5.4.2.5 Network Topology

In addition to the communication processor, another difference between the QCDOC and Bluegene/L machines is the point-to-point communication network topology. QCDOC machine has a six-dimensional torus topology: for the HASE experiments a four-dimensional physical network or HASE `MESH4D` template was exploited. In the project Entity Description Language (EDL) file, the communication network topology can be altered by altering `MESH4D` to `MESH3D`. When the HASE project is built, the three-dimensional bi-directional torus is automatically generated by HASE.

Altering the network topology is presently different from altering all other parameters mentioned earlier. A parameter change in HASE does not require project rebuilding. A modification in the EDL file, on the other hand, needs recompilation as the entities instances generated by HASE take different forms. A four dimensional entity is identified by `MESH4DInstanceName._dim1._dim2._dim3._dim4` while entities instances generated by `MESH3D` are `MESH3DInstanceName._dim1._dim2._dim3`. These instances are generated statically in the `project.c` file. Thus, the metamodel generates parallel machines with different sizes and dimensions (mesh and torus topologies) using the HASE template facility with no redesign overhead.

Finally, global sum performance can be compared with existing HASE QCDOC models design parameters. Chapter 4 identified the limits of simulating a large number of entities within the HASE template generation mechanism. A high-level abstraction HASE QCDOC model, without detailed on-chip memory arrays, is used for the global sum performance estimates of the system. As explained in section 4.5, a high-level abstraction model extends the SCU `ENTITY` to emulate the 4-D torus network as well as a binary tree network. This extended model facilitates the collective communication performance analysis on the QCDOC's four-dimensional torus network with "pass-through" capability and on the tree network of the Bluegene/L system.

5.4.3 Constraints

The HASE parameterised metamodelling design approach provides an efficient and effective mechanism to perform a range of performance evaluation experiments for the parallel QCD application on the QCDOC and Bluegene/L simulation models. At the same time, it is recognised that implementing and extending the HASE metamodelling scheme to other parallel applications or application-driven simulations will require a number of other features to be considered and incorporated in the simulation design. In particular, the features that contribute to the application execution scaling and load balancing characteristics of an application code over the underlying parallel hardware configurations. For instance, the QCD application code runs in a single thread so there are no context switching in the QCD kernel code execution on the QCDOC and Bluegene/L simulation models. In many other systems, this may not be as straightforward. The metamodelling approach in these situations must take into account the operating system characteristics in generating the alternate simulation models.

In addition to the hardware and operating system characteristics of the modelled systems, the application load balancing and scaling properties may require additional factors/parameters to be incorporated in a metamodel. The communication pattern and volume are the key considerations for designing a metamodel. A parallel QCD application has a regular and homogeneous communication pattern; only the data structures of the boundary elements need to be communicated. Thus, when the underlying physical network topology of a system changes (different dimensional meshes and tori), it is straightforward to adapt it into the metamodel. For a number of other parallel scientific applications, this is not the case. Depending on the workload mapping per processing node and the underlying network topology, the metamodel may need to incorporate a significantly large number of features.

In the view of the above, a decision to adopt a parameterised metamodelling approach depends on getting the balance right between the:

1. cost of re-designing and re-modelling separate machines and their application code generation; and

2. the complexity of incorporating processing node and inter-node communication design parameters as well as the application code load balancing and scaling properties within a metamodel. This was found to be a cost-effective approach in studying the parallel QCD code performance on the QCDOC and the Bluegene/L machine simulation models.

5.5 Performance Comparison Experiments

Performance comaprison experiments on the simulation models of the QCDOC and Bluegene/L machines are conducted using workload configurations that represents the large practical lattice volumes. Table 5.6 how these lattice volumes would be mapped onto the available QCDOC and Bluegene/L machines.

Lattice Volume	QCDOC Configuration	QCDOC Sites per node	Bluegene/L Configuration	Bluegene/L Sites per node
32x32x32x32	16x8x8x8	2x4x4x4	16x16x16	2x2x2x32
48x32x32x32	16x8x8x8	3x4x4x4	24x16x16	2x2x2x32
96x32x32x32	16x8x8x8	6x4x4x4	48x16x16	2x2x2x32
96x48x48x48	24x12x6x6	4x4x8x8	48x24x24	2x2x2x48
32x64x64x64	4x8x16x16	8x8x4x4	32x32x32	2x2x2x32

Table 5.6: Practical lattice QCD calculations onto QCDOC and Bluegene/L machines

A 10-Teraflops QCDOC is expected to have 10K+ processing nodes and it can be reconfigured and partitioned in any combination of a four-dimensional torus network. Bluegene/L will be configured as three-dimensional, 64x32x32 processing nodes system. For the two largest practical lattice volumes, 96x48x48x48 and 32x64x64x64 sites per node, a QCDOC processing node will handle more sites per node than a Bluegene/L processing node. Nevertheless, the three-dimensional Bluegene/L machine has one less dimension for a four-dimensional calculation, therefore the smallest dimension of the calculation is stored over all Bluegene/L processing nodes.

Two sets of performance comparison experiments have been performed with the HASE QCDOC and Bluegene/L simulation models. The first set of experiments investigated the computation requirements of the QCD kernel for a range of workloads. These experiments were conducted with identical sets of workloads on the two modelled systems. The second set of experiments provided an insight into the QCD code execution for practical lattice configurations. The machine topology and lattice mapping in the second set of experiments are taken from table 5.6.

For both sets of experiments, the hardware parameters of the QCDOC and Bluegene/L are set according to their default configuration as specified in [BCC^{+}01] and [Tea02] respectively. This default configuration applies to the relative on-chip processing node clock frequencies, memory sizes, latencies and bandwidths, and the machine inter-connection network topologies.

5.5.1 Workload Scaling

QCD code is a floating-point computation intensive code. Every effort has been taken to maximise the on-chip memory bandwidth and to minimise the memory latencies. The Bluegene/L system targets the memory bandwidth and latency problems but not in a QCD specific way. In QCD, the ratio of memory access operations to the floating-point arithmetic operation is small as compared to many other scientific applications.

Experiments were performed to observe the relative advantage of the QCD memory hierarchy over the Bluegene/L memory hierarchy. With varying system workloads, it was established how the computation scales change with changes in workload. One would expect Bluegene/L to out-perform the QCDOC because it has been based on a relatively advanced processor, SOC components and operates at a faster clock frequency.

Figure 5.15 shows the workload scaling on the two systems.

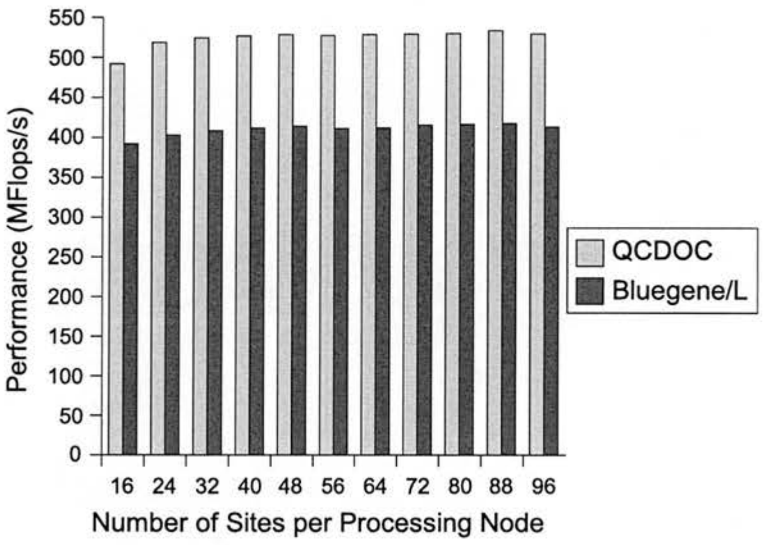


Figure 5.15: Parallel QCD workload scaling on QCDOC and Bluegene/L machine

For the same workload, a QCDOC node was able to finish the job more quickly as compared to a Bluegene/L node. However, the scaling behaviours of both systems are identical. These results highlight that a higher processor clock frequency (700 MHz compared to 500 MHz) and larger on-chip memory arrays do not necessarily result in a higher performance and efficiency of code execution. In fact, for the QCD application code, the relative on-chip memory latencies and on-chip memory bandwidths contribute more to fast execution times.

5.5.2 Efficiency

Performance comparison experiment presented in the previous section do not take into account the lattice volume for a QCD calculation; workload or the computation load (number of sites per processing node) was the focus. The practical lattice volumes of table 5.6 are compared on the two machines for estimating the achievable performance efficiency.

For the larger QCD lattice volumes, 96x48x48x48 and 32x64x64x64 lattice sites per processing node, the 64K node Bluegene/L executes less workload per processing than a 10K QCDOC, thereby finishing the calculations more quickly. The results presented in this section therefore compare the relative performance, by taking a ratio of the number of nodes available and number of sites mapped per processing node. The central aim of the performance comparison experiments is to establish parallel load balancing, efficiency and scalability of the parallel QCD code with respect to the processing node design configurations of the two systems — independent of the total number of processing nodes of two machines.

Figure 5.16 shows the execution times for a single iteration of the QCD kernel against the number of lattice sites mapped per node.

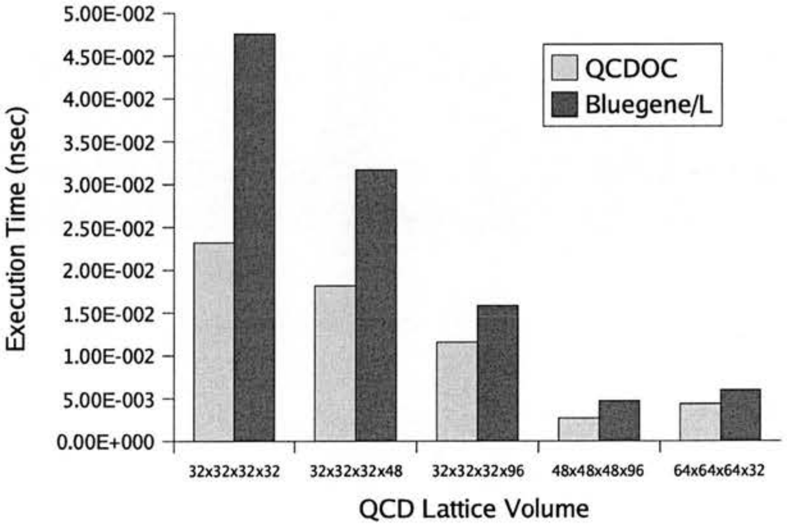


Figure 5.16: Execution time per lattice site

Figure 5.17 shows the ratios of workload distribution per node and total execution time between QCDOC and Bluegene/L for increasing lattice volumes.

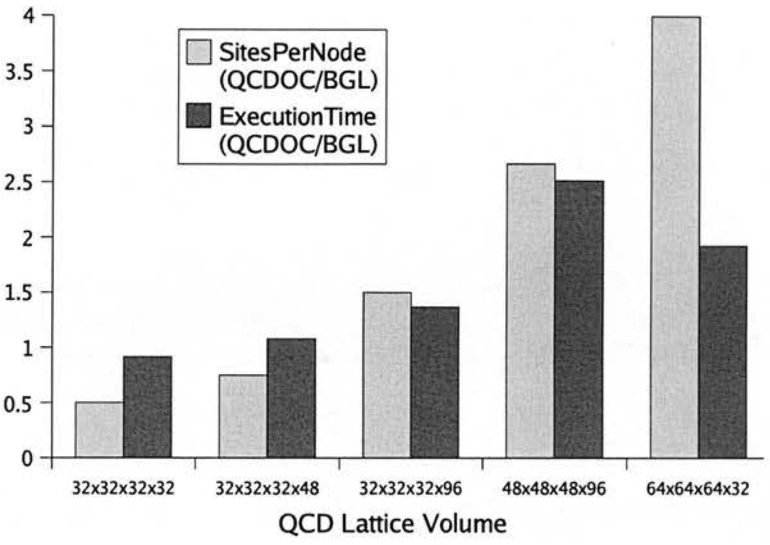


Figure 5.17: QCDOC and Bluegene/L (BGL) Performance

The main observation (from figure 5.16 and figure 5.17) is the difference in overall efficiency of the two systems: QCDOC machine handles parallel QCD code more efficiently than the Bluegene/L machine. Particularly, for 64x64x64x32 lattice, local lattice volume for a QCDOC processing node is four times greater than the local lattice volume of a Bluegene/L processing node, but the execution time on the former is only twice as long as the latter.

The results represented in figure 5.16 and figure 5.17 do not include the communication costs of the QCD application: the local communication cost and the global-sums communication costs. Global sums, like any other collective communication on a large dimension system, would suffer from the comparatively large number of nodes in the Bluegene/L system. In section 5.3.1, nearest-neighbour communication performance results were compared over three-dimensional and four-dimensional torus configurations with identical communication latencies. It would be useful however, to observe the scaling of communication cost with respect to communication volume in the nearest-neighbour communication operations on the two different machines.

Bluegene/L's communication processor is expected to reduce the nearest-neighbour communication startup times by off-loading the computation CPU — this however would require some modification of the existing QCD kernel. Moreover, on-chip communication data transfers on the Bluegene/L will be more efficient since the communication node's L2 cache is directly connected to the network unit — QCDOC connects them via the system bus with multiple masters. Nevertheless, simulation experiments with variable on-chip and off-chip communication data transfer latencies (section 4.4.1) demonstrated that the nearest-neighbour communication times predominantly depend on off-node data transfer latencies.

The effects of frequent nearest-neighbour communication operations are not explored on the Bluegene/L and the QCDOC simulation models. In case of the QCDOC machine modelling, access to the confidential design documents [BCC$^+$02b] provided precise timing information over the off-node links and the custom communication protocol. A comprehensive description of Bluegene/L off-node point-to-point communications is not available at the time of writing. In the absence of precise timing information, and information about the storage and buffering characteristics of the communication unit, the comparison experiments were not possible. The HASE model communication unit allows a range of buffer and storage options to be specified together with a provision to specify send and receive latencies. It is therefore possible, when the Bluegene/L communication details are known, to compare the nearest-neighbour communication performance on the two systems.

Similarly, the metamodelling approach allows for the Bluegene/L collective communication network to be simulated as a HASE model. The SCU `ENTITY` contains parameters that specify six options for global sum. One global sum option is the one implemented for the Bluegene/L architecture: the two-round trip to tree global sum using a serial adder in the communication unit implementation. Hence, it is possible to estimate global sum times on the two machines. But, in addition to global sum options, another requirement critical in global sum estimates is the send and receive communication latencies over the Bluegene/L network.

The Bluegene/L tree network architecture and functioning of its ALU are known at the time of writing this thesis, but the communication latencies are not available. Therefore, for the lattice configuration outlined in table 5.6, the double-precision floating-point reduction times cannot be compared between the QCDOC machine model and the Bluegene/L machine model. Like nearest-neighbour experiments, once these timings are available it would be a matter of altering the input parameters to the model. It would not require re-designing or re-coding the simulation model in HASE.

5.6 Summary

Application-driven metamodelling and the Bluegene/L architecture have been introduced along with an overview of Bluegene/L architecture's similarities to and differences from the QCDOC machine. The advantages of hardware and software parameters of the HASE model have been described with an introduction to the metamodelling concept and its importance in SOC design exploration. The benefits of the metamodelling approach include component reuse, flexibility and efficiency in generating complex simulation models. This scheme allowed parallel QCD performance comparison experiments on the HASE simulation models simulating QCDOC and Bluegene/L design configurations. Furthermore, HASE simulation results have confirmed that a cost-effective, custom-built system achieves a better fraction of peak performance compared to a general-purpose supercomputer, even though the two systems are based on similar technologies.

Chapter 6

Conclusions

The purpose of this chapter is to summarise the contents of the thesis and to identify the notable contributions. Suggestions are outlined for future directions of this research.

6.1 Thesis Summary

Theoretical peak performance of a parallel computer is quite straightforward to determine; one can obtain it by multiplying the number of processing nodes in a parallel system by the theoretical peak performance of its processing nodes. However, the achievable performance depends on the characteristics of computation and communication components of a parallel system, most notably, the memory hierarchy and interconnection network latencies. Furthermore, the *achievable* performance of an application is attributed to its workload characteristics and the *sustainable* performance depends on the application software load balancing and scaling on a parallel system with varying problem sizes.

Thus, in order to investigate the performance limiting factors and to explore the design and performance search space of a parallel machine, it is essential to incorporate a number of system hardware and application software properties in performance studies. UKQCD computer simulation research is based on an application-driven simulation approach to study performance of a recent, application-specific supercomputer. In order to conduct performance experiments, the HASE UKQCD computer (QCDOC) parameterised simulated models are created. The application software is parameterised and simulated along with the parameterised hardware model of the system. Thus, along with the behaviour modelling of the hardware components, system software, the optimised QCD application code's problem size is made a model parameter. Hardware parameters of the QCDOC model enabled a wider design exploration of the system architecture, while the ability to alter problem size during a simulation allowed for the scalability studies of the application workload.

The hardware-software co-simulation approach paved the way for not only QC-DOC performance characteristics exploration but also Bluegene/L's design space exploration, QCDOC successor supercomputer, through simulation metamodelling. The metamodel scheme permits efficient generation of HASE simulation models for parallel QCD performance studies with different design configurations by reusing the existing design components. Bluegene/L is a System-On-a-Chip (SOC) based supercomputer like QCDOC. Although, unlike the QCDOC system, it has not been optimised for the QCD calculations, it shares many of its design features, particularly the on-chip memory hierarchy and the instruction set architecture with the QCDOC system. The optimised QCD application kernel written in PowerPC assembly language is the benchmark software for comparing the two systems performance. Moreover, it is argued that the rapid and flexible prototyping facilities of HASE for parameterised model development and the extensions made to the HASE during the course of this research are essential for the metamodel creation. As a result, SOC-based massively-parallel processor (MPP) HASE models are generated efficiently for parallel QCD code performance comparison experiments.

It is believed that the application-driven simulation metamodelling approach is useful in scientific supercomputing research since these supercomputers are mostly optimised for and dedicated to a few highly-demanding applications. The main drawback of the cycle-accurate, instruction-level simulation scheme proposed in this thesis is the wall clock run time of the simulation and the integration of workload simulation within the architectural simulation. Additional design efforts are required for simulating other workloads.

6.2 Key Contributions

Computer architecture simulation frameworks have been employed in research and innovation of small- and medium-scale parallel systems hardware, which are designed and optimised for commercial applications. Software simulation techniques provide trade-offs between accuracy, efficiency and flexibility. At the time of writing this thesis, dedicated software simulations are created from scratch for scientific, high-end supercomputer architecture performance studies: ASCI Q [PKP03] and BGLSim [MCS$^+$03]. The UKQCD computer simulation research, in contrast, employs the HASE platform to conduct performance studies of a recent scientific supercomputer. During the research, useful new mechanisms as well as limitations of the HASE platform for the high-end supercomputer simulations have been identified. Extensions have been made to the HASE to facilitate the design and use of the QCDOC system simulation model.

The primary design target was accuracy of the simulation models, since the precise design details of the architecture were made available. Prototypes of the QCDOC, SOC-based processing node MPP system, have been designed in the HASE that emulate the correct functional and timing behaviour of the existing system. In addition to prototyping the system, the simulation model is parameterised, such that the performance limiting factors are exposed as a result of experiments with varying hardware configurations.

The QCDOC system has been specifically built for QCD applications, especially for application domains which are important for Columbia and UKQCD collaboration researches. A customised but highly optimised QCD kernel is the basis for real system prototype performance testing as well as the HASE simulation model performance exploration. Because of the long simulation times associated with the VHDL simulations,[1] the highly optimised QCD kernel was primarily used to model a small problem size. At the same time, this optimised kernel is intended to provide library routines for QCD application code calculations as it represents the most frequently-used and time-consuming component of a QCD application. Hence, this

[1]QCDOC design team prototypes.

optimised QCD kernel is not restricted to a single problem size; it is parameterised for a number of problem and machine sizes. In order to perform computation and communication load balancing and scalability experiments, the application kernel was parameterised in HASE QCDOC model. The workload parameters of the model have allowed the workload size to vary for the simulation experiments — removing the need for time-consuming off-line code generation. Addition of application workload parameters in HASE model required a thorough understanding of the application code distribution and mapping onto a parallel architecture and the QCDOC operating system characteristics.

Parameterised hardware and software simulation in HASE allowed a range of experiments to be performed on the QCDOC simulation models including: memory hierarchy characteristics and QCD frequent nearest-neighbour communication and global sum performance exploration. Results from these experiments confirm that the QCD-specific design optimisations, the Prefetch EDRAM Controller (PEC) and the four-dimensional communication topology along with the Serial Communication Unit (SCU), contributes to the QCD performance efficiency. Yet, alternate combinations of parameters can further enhance the achievable performance. The level 1 data cache line size and an unbalanced QCD workload distribution are found to be the main performance bottlenecks.

A parameterised HASE simulation model allows for experiments by altering the system configuration parameters. Simulation with multiple global sum options provided scaling behaviour of floating-point global sums of QCD calculations. Also, it was found that the large communication latencies inherent in commercial networks and routing schemes can have a significant impact on small-sized QCD communication packets. In addition to the performance evaluation experiments, simulation runs with ideal and hypothetical system configurations are executed. Without these experiments, it would be impractical to predict the precise behaviour of the QCD application and to quantify the hardware limitations of the system. Experiments with varying workload sizes enabled understanding of behaviour of the hardware components, computation to communication ratios and parallel code scalability.

Finally, the Bluegene/L and QCDOC HASE models for the QCD applications were generated by employing an application-driven simulation metamodelling scheme. Metamodelling methodology enables an efficient and seamless creation of HASE simulation models for parallel QCD code performance experiments based on different hardware characteristics and interconnection network topologies. The HASE hardware-software co-simulation and component reuse mechanisms have been exploited for the HASE simulation metamodel design. Performance comparison experiments have been conducted on the HASE QCDOC and Bluegene/L simulation models. These comparison experiments provided an insight into the performance characteristics of the QCD code on a special purpose QCDOC machine and a non-QCD-specific system, the Bluegene/L. It was confirmed that the balance between processor speed and on-chip memory latencies and bandwidth, offered by the QCDOC processing node, favour QCD application code execution.

6.3 Future Directions

Simon *et. al.* [SMK$^+$03] suggested that current and future state-of-art scientific High Performance Computing (HPC) research relies on a sound understanding of the behaviour of the underlying system for an application code execution. Computer simulations are believed to be the only cost-effective, flexible and yet precise mechanism that can enable achievement of these targets. Nonetheless, as Dongarra *et. al.* [DSSS03] concluded, the current techniques of system simulations are not appropriate for a simulation designer to explore and to benchmark a current, or a future system with certain degree of speed, flexibility and accuracy. The research presented in this thesis demonstrated an approach to hardware-software integration in modelling and parameterisation of system design. The experience of this research provided an insight into limitations and short-comings for future research projects within the HASE framework and for a wider scientific multiprocessor simulation.

First, analysis of enormous simulation trace files require additional capabilities in the HASE platform. Multiprocessor simulations generate huge trace files, and filtering information from these traces is currently a manual task. This is currently a responsibility of the user at coding time, to selectively store useful information in separate files. A selection mechanism in the HASE trace generation mechanism would allow useful information to be captured and filtered at the user's request at simulation run time. The overall behaviour of selective input and output parameters can then be compared and contrasted for successive simulation runs. However, the filtering mechanism should not compromise the design flexibility and freedom offered by the HASE, by imposing restrictions on the way in which parameters are included and altered.

Second, existing design validation capabilities within HASE need to be extended. Presently, the HASE system validates the `LINK` consistency and memory `ARRAY` inputs in `.mem` files. Two HASE entities connected via a `LINK` can only declare a `PORT` connected via the same `LINK` definition. Similarly, at the simulation run time, when the physical memory file is read, HASE identifies and prints warning messages if an invalid data structure is found. Apart from the `LINK` and memory `ARRAY` validation, design validation is a simulation designer's responsibility. Typically, run-time errors are identified during a simulation run, during animation and via timing diagram inspections.

Third, as Coe [Coe00] suggested in his PhD thesis on distributed shared-memory multiprocessor which are smaller in size and dimensions than the QCDOC systems, Hase++, the discrete event simulation library of the HASE platform, should be ported and run on a parallel machine. Bagrodia *et. al.*[BDP01] suggested several parallel discrete-event simulation approaches for parallel applications. Alexander *et. al.* [ABB$^+$01] exploit extreme-scale simulation techniques, which presently focuses on simulating large interconnection networks. It does not however, capture and explore node-level processing and memory design optimisations.

Fourth, the HASE platform needs scalable and customisable visualisation capabilities for multiprocessor simulation models. Currently, C++ QT APIs provide the GUI and animation facility in the HASE. The on-screen complexity of the MPP QCDOC model, at the processing node level and for a larger system, require a scalable visualisation mechanism for the HASE template-based entities. A zoom-in and zoom-out facility should exist in conjunction with the current hierarchical user interface, such that entities can be hidden and expanded by a user through the HASE GUI. Currently, developments are under way to separate the graphical view of a HASE model from its behavioural contents. This would allow additional graphical features to be incorporated in a HASE model without the restrictions imposed by the behavioural code and the simulation libraries.

Fifth, the scope and advantages of the metamodelling techniques for other scientific applications and architectures can be explored. Metamodelling is widely used in embedded system architecture exploration. However, application of these techniques requires a thorough understanding of a parallel application code mapping and its computation and communication characteristics. Furthermore, a detailed knowledge of the architectural configurations of alternate simulation models would be necessary to conduct high fidelity performance studies. Extensive multi-disciplinary research efforts are required to establish whether, for a given application and architecture, re-design overheads are greater than the parameterised model design overheads.

Finally, analysis of enormous trace files generated as a result of multiprocessor simulations require statistical methods to identify and to report statistically significant parameters. Currently, the timing simulators report execution time and functional simulators focus on representing hit/miss cache ratio or busy/idle network time. But they fail to capture global variations in program behaviour and performance. The end result may be a function of a number of system variables and application of statistical algorithm would enable a comprehensive analysis and understanding of simulation results. At the microprocessor design level for instance, the Sampling Microarchitecture Simulation (SMARTS) [WWFH03] addressed the need for improved simulation accuracy and performance by applying statistical sampling to microarchitecture simulations.

It is envisaged that the design optimisation and exploration of supercomputers would rely on effective and efficient computer generated simulations as is the case in researches in other disciplines. No doubt, supercomputers would be needed for these simulation studies and the current issue, as concluded in 2003 Conference on High-Speed Computing:

> "Computational scientists have become quite expert in using high-end computers to model everything except the systems they run on."

By conducting an in-depth performance analysis of the QCDOC machine through its HASE simulation model and by predicting parallel QCD code performance scaling on a future-generation supercomputer using a flexible and efficient metamodelling scheme, the UKQCD computer simulation project demonstrates that scientific supercomputing modelling issues and challenges can be addressed within a simulation framework by aggregating and by exploiting existing simulation strategies and techniques.

Appendix A

Benchmark Code

An optimised QCD kernel and a custom operating system were developed by the QCDOC design team at the Columbia University to benchmark and to test the initial QCDOC prototypes as well as the physical machines. The QCDOC operating system features are incorporated and simulated along with the behaviour modelling of HASE hardware design entities. The benchmark kernel executable image is included as an input to the HASE models (in form of HASE memory ARRAY).

A.1 QCD Kernel

The kernel comprises the most time consuming part of a QCD application code called the "Dslash". The "Dslash" computation in turn consists of four computation functions and two nearest-neighbour communication operations. The function signatures are below:

```
void QCDOC_ChDecom      (void *psi,void *len,void *tab_bwd)
void QCDOC_ChDecom_hsu3 (void *psi,void *Ucb ,void *len,void *tab_fwd)
void QCDOC_ChRecon_su3  (void *chi,void *Un  ,void *chib,void *len)
void QCDOC_ChRecon_add  (void *chi,void * chif,void *len)
```

The addresses the above functions' parameter pointer points to remain the same if the underlying workload of the system changes. However, the values they point to depends on the local lattice volume (or workload), i.e., lattice volume mapped per processing node. For instance, the `len` variable is calculated by taking the half of the product of the number of sites in the four space-time directions.

In order to take the maximum advantage of the custom designed features of the QCDOC hardware, the above-listed kernels have been hand-coded. Code listing of `QCDOC_ChDecom` (the smallest method) in PowerPC assembly language is provided below:

```
QCDOC_ChDecom:
        or    %r9 , %r3 , %r3
        la   %r1,       -928(%r1)
        or    %r16 , %r4 , %r4
        li   %r26,0
        addi  %r14 , %r1 , 864
        li   %r27,32
        or    %r19 , %r5 , %r5
        li   %r28,64
        dcbt    %r26,%r19
        lwz   %r16,      0(%r16)
        or.  %r16 , %r16 , %r16
        bf gt,    lab0
        addi  %r24 , %r9 , 96
        lwz   %r13,      0(%r19)
        addi  %r14 , %r1 , 864
        dcbt    %r26,%r9
        dcbt    %r27,%r9
        dcbt    %r28,%r9
        dcbt    %r26,%r24
        dcbt    %r27,%r24
        dcbt    %r28,%r24
        lfd  0,         0(%r9)
        lfd  1,         8(%r9)
        lfd   2,         16(%r9)
        lfd   3,         24(%r9)
        lfd   4,         32(%r9)
        lfd   5,         40(%r9)
        lfd   6,         48(%r9)
        lfd   7,         56(%r9)
        lfd   8,         64(%r9)
        lfd   9,         72(%r9)
        lfd   10,        80(%r9)
        lfd   11,        88(%r9)
        lfd   12,        96(%r9)
        lfd   13,        104(%r9)
        lfd   14,        112(%r9)
        lfd   15,        120(%r9)
        lfd   16,        128(%r9)
        lfd   17,        136(%r9)
        lfd   18,        144(%r9)
        lfd   19,        152(%r9)
        lfd   20,        160(%r9)
        lfd   21,        168(%r9)
        lfd   22,        176(%r9)
        lfd   23,        184(%r9)
        mtctr   %r16
        b          lab1
lab1:
        addi  %r20 , %r13 , 0
        la   %r19,       4(%r19)
        fsub   24 , 12 , 7
        addi  %r9 , %r9 , 192
        fadd   25 , 13 , 6
        lwz   %r13,      0(%r19)
```

```
fsub   26 , 18 , 1
dcbt    %r27,%r19
fadd   27 , 19 , 0
stw  %r9,       16(%r14)
fsub   28 , 14 , 9
addi %r21 , %r13 , 0
fadd   29 , 15 , 8
stfd 24,        0(%r20)
fsub   30 , 20 , 3
la  %r19,       4(%r19)
fadd   31 , 21 , 2
stfd 25,        8(%r20)
fsub   24 , 16 , 11
lwz  %r13,      0(%r19)
fadd   25 , 17 , 10
stfd 26,        16(%r20)
addi %r22 , %r13 , 0
la  %r19,       4(%r19)
fsub   26 , 22 , 5
stfd 27,        24(%r20)
addi %r24 , %r9 , 192
lwz  %r13,      0(%r19)
fadd   27 , 23 , 4
stfd 28,        32(%r20)
addi %r23 , %r13 , 0
la  %r19,       4(%r19)
fsub   28 , 12 , 6
stfd 29,        40(%r20)
lwz  %r13,      0(%r19)
fsub   29 , 13 , 7
stfd 30,        48(%r20)
```

```
dcbt    %r26,%r24
fadd  30 , 18 , 0
stfd 31,        56(%r20)
la  %r16,       -1(%r16)
fadd  31 , 19 , 1
stfd 24,        64(%r20)
fsub  24 , 14 , 8
stfd 25,        72(%r20)
fsub  25 , 15 , 9
stfd 26,        80(%r20)
dcbt    %r27,%r24
fadd  26 , 20 , 2
stfd 27,        88(%r20)
fadd  27 , 21 , 3
stfd 28,         0(%r21)
fsub  28 , 16 , 10
stfd 29,         8(%r21)
fsub  29 , 17 , 11
stfd 30,        16(%r21)
dcbt    %r28,%r24
fadd  30 , 22 , 4
stfd 31,        24(%r21)
addi %r24 , %r24 , 96
fadd  31 , 23 , 5
stfd 24,        32(%r21)
fsub  24 , 12 , 1
stfd 25,        40(%r21)
fadd  25 , 13 , 0
stfd 26,        48(%r21)
dcbt    %r26,%r24
fadd  26 , 18 , 7
```

```
stfd 27,        56(%r21)
fsub  27 , 19 , 6
stfd 28,        64(%r21)
fsub  28 , 14 , 3
stfd 29,        72(%r21)
fadd  29 , 15 , 2
stfd 30,        80(%r21)
dcbt    %r27,%r24
fadd  30 , 20 , 9
stfd 31,        88(%r21)
fsub  31 , 21 , 8
stfd 24,         0(%r22)
fsub  24 , 16 , 5
stfd 25,         8(%r22)
fadd  25 , 17 , 4
stfd 26,        16(%r22)
dcbt    %r28,%r24
fadd  26 , 22 , 11
stfd 27,        24(%r22)
fsub  27 , 23 , 10
stfd 28,        32(%r22)
fsub  28 , 12 , 0
stfd 29,        40(%r22)
fsub  29 , 13 , 1
stfd 30,        48(%r22)
lfd  0,          0(%r9)
fsub  30 , 18 , 6
stfd 31,        56(%r22)
lfd  1,          8(%r9)
fsub  31 , 19 , 7
stfd 24,        64(%r22)
```

```
lfd  6,         48(%r9)
fsub  24 , 14 , 2
stfd 25,        72(%r22)
lfd  7,         56(%r9)
fsub  25 , 15 , 3
stfd 26,        80(%r22)
lfd  2,         16(%r9)
fsub  26 , 20 , 8
stfd 27,        88(%r22)
lfd  3,         24(%r9)
fsub  27 , 21 , 9
stfd 28,        0(%r23)
lfd  8,         64(%r9)
fsub  28 , 16 , 4
stfd 29,        8(%r23)
lfd  9,         72(%r9)
fsub  29 , 17 , 5
stfd 30,        16(%r23)
lfd  4,         32(%r9)
fsub  30 , 22 , 10
stfd 31,        24(%r23)
lfd  5,         40(%r9)
fsub  31 , 23 , 11
stfd 24,        32(%r23)
lfd  10,        80(%r9)
stfd 25,        40(%r23)
lfd  11,        88(%r9)
stfd 26,        48(%r23)
lfd  12,        96(%r9)
stfd 27,        56(%r23)
lfd  13,        104(%r9)
```

```
        stfd  28,         64(%r23)
        lfd   14,         112(%r9)
        stfd  29,         72(%r23)
        lfd   15,         120(%r9)
        stfd  30,         80(%r23)
        lfd   16,         128(%r9)
        stfd  31,         88(%r23)
        lfd   17,         136(%r9)
        lfd   18,         144(%r9)
        lfd   19,         152(%r9)
        lfd   20,         160(%r9)
        lfd   21,         168(%r9)
        lfd   22,         176(%r9)
        lfd   23,         184(%r9)
        bdnz       lab1
lab0:
        la   %r1,         928(%r1)
        blr
```

Like `QCDOC_ChDecom`, other methods branch and jump to several other labels within a signal call. The number of iterations is determined statically. In other words, the number of iterations do not depend on a value computed dynamically, instead, it is computed at the lattice initialisation stage.

A.2 Object File Format

Labels in the QCDOC kernels have been named as lab*n*, where *n* can be a number from 0, 1, ... The duplication of the labels was avoided by using the complete executable image of the kernel rather than individual files for each subroutine. An executable image format in the instruction memory for the `QCDOC_ChDecom` is shown in figure A.2.

```
0000ca20 <QCDOC_ChDecom>:
    ca20:        7c 69 1b 78     mr       r9,r3
    ca24:        38 21 fc 60     addi     r1,r1,-928
    ca28:        7c 90 23 78     mr       r16,r4
    ca2c:        3b 40 00 00     li       r26,0

..............

    cad4:        7e 09 03 a6     mtctr    r16
    cad8:        48 00 00 08     b        cae0 <lab1>
0000cae0 <lab1>:
    cae0:        3a 8d 00 00     addi     r20,r13,0
    cae4:        3a 73 00 04     addi     r19,r19,4

..............
```

Figure A.1: Contents of the `.obj` file in human-readable format

A.3 Input Format for the HASE Model

A HASE EDL construct `INSTR` has been employed for defining the instruction set for the model. A subset of the PowerPC instruction set has been implemented. Several instruction mnemonics in PowerPC instruction set are keywords in C/C++ programming language and are not allowed to be used in a program. Therefore, the lower case letters of PowerPC instruction mnemonics and labels are converted to upper case. Furthermore, an "0x" is appended to the addresses and offsets to represent them in C/C++ compliant hexadecimal values. This process is automated by developing a Java utility that converts input in the PowerPC object code format to the HASE QCDOC model conformant. The final format of the instruction which is loaded to instruction cache is shown in figure A.3.

```
0x0000ca20 LABEL QCDOC_CHDECOM:
0xca20 MR R9,R3
0xca24 ADDI R1,R1,-928
0xca28 MR R16,R4
0xca2c LI R26,0

..............

0xcad4 MTCTR R16
0xcad8 B CAE0<LAB1>
0x0000cae0 LABEL LAB1:
0xcae0 ADDI R20,R13,0
0xcae4 ADDI R19,R19,4

..............
```

Figure A.2: Contents of the instruction cache in the `.mem` file

Appendix B

Input Parameters and Output Variables

The design space of the QCDOC architecture was explored by altering and experimenting with a range of parameters included in the HASE models (i.e. by parameterising the HASE design entities). HASE provides an efficient and flexible mechanism to alter and to include entities and their parameters in a model. For conducting experiments, the HASE parameter window interface allows a user to alter a value or set of values for successive simulation runs.

B.1 `ENTITY`s and their Parameters

Theoretically, a number of parameters can be included in a parameterised model. However, in the QCDOC simulation model design, parameterisation and result gathering in the model are focused on (a) custom-built QCDOC components, (b) entities that are involved in code execution and on-chip data movements and (c) factors that influence code scalability and load balancing.

B.1.1 The PowerPC `COMPENTITY`

This compound entity (COMPENTITY) has three sub-entities: Central Processing Unit (CPU), Memory Management Unit (MMU) and an auxiliary Floating Point Unit (FPU).

B.1.1.1 The CPU `ENTITY`

Parameter	Description	Possible Values
`InstrFetchPerCycle`	the number of instructions fetched from the MMU in each clock cycle	1-4
`qcdoc_kernel`	test routine name	QCD Kernel routines
`CommType`	Direction of the Nearest Neighbour Communications	None, Forward or Backward
`PrefetchValues`	Whether prefetch values in the parameter registers	1-4

Table B.1: CPU parameters

Table B.1 lists the parameters a user can alter for a simulation run. The results filtered by a CPU entities are:

1. PowerPC instructions issued per clock cycle.
2. Instruction movement from a pipeline stage to other with source register values and effective address (load store instructions).
3. Instructions committed along with value of destination register.
4. Number of instructions in load-store, simple integer and complex integer pipelines.
5. Status of execution pipelines: blocked or not.

After a simulation run, the `$report` section of all entities in a HASE model are executed. The `$report` section of the CPU `ENTITY` prints the execution time and number of instruction issued and committed along with the `unique_identity` of a CPU. In a four dimensional mesh, each instance of an entity has a unique identifier.

Parameter	Description	Possible Values
`InstrFetchPerCycle`	Number of instructions from CPU every clock cycle.	1-4
`IdealMode`	Assume load and store hits in Level 1 cache	boolean

Table B.2: FPU parameters

B.1.1.2 The FPU `ENTITY`

FPU `ENTITY` parameters are listed in Table B.2. For extensive instrumentation of floating-point intensive QCD code, values below are recorded at each time step of the code execution:

1. PowerPC floating-point instructions from CPU each clock cycle.
2. Instructions committed by load-store and arithmetic pipelines with destination register values.
3. Load wait - latency of load data arrival for each load instruction.
4. Instruction movement from a pipeline stage to other with source register values.

In the `$report` section, an FPU entity prints the total execution time from first instruction issue to last instruction committed and the total number of load-store and arithmetic instructions.

B.1.1.3 The MMU `ENTITY`

Detailed knowledge of the MMU operations and its behaviour during a simulation is of pivotal importance in investigating and scrutinising the performance of the QCD code.

The allocate and write policy attributes of Table B.3 can also be controlled via a TLB entry. An MMU`ENTITY` records the following values at each clock cycle:

1. read address requests together with the data cache touch `dcbt` or line fill requests.
2. Read replies with effective addresses and value to FPU and CPU.

Parameter	Description	Possible Values
`d_cache_sizeX32K`	Size of the data cache	32K and 64K
`d_cache_ways_log2`	Number of cache ways	4 to 64
`d_cache_sets_log2`	Number of cache sets	4 to 64
`burst_size_log2`	Data read and write bus size	4 to 32 words
`write_policy`	Whether write through or write back	write through, copy back
`allocate_policy`	What should happen on a write miss	allocate, no allocate

Table B.3: MMU parameters

3. Write data plus addresses from CPU and FPU.

The MMU `$report` section reports number of hits, misses, total requests for read and write operations to the CPU.

B.1.2 PDB and PLBDBLK `ENTITY`

The PDB provides a high bandwidth access to EDRAM, serves as a prefetch engine, buffers write from core's data bus and interfaces core masters to Processor Local Bus. PLBDBLK is similar to the PDB except for the MMU's processor read and write bus master interfaces. Table B.4 lists the PDB and PLBDBLK `ENTITY` parameters.

Parameter	Description	Possible Values
`PrefetchSize_W_log2`	Size of prefetch and buffer registers	8-64 words
`ReadRegisters`	Number of prefecth read registers	2-32
`WriteRegisters`	Number of write buffer registers	1-16
`ReplacementPolicy`	Read registers replacement policy	LRU, Round Robin and Random

Table B.4: PDB and PLBDBLK parameters

A PDB `ENTITY` stores in text files during a simulation run:

1. Read requests from MMU;
2. Write requests from MMU;
3. Read data replies to the MMU;
4. Write flushes to the EDRAM; and
5. Prefetch read requests to the EDRAM.

The `$report` section prints total number of requests, hits and misses for read and write requests from MMU.

B.1.3 The EDRAM `ENTITY`

EDRAM stores on-chip instructions/data and supports high bandwidth read and write accesses to PDB, PLB slave and DMA. One parameter `edram_burst_log2` indicates the data transfer bus width to the PEC interfaces. The EDRAM `ENTITY` filter the read and write requests from the PEC interfaces as well as prefetch read broadcasts.

B.1.4 The SCU `ENTITY`

The communication unit, SCU, a custom designed component's parameters are listed in Table B.5.

The effect of the above parameter changes are recorded in form of the values below:

1. Packets send/received from/to 16 send and receive ports.
2. Read requests to PLB with send buffer identifier.
3. Read data from PLB with send buffer identifier.
4. Write data to PLB with receive buffer identity.
5. Write acknowledge from PLB with receive buffer identity.

In addition to the above values stored filtered in separate files, an SCU entity prints simulation results in form of number of double-word transfers from each port and simulation execution time in clock cycles.

Parameter	Description	Possible Values
register_size	Size of send and receive registers	1-11 double words
buffer_size	Size of send and receive buffers	1-4 double words
send_latency	Latency of a double word send	up to 200 clock cycles
ack_latency	Latency of a double word acknowledgement	up to 100 clock cycles
plbxfer_latency	Time to load 2 double-words to and from registers	up to 20 clock cycles

Table B.5: SCU parameters

B.2 Sample Outputs

As mentioned in the last section, several entities filter information, which are not recorded by the simulation tracefile, and write them into a text file. These files can later be used for simulation validation and analysis of an entity's behaviour during a simulation.

For example, the output from the CPU ENTITY commit unit is as follows. It identifies not only the execution pipeline and clock cycle value but also the values in registers and the effective address (EA) for load and store instructions.

```
..............

7 J-Pipe MR R9,R3 dest=R9(B000C400,-1342127104)
src1=R3(B000C400,-1342127104) src2=R0(0,0) EA=(0,0) data=(0,0)

8 J-Pipe ADDI R1,R1,-1408 dest=R1(B00FF718,-1341130984)
src1=R1(B00FFC98,-1341129576) src2=R0(FFFFFA80,-1408) EA=(0,0)
data=(0,0)

9 J-Pipe MR R11,R4 dest=R11(B00FFD04,-1341129468)
src1=R4(B00FFD04,-1341129468) src2=R0(0,0) EA=(0,0) data=(0,0)
```

```
10 J-Pipe MR R23,R5 dest=R23(B002BC00,-1341998080)
src1=R5(B002BC00,-1341998080) src2=R0(0,0) EA=(0,0) data=(0,0)

11 J-Pipe ADDI R22,R1,864 dest=R22(B00FFA78,-1341130120)
src1=R1(B00FF718,-1341130984) src2=R0(360,864) EA=(0,0) data=(0,0)

12 J-Pipe MR R24,R6 dest=R24(B00FFCF4,-1341129484)
src1=R6(B00FFCF4,-1341129484) src2=R0(0,0) EA=(0,0) data=(0,0)

13 J-Pipe ADDI R13,R22,64 dest=R13(B00FFAB8,-1341130056)
src1=R22(B00FFA78,-1341130120) src2=R0(40,64) EA=(0,0) data=(0,0)

20 LS-Pipe LWZ R24,0(R24) dest=R24(8,8) src1=R24(B00FFCF4,-1341129484)
src2=R0(0,0) EA=(B00FFCF4,-1341129484) data=(8,8)

23 I-Pipe MRCR R24,R24 dest=R24(0,0) src1=R24(8,8) src2=R0(0,0)
EA=(0,0) data=(0,0)

24 I-Pipe BLE EA3C<LAB0EA3C> dest=R0(0,0) src1=R0(0,0) src2=R0(0,0)
EA=(0,0) data=(E304,58116)

28 J-Pipe ADDI R20,R11,96 dest=R20(B00FFD64,-1341129372)
src1=R11(B00FFD04,-1341129468) src2=R0(60,96) EA=(0,0) data=(0,0)

29 J-Pipe LI R25,0 dest=R25(0,0) src1=R1(0,0) src2=R0(0,0) EA=(0,0) data=(0,0

30 J-Pipe ADDI R15,R22,160 dest=R15(B00FFB18,-1341129960)
src1=R22(B00FFA78,-1341130120) src2=R0(A0,160) EA=(0,0) data=(0,0)

31 J-Pipe LI R26,32 dest=R26(20,32) src1=R0(20,32) src2=R0(0,0)
```

```
EA=(0,0) data=(0,0)

32 J-Pipe ADDI R16,R22,256 dest=R16(B00FFB78,-1341129864)
src1=R22(B00FFA78,-1341130120) src2=R0(100,256) EA=(0,0) data=(0,0)

33 LS-Pipe DCBT R25,R13 dest=R0(0,0) src1=R25(0,0)
src2=R13(B00FFAB8,-1341130056) EA=(B00FFAB8,-1341130056) data=(0,0)

..............
```

The main source of the pipeline blocking is the unavailability of the load data. A file stores the number of wait cycles for a load instruction with respect to the clock cycle value.

```
..............
13591 1
13594 2
13597 6
13599 1
13606 2
13608 2
13609 1
13610 1
13611 1
13612 1
13614 2
13616 2
13619 2
13622 2
13623 1
13625 2
13635 4
..............
```

Appendix C

PowerPC ELF and ABI

In order to be able to parameterise the QCD software, the QCD operating system and PowerPC process handling features are explored and incorporated in the HASE model design. The QCD operating system is a custom operating system that follows the Unix binary specification known as System V Application Binary Interface (ABI). Another important issue was the object file implementation conforming System V ABI specification, called Executable and Linking Format (ELF). This appendix briefly introduces PowerPC ABI and ELF concepts[1] and explains these are exploited in shaping the software parameters for the HASE QCDOC model.

C.1 System V ABI

An ABI describes binary-level conventions for applications running on a particular system, and establishes conventions for operations such as register usage, parameter passing, and layout of data. Typically, these conventions are embodied in development tools, particularly compilers. Assembly-level programmers must be aware explicitly of these conventions, especially if they wish to interface their assembly code with code produced by a compiler.

The primary goals of an ABI are:

[1]Extracted from PowerPC compiler writer guide, available at <http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/>, and Macintosh C/C++ ABI overview document, available at <http://developer.apple.com/tools/mpw-tools/compilers/docs/abi.html>

1. To establish machine-level run time conventions for a processor family;

2. To ensure object code compatibility between compilers for a platform.

While it is possible for programs to depart from the conventions of an ABI, particularly within isolated sections of a program (such as sections of hand-crafted assembly code), conformance to the ABI is often required to make use of system-level code and code produced by other compilers. To the extent that a program is monolithic and is built with the same set of tools conformance to the ABI is only an issue when the program interfaces with the system. To the extent that a program is made up of (or accesses) components which may have been built with other tools conformance to the ABI is more critical. QCD benchmark kernel not only interfaces with code written in a high-level language (C++) but also calls communication routines embedded in its operating system.

C.2 PowerPC ELF

ELF defines a linking interface for compiled application programs. ELF is described in two parts. The first part is the generic System V ABI. The second part is a processor specific supplement.

To be ABI-conforming, the processor must implement the instructions of the architecture, perform the specified operations, and produce the expected results.

C.2.1 Function Calling Sequence

This section discusses the standard function calling sequence, including stack frame layout, register usage, and parameter passing.

C.2.1.1 Registers

The 32-bit PowerPC Architecture provides 32 general purpose registers, each 32 bits wide and several special purpose registers. In addition, the auxiliary processor provides 32 floating-point registers, each 64 bits wide.

All integer, special purpose, and floating-point registers are global to all functions in a running program. Table C.1 shows how the registers are used.

r0	Volatile register used in function prologs
r1	Stack frame pointer
r2	TOC pointer
r3	Volatile parameter and return value register
r4-r10	Volatile registers used for function parameters
r11	Volatile register used in calls by pointer and as an environment pointer for languages which require one
r12	Volatile register used for exception handling
r13	Reserved for use as system thread ID
r14-r31	Non-volatile registers used for local variables
f0	Volatile scratch register
f1-f4	Volatile floating point parameter and return value registers
f5-f13	Volatile floating point parameter registers
f14-f31	Nonvolatile registers
LR	Link register (volatile)
CTR	Loop counter register (volatile)
XER	Fixed point exception register (volatile)
FPSCR	Floating point status and control register (volatile)
CR0-CR1	Volatile condition code register fields
CR2-CR4	Non-volatile condition code register fields
CR5-CR7	Volatile condition code register fields

Table C.1: PowerPC Registers

Registers r1, r14 through r31, and f14 through f31 are non-volatile, which means that they preserve their values across function calls. Functions which use those registers must save the value before changing it, restoring it before the function returns. Register r2 is technically non-volatile, but it is handled specially during function calls. In some cases the calling function must restore its value after a function call.

Registers r0, r3 through r12, f0 through f13, and the special purpose registers LR, CTR, XER, and FPSCR are volatile, which means that they are not preserved across function calls. Furthermore, registers r0, r2, r11, and r12 may be modified by cross-module calls, so a function can not assume that the values of one of these registers is that placed there by the calling function.

The condition code register fields CR0, CR1, CR5, CR6, and CR7 are volatile. The condition code register fields CR2, CR3, and CR4 are non-volatile; a function which modifies them must save and restore at least those fields of the CR.

C.2.1.2 The Stack Frame

In addition to the registers, each function may have a stack frame on the run time stack. This stack grows downward from high addresses. The stack pointer (general purpose register r1) of the called function after it has executed code establishing its stack frame.

C.2.1.3 Parameter Passing

For a RISC machine such as PowerPC, it is generally more efficient to pass arguments to called functions in registers (both general and floating-point registers) than to construct an argument list in storage or to push them onto a stack. Since all computations must be performed in registers anyway, memory traffic can be eliminated if the caller can compute arguments into registers and pass them in the same registers to the called function, where the called function can then use them for further computation in the same registers. The number of registers implemented in a processor architecture naturally limits the number of arguments that can be passed in this manner.

C.2.1.4 Function Prologue and Epilogue

This section describes functions' prologue and epilogue code. A function's prologue establishes a stack frame, if necessary, and may save any non-volatile registers it uses. A function's epilogue generally restores registers that were saved in the prologue code, restores the previous stack frame, and returns to the caller. Except for the rules below, this ABI does not mandate predetermined code sequences for function prologues and epilogues. However, the following rules permit reliable call chain back-tracing:

1. If the function uses any non-volatile general registers, it shall save them in the general register save area. If the function does not require a stack frame, this may be done using negative stack offsets from the caller's stack pointer.

2. If the function uses any non-volatile floating point registers, it shall save them in the floating point register save area. If the function does not require a stack frame, this may be done using negative stack offsets from the caller's stack pointer.

3. Before a function calls any other function, it shall establish its own stack frame, whose size shall be a multiple of 16 bytes, and shall save the link register at the time of entry in the LR save area of its caller's stack frame.

4. If the function uses any non-volatile fields in the CR, it shall save the CR in the CR save area of the caller's stack frame.

5. If a function establishes a stack frame, it shall update the back chain word of the stack frame atomically with the stack pointer (r1) using one of the "Store Double Word with Update" instructions.

6. When a function deallocates its stack frame, it must do so atomically, either by loading the stack pointer (r1) with the value in the back chain field or by incrementing the stack pointer by the same amount by which it has been decremented.

C.3 QCDOC Operating System

QCDOC operating system is a light weight UNIX kernel that is installed on each processing node. One of the key purpose of the kernel is the QCDOC-specific communication calls interfaces and memory allocation for the processing and handling of QCD compute and communication data. Communication call outs include initialisation of the communication routines, monitoring and polling and their successful completion. Since the QCDOC address space has two sets of physical addresses, one for the EDRAM addresses and another for the external memory addresses, special functions are installed for allocating and transferring data explicitly to these addresses.

The QCD kernel initialisation process relies on the operating system, therefore, an understanding of the relationship between the kernel code memory allocation and operating system calls to the underlying operating system was essential. It was also necessary to have a knowledge of memory partition into normal cacheable memory, stack and heap sections and transient and non-cacheable memory allocations for the send and receive addresses. The basic data structure of the calculation, the Wilson fermion matrix, the initialisation process relies on the information of the number of sites per processing node. Number of sites per processing node in turn depends on the lattice volume and the size of the available machine.

```
/*--------------------------------------------------------------*/
#define ND                    4      /* Space time dimension      */

........


/* The Wilson structure typedef */
typedef struct{
   unsigned long *ptr;
   unsigned long *shift_table[2][2];
   /*pointer to an array with addressing offsets */
   unsigned long *face_table[2][2][ND];
   /*pointer to an array with addressing offsets */

   int   vol[2];
   IFloat *spinor_tmp;      /* temp spinor needed by mdagm        */
   IFloat *af;
   /* point. array to 4 interleaved fwd proj half spinors  */
   IFloat *ab;
   /* point. array to 4 interleaved bwd proj half spinors  */
   IFloat *send_f[ND];
```

```
    IFloat *recv_f[ND];
    IFloat *send_b[ND];
    IFloat *recv_b[ND];

    int local_latt[ND];
    int nbound[4];
    int allbound;
    int local_comm[4];
    SCUDirArgMulti *comm_f;
    SCUDirArgMulti *comm_b;
} Wilson;
```

In the HASE QCDOC model, when the number of sites per processing node are known, this initialisation process has been performed in the EDRAM `ENTITY`. Part of the process is re-writing the volume of the lattice, the send and receive pointers and other calculation pointers. When the CPU `ENTITY` starts executing a function with a different problem size, it will still execute the same function but the pointers would read different values.

```
void QCDOC_ChDecom       (void *psi,void *len,void *tab_bwd)
void QCDOC_ChDecom_hsu3 (void *psi,void *Ucb ,void *len,void *tab_fwd)
void QCDOC_ChRecon_su3  (void *chi,void *Un  ,void *chib,void *len)
void QCDOC_ChRecon_add  (void *chi,void * chif,void *len)
```

In the above functions, all the values are passed by pointers. These are in fact pointers to the values within the Wilson struct which in turn point to memory locations where the actual values are stored. The number of iterations depend on the `vol`, the volume of the lattice and `local_latt` and `nbound` array values which are initialised with the number of sites in each direction. For example:

```
nbound[0] = lx * ly * lz /2
lx = number of sites in x direction
ly = number of sites in y direction
lz = number of sites in z direction
```

These values are computed as part of the behaviour code in the `.hase` file and the successive simulation runs input the new problem size or lattice configuration.

Bibliography

[ABB$^+$00] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. N. Houstis, J. R. Rice, R. Sakellariou, D. J. Sundaram-Stukel, P. J. Teller, and M. K. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. *IEEE Transactions on Software Engineering*, 26(11), 2000.

[ABB$^+$01] F. J. Alexander, K. Berkbigler, G. Booker, B. Bush, K. Davis, and A. Hoisie. An Approach to Extreme-Scale Simulation of Novel Architectures. *Fourth Biennial Tri-Laboratory Engineering Conference on Modelling and Simulation*, 2001.

[ABDS95] S. Antonelli, M. Bellacci, A. Donini, and R. Sarno. Full QCD on APE100 Machines. *International Journal of Modern Physics C6*, 25, 1995.

[ABK$^+$99] S. Aoki, R. Burkhalter, K. Kanaya, T. Yoshi, T. Boku, H. Nakamura, and Y. Yamashita. Performance of Lattice QCD Programs on CP-PACS. *Parallel Computing*, 25(10-11), 1999.

[ACC$^+$02] G. S. Almasi, C. Cascaval, J. G. Castanos, M. Denneau, W. Donath, M. Eleftheriou, M. Giampapa, H. Ho, D. Lieber, J.E. Moreira, D. Newns, M. Snir, and Jr. H. S. Warren. Demonstrating the Scalability of a Molecular Dynamics Application on a Petaflops Computer. *International Journal of Parallel Programming*, 30(4), 2002.

[Aga91] A. Agarwal. Limits on Interconnection Network Performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), 1991.

[ALE02] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2), 2002.

[Alt] *AltiVec Technology Programming Environments Manual.*

[BBJ91] R. G. Brickner, C. F. Baillie, and L. Johnsson. QCD on the Connection Machine: Beyond *Lisp. *IMACS First International Conference on Computational Physics*, 1991.

[BCC+01] P. Boyle, D. Chen, N. Christ, C. Cristian, Z. Dong, A. Gara, B. Joo, C. Kim, L. Levkova, X. Liao, G. Liu, R.D. Mawhinney, S. Ohta, T. Wettig, and A. Yamaguchi. Status of the QCDOC Project. *Nuclear Physics Proceedings Supplement*, 106, 2001.

[BCC+02a] P. Boyle, D. Chen, N. Christ, C. Cristian, Z. Dong, A. Gara, B. Joo, C. Jung, C. Kim, L. Levkova, X. Liao, G. Liu, R. D. Mawhinney, S. Ohta, K. Petrov, T. Wettig, and A. Yamaguchi. Status of and Performance Estimates for QCDOC. *Lattice2002 (Machines)*, 2002.

[BCC+02b] P. Boyle, D. Chen, C. Cristian, N. Christ, Z. Dong, A. Gara, B. Joo, C. Kim, L. Levkova, X. Liao, G. Liu, R. Mawhinney, S. Ohta, T. Wettig, and A. Yamaguchi. QCDOC Design. Technical report, Columbia University, 2002.

[BCGV02] G. Bhanot, D. Chen, A. Gara, and P. Vranas. The BlueGene/L Supercomputer. *Lattice2002 (Plenary) proceedings*, 2002.

[BDCW92] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS : A High-Performance Parallel-Architecture Simulator. *International Conference on Measurement and Modelling of Computer Systems*, 1992.

[BDG+95] C. Bernard, C. DeTar, S. Gottlieb, U. M. Heller, J. Hetrick, N. Ishizuka, L. Karkkainen, S. R. Lantz, K. Rummukainen, R. Sugar, D. Toussaint, and M. Wingate. Lattice QCD on the IBM Scalable POWER Parallel Systems SP2. *Supercomputing*, 1995.

[BDP01] R. Bagrodia, E. Deelman, and T. Phan. Parallel Simulation of Large Scale Parallel Applications. *International Journal of High-Performance Computing Application*, 15(1), 2001.

[BG02] G. Bell and J. Gray. What's Next in High-Performance Computing? *Communications of the ACM*, 2002.

[BGK94] M. W. Berry, C. Grassl, and V. K. Krishna. Blocked Data Distribution for the Conjugate Gradient Algorithm on the CRAY T3D. *Cray Research*, 1994.

[BH99] K. C. Bowler and A. J. G. Hey. Parallel Computing and Quantum Chromodynamics. *Parallel Computing*, 25(13-14), 1999.

[BJW03] P. A. Boyle, C. Jung, and T. Wettig. The QCDOC Supercomputer: Hardware, Software, and Performance. *Conference for Computing in High Energy and Nuclear Physics*, 2003.

[Bow98] K. C. Bowler. Why QCD needs High Performance Computing. *Conference on Computational Physics*, 1998.

[Boy01] P. Boyle. QCDOC Assembler Kernels - Proposal for Wilson Fermion Matrix. Draft Document, 2001.

[BPE$^+$99] C. Best, M. Peardon, N. Eicker, P. Uberholz, T. Lippert, and K. Schilling. Lattice Field Theory on Cluster Computers: Vector- vs. Cache-Centric Programming. *1st IEEE Computer Society International Workshop on Cluster Computing*, 1999.

[BS92] F. R. Bailey and H. D. Simon. Future Directions in Computing and CFD. *Proceedings of AIAA 10th Applied Aerodynamics Conference*, 1992.

[CCC$^+$01] D. Chen, N. H. Christ, C. Cristian, Z. Dong, A. Gara, K. Garg, B. Joo, C. Kim, L. Levkova, X. Liao, R. D. Mawhinney, S. Ohta, and T. Wettig. QCDOC: A 10-teraflops Scale Computer for Lattice QCD. *Nuclear Physics Proceedings Supplement*, 94, 2001.

[CCC+02] C. Cascaval, J. G. Castanos, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J. E. Moreira, K. Strauss, and H. S. Warren Jr. Evaluation of a Multithreaded Architecture for Cellular Computing. *Eighth International Symposium on High-Performance Computer Architecture*, 2002.

[CDJ+91] R. G. Covington, S. Dwarkadas, J. R. Jump, J. B. Sinclair, and S. Madala. The Efficient Simulation of Parallel Computer Systems. *International Journal in Computer Simulation*, 1, 1991.

[CGS98] D. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[CHIW98] P. S. Coe, F. W. Howell, R. N. Ibbett, and L. M. Williams. A Hierarchical Computer Architecture Design and Simulation Environment. *ACM Transactions on Modelling and Computer Simulation*, 8(4), 1998.

[CHKM93] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. *Proceedings of the International Symposium on Computer Architecture*, 1993.

[Chr99] N. H. Christ. Computers for Lattice QCD. *Nuclear Physics Proceedings Supplement*, 83, 1999.

[CIRW98] P. Coe, R. Ibbett, N. Rafferty, and L. Williams. HASE: An Environment for Hardware/Software Codesign. Technical Report CSG-41-98, University of Edinburgh, 1998.

[CIS99] N. Cabibbo, Y. Iwasaki, and K. Schilling. High Performance Computing in Lattice QCD. *Parallel Computing*, 25(10-11), 1999.

[CK94] A. A. Chien and M. Konstantinidou. Workloads and Performance Metrics for Evaluating Parallel Interconnects. *IEEE TCCA Newsletter*, 1994.

[CKP$^+$93] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP : Towards a Realistic Model of Parallel Computation. *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[Coe00] P. Coe. *Simulation Models of Shared-Memory Multiprocessor Systems*. PhD thesis, Unversity of Edinburgh, 2000.

[Cre83] M. Creutz. *Quarks, Lattices and Gluons*. Cambridge University Press, 1983.

[Dal90] W. J. Dally. Performance Analysis of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, 39(6), 1990.

[Doc01] K. Dockser. "Honey, I Shrunk the Supercomputer!" - The PowerPC 440 FPU Brings Supercomputing to IBM's Blue Logic Library. *IBM Micronews*, 2001.

[DSSS03] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier. High Performance Computing. Clusters, Constellations, MPPs, and Future Directions. *Communications of the ACM*, 2003.

[EAB$^+$01] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A Performance Model Framework. *IEEE Computer*, 2001.

[Fer78] D. Ferrari. *Computer Systems Performance Evaluation*. Prentice Hall, 1978.

[FJL$^+$88] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Computers*. Prentice Hall, 1988.

[FWM94] G. C. Fox, R. D. Williams, and P. C. Messina. *Parallel Computing Works*. Morgan Kaufmann, 1994.

[GAB$^+$02] A. Gara, G. S. Almasi, D. K. Beece, R. E. Bellofatto, G. V. Bhanot, H. R. Bickford, M. A. Blumrich, A. A. Bright, J. R. Brunheroto, G. C. Cascaval, J. G. Castanos, L. H. Ceze, P. W. Coteus, S. Chatterjee, D. Chen, and R. K. Sahoo. Cellular Supercomputing with System-On-a-Chip. *IEEE International Solid-State Circuits Conference*, 2002.

[GLS99] S. Gusken, T. Lippert, and K. Schilling. Lattice QCD with Two Dynamical Wilson Fermions on APE100 Parallel Systems. *Parallel Computing*, 25(10-11), 1999.

[GNA02] P. Galloway, W. Ng, and M. Annand. Copper Cabling for Multiple-Gigabit Serials Links for Inter-Cabinet Connections. *High-Performance System Design Conference*, 2002.

[Gri98] J. Gribbon. *Q is for Quantum*. Weidenfeld and Nicolson, London, 1998.

[Gup99] R. Gupta. General Physics Motivations for Numerical Simulations of Quantum Field Theory. *Parallel Computing*, 25(10-11), 1999.

[Gus99] S. Gusken. Stochastic Estimator Techniques and their Implementation on Distributed Parallel Computers. *Parallel Computing*, 25(10-11), 1999.

[HASa] *HASE Home Page, Available at: <http://www.icsa.informatics.ed.ac.uk/research/groups/hase/>*.

[HASb] *The QCD Computer Simulation Project, Available at: <http://www.icsa.informatics.ed.ac.uk/research/groups/hase/projects/qcd/qcd.html>*

[Her93] S. Herrod. Tango Lite: A Multiprocessor Simulation Environment. Technical report, Computer Systems Laboratory, Stanford University, 1993.

[Ibb00] R. N. Ibbett. HASE DLX Simulation Model. *IEEE Micro*, 20(3), 2000.

[IHH95] R. N. Ibbett, P. E. Heywood, and F. W. Howell. HASE : A Flexible Toolset for Computer Architects. *The Computer Journal*, 38(8), 1995.

[IMMM99] I. Ikodinovic, D. Magdic, A. Milenkovic, and V. Milutinovic. Limes: A Multiprocessor Simulation Environment for PC Platforms. *Proceedings of the 3rd International Conference on Parallel Processing and Applied Mathematics*, 1999.

[IN00] N. Isgur and J. W. Negele. Nuclear Theory with Lattice QCD. A proposal submitted to the U.S. Department of Energy, 2000.

[KBD93] M. Kumar, Y. Baransky, and M. Denneau. The GF11 Parallel Computer. *Parallel Computing*, 19(12), 1993.

[Ken99] A. D. Kennedy. The Hybrid Monte Carlo Algorithm on Parallel Computers. *Parallel Computing*, 25(10-11), 1999.

[Ken00] R. Kenway. Lattice QCD in Europe and the UKQCD-QCDOC Collaboration. *Workshop on a New Computing Venue for Lattice Gauge Theory Calculations*, 2000.

[KHW02] D. J. Kerbyson, A. Hoisie, and H. J. Wasserman. Exploring Advanced Architectures using Performance Prediction. *Innovative Architecture for Future Generation High-Performance Processors and Systems*, 2002.

[Kon99] J. Konstas. Converting Wide, Parallel Data Buses to High Speed Serial Links. *International IC '99 Conference Proceedings*, 1999.

[Lep99] G. P. Lepage. Improved Discretisations for Lattice QCD. *Parallel Computing*, 25(10-11), 1999.

[Lip99] T. Lippert. Parallel SSOR Preconditioning for Lattice QCD. *Parallel Computing*, 25(10-11), 1999.

[Lip03] T. Lippert. Recent Development of Machines for Lattice QCD. *The XXI International Symposium on Lattice Field Theory*, 2003.

[Lus02] M. Luscher. Lattice QCD on PCs? *Nuclear Physics Proceedings Supplement*, 2002.

[Mar88] J. L. Martin, editor. *Performance Evaluation of Supercomputers*. North Holland, 1988.

[Maw99] R. D. Mawhinney. The 1 Teraflops QCDSP Computer. *Parallel Computing*, 25(10-11), 1999.

[May94] C. May. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann, 2nd edition, 1994.

[MCE$^+$02] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2), 2002.

[McN00] C. McNeile. Progress in Lattice QCD Calculations on the T3E. *Proceedings of the Sixth European SGI/Cray MPP Workshop*, 2000.

[MCS$^+$03] J. Moreira, L. Ceze, K. Strauss, G. Almasi, P. J. Bohrer, J. R. Brunheroto, C. Cascaval, J. G. Gastranos, and D. Lieber. Full Circle: Simulating Linux Clusters on Linux Cluster. *The Fourth Linux Clusters: the HPC Revolution 2003 Conference*, 2003.

[MHW02] C. J Mauer, M. D. Hill, and D. A. Wood. Full System Timing-First Simulation. *ACM SIGMETRICS*, 2002.

[MI03] F. Mallet and R. N. Ibbett. JavaHASE: Automatic Generation of Applets from HASE Simulation Models. *Summer Computer Simulation Conference*, 2003.

[MIA02] F. Mallet, R. N. Ibbett, and S. R. Alam. An Extensible Clock Mechanism for Computer Architecture Simulations. *13th IASTED International Conference Modelling and Simulation*, 2002.

[MIL] *The MIMD Lattice Computation (MILC) Collaboration, Available at: <http://www.physics.utah.edu/~detar/milc/>.*

[MP01] V. Mathur and V. K. Prasanna. A Hierarchical Simulation Framework for Application Development on System-on-Chip Architectures. *4th IEEE International ASIC-SOC Conference*, 2001.

[MRF+00] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus. Wisconsin Wind Tunnel II : A Fast, Portable Parallel Architecture Simulator. *IEEE Concurrency*, 8(4), 2000.

[NKP+00] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, J. S. Harper, S. C. Perry, and D. V. Wilcox. PACE: A Toolset for the Performance Prediction of Parallel and Distributed Systems. *Journal of High Performance Application*, 14(3):228–251, 2000.

[NYHG+98] A. K. Nanda, M. Ohara Y. Hu, M. Giampapa, C. Benveniste, and M. Michael. The Design of COMPASS : An Execution Driven Simulator for Commercial Applications Running on Shared Memory Multiprocessors. *Proceedings of International Parallel Processing Symposium*, 1998.

[PCD+01] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and Memory Optimization Techniques for Embedded Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2001.

[Pie00] M. Di Pierro. Matrix Distributed Processing and FermiQCD. *Proceedings of the Workshop on Advanced Computing and Analysis Techniques*, 2000.

[PKP03] F. Petrini, D. J. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. *Annual Supercomputing Conference*, 2003.

[ppca] *Macintosh C/C++ ABI overview document, Available at: <http://developer.apple.com/tools/mpw-tools/compilers/docs/abi.html>.*

[ppcb] *PowerPC compiler writer guide, Available at: <http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/>.*

[PRA97] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM : An Execution-Driven Simulator for ILP-Based Shared Memory Multiprocessors and Uniprocessors. *IEEE TCCA Newsletter*, 1997.

[PV97] F. Petrini and M. Vanneschi. SMART : a Simulator of Massive ARchitectures and Topologies. *International Conference on Parallel and Distributed Systems, Euro-PDS'97*, 1997.

[RBDH97] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions in Modelling and Computer Simulation*, 7(1), 1997.

[Rue97] U. Ruede. Iterative Algorithms on High Performance Architectures. *Lecture Notes in Computer Science*, 1300, 1997.

[SA102] IBM. *PowerPC 440 Embedded Processor Core*, 2002.

[SB01] T. J. Schriber and D. T. Brunner. Inside Simulation Software: Inside Discrete-Event Simulation Software: How it Works and Why it Matters. *Winter Simulation Conference*, 2001.

[SHM97] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Journal of Scientific Programming*, 1997.

[Siv97] A. Sivasubramaniam. Execution-Driven Simulators for Parallel Systems Design. *Proceedings of 1997 Winter Simulation Conference*, 1997.

[SKSZ99] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang. The Case for Application-Specific Benchmarking. *The Seventh Workshop on Hot Topics in Operating Systems*, 1999.

[SMA$^+$03] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai. Challenges in Computer Architecture Evaluation. *IEEE Computer*, 2003.

[SMK$^+$03] H. D. Simon, C. W. McCurdy, W. T. C. Kramer, R. Stevens, M. McCoy, M. Seager, T. Zachariaa, J. Nichols, R. Bair, S. Studham, W. Camp, R. Leland, J. Morrison, and B. Feiereisen. Creating Science-Driven Computer Architecture: A New Path to Scientific Leadership. NERSC Document, 2003.

[Sro02] Z. Sroczynski. Improved Performance of QCD Code on ALiCE. *Contribution to Lattice2002 (Machines)*, 2002.

[SZ02] T. L. Sterling and H. P. Zima. Gilgamesh: A Multithreaded Processor-In-Memory Architecture for Petaflops Computing. *Supercomputing*, 2002.

[SZI] *The SZIN Software System, Available at: <http://www.jlab.org/∼edwards/szin/>.*

[Tea01a] IBM Bluegene Team. Blue Gene: A Vision for Protein Science using a Petaflop Supercomputer. *IBM Systems Journal*, 40(2), 2001.

[Tea01b] QCDOC Design Team. PEC Design Document. Confidential, 2001.

[Tea02] IBM Bluegene/L Team. An Overview of the BlueGene/L Supercomputer. *Supercomputing*, 2002.

[TH99] T. Touyama and S. Horiguchi. Performance Evaluation of Practical Parallel Computation Model LogPQ. *International Symposium on Parallel Architectures, Algorithms and Networks*, 1999.

[Tri99] R. Tripiccione. APEmille. *Parallel Computing*, 25(10-11), 1999.

[Uka99] A. Ukawa. Lattice QCD Results from the CP-PACS Computer. *Parallel Computing*, 25(10-11), 1999.

[UM97] R. A. Uhlig and T. N. Mudge. Trace-Driven Memory Simulations: A Survey. *ACM Computing Surveys*, 29(2), 1997.

[VF94] J. E. Veenstra and R. J. Fowler. MINT : A Front End for Efficient Simulation of Shared-memory Multiprocessors. *Modelling and Simulation of Computers and Telecommunications Systems*, 1994.

[VM02] J. S. Vetter and F. Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. *International Parallel and Distributed Processing Symposium*, 2002.

[WI96] L. M. Williams and R. N. Ibbett. Simulating the DASH Architecture in HASE. *29th Annual Simulation Symposium*, 1996.

[WWFH03] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. *The 30th Annual International Symposium on Computer Architecture*, 2003.