Profile-driven Parallelisation of Sequential Programs

Georgios Tournavitis

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2011

Abstract

Traditional parallelism detection in compilers is performed by means of static analysis and more specifically data and control dependence analysis. The information that is available at compile time, however, is inherently limited and therefore restricts the parallelisation opportunities. Furthermore, applications written in *C* – which represent the majority of today's scientific, embedded and system software – utilise many low-level features and an intricate programming style that forces the compiler to even more conservative assumptions. Despite the numerous proposals to handle this *uncertainty* at compile time using *speculative* optimisation and parallelisation, the software industry still lacks any pragmatic approaches that extracts coarse-grain parallelism to exploit the multiple processing units of modern commodity hardware.

This thesis introduces a novel approach for extracting and exploiting multiple forms of coarse-grain parallelism from sequential applications written in *C*. We utilise profiling information to overcome the limitations of static data and control-flow analysis enabling more aggressive parallelisation. Profiling is performed using an instrumentation scheme operating at the Intermediate Representation (Ir) level of the compiler. In contrast to existing approaches that depend on low-level binary tools and debugging information, Ir-profiling provides precise and direct correlation of profiling information back to the Ir structures of the compiler. Additionally, our approach is orthogonal to existing automatic parallelisation approaches and additional fine-grain parallelism may be exploited.

We demonstrate the applicability and versatility of the proposed methodology using two studies that target different forms of parallelism. First, we focus on the exploitation of loop-level parallelism that is abundant in many scientific and embedded applications. We evaluate our parallelisation strategy against the Nas and Spec Fp benchmarks and two different multi-core platforms (a shared-memory Intel Xeon Smp and a heterogeneous distributed-memory Ibm Cell blade). Empirical evaluation shows that our approach not only yields significant improvements when compared with state-of-the-art parallelising compilers, but comes close to and sometimes exceeds the performance of manually parallelised codes. On average, our methodology achieves 96% of the performance of the hand-tuned parallel benchmarks on the Intel Xeon platform, and a significant speedup for the Cell platform. The second study, addresses the problem of partially sequential loops, typically found in implementations of multimedia codecs. We develop a more powerful whole-program representation based on

the Program Dependence Graph (PDG) that supports profiling, partitioning and code-generation for pipeline parallelism. In addition we demonstrate how this enhances conventional pipeline parallelisation by incorporating support for multi-level loops and pipeline stage replication in a uniform and automatic way. Experimental results using a set of complex multimedia and stream processing benchmarks confirm the effectiveness of the proposed methodology that yields speedups up to 4.7 on a eight-core Intel Xeon machine.

Acknowledgements

First of all, I would like to thank my advisor, Björn Franke, for his guidance and support throughout the four years of my PhD studies. Most importantly, I am grateful to Björn for his understanding, optimism and confidence in me. I would also like to thank Michael O'Boyle for his invaluable advice, collaborative spirit and spirited leadership. In addition, I would like to express my gratitude to Nigel Topham, Marcelo Cintra, Murray Cole, Aris Efthymiou and Vijay Nagarajan, each one of them contributing with his own special way to making ICSA a great research institute. Finally, special thanks to all my colleagues in CARD for the endless hours that we spent together and for making our lab such a fun place to study.

Special thanks to Murray Cole, Mikel Luján and Peter Marwedel not only for undergoing the torture of studying this thesis, but also for their insightful comments and making my PhD defence a rewarding experience.

This thesis would not have been possible without the support of all my friends; those who either bearing with my idiosyncrasies on a daily basis or sharing a few minutes on the phone every now and then have been there for me all these years.

I am dedicating this thesis to my family, for without their love and affection I would not have achieved any of this.

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.

(*Georgios Tournavitis*)

Publications

Parts of this thesis have been published in the following refereed conference and workshop papers (in reverse chronological order):

1. Semi-Automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism Using Profiling Information. Georgios Tournavitis and Björn Franke. In *PACT' 10: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 377–388, Vienna, Austria, 2010. ACM.

2. Towards a Holistic Approach to Auto-parallelisation: Integrating Profile-driven Parallelism Detection and Machine-Learning Based Mapping. Georgios Tournavitis, Zheng Wang, Björn Franke and Michael F.P. O'Boyle. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 177–187, Dublin, Ireland, 2009. ACM.

3. Towards Automatic Profile-Driven Parallelisation of Embedded Multimedia Applications. Georgios Tournavitis and Björn Franke. In *MULTIPROG-2009: Proceedings of the Second Workshop on Programmability Issues for Multi-Core Computers*, pages 53–64, Paphos, Cyprus, 2009.

Contents

Abbreviations

API	Application Programming Interface	**OoO**	Out-Of-Order
AST	Abstract Syntax Tree	**OS**	Operating System
AVX	Advanced Vector Extensions	**PC**	Program Counter
BSS	Block Started by Symbol	**PDG**	Program Dependence Graph
BB	Basic Block	**PPE**	Power™ Processing Element
CCMIR	CoSy Common Medium-level IR	**RLE**	Run-Length Encoding
CDFG	Control Data Flow Graph	**SCC**	Strongly Connected Component
CFG	Control Flow Graph	**SDK**	Software Development Kit
CMP	Chip Multi-Processor	**SEME**	Single-Entry-Multiple-Exit
CoSy	Compiler [1]	**SESE**	Single-Entry-Single-Exit
DAG	Directed Acyclic Graph	**SIMD**	Single Instruction Multiple Data
DSWP	Decoupled SoftWare Pipelining	**SMP**	Symmetric Multi-Processor
DMA	Direct Memory Access	**SMT**	Simultaneous Multi-Threading
DSP	Digital Signal Processor	**SON**	Scalar Operand Network
DWT	Discrete Wavelet Transform	**SPSC**	Single Producer Single Consumer
FIFO	First-In-First-OUT	**SPE**	Synergistic Processing Element
FP	Floating Point	**SSE**	Streaming SIMD Extensions
FUD	Factored Use-DEF	**STM**	Software Transactional Memory
IDE	Integrated Development Environment	**TLP**	Thread-Level Parallelism
ILP	Instruction-Level Parallelism	**TLS**	Thread-Level Speculation
IR	Intermediate Representation	**TM**	Transactional Memory
LS	Local Storage	**VLIW**	Very Long Instruction Word
MPSoC	Multi-Processor System-on-Chip	**VMX**	Vector Multimedia eXtension
MMX	Matrix Math Extensions	**WAR**	Write-After-Read dependence
MPI	Message Passing Interface	**WAW**	Write-After-Write dependence

Chapter 1.

Introduction

Transistor miniaturization, increasing clock frequency and Instruction-Level Parallelism (ILP) exploitation techniques have been leading the semiconductor industry for the last thirty years to deliver exponentially increasing performance. Nevertheless, moving towards the end of the nanoscale transistors era, power density and heat dissipation issues as well as the lack of the micro-architectural breakthroughs of the past has forced the industry to shift towards multi- and many-core architectures [55, 109, 120, 127] to continue shipping products that follow this trend, also known as the Moore's Law [93]. These architectures have the potential to deliver unprecedented peak performance of Teraflop scale on a single die [154], however, this entails a considerable risk. The established model of providing higher performance through technology and micro-architectural improvements allowed the overlying software stack to be largely unchanged. Multi-processing on the other hand requires the software to be efficiently parallelised and thus breaks this separability between the hardware and software layer.

Only to make things worse, most of today's applications are sequential and thus additional effort is required during this transition to the multi-core era. Figure 1.1 shows the implementation cost of portable consumer devices – a market that parallel processing has only recently been introduced as a means to meet the ever incresing consumer demand for higher functionality – and its breakdown to software and hardware aspects. Besides the extremely high increase of the overall cost that the adoption of multi-processing transition causes, this figure clearly shows the widening gap of software and hardware *productivity*. The projection for the following years shows that a more than two-fold decrease in software development cost is required to rebalance the two. However, this is based on the optimistic prediction that research on parallel compiler technology will lead to a significant increase in software productivity. For this to happen the research community has to create new methodologies and design tools that leverage the analyses developed in the last decades but overcome their limitations and increase their potential.

Multi-core architectures, in contrast to Single Instruction Multiple Data SIMD and Very Long Instruction Word (VLIW) architectures that target word and instruction-level parallelism respectively, necessitate the extraction of parallelism in the form of Thread-Level Parallelism (TLP). At an algorithmic level, there is a plethora of sequen-

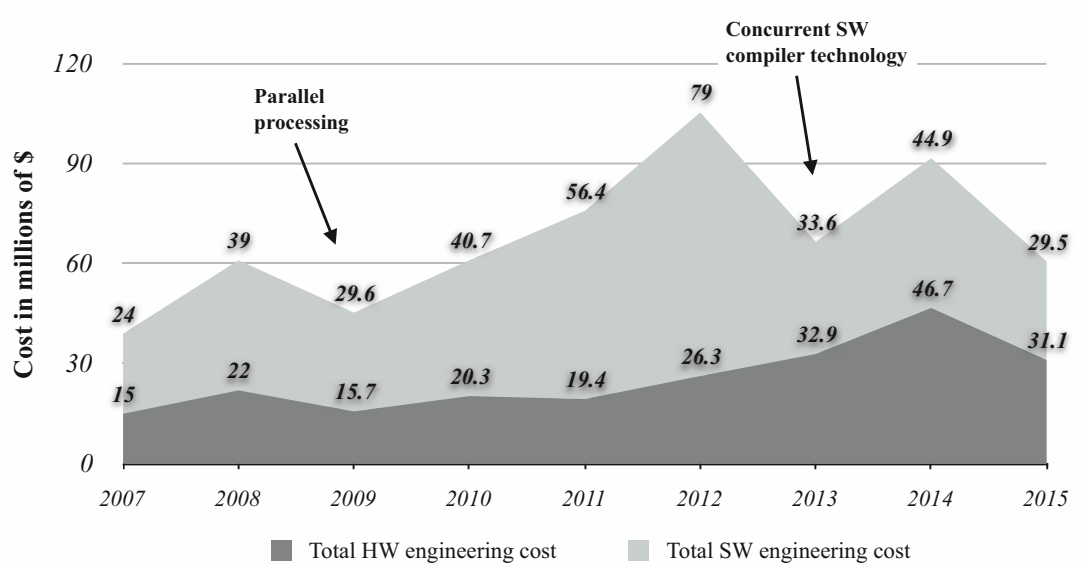


Figure 1.1.: Impact of design technology on the implementation cost of portable consumer devices (based on data published from the International Technology Roadmap Committee in [59, 58]). The introduction of parallel processing in 2009 resulted in a steep increase in the cost spent on the software stack. The projection for 2013 though shows a more than two-fold decrease in software development cost. However, this is based on the optimistic prediction that research on parallel compiler technology will lead to a significant increase in software productivity.

tial applications that consists of high-level operations that can be largely computed in parallel. At the level of a sequential implementation though, identifying the structure of parallelism has proven to be an extremely demanding task since it requires a global view of the whole-program and a precise knowledge of the interaction between its numerous components. This problem is further complicated if the additional problem of partitioning an application for a specific target is considered. This task entails the modelling of different resources, constraints and the complex interplay between the application, the operating system and the architecture. These problems are definitely not new to the compiler and design automation community. Many ambitious research projects and countless papers have addressed individual aspects of this topic. Nevertheless, the lack of a general and applicable methodology in addition to the inadvertent market domination of the multi-cores and the high stakes involved have posed a challenge and at the same a unique opportunity to investigate more *pragmatic* approaches.

Parallel Programming High-level parallel programming languages – such as Cilk [38], OpenMp [103], StreamIt [140], Upc [50], X10 [126] and OpenCL [71] – and programming models – such as Stapl [117], Hta [41], Galois [73] and CnC [18] – have long be proposed as a means to facilitate *intuitive*, *expressive* and *efficient* application design and implementation for parallel architectures. This high potential, however, comes at a cost. Parallel programming not only presumes a long learning process, but

is inherently a cumbersome, time-consuming and error-prone process. Therefore, it is not a surprise that a considerable part of parallel application development is spent in addressing hard to identify and even harder to correct *errors* and *performance pathologies* related to *concurrency*. This, inevitably, leads to higher software engineering cost and increased time-to-market (figure 1.1). But most importantly, programming models do not resolve, nor even address, the immense problem of legacy sequential code which accounts for the vast majority of existing commercial and industrial applications.

Auto-parallelisation Auto-parallelisation, on the other hand, bears the promise to alleviate the burden of identifying and exploiting parallelism, while ensuring formal correctness of the derived implementation. In fact, remarkable progress has been achieved in this domain in the past three decades [6] that has also resulted in the emergence of a new generation of parallelising research compilers, e.g. Polaris [107], SUIF [132] and, more recently, Open64 [21]. These compilers employ sophisticated analyses like interprocedural array data-flow analysis [19], symbolic analysis [43] and array region analysis [45, 30], and perform code transformations like array privatisation, parallel reduction, loop fusion and unimodular loop transformations in order to increase parallelisation opportunities. Nevertheless, this large body of work has been primarily focused on scientific applications written in Fortran.

1.1. The Problem

This thesis is concerned with the problem of automatically extracting Thread-Level Parallelism (TLP) from sequential applications written in *C*. The choice of *C* as a source language is dictated by its wide use in the large body of legacy scientific and embedded applications as well as its dominating position in the consumer electronics industry. Other languages, like Fortran or StreamIt, offer greater scope for compiler-based analysis and parallelisation. However, these languages lack the market dominance of *C*. Unfortunately, *C* offers a powerful set of low-level language features that often lead to obscure programming idioms and practices. The extensive use of pointer arithmetic, dynamic memory allocation, indirect function calls and resizeable arrays in *C* programs has proven to be extremely challenging for static analysis. Recent proposals, like context and field-sensitive pointer analysis [48], shape analysis [40] and other abstract interpretation based techniques like commutativity analysis [4] have definitely increased the scope and precision of compiler optimisation for pointer intensive programs. Nevertheless, the problem of effectively parallelising full-scale real-life *C* applications persists [66, 88, 17, 69, 67, 68]. This is mainly due to the inability of static analysis to disambiguate the sequential semantics on a whole-program level. Unavoidably, to preserve correctness compilers make conservative assumptions and therefore automatic parallelisation fails even for applications that inherently feature abundant parallelism.

Multi-core architectures offer orders of magnitude faster on-chip communication and synchronisation in comparison to the scalable distributed memory parallel systems of the past. Still, threading overheads typically range in the scale of thousands of cycles, thus preventing the efficient parallelisation of loops that require fine-grain communication (e.g. DOACCROSS execution [6]). Despite the emergence of many research proposals that demonstrate the potential of dedicated hardware support for register communication [115], cache-to-cache transfers [134], or asynchronous thread messaging [125] these are features that have not yet gained the support of any of the major microprocessor vendors. Similarly, hardware *speculation* has long being presented as the "silver bullet" for parallelisation [72, 85, 129, 133, 119]. Highly complex designs, interaction with the cache-coherency protocol and high chip-area and validation cost of such hardware schemes has so far doomed their implementation [36]. To address the escalating problem of parallelisation on the multi-core systems of the next decade, however, more grounded and pragmatic approaches are necessary.

Multimedia codecs typically exhibit a *stream* processing pattern. Input is taken in the form of packets or blocks of data, then processed by a sequence of transformations/ filters and finally the resulting formatted data are written in the output. Additionally, coarse-grain loops with statically unknown loop bounds and loops that include operations with loop-carried dependences are partially sequential but still include exploitable data and task-level parallelism. The automatic extraction of these complex parallelisation skeletons from real-life implementation, however, poses many difficulties. Independent tasks are often coupled deep in the call hierarchy, intertwined with I/O or other inherently sequential operations (e.g. Run-Length or Huffman coding), or span multiple nested loops further complicating the task of deriving a parallel implementation. To address this problem interactive tools that assist the developers to identify and estimate the potential of alternative parallelisation schemes have been recently proposed [64, 31, 52, 155]. To overcome the limitation of traditional auto-parallelisers these tools either rely on the user to indicate parallelisation candidates or utilise dynamic dependence profiling. The latter technique is particularly promising since it is relaxing the conservative view of static dependence analysis, focusing only on those dependences that manifest during execution. Nevertheless, these tools heavily depend on user intervention and thus presume detailed knowledge of the specific implementation. Most importantly, existing approaches either do not address or have very limited support for automatic parallel code generation. It is clear that going beyond DOALL parallelism without compromising programmer's productivity does not only demand more flexible tools but a holistic approach that automates every step of the parallelisation process.

1.2. Goals

The primary objective of this thesis is to identify and overcome some of the limitations that have till now prevented compilers to exploit the full potential of *coarse-grain*

parallelism from sequential applications. Although general, our approach mainly focuses on programs written in *C*. In addition, our approach is *orthogonal* to alternative parallelisation techniques that aim at extracting fine-grain parallelism (e.g. utilising SIMD extension). In fact our experimental methodology includes state-of-the-art auto-vectorisation in each baseline sequential configuration. To remove the hurdle of conservative dependence analysis we employ a powerful profiling mechanism for extracting data dependence information. Although dependence profiling can not be conclusive, many applications exhibit stable data-flow behaviour. For these applications and given a set of representative inputs, profiling can provide the detailed dependence information that is necessary to enable aggressive parallelisation on a whole-program level.

We should stress that our goal is to present a methodology that addresses not only parallelism identification but also effective *exploitation* on commodity hardware. The main challenge in this goal is to bridge the *information gap* between profiling and compilation. Typically profiling is performed in low-level binary or assembly representations that lack high-level information which is critical to enable data and control-flow transformations. Furthermore, our plan is to develop a unified framework that not only addresses established paradigms like loop-level or task parallelism but also extends the scope of parallelisation tools to more advanced skeletons like parallel pipelines. Finally, our ultimate goal is to demonstrate that our methodology is applicable not to just "toy" applications but full-scale real-life implementations.

This thesis is deliberately not targeting sequential applications that exhibit limited parallelism and therefore have very low performance potential from the exploitation of TLP. In other words our goal is to overcome the *limitations* of coarse-grain parallelisation rather than quest for the "holy grail" of extending the *limits* of parallelism itself. This choice is motivated by recent studies [69, 67, 68] that not only confirm the marginal speedups of TLP, but also demonstrate the surprisingly small benefits that hardware support for *speculative* TLP can offer – even if ideal threading overheads are assumed. Our view is that the optimisation of this class of sequential applications should be pursued using alternative micro-architectural or co-operative software/hardware methods. For instance, *helper threads* [27, 163, 81] have been proposed to increase the efficiency of sequential applications by exploiting the additional cores for the execution of subordinate threads that prefetch data in higher levels of the cache hierarchy. Nevertheless, we consider such approaches to be beyond the scope of this thesis.

1.3. Contributions

The contributions of this thesis can be summarised in the following:

- We demonstrate that IR-profiling is a powerful tool for the extraction and exploitation of parallelism beyond loop level. IR-profiling bridges the *information gap* between the low-level execution profile and the IR maintained within the

parallelising compiler. This simple, yet effective *back annotation* capability distinguishes our approach from the majority of existing profiling methodologies.

- We propose the use of profile-driven parallelisation for the extraction of data-level parallelism in both scientific and embedded applications. We show that profile-driven analyses can detect more parallel loops than static techniques. In addition, we demonstrate that our approach when coupled with a Machine-Learning profitability analysis not only yields significant improvements compared with state-of-the-art parallelising compilers, but also comes close to and sometimes exceeds the performance of manually parallelised codes.

- We develop a top-down approach for the extraction of processing pipelines from sequential applications. We exploit the power of a whole-program IR, but avoid exposing overly detailed (and possibly redundant) dependence information to the pipeline extraction pass. Using an iterative, *selective unfolding* strategy we specifically target the levels of computationally intensive code regions that will yield exploitable parallelism.

- We extend conventional linear-pipeline parallelisation with two concepts borrowed from streaming languages, namely *multi-level pipelines* and *stage replication*. Furthermore, we present evidence that profile-driven dependence analysis is powerful enough to uncover such high-level structure from *real-life* applications. Finally, we demonstrate how their combination can uncover additional parallelisation opportunities that existing approaches fail to exploit.

1.4. Structure

The remainder of this thesis is organised as follows:

Chapter 2 presents a critical review of existing parallelisation approaches for data and pipeline parallelism. In addition, we present the results of existing limit studies on the potential of static parallelisation.

Chapter 3 describes the CoSy compiler infrastructure that was used for the implementation of IR-profiling and the source-to-source transformations proposed in this thesis. Additionally, we present a brief overview of the main features of the OpenMP parallel programming interface that is utilised for code-generation in chapter 5.

Chapter 4 introduces IR-profiling, a dependence profiling approach that enables precise and straightforward correlation of the extracted information back to the IR of the compiler.

Chapter 5 addresses the problem of extracting and exploiting data-level parallelism. In addition, we introduce a novel approach that combines profile-driven parallelism detection with Machine-Learning profitability analysis. Finally, we demonstrate the potential of our approach on two different parallel architectures by comparing with both sequential execution and manual parallelisation.

Chapter 6 considers the case of partially sequential loops that exhibit a stream processing structure. We present a methodology which exploits profiling information to automatically restructure computation that spans multiple loop-levels and functions into a parallel pipeline. Finally, we demonstrate the applicability and performance of our proposal using a set of widely used embedded and multimedia benchmarks.

Chapter 7 concludes this thesis with a summary of the main contributions, a retrospective critical analysis, and a discussion of topics for future work.

Chapter 2.

Related Work

This chapter presents a critical review of related work from the fields that are most relevant to this thesis. The primary objective is to compare and contrast the approach and techniques proposed in this thesis with the existing body of scientific work. We deliberately focus on approaches that go beyond static-only parallelisation for two reasons: (i) this is an extremely wide and well-studied topic, and (ii) despite the sophistication and the volume of prior work it remains ineffective for the application domains that we are targeting. The structure of this chapter is as follows. In section 2.1 we present limit studies, dynamic and hardware-assisted approaches that target data-parallel loops. A review of recently proposed techniques to parallelise partially sequential loops by exploiting intra-iteration level parallelism follows in section 2.2. Finally, in section 2.3 we compare our approach with state-of-the-art commercial tools that focus on interactive parallelisation. Some approaches cross multiple domains, however, we choose to present them along with the research which is more relevant to their main contributions.

2.1. Data Parallelism

2.1.1. Limit Studies

Compiler approaches to the problem of automatic extraction of coarse-grain parallelism have a long history starting from the early methods developed for array data-flow and loop dependence analysis, like Banerjee's *gcd test* [14], and reaching to more advanced topics like loop transformations for improving the locality of array accesses. Rather than presenting such an extended literature – a task far beyond the scope of this thesis –, we present recent studies on the potential of static-only parallelisation approaches. This way we attempt to highlight the fundamental restrictions of static analysis and motivate our study on dynamic and profile-driven parallelisation that follows.

Islam et al. An early study from M. Islam [62] uses Intel's ICC v.9.1 to auto-parallelise a subset of the MediaBench and EEMBC 1.1 embedded benchmarks. Results show that the loops that were successfully parallelised account for only a minor 10% of the total execution time. In addition, the author reports that in 38% of the cases that parallelisation failed, this was due to unknown loop trip count. The main limitation of

this work is that it does not identify or quantify available parallelism, i.e. parallelism that would be available if the limitation of static-analysis are relaxed. Additionally, it does not provide any details about the speedup potential, the granularity of the parallelism extracted or the means of exploitation (e.g. TLP vs. SIMD).

Kejariwal et al. The coverage studies from Kejariwal et al. [69, 67, 68] largely overcome these limitations and present a more thorough evaluation of the speedup potential that covers SIMD, TLP and speculative-TLP (TLS). One of the main contributions of these studies is that they use manual inspection in addition to automatic analysis (using the Intel ICC) in order to identify the amount of true loop-level TLP. In addition, their reported results are based on a baseline that incorporates both auto-vectorisation and software-pipelining. As we will discuss later in the experimental methodology chapter, it is crucial for providing a *fair* comparison to perform these single-threaded optimisations in the sequential baseline.

More specifically, in [67] the authors used SPEC2000 for their evaluation and their figures show that the coverage of DOALL parallelism for the majority of the floating point applications is very high (higher than 70%), considering either most or outermost loops. At the same time, however, their choice to report both automatically and manually extracted DOALL loops using one metric conceals the inability of modern production and research compilers to be effective in this task. In fact numerous studies, including [66, 145], have reached the conclusion that although advanced analyses like the ones referenced in [69], can be effective in individual cases the extraction of coarse-grain parallelism from full applications is bound to fail. For instance, in [66] the authors use the SUIF research compiler on a set of embedded benchmarks. Their conclusion is that there is no single-point of failure for static analysis and even if oracle pointer disambiguation, array privatisation and reduction is assumed, only 29% of the true DOALL will be statically identified.

Another significant result of these studies is that if DOALL loops are excluded, the speedup potential of any of the available TLS methods is surprisingly low; 39% on average for SPEC2000 [67] and only 6% (geometric mean) for SPEC2006 [69]. The latter is actually reduced to the scale of 1% if a realistic threading overhead of 1000 cycles is accounted. These results confirm our insight and strengthen our original motivation to focus on methods that leverage profile-driven analysis to uncover coarse-grain loops that are inherently parallel but compilers fail to automatically parallelise. Nevertheless, it is important to note that both studies evaluate the potential of relatively simple parallelisation techniques, exploiting only loop-level parallelism (speculative or not). In fact, the experimental results presented in this thesis clearly show that in many cases employing more sophisticated parallelisation transformations, like hierarchical pipelining, leads to significantly higher coverage and speedups than the ones reported in [67, 69]. For instance, in section 6.3 it is shown that for *bzip2* from the SPEC CPU2000 benchmark suite the proposed profile-driven approach effectively parallelises

loops that account for more than 98% of the sequential execution time, and lead to speedups of up to 4.7. In contrast, for the same benchmark in [67] the authors report a coverage of less than 80% for both speculative and non-speculative parallelisation. Therefore, the aforementioned findings should not be accounted as a conclusive study on the performance potential of parallelisation in general. Additionally, in section 2.2 we review and compare with the most relevant parallelisation techniques that address the limitations of the parallelisation model assumed by these studies.

Finally, the same authors in [68] investigate the challenges of auto-parallelisation but this time in the context of embedded applications. They compile EEMBC 1.1, EEMBC 2.0 and MiBench using Intel ICC. A very interesting conclusion of this work is that many outer-loops in embedded applications cannot be parallelised due to library calls which are not analysed by the compiler or are inherently sequential (e.g. I/O). This observation actually motivates the parallelisation of partially-parallel loops using pipelining, a technique targeted by the techniques presented in chapter 6.

No matter how successful auto-parallelisation has proven to be in the domain of scientific applications, the main caveat of the vast majority of the approaches which are based on static analysis is that their application is restricted to subsets of general purpose programming languages. Pointer-based code, unknown loop bounds, complex control-flow or non-affine array-access functions force static analysis to be overly conservative. Consequently, production compilers or even sophisticated dependence analyses employed in research compilers fail to extract parallelism beyond the scope of the innermost loops from embedded applications [68, 62] or general purpose integer codes [17, 69, 161].

2.1.2. Dynamic Approaches

Static analysis no matter how sophisticated, is still bound to its inherent limitations. This, for instance, prevents the parallelisation of dynamic and sparse scientific applications even when coded in more restrictive languages than *C* like Fortran. In this context, *dynamic* (or *runtime*) analysis aims at exploiting the increased accuracy of the dependence information that becomes available at execution time to increase the parallelisation opportunities. Most software techniques that emerged from this concept during the last two decades fit in one of the two following major classes: (i) analysing the trace of memory references, and (ii) deferring static analysis tests.

Reference-based Approaches The first records all the possibly offending memory references at runtime. Then, based on this *perfect* information it reasons about independence. Reference-based approaches can be further divided in two groups based on the runtime mechanism utilised: (i) *inspector/executor* that first performs preprocessing at runtime and then executes iterations in an order that maximises parallelism (e.g. [124]), and (ii) *speculation* that optimistically executes loop iterations in parallel and if a dependence materialises computation is squashed and re-executed in a sequential

order (e.g. the LRPD test [116]). The complexity of a trace-based approach is proportional to the number of dynamic memory references and therefore its parallelisation potential is often diminished by the high runtime overhead. These studies although pioneering and influential for later hardware or hybrid software/hardware speculation systems have largely failed to demonstrate significant performance improvements in a more general context. Essentially, this first class of dynamic techniques together with static analysis represent the two opposite extremes in parallelisation technology.

Hybrid Approaches The second class in dynamic parallelisation, on the other hand, tries to fuse these two and largely independent worlds. It delays the computation of some symbolic expressions extracted during the static dependence testing until the point of the execution where the values of the variables are known. These techniques bear the potential of significantly reducing the runtime overhead while retaining the benefits of precise runtime information.

Predicated Array Data-flow Analysis Moon et al. in [91, 92, 90] propose Predicated Array Data-flow Analysis which is a framework that associates predicates to each data-flow value. Based on these predicates the compiler can extract simple tests that describe the conditions under which a dependence is not materialised or transformations like privatisation can be safely applied. An extensive evaluation of predicated analysis using the SUIF research compiler, presented in [91], reports successful parallelisation of more than 40% of the available parallel loops that were not parallelised by the baseline compiler. The benchmarks consisted of scientific applications from the PERFECT, SPECFP95 and NAS suites. One of the main drawbacks of this approach, however, is that it restricted to loop-invariant assertions that can be easily hoisted out of the loop body.

Hybrid and Sensitivity Analysis Hybrid Analysis (HA) [122] overcomes this limitation and additionally it offers a holistic approach that integrates the whole spectrum of runtime data-flow analysis, ranging from simple runtime predicates to detailed reference-based validation (LRPD test). This is accomplished using an inter-procedural aggregated reference representation (Run-Time Linear Memory Access Descriptor, RT-LMAD). Aggregated references can be used in different program levels (e.g. procedure, loop or iteration-level) to perform set and interval operations (e.g. union, intersection, subtraction). Based on these operations – that can be evaluated fully or partially either at compile or run-time – the compiler can effectively defer the decision for optimisations like *privatisation* and parallel *reduction* until the point of the execution that all the operands can be evaluated. The same authors only recently extended HA with a novel predicate extraction algorithm, called Sensitivity Analysis (SA) [123]. SA is based on a graph of predicates which enables the compiler to extract simpler optimistic approximations of the aggregate references, and thus to considerably reduce

the runtime overhead of dynamic parallelisation. The authors implemented SA using OpenMP directives, a method we also followed for parallel-code generation in chapter 5. The detailed evaluation included in [123] demonstrates that Sensitivity Analysis can effectively extract and exploit almost all the available parallelism from an extended set of scientific applications drawn from the Perfect and SpecFp benchmark suites.

STMlite In parallel to our work Mehrara et al. demonstrated a simple dynamic parallelisation system that targets scientific applications written in *C* [87]. The authors use a lightweight software Transactional Memory (Tm) system called STMlite to guarantee correctness. Their key observation is that restricting the use of a Tm to loop-level parallelisation only allows the relaxation of some requirements of general-purpose Tms that are critical for performance. More specifically, STMlite (i) does not monitor accesses to local variables, (ii) does not provide *strong atomicity* guarantees [1], and (iii) does not support *zombie* transaction detection. The last point is very critical since it enables STMlite to perform lazy conflict detection and subsequently it allows conflict detection to be decoupled from the parallel execution itself and be centrally performed by a separate thread, the Transaction Commit Manager (Tcm). Despite all these performance enhancements, the experimental evaluation in [87], which presents the speedups for six benchmarks from various SpecFp suites shows a limited potential. Only two applications show a speedup more than 2× on a system with 8-cores. In contrast to profile-driven parallelisation, this approach does not utilise profile-information to perform speculative privatisation or parallel reduction, using it only for profitability analysis. Thus many parallel loops that are in fact parallelisable but in their original form violate the sequential semantics are not parallelised at all or suffer from very frequent squashes. This is clearly demonstrated by the speedup achieved by *183.equake*, where STMlite fails to get any speedup but our approach achieves a significant speedup of almost 2× on 8-cores [2].

The majority of the dynamic or hybrid methods is focused on array-based scientific codes. Most importantly, they presume programming languages like Fortran with well-defined array operations, language-visible heap allocation, constrained pointers, and limited or no support for pointer arithmetic. For instance, for the majority of the applications that we consider in the data-parallelism study of chapter 5 – which are coded in *C* – sensitivity analysis would be unable to extract cheap scalar or aggregated reference assertions. Pointer aliasing, complicated control flow and external library routines, just to name a few, preclude precise array data-flow analysis. Thus dynamic methods in these languages are bound to resort to full reference tracing which leads to poor scal-

[1] *Strong atomicity* is required in general-purpose Tms to guarantee atomicity between transactional and non-transactional accesses to same memory location. During the execution of a parallel loop, however, all accesses are performed within the transactions and therefore *weak atomicity* suffices.

[2] Actually the scalability of *183.equake* when parallelised using the methods proposed in this thesis is only constrained from poor spatial locality that is caused from a poor memory-allocation policy in the original code. Experiments using a simple modification to the allocation code – which is also incorporated in a later version of the benchmark – achieves a speedup of 5.95.

ability. Therefore, dynamic analysis has found limited or no applicability to application domains where *C* is predominant, including and not limited to embedded, multimedia, data-mining, bio-informatics and scientific applications (e.g. EEMBC, MIBENCH, MEDIABENCH, PARSEC and BIOBENCH benchmark suites are entirely written in *C/C++*). As it is demonstrated in chapter 5 for these languages even state-of-the-art production compilers fail almost completely to extract exploitable data-parallelism.

Compared to profile-driven parallelisation, proposed in this thesis, hybrid approaches share the concept of utilising the precise dependence information which becomes available only at program execution time. Profile-driven parallelisation, however, utilise this information only off-line, at compile-time, aiming at the exploitation of *always-true* parallelism, which, although it cannot be statically verified, is safe for all *valid* inputs. This last hypothesis, although it can be systematically asserted using multiple representative inputs, ultimately demands user-approval to guarantee the safety of the proposed parallelisation. This usage scenario seems far less attractive than automatic parallelisation. It is, however, *pragmatic* since it applies in different application domains and configurations and most importantly in cases where both static and dynamic approaches have been proven to utterly fail. In addition, as this thesis demonstrates, when profiling is performed in the appropriate program representation it bears the potential of not just parallelism detection but also exploitation, thus alleviating the burden of manual parallelisation.

2.1.3. Hardware-assisted Approaches

In the section about hybrid approaches to parallelisation we reached the conclusion that despite the innovative ideas of the last two decades software-only approaches failed to deliver performance beyond niche settings (e.g. array-based scientific applications in Fortran). Hardware support for Thread-Level-Speculation (TLS) [72, 85, 129, 133, 119] revisits speculative execution originally proposed in the context of hybrid parallelisation but in a more general framework that addresses more forms of parallelism and applies to programs and languages that are not statically analysable. This is accomplished either with a hardware-only or software/hardware hybrid infrastructure that facilitates low-overhead program-state checkpointing and conflict detection. Including a complete review of the numerous mechanisms that have emerged in this field during the last decade is definitely beyond the scope of this thesis. Instead, we briefly discuss the key reasons that, although complementary to ours, such approaches fail to provide a realistic alternative.

Hardware-assisted approaches achieve modest speedups for sequential integer applications (e.g. SPEC CINT) which are hard to parallelise with conventional methods. However, this comes at the cost of increased complexity in the coherency protocol and the memory subsystem. Evaluating these factors along with the area, power and verification overheads of such hardware proposals makes their adoption from the micro-

processor industry even more challenging. In fact, the only commercial general-purpose CMP with thread speculation, the Sun® Microsystems Rock[3], was permanently discontinued after five years of development and before its final release [22]. This is also magnified in the embedded systems design world where such intrusive and costly approaches are of limited applicability. Additionally, the majority of TLS systems utilises $L1$ cache as a speculative buffer and therefore it is bound to relatively fine-grain TLP. Applications that feature coarse-grain parallelism like floating-point and streaming applications – and therefore have a greater potential for effective parallelisation under realistic constraints–, however, demand speculative state that exceeds by far the size of typical $L1$ caches (typically 32KB).

2.1.4. Interactive Parallelisation

.

Interactive parallelisation tools [60, 70, 16, 61] provide a way to actively involve the programmer in the detection and mapping of application parallelism. Although promising, the majority relies heavily on the user to provide the critical parallelisation information. In addition, only a few interactive tools automate the parallel-code generation, demanding from the user to perform this complex and error-prone task.

SUIF Explorer An interesting example which is focused on data-parallelism for Fortran programs is SUIF Explorer [157]. It combines the advanced static analysis and transformations of the SUIF compiler with profile information and a powerful interactive interface. The first profile analyser, called Loop Profile Analyser (LPA), collects information regarding the execution time of individual loops. Thus, it can provide the user with suggestions about the loops that are more likely to yield significant parallel speedups. However, the profitability hints are derived using simple heuristics like sorting loops based on the execution time of a specific input. On the contrary, the Machine-Learning approach adopted in chapter 5 offers automatically tuned rapid performance prediction across platforms. The feature of SUIF Explorer that is most relevant to this thesis is the Dynamic Dependence Analyzer (DDA), a profile-based technique to identify potential parallel loops. Although, DDA takes *induction* and *reduction* variables into account, the relevant information is still statically extracted and thus conservative. In contrast, we propose a more aggressive hybrid static/profiling-based technique that marks potential reductions at compile time and then verifies them during profiling of the IR. This way we overcome the limitations of imprecise alias analysis which in languages like C normally constraints reduction detection to single scalar statements.

[3]The underlying micro-architectural mechanism of the Rock processor, called Simultaneous Speculative Threading (SST), was originally designed to provide speculative execution under cache miss events. Then it was extended to provide simple Transactional Memory (TM) primitives that can be used appropriately by system software designers. Nevertheless, the length of speculation is at most a few thousand instructions.

2.2. Pipeline Parallelism

In the past parallelising compiler technology targeted mainly scientific applications with an abundance of data level parallelism. Hence, the scope was largely restricted to the detection and mapping of this particular kind of parallelism. The advent of chip multi-processors (CMPs) as well as the proliferation of multimedia consumer electronics has emphasised the importance of compiler tools that address the overwhelming task of parallelising popular multimedia and other integer applications that are typical to these domain. Unfortunately, these applications typically feature loops with inherently sequential operations, like I/O, and therefore demand the adoption of more flexible parallelisation schemes. *Pipelining* exploits parallelism by executing multiple regions (*stages*) from multiple iterations of a single loop across the available cores of CMP. In the paragraphs following we present a critical review of the most significant studies in the area of pipeline-parallelism extraction and compare it with the approach presented in chapter 6 of this thesis.

2.2.1. Non-speculative Approaches

Pedigree Among the first to propose pipeline parallelism as an alternative to DOACROSS [6] was Newburn et al. in [98]. They motivate their approach using the observation that single-directional dependences can sustain longer inter-processor communication latencies and thus achieve better performance and scalability in more realistic multi-processor configurations. Their approach is based on post-pass tool, called Pedigree [100, 99], that constructs a Program Dependence Graph (PDG) representation of the whole program starting from the optimised assembly code. PDG enables the extraction of parallelism beyond the scope of a basic block and ranging from individual instructions to nested loop iterations and conditionals. Special nodes are inserted in the representation to group different forms of parallelism (e.g. DOALL loops). Code-generation is then performed hierarchically on the PDG. A very important constraint, however, of the code-generation proposed in [98] is that threads have to synchronise at the end of each iteration in the case of an iteration body being partitioned to more than one thread. Therefore, inter-iteration parallelism cannot be exploited for non-DOALL loops. Additionally, in contrast to our approach, Pedigree uses a bottom-up approach to reconstruct the program representation and it is based on low-level assembly. No matter any benefits this choice might yield, it restricts memory-dependence analysis and thus it is difficult to be effective in the extraction of coarse-grain parallelism. Finally, the evaluation presented in [98] cannot be used to generalise its findings since it is limited to relatively simple data and signal-processing kernels (SDIO Benchmark Suite) used in aerospace and military applications.

DDA In [66], Dynamic Dependence Analysis is proposed as an aggressive, i.e. not safe and thus verified by user feedback, dependence checking method to uncover loop-level

parallelism. The authors extract the traces using a source-to-source transformation tool similar to ours implemented in SUIF [44]. In addition, they present a study of the parallelisation potential previously unexploited by the SUIF parallelising compiler for a set of DSP kernels and multimedia applications. They reach the conclusion that static analysis is too restrictive and parallelising compiler technology is failing to gain significant speedups on the studied benchmarks even in combination with advanced techniques like interprocedural analysis, reduction and perfect pointer disambiguation. The same authors extend their previous work in [65, 64] and show that dependence vectors extracted by means of DDA can be exploited to combine loop parallelisation and functional pipelining. This work, however, is focused on the general feasibility of using dynamic dependence information, and does not present a methodology for deriving a parallel implementation.

Valerio et al. Another early approach to task-graph extraction for embedded system synthesis is presented in [150]. The authors construct a DAG from the source code, with each node representing a set of program statements that execute sequentially and edges indicating (control/data) dependence between the nodes. Loops are encapsulated in a single node. Thus, this approach is not addressing the cases of loop-level parallelism, task-level parallelism within each iteration of a loop, or loops containing function calls. Inter-procedural propagation of dependences among the different functions is based on a call-graph that they extract by executing an appropriately instrumented version of the code. In addition, the instrumentation provides information about the size of the data structures and the execution time of each task. Comparing to our approach, which is also using source-code instrumentation, this approach is not addressing the problem of variability across different data inputs and the information which is extracted involves control flow information only. Dependence analysis is performed based on *def/use* information extracted by sequentially traversing an AST representation of the source code, resembling a single-pass forward data-flow analysis. Regarding the disambiguation of pointer references the authors follow a particularly conservative approach and do not comment at all on array accesses, which leads us to the conclusion that they handle an access to an array-element as an access to the whole object. The granularity as well as other interesting properties of the extracted parallelism are largely unknown since the authors present only the number of tasks extracted for each of the benchmarks. As a concluding remark, this approach seems to be tightly oriented to hardware/software co-synthesis, and thus overly restrictive and of doubtful applicability to other domains.

Dai et al. In [32] Dai et al. consider the case of automatically parallelising packet-processing applications for parallel Network Processors. The authors present a partitioning algorithm that incorporates quite a few interesting features like dissecting the CFG of the loop-body in non-control-equivalent points and handling multiple loop-exits. Despite its valuable contributions, this approach seems to be considering only scalar

inter-thread dependences and in general, it lacks the generality of subsequent studies that address the parallelisation of a wider domain of applications.

Thies et al. The work presented by Thies et al. [141] is most relevant to our work. The authors propose a profile-based parallelisation methodology for applications that exhibit streaming computation patterns. They use a library of macros to *manually* annotate the boundaries of the pipeline stages at the source-level. In a subsequent step the correctness of the user-defined partitioning is verified using a binary-instrumentation tool. A runtime systems manages multiple processes communicating through pipes. This simplifies code-generation as each pipeline stage (i.e., a process) has its own private address space and no explicit privatisation is necessary. However, unnecessary overhead for communication and process management is introduced. We consider it a major disadvantage that the user of this methodology is required to both have a good knowledge of the algorithm to parallelise and its actual sequential implementation. Manually determining profitable partitionings is generally hindered by deeply nested functions or loops and idiosyncratic programming styles. In addition, the use of macros and the manual "matching" of the communication operations, especially in the case of variably-sized objects further complicates the work flow. This is actually demonstrated by the fact that most of the proposed partitioning in [141] are highly unbalanced or fail to unwind sequential (e.g. I/O) and parallel operations. Finally, the choice to delegate work mapping and scheduling decisions to the OS scheduler and resorting to system-call based communication primitives may limit parallel scalability.

DSWP *Decoupled Software Pipelining* (DSWP) is a recent development and works by statically splitting programs into critical path and off-critical path threads that run concurrently on thread-parallel architectures. Unlike, DOACROSS parallelism or approaches that extract completely independent threads, DSWP requires the flow of data among the threads to be acyclic.[106] first proposed DSWP for parallelising sequential general-purpose applications using fully-automatic analysis and code-generation. DSWP offers high flexibility and precision since it operates on a low-level instruction based representation, augmented with high-level memory dependence information. The key advantage of this approach is in the exploitation of latency-tolerant pipeline fashion parallelism across multiple basic blocks and control-flow paths. Effectively, DSWP performs global instruction scheduling and exploits ILP like traditional software-pipelining, but at the same time has broader scope and utilises multiple cores and functional units. The main difference to the approach presented in this thesis is that DSWP operates on a low-level instruction based representation and despite the detailed memory dependence information it cannot apply aggressive privatisation. This has an impact on the scope of the resulting thread-level parallelism. DSWP requires fine-grain communication primitives to communicate register values, and relies on the cache hierarchy to communicate memory dependences. In contrast, we target coarser-grain parallelism and memory-based

communication which gives rise to pipelines broadly resembling the algorithmic stages of the application. This enables us to use commodity hardware and longer communication free intervals, and unlike DSWP we do not rely on special micro-architectural support to unfold the full performance potential.

PS-DSWP Subsequently, Raman et al. revisited the original non-speculative DSWP and presented PS-DSWP [113], an approach exploits data-parallelism of pipeline stages with no loop-carried dependences. This is data-parallelism that spans more than two iterations of a partially sequential loop and it can be more beneficial than intra-iteration parallelism of previous approaches like [10, 28]. This feature is also supported by our approach that selectively applies *stage-replication* (6.2.4) when this reduces the execution time of the slowest stage. Nevertheless, PS-DSWP shared the same restrictions with DSWP since it is focused in finer-grain parallelism and relies on dedicated hardware support.

Rul et al. Rul et al. in [121] demonstrate the significance of profiling information in uncovering pipeline parallelism in general-purpose applications. They manually parallelise two applications using PTHREADS, but their approach lacks an automated code-generation methodology. In addition, they follow a simulation-based profiling approach which incurs a very high runtime overhead and lacks any facility to back annotate profiling information to higher level code in order to drive the code transformation and parallelisation process.

MAPS In [20] the MAPS methodology for extracting task parallelism from sequential C applications and mapping onto MPSOC platforms is presented. By clustering individual statements into larger clusters a *bottom-up* approach is taken, resulting in relatively fine-grained threads that for their efficient execution require a tightly-coupled architecture with special thread and communication support. MAPS assumes cleaned up code and only operates on single-level loops. It has been shown to deliver good results on small embedded benchmark kernels and the proprietary, low communication latency TCT platform, but it is unclear if the MAPS approach scales to larger applications or more generic hardware platforms.

MPA MPA [10] is a parallelisation exploration tool for MPSOC platforms. It requires the user to provide a parallelisation specification that specifies the functions and loops to be assigned to each thread. Then the tool utilising a static data-flow analysis based on Factored Use-Def chains (FUDS) generates separate functions for each thread and inserts the inter-thread communication/synchronisation. Contrary to our pipeline approach, MPA allows arbitrary communication between the threads and not only unidirectional transfers, thus it follows a paradigm resembling DOACROSS execution. Communication is performed using First-In-First-Out (FIFO) queues but it only

supports scalar variables and relies on the user to provide synchronisation for more complex structures (arrays or dynamically allocated memory). Furthermore, MPA assumes an execution model where all threads are synchronised at the end of each iteration using a barrier. Therefore, intra-iteration parallelism only is exploited. Finally, in MPA the iteration space of nested loops can be distributed across more than one thread but only using a static iteration scheduling specified in the parallel specification. Hence, compared to our code-generation approach, MPA does not support uncounted nested loops and dynamic iteration distribution.

Cordes et al. Cordes et al. in [28] focus on the problem of optimally partitioning a program into parallel tasks such that the total execution time is minimised under the specific architectural constraints given as additional input. Their bottom-up approach considers different levels for the partitioning of the hierarchical task graph. For the evaluation of the different alternative partitions at each level the authors propose an Integer Linear Programming formulation that models both the data communication and task creation overheads. Hence, the algorithm determines which parts of the program should form a task as well as the number of processors that should be assigned for each task if it is parallel. Task extraction utilises a hierarchical program representation but unlike the Program Dependence Graph introduced in section 6.2.1 control dependences are not explicit. Therefore, their partitioning can only extract parallelism from control-equivalent sections that cannot span multiple loop levels and conditionals. Additionally, this implies that their approach can only extract data-parallelism from within a specific instance of a task or tasks that can be independently executed. Compared to *stage-replication* (6.2.4) that exploits data-parallelism across tasks of different loop-iterations the scheme of Cordes et al. is less flexible and thus less applicable to applications with no intra-task parallelism.

Paralax Only recently and in parallel with our work Vandierendonck et al. presented in [153] a parallelisation framework, called Parallax, that enables the parallelisation of integer applications. Parallax is based in a set of user annotations that assign data-flow properties to data and functions which are critical for parallelisation. For instance, one of the most useful annotations is KILL which informs the compiler that the memory pointed-to by the specified pointer is invalidated. This is in deep contrast with previous approaches like [141] or OPENMP which demand from the user to explicitly designate parallelisation structures (e.g. DOALL loops or pipeline stages). Although these implicit parallelisation hints give the flexibility to the compiler to automatically transform the code without being constrained from a specific parallelisation paradigm, it remains unclear whether these are more *intuitive* and *easy* to use from the programmers perspective. The authors try to address this issue by utilising a profiling-based initial stage similar to our previously published approach [145]. Profiling information is then used

to hint the users for the structures which if annotated will yield more parallelisation opportunities.

The key difference with our approach is that our compiler is aiming at performing aggressive parallelisation based on the profile-information as a first step. And only then it informs the user for the exact data and control-flow dependences that had to be relaxed to achieve this. Additionally, although the code-generation of Paralax is not described in detail, the experimental evaluation leads us to the conclusion that they do not consider the case of multi-level loops. Furthermore, the authors do not present a specific partitioning strategy and do not perform guided *function-splitting*, a technique that enables us to uncover and exploit parallelism which is deeply nested in the call hierarchy [4]. This seems to be the reason that the achieved scalability of their approach for *bzip2* compression – the only common benchmark with our experimental evaluation – is considerably lower (by a factor of two). Finally, Paralax annotations like SYSCALL that instructs the compiler to assume external side-effects and COMMUTATIVE that designates that changing the order of calls to a function is permitted are very similar to the instrumented wrapper functions used in our approach for memory allocation and I/O.

Nevertheless, we consider the work from Vandierendonck et al. to be complementary to our work. Our view is that these are examples of a new, rapidly developing class of semi-automatic approaches that prioritise automatic detection and exploitation over safety and only rely on the user for final approval. The precise semantics and the level of interaction with the developer, however, still remain to visited by future research.

2.2.2. Speculative Approaches

BOP Ding et al. introduce in [33] Behavior-Oriented Parallelisation (BOP). It is based on simple annotations where the user or an analysis tool specifies code *regions* that can be executed in parallel. Then, at runtime a software speculation system, which exploits the virtual-memory protection infrastructure (page-fault handlers) available in modern operating systems, guarantees that the proposed parallelisation does not violate program semantics. Several techniques, like value-based conflict detection, to minimise the overhead of speculation which in any case will never lead to slowdowns compared to sequential execution. Parallel regions in BOP consist of either independent tasks or segments of successive loop iterations. Nevertheless, BOP does not support partitioning a loop iteration to multiple stages, a key feature to enable true pipeline parallelisation. Another, important drawback of this proposal is that it does not support I/O operations within the parallel regions. As it is demonstrated in the evaluation section of chapter 6 this is a common pattern in the majority of streaming applications where computation is effectively decoupled from I/O operations that execute in

[4] Figure 6 in [153] clearly shows that the authors consider *sndMTF* to be sequential. In contrast, our partitioning strategy was able to extract two stages out of it. One that is subsequently merged with the *replicable* compression stage and a second sequential one that performs the file output

separate threads but in parallel to the data processing threads. Finally, most of the few applications considered in the empirical evaluation of [33] required a quite detailed understanding of the implementation and non-trivial modification in order to achieve good speedups.

Copy-or-Discard Tian et al. [142] focus on the efficient exploitation of pipeline parallelism using a data speculation runtime system which creates copies of static as well as dynamically allocated data on-demand. Similar to [112], this study handles only single-level loops and using a fixed pipeline skeleton of three stages. Effectively, this is only applicable to loops with only a few statements with loop-carried dependences (e.g. induction variable increment) in the beginning or end of the loop and a dominating middle stage. In that sense the scope of our approach is broader and it can be applied in more complex cases which as we show in section 6.3 are predominant in multimedia applications. In addition, the authors do not discuss or give any information about the problem of correlating the profiling information which is based on debugging information and binary instrumentation. This information is necessary to perform parallelism extraction and thread partitioning automatically. Their results show excellent scaling but do not report whole application speedups and in some cases presume that applications process multiple input files in parallel.

Speculative DSWP Vachharajani et al. [149] extended DSWP with support for speculation over rarely occurring control or memory dependences, thus extending its applicability and facilitating better balanced partitionings to more than one thread. Nevertheless, this extensions still relies on dedicated hardware support for both low-latency register communication and thread speculation.

SMTX In parallel to our work, Arun-Raman et al. presented Software Multi-Threaded Transactions (SMTX) [112] a speculation system that aims at supporting coarse-grain pipeline parallelism on top of commodity hardware. Its design, similarly to BOP, utilises multiple OS-processes instead of threads, and therefore it supports transparent private address-space for each pipeline stage. For the performance evaluation of [112] the authors follow a parallelisation scheme similar to PS-DSWP and report impressive scalability up to 24 cores for a set of integer applications. Parallelisation candidates were identified using a loop-level profiling approach built on top of the LLVM compiler. Unlike our proposal, parallelism extraction and the necessary code transformations were performed manually, but systematically. In addition, the case of multi-level loops and function-splitting, two techniques proposed in chapter 6 is not considered.

2.3. Commercial Tools

As we have already discussed, the ubiquitous adoption of multicore systems has led to an unprecedented increase in the market potential of commercial interactive parallelisation tools. In the following paragraphs we briefly present a few of them that appear to be relatively mature and more promising. In addition, we attempt to provide a high-level comparison with the main features of the work of this thesis [5].

CriticalBlue Prism In the first quarter of 2009 CriticalBlue, a company specialised in automated embedded system design solutions, released the first version of their multicore-software analysis tool called Prism™ [31]. Prism offers an integrated environment to support optimisation and debugging of parallel applications. It is based on hardware-specific instrumentation which also allows the developer to inspect the parallel overheads and bottlenecks of a parallel implementation on the specific target. Data dependence information is extracted using profiling. However, Prism uses a binary instrumentation approach and thus, unlike IR-profiling, it is fundamentally limited to use *imprecise* debugging symbol information to correlate dependence relations back to the source code. Another interesting feature of Prism is an interactive interface that enables the exploration of alternative parallelisation scenarios and their potential benefits. Interestingly, this is accomplished without demanding from the user to provide a full parallel implementation. Although these features can reduce the effort put by the developers in the exploration of the parallelisation design space, the great problem of deriving a parallel implementation is not addressed. Additionally, Prism is based on a instrumentation approach and therefore it is fundamentally restricted of utilising imprecise debugging symbols. Finally, the parallelism-mapping approach presented in chapter 5 bears the potential to substitute hard-tuned analytical models like the one implemented in Prism [105] with automatic and portable profitability analysis.

Intel Parallel Advisor Lite Intel® has also recently released an experimental tool called Parallel Advisor Lite [52]. Unlike existing tools from the same company that focus on interactive analysis, debugging and optimisation of already-parallelised code [53, 54], Parallel Advisor provides a set of simple annotations to let the user specify potentially parallel constructs in the original sequential code. Then, the tool informs the user for potential data-sharing problems that will arise and the performance potential of the proposed parallelisation. In contrast to the approaches proposed in this thesis, parallelisation should be manually performed by the developer, a process which is both long and error-prone. Moreover, Parallel Advisor is solely focused on data and a simple form of task parallelism. Thus, there is no support to express more advance parallelisation schemes like pipeline parallelism.

[5] This is, of course, inherently difficult due to limited access to the documentation and the technical specifications of these products.

Vector Fabrics vfAnalyst Only recently – Q1 of 2010 –, the startup company Vector Fabrics released a novel parallelisation environment called vfAnalyst™ [155]. vfAnalyst shares many features with CriticalBlue's Prism, however, it profiles dependences with an infrastructure that allows tracing of individual accesses back to specific source code expressions and data variables. In addition, it is able to automatically parallelise code that features data-parallel or *streaming* parallelism. In the case of memory dependences, however, vfAnalyst relies on the user to provide the appropriate communication and synchronisation code. As we will demonstrate in chapter 6, embedded and multimedia applications typically exhibit coarse-grain parallelism that requires privatisation and synchronisation of heap-allocated data structures, therefore this code-generation limitation of vfAnalyst can considerably constrain its parallelisation potential. On the contrary, our approach can effectively handle these issues using precise profiling information and runtime pointer disambiguation (section 6.2.3). Furthermore, our flexible code-generation scheme coupled with an automated top-down partitioning strategy (section 6.2.2) automatically identifies the candidate schemes that are likely to perform better and does not rely on the user to traverse the complex program hierarchy to uncover these loops manually.

2.4. Conclusion

In this chapter we presented a detailed comparison of profile-driven parallelisation with the current state-of-the-art. We addressed techniques that target DOALL loop parallelism but also proposals like *task* and *pipelining* that go beyond iteration-level parallelism.

As an alternative to a summary of the key features that differentiate our approach from prior-art in parallelisation we summarised the approaches that were reviewed in table 2.1. Some features are deliberately summarised or simplified for the sake of visualisation. The main observation is that IR-profiling despite its requirement for user-approval, it represents an interesting and competitive design point in the space of parallelisation methods. Not only it enables the implementation of many interesting features that may be individually or partially supported from existing approaches, but it also enhances their applicability. This is achieved with an automatic approach that identifies and exploits different levels of parallelism automatically, relying to the user only for the final verification. In contrast to leading static approaches that assume dedicated hardware support (Scalar Operand Networks or hardware speculation support), profile-driven parallelisation exploits high-level patterns in computation, extracting coarse-grain parallelism and thus enabling the exploitation of readily available commodity hardware.

Title	Citation	Source	Parallelism		TLS	H/W extension	Parallelism granularity	Dependence profiling	Approach	Heap objects	Array reduction	Pipeline specific features		
			data	pipeline								x-iteration	multi-level	replication
PADFA	[90]	Fortran	✓		software		coarse		auto	n.s.	static	n.a.	n.a.	n.a.
HA & SA	[122, 123]	Fortran	✓		software		coarse		auto	n.s.	static	n.a.	n.a.	n.a.
STMlite	[87]	*C*	✓		software		coarse	✓	auto	✓	static	n.a.	n.a.	n.a.
SUIF Explorer	[157]	Fortran	✓				coarse		interactive	n.s.	static	n.a.	n.a.	n.a.
Pedigree	[6]	*C*	✓				fine		auto					
DDA	[66]	*C*					coarse	✓	n.s.	✓	static	n.s.		
Dai et al.	[32]	*C*		✓			fine			n.s.		✓		
Thies et al.	[141]	*C*		✓			coarse	✓	annotation	manual		✓	✓	✓
DSWP	[106]	*C*		✓		✓	fine		auto			✓	✓	
Rul et al.	[121]	*C*		✓			coarse	✓	manual	n.a.		✓		✓
MAPS	[20]	*C*		✓		✓	fine		annotation			✓		
MPA	[10]	*C*		✓			coarse		annotation	n.s.				
Cordes et al.	[28]	*C*	✓	✓			coarse		auto	n.s.				
Paralax	[153]	*C*		✓			coarse	✓	annotation	annotate		✓		✓
BOP	[33]	*C*	✓		software		coarse	✓	annotation	✓		n.a.		
CoD	[142]	*C*		✓	software		coarse	✓	auto	✓		✓		✓
PS-DSWP	[113]	*C*		✓		✓	fine		auto			✓	✓	✓
Spec-DSWP	[149]	*C*		✓	hardware	✓	fine		auto			✓	✓	
SMTX	[112]	*C*		✓	software		coarse	✓	manual	✓		✓	✓	✓
Prism	[31]	*C*	✓				coarse	✓	manual	n.a.		n.a.		
P. Advisor	[52]	*C*	✓				coarse	✓	manual	n.a.		n.a.		
vfAnalyst	[155]	*C*	✓	✓			coarse	✓	semi-auto			n.s.		
IR-profiling	[145, 144]	*C*	✓	✓			coarse	✓	semi-auto	✓	hybrid	✓	✓	✓

Table 2.1.: Summary of important features for the approaches that are most relevant to this thesis (*n.a* and *n.s.* stand for *not applicable* and *not specified* respectively).

Chapter 3.

Background

This chapter presents background material about the programming models, benchmarks, compiler and hardware infrastructure used in this thesis. First, we present the *CoSy* compiler infrastructure which was used for the source-to-source parallelisation in section 3.1. Section 3.2 provides information about the architecture-specific compilers. Section 3.3 is a brief overview of the main features of OpenMP, a parallel programming model which is used extensively in the remaining of this thesis. Finally, we describe the evaluation platforms in section 3.5 and the application benchmarks in 3.4.

3.1. Source-to-source Compiler Framework

This section presents the *CoSy* compiler development system [1], a modular and extendible compiler infrastructure, on top of which we built the source-to-source tools proposed in this thesis. More specifically, CoSy was utilised for the implementation of both, the Ir profiling tool introduced in chapter 4, and the code-generation phase of the pipeline-parallelism approach which is presented in chapter 6. Although CoSy is an end-to-end compiler solution with features that range from multiple front-ends (Iso C 89, Dsp-C, etc.) to a very powerful and flexible Back-End Generator (Beg), we will only refer to the modules which are more relevant to this thesis.

First, in 3.1.1 we provide an overview of the overall architecture of the CoSy framework. Finally, a description of the middle-level Ir follows in 3.1.2.

3.1.1. Overview

A high-level picture of the CoSy compiler development system architecture is illustrated in figure 3.1. At the core of CoSy is Ccmir [2], a common middle-level Ir that serves as both the source and target representation for all the middle-level compiler passes (referred to as *engines* in the CoSy terminology). All the frontends (*anc0* in the case of Iso C) produce as output a program representation in Ccmir. Middle-level transformations and optimisations take as input and produce a Ccmir representation. This includes common optimisations like global subexpression elimination, dead-code elimination, etc. Although the program might be *"lowered"* as a result of a transformation pass, it is still expressed in Ccmir. Nevertheless, additional information like

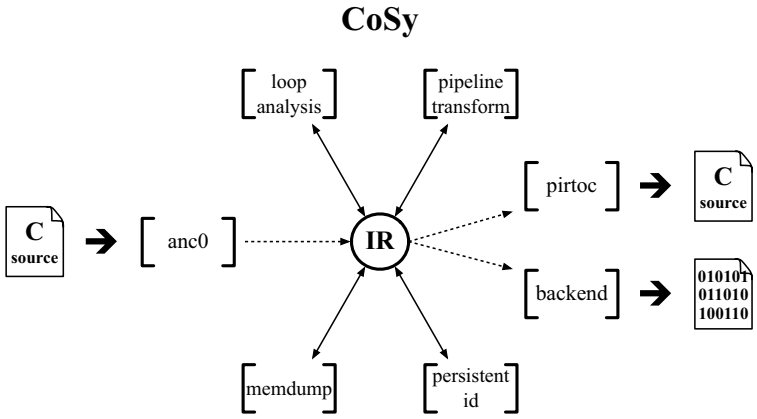


Figure 3.1.: High-level picture of the CoSy compiler development system. The infrastructure partly consists of multiple *engines* which perform transformations on a core intermediate representation of the program.

debugging information or high-level loop informations can be conveyed in the form of extensions to CCMIR, which is also a very powerful feature of CoSy. The rest of this thesis presumes programs written in *C*, however most of the compiler passes developed are building on top of CCMIR 's language independent representation. In symmetry to the frontend, the backend takes as input CCMIR and produce programs in the target-specific machine language. A special case of a backend is *pirtoc* which generates code in *C* and effectively enables the use of CoSy as a source-to-source transformation tool. Despite the fact that CoSy ships with a high-quality code generator for *x86* architectures – which is the main target architecture used of this thesis – the choice to use it only as a source-to-source tool was primarily dictated by our objective to provide a fair and robust comparison with existing auto-parallelising compilers (e.g. Intel ICC). In addition, this choice enabled us to utilise the support for established parallel programming interfaces (e.g. OPENMP) which is already built-in in some of the target-specific compilers.

3.1.2. CCMIR

CCMIR, briefly introduced in the previous section, is the common middle-level IR of CoSy. A key property of CCMIR is its generic and versatile design which enables it to be *independent* of the source language and the target architecture. It is a graph-based language where nodes represent statements and expressions, but contrary to common languages it does not have a syntax or textual representation. CCMIR is *fully-typed* and thus each node has a type throughout the middle-level transformations. Nodes also contain fields which are either *by-value* or *by-reference*. By-reference fields point to other nodes and effectively represent the edges of the graph. By-value are self-contained fields that do not extend the graph structure, e.g. the value of a constant. Although the structure of CCMIR is a directed graph, it can also be viewed as a tree

with an overlay of additional arcs. At the root of the tree is a node representing the whole compilation unit (i.e the code in a pre-processed source file) `mirUnit`, figure 3.2. Its descendants include nodes representing lists of types, global data and procedures.

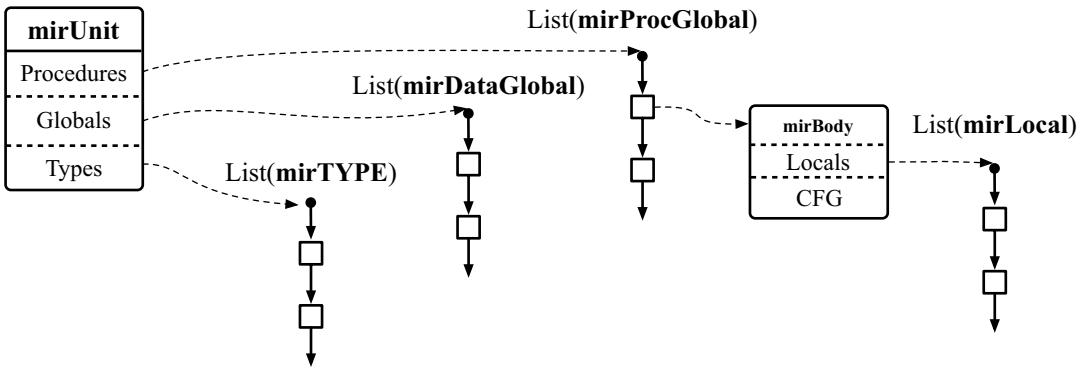


Figure 3.2.: CCMIR is a graph-based representation. Nodes are connected with edges that represent references. This figure shows a simplified view of `mirUnit`, which is the topmost node of every compilation unit (source file). `mirUnit` has pointers to lists containing Procedures, global variables and types defined in the current file.

Procedures

Besides fields like its name, type, and linkage each procedure contains a body which, if defined, consists of a Control Flow Graph (CFG) and a list of local variables. The CFG is a rooted directed graph. There are two special nodes, `mirBeginProcedure`, which is also the root of the CFG and `mirEndProcedure`, which is the only node with no successors. Nodes represent Basic Blocks (BBs), i.e. Single-Entry Multiple-Exit (SEME) lists of *statements*. Directed edges show the possible successors of the current BB in the control-flow (i.e. jump targets).

Statements

CCMIR makes a distinction between *statements* and *expressions*. Expressions are guaranteed to not modify the program state (variables, control-flow, dynamic memory etc.), i.e. expressions are *side-effect free*. Therefore, there is no dependence between the expressions of the same statement. On the contrary, statements can modify the program state and hence are not allowed to be reorder without prior dependence analysis. Statements are further divided in two abstract domains: (i) *Simple* statements like assignments to variable and function calls and (ii) *Control* statements that define the BBs where the control can be transferred after the current BB. Table 3.1 summarises the most important types of CCMIR statements. In every BB the last statement is always a control statement, i.e. a conditional or unconditional *jump*. `mirEndProcedure` has the additional requirement to be the only statement in its BB.

CCMIR does include special control statements (`mirBeginLoop` and `mirEndLoop`) to designate the *loop header* and *pre-header* of *natural* loops [1], however, these constructs

[1] The terminology follows the definitions in [95]

Statements		
Simple statements		
Name	**Description**	**Fields**
`mirAssign`	Rhs expr. evaluation and assignment of the result to the Lhs	*<Lhs expr. (address)>, <Rhs expr.>*
`mirEvaluate`	Evaluate expr. with no assignment of the result	*<expr.>*
`mirCall`	Function call with no return type	*<procedure expr.>, <actual parameters list>*
`mirFuncCall`	Function call with result assignment	*<procedure expr.>, <actual parameters list>, <result expr. (address)>*
`mirStackAllocate`	Stack memory allocation	*<type of pointer>, <size of memory>, <result expr. (address)>*
Control statements		
`mirCondAssign`	Conditional assignment	*conditional expr., Lhs expr. (address), Rhs expr.*
`mirBeginProcedure`	First statement of the entry BB	-
`mirIf`	Binary conditional branch	*<conditional expr.>, <then BB>, <else BB>*
`mirGoto`	Unconditional branch	*<target BB>*
`mirReturn`	Return value and then exit procedure	*<return value>, <exit BB>*
`mirSwitch`	Multi-target conditional branch	*<expr.>, <list of (value range, target BB)>*
`mirBeginLoop`	Loop pre-header	*<iteration count>, <loop header BB>*
`mirEndLoop`	Loop header	*<successor BB in loop body>, <loop exit BB>*
`mirEndProcedure`	The last and only statement of the exit block	-

Table 3.1.: Statements are divided into two groups: *simple* and *control*. This table lists the most important instances of these two types and their private fields.

are tailored for supporting zero-overhead loops in the code-generator rather than performing loop analyses. Loops are generally expressed CCMIR using lower-level control statements from table 3.1 like `mirIf` and `mirGoto`. As a result, CCMIR is not the appropriate representation for the needs of parallelism extraction – which primarily targets and performs transformations on loop structures. Therefore, for the rest of this thesis we utilise the *Loop-markers* extension of CCMIR [3], which provides high-level information about the loop structures. Due to their significance, Loop-markers are presented in detail in the next section, 3.1.3.

Expressions

Expressions are recursive data types and represent side-effect free computation. In the interest of brevity a list of available expressions is omitted. Figure 3.3 shows an example of the expression hierarchy for a simple *C* expression.

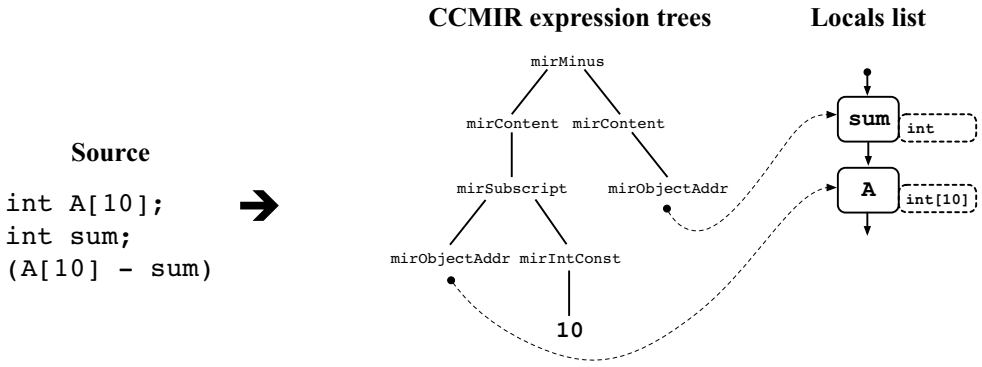


Figure 3.3.: The CCMIR hierarchical representation of a simple expression in *C*. Sub-expressions form a tree. The leaves point to objects (locals `A`, `sum`) or other structures (*opaques*) like integer literals (`10`).

3.1.3. Loop-markers

Loop-markers are an extension of the CCMIR that provide high-level information about the loop structures of a procedure. Loop-markers are constructed by the front-end for languages like *Fortran*. On the contrary, for *C* which lacks well-defined loop constructs CoSy extracts this information from the middle-level IR representation that is produced from the frontend. The loop-markers information can be updated using the *loopanalysis* pass. In the case of *irreducible* CFGs, i.e. containing loops with multiple entries, loopanalysis is not constructing any annotation. Contrary to low-level program representations where irreducible loops are common side-effects of back-end optimisations like global instruction scheduling, software-pipelining and code replication [147], in the relatively high-level representation that this thesis is focusing on is a rare phenomenon. In practice, for the wide range of programs used in the empirical evaluation this property was not found to be a significant constraint. Nevertheless, approaches similar to [46] can construct a loop-nesting tree with maximal reducible loops even in the presence of irreducible subgraphs, thus minimising the size of irreducible code. We plan to utilise such techniques in future work.

For the sake of clarity a few definitions are given before describing the main features of the loop-markers. These definitions follow the documentation in [3] but are also in line with the related bibliography [95].

3.1.3.1. Definitions

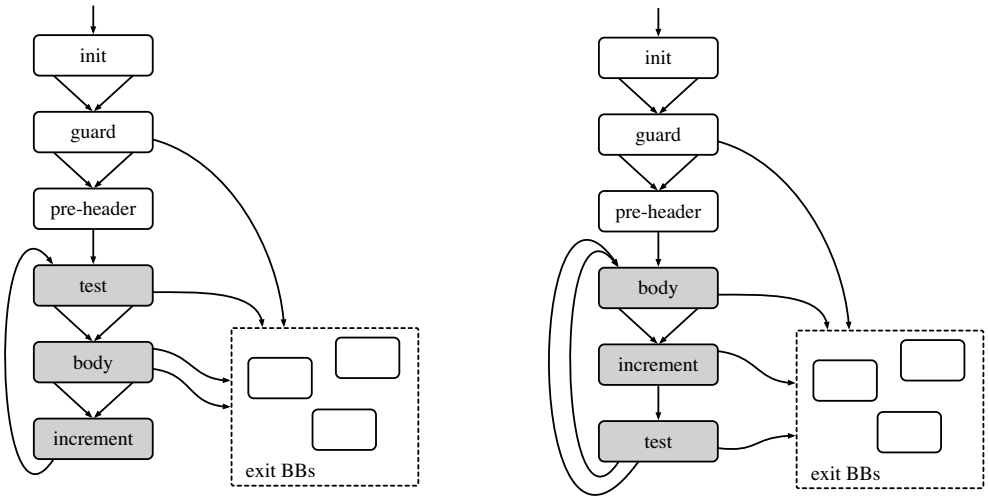
Natural loop *Natural loop* of a back-edge $v \rightarrow h$ in a CFG G is the subgraph of G consisting of: (i) the set of nodes V containing h and all the nodes from which v can be reached in the flowgraph without passing through h, and (ii) the set of edges E connecting all the nodes in the node set V.

Loop header In a natural loop characterised by the back-edge $v \rightarrow h$ BB h is called $loop-header$.

Loop entry $Loop-entry$ is a BB that belongs to the loop but it has a predecessor that it is outside the loop.

Loop exit A BB that is the successor of a BB belonging in a loop is called a $loop-exit$ of this loop.

Loop pre-header A BB that has as its only successor the header of a loop is called the $pre-header$ of this loop.



(a) $while-do$ loop **(b)** $repeat-until$ loop

Figure 3.4.: This figure shows how the different loop-elements defined by the Loop-markers framework connect depending on the form of the loop structure ($while-do$ or $repeat-until$). Shaded nodes denote *primary* elements i.e. elements that are part of the loop. Nodes with just one successor are SESE subgraphs. The rest are SEME.

The loop-markers infrastructure considers two different loop structures, *while-do* and *repeat-until*. Each such loop structure contains a well-defined set of $Single-EntryMultiple-Exit$ (SEME) and $node-disjoint$ subgraphs which are called *loop-elements*. Figure 3.4 shows the connections between the elements. In addition, table 3.2 summarise their characteristics. If a CFG does not contain any statements or BB that fit with the specification of a *primary* loop-element, the loopanalysis pass will create one with only a constant `mirGoto` statement. This feature is actually utilised extensively in the pipeline code-generation pass of chapter 6 to inject the inter-stage communication code.

3.2. Target-specific Compilers

This section summarises information about the compiler infrastructure used to generate executable machine code for the empirical evaluation.

Loop elements			
Name	Type	Primary?	Description
`Init`	SEME	–	Code for the initialisation of loop variables. Single/multiple exits to the loop-header (`Test` or `Body`).
`Guard`	SEME	–	Code that determines whether the loop will execute.
`Pre-header`	SESE	–	Loop-invariant code. Single exit to the loop-header (`Test` or `Body`).
`Test`	SEME	✓	Code executed at every iteration and determines whether the loop-body will execute. Multiple exits to the loop-body or/and loop-exit BBs.
`Body`	SEME	✓	Every BB that belongs to the loop but is not in the Test or Increment loop element. Multiple exits to loop-exit BBs and the `Increment` loop element.
`Increment`	SESE	✓	Code executed at every iteration and updates the loop variables. Single exit to the loop-body.

Table 3.2.: Brief description and main characteristics of loop-elements.

Most of the applications that we consider in this thesis feature ample fine-grain loop-level parallelism and therefore can be further optimised utilising the SIMD extensions available in most modern embedded and general purpose processors. Nevertheless, the primary focus of this thesis is on the extraction of coarse-grain Thread-Level Parallelism (TLP). We consider the exploitation of this form of parallelism to be *orthogonal* to our approach and therefore rely solely on the *auto-vectorisation* features of the target-specific compilers. All the measurements reported hereafter, including the sequential versions of the programs, are performed with auto-vectorisation enabled, thus guaranteeing comparison with the *strongest* available baseline.

3.2.1. Compilers for x86_64 Architectures

We used two different compilers when targeting the *x86_64* architectures, namely the auto-parallelising Intel ICC and GNU GCC. Version details, optimisation features and the exact command lines arguments that we used are shown in table 3.3 [2]. For the study on data-parallelism in chapter 5 we have used ICC for two reasons: (i) it offers state-of-the-art auto-parallelisation features and thus it can be used to contrast and compare to profile-driven techniques, and (ii) ICC's OPENMP implementation performed significantly better both in terms of absolute runtime and speedup over sequential execution for the benchmarks that we considered. On the other hand, for the study about pipeline-extraction we preferred GCC because it achieved higher performance when compiling the *C* code which is produced by *pirtoc*, the IR-to-*C* CoSy engine.

[2] The compilation flags used in the case of ICC are the ones Intel is using for reporting published SPEC results for systems similar to *M1*.

Intel ICC		
Version	10.1 (Build 20070913)	
Flags	-O2	Enable generally recommended optimisations. Optimisations include:
		-aggressive data dependence analysis -software pipelining -prefetch insertion -partial redundancy elimination -loop unrolling -copy and constant propagation -global register allocation -global instruction scheduling
	-parallel	Auto-parallelisation using threads.
	-ipo	Enable multi-file interprocedural optimisations.
	-openmp	Enable OpenMp.
	-ansi-alias	Assume conformance to Iso *C* aliasing rules.
	-xS	Optimised code-generation and auto-vectorisation for Sse, Sse2, Sse3, Ssse3 and Sse4.
	-axS	Auto-vectorisation targeting the Simd referenced above.
Environment variables	`KMP_AFFINITY=compact,0`	Bind threads to processors as closely together as possible.
	`KMP_LIBRARY=turnaround`	Optimise OpenMP runtime for turnaround time.
	`KMP_BLOCKTIME=infinite`	Enables busy-waiting when a thread remains idle.

GNU GCC		
Version	4.4.1	
Flags	-O3	Maximum optimisation level. Optimisations include:
		-loop vectorisation on trees -global interprocedural subexpression elimination, constant and copy propagation -interprocedural constant propagation -range value propagation -partial redundancy elimination -function inlining
	-march=core2	Tune and generate code specifically for Intel Core2 architecture, 64-bit extensions and Mmx, Sse, Sse2 and Sse3 Simd extensions.

Table 3.3.: Details of the two compilers that were used for generating native code on the *x86_64* platform (*M1*).

IBM XL for Multicore Acceleration		
Version	0.9	
Flags	-O5	Maximum optimisation level. Optimisations include:
		-interprocedural data-flow and alias analysis -aggressive code motion -inlining & cloning -constant propagation -dead-code elimination
	-qstrict	Ensures correctness for -O5.
	-qarch=cell	Specifies target architecture.
	-qipa=partition=minute	Minimum code-overlay buffer size.
	-qipa=overlay	Automatic code overlays.

Table 3.4.: Details of the compiler that was used for generating native code on the Cell BE platform (*M2*).

3.2.2. Compilers for the IBM Cell BE

Applications for the Cell Be were compiled using the Ibm Xl single-source compiler. Xl supports OpenMP directives and generates code for both the general-purpose processor and the synergistic units. To leverage the constraints of the Spe local storage, the Xl compiler is employing advanced compilation techniques like *code overlays* and a *software caching* [34, 102]. Still for a few programs or data inputs we were unable to produce a valid executable. Table 3.4 includes version information and the compilation flags that were used.

3.3. OpenMP

In the study on extraction of data-level parallelism we opted to use the OpenMP programming interface for parallel code-generation. This decision was primarily dictated by the following facts:

- OpenMP is the *de-facto* standard for manual data parallelisation in *C* and *Fortran*. Its specification, which is the outcome of a multi-vendor board, includes an *architecture-independent* and *well-defined* execution and memory model. These provided an abstraction layer that removed the burden of specifying an intermediate parallel architecture from scratch.

- The availability of mature and robust industrial compiler products for many parallel architectures, e.g. *x86_64*, Power® , Sparc, guarantees *performance* and *portability*. Additionally, it enables us to exploit the high-quality target-specific code-generation – which is available from these compilers – without sacrificing target-independence in our parallelisation approaches.

- Availability of well studied and widely used sequential benchmarks (e.g. Nas-Pb and Spec2000 Fp/Spec Omp2001) that are also *hand-parallelised* using

OpenMP. This provides a *strong* and *realistic* upper limit for the speedup that can be achieved with parallelisation. Contrary to the established methodology of comparing against a sequential baseline – which always conceals the common fallacy of assuming linear speedups – this approach evaluates parallelisation into a more realistic context. In addition, this is particularly useful in identifying which are the main obstacles that prevent existing compiler technology from matching the performance achieved by parallel programming experts.

- *Gcc*'s support for OpenMP offers an *open* and highly portable alternative for many platforms. Thus, making our parallelisation approach readily available to test in many platforms.

3.3.1. Overview

OpenMP is a Application Program Interface (Api) that supports multi-platform parallel programming for *C/C++* and Fortran [103]. It follows the shared-memory programming model. Nevertheless, it has been successfully ported to architectures which are not strictly adhering to the shared-memory model. For instance, the Ibm Xl *C/C++ for Multicore Acceleration for Linux* [51, 34, 102] targets the Cell BE where each accelerator has access to the coherent shared memory through a Dma controller but is restricted to perform computation using direct access to a private software-managed local memory. The main components of OpenMP are a collection of (i) compiler directives, (ii) library routines, and (iii) environment variables.

The sole purpose of the following subsections is to provide the necessary background for the chapter about data-level parallelism extraction where we use a code-generation methodology built on top of OpenMP. The main compiler directives and library routines are presented in 3.3.2. A more detailed description of the execution and memory model follows in subsections 3.3.3 and 3.3.4 respectively.

3.3.2. OpenMP Directives

3.3.2.1. Definitions

Structured block An executable statement with a single-entry at the top and a single-exit at the bottom. The statement can be either simple or compound.

Construct A *construct* includes the directive plus the code included in the lexical extent of the associated statement, loop or code block.

Region A *region* includes all the code encountered during the execution of a specific instance of a construct. Hence, it contains the code of all the routines called within the associated construct.

Constructs		
Name	**Arguments**	**Description**
`parallel`	—	Starts parallel execution, creating a new team of threads. A barrier is implied both on entry and exit from the region.
`for`	—	The iterations of the FOR loop that follows are distributed among the threads of the current team based on the current scheduling policy (default or specified with the `schedule` clause). An implied barrier is enforced at the end of the loop.
`parallel for`	—	A `parallel` construct combined with just one nested FOR.
`threadprivate`	*(list)*	Defines that a global-lifetime variable will be replicated, so that each thread will have access to a private and persistent copy of the original variable (see 3.3.4.1)
Synchronisation Constructs Constructs		
`barrier`	—	Specifies a point of a parallel region that no thread of the current team is allowed to proceed until all the threads of the team have reached it.
`single`	—	Specifies that the structured block that follows will be executed only by one thread of the current team. A barrier for the whole team is implied at the exit of the block.
`master`	—	Specifies that the structured block that follows will be executed only by the *master* thread of the current team.
`critical`	*[(name)]*	Enforces exclusive access to all the structured blocks that have the same name or the default unspecified name.
`flush`	*[(list)]*	The *private view* of the specified variables, or in the default case the whole program state of the executing thread, is written back to the *memory*. Upon a new access to this variable a new copy will be fetched from *memory* (see 3.3.4.3)
Synchronisation Clauses		
`nowait`	—	The implied barrier at the end of a worksharing construct (e.g. `for`) is removed.
Data-sharing Clauses		

`private`
`firstprivate`
`lastprivate`
	(list)	Assigns the respective sharing attribute to the specified list of variables within the region of the relevant construct (see 3.3.4.1).

`copyin`	*(list)*	Assigns the *copyin* sharing attribute to the specified list of *threadprivate* variables (see 3.3.4.1).
`reduction`	*(operator:list)*	Assigns the *reduction* sharing attribute to the list of specified variables. In addition it specifies the specific reduction operation to be used (e.g. a parallel reduction for a specific variable using the specified *operator*.
Scheduling Clauses		
`schedule`	*(kind[, chunk])*	Specifies the iteration distribution policy of the relevant `for` construct. Kind can be one of `static`, `dynamic`— or `guided`. *chunk* specifies either the number of iterations distributed among the threads in the case of `static` and `dynamic`, or the minimum number of iterations distributed in the case of `guided`.

Table 3.5.: Directives utilised by the code-generator in chapter 5.

Library Routines	
Prototype	**Description**
`int omp_get_num_threads()`	Returns the number of threads in the current team.
`int omp_get_num_threads()`	Returns the number of threads that would form a team if a parallel construct were to be executed.
`int omp_get_thread_num()`	Returns the sequence number of the calling thread with respect to the current team ([0,`omp_get_num_threads()`])
`void omp_init_lock() (omp_lock_t *)`	The specified lock is initialised in the *unlocked* state.
`void omp_destroy_lock() (omp_lock_t *)`	The specified lock is set to the *unitialised* state.
`void omp_set_lock() (omp_lock_t *)`	The calling thread blocks at least the specified lock is in *unlocked* state.
`void omp_unset_lock() (omp_lock_t *)`	The calling thread sets the specified lock in the *unlocked* state and returns.

Table 3.6.: library routines utilised by the code-generator in chapter 5.

Worksharing construct is a construct defining the distribution of work among the threads in the team executing the construct. In *C* these are the following: `for`, `sections` and `single`.

The main *directives*, *clause* and library routines utilised by the code generator in chapter 5 are listed in tables 3.5 and 3.6. Figure 3.5 is a simple code excerpt from the NAS PB benchmark CG which demonstrates the parallelisation of a FOR loop using a parallel *reduction* `reduction` clause. Parallel reductions are a common and extremely useful design pattern in parallel computation. It enables the parallelisation of loops in the presence of loop-carried dependence that are caused by *associative* and *commutative* operations like the one in line 8 of figure 3.5.

```
1  void compute{
2      int x[N];
3      int y[N];
4      int i, d, sum;
5
6      #pragma omp parallel for reduction(+:sum) private(d)
7      for (i=1; i < N; i++) {
8          d = x[i] - y[i];
9          sum = sum + d * d;
10     }
11 }
```

Figure 3.5.: A `parallel for` construct in OPENMP. Variable `d` is privatised and thus each thread accesses a private copy with the same name. Variable `sum` is part of a parallel reduction using a + operator. The compiler will automatically generate code to accumulate the partial results in the original variable sum at the end of the parallel region (line 10).

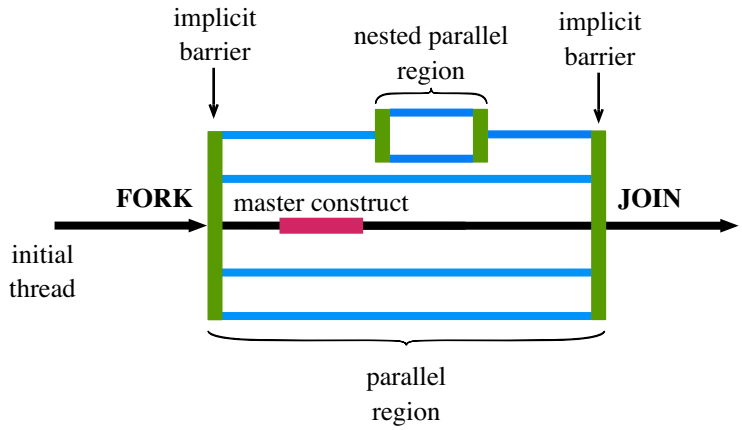


Figure 3.6.: OpenMP follows the fork-join execution model. In the beginning, there is only the initial thread. New threads are forked when a `parallel` construct is encountered by one of the existing threads. Hence, as shown in the figure, nested fork-join hummocks can be constructed. At the end of each work-sharing construct threads are synchronised on a barrier.

3.3.3. Execution Model

OpenMP follows the *fork-join* model of parallel execution, figure 3.6. A program in OpenMP begins its execution as a single sequential thread, named the *initial thread*. The initial thread also defines a virtual parallel *region* which consists of the whole sequential program. When a new work-sharing construct is encountered by any of the existing threads, a *team* of threads is forked. The thread encountering the construct is called *master* thread. Each thread in the team is executing the code in the construct. At the end of the parallel construct all the threads synchronise on an implicit barrier. After the barrier (*join*) only the master of the team resumes execution. Upon entering a worksharing construct the work is split among the threads of the current team. Finally, at the end of the worksharing construct there is also an implicit barrier like in the case of a parallel construct but this time all the threads in the team resume executing the subsequent code of the parallel region.

A team consists of the master plus zero or more threads. The exact number is determined dynamically based on a set of dynamic and static information and policies (e.g. `num_threads` clause, support for nested parallelism, `OMP_NESTED` and `OMP_DYNAMIC` environment variables). In addition, nested regions are valid, hence nested thread teams are also possible [3]. Nevertheless, in the context of this thesis we are only utilising the simplest option of using single level (i.e. `OMP_NESTED` is false) parallelism and a team with as many threads as the number of available cores (i.e. `OMP_NUM_THREADS`).

It is important to clarify that the execution model defines the intended behaviour of an OpenMP program and not the implementation specifics. For instance, most efficient implementations would use a single or multiple pools of Os-threads which are

[3] Actually supporting nested regions of more than one thread is implementation defined.

spawned on-demand but are not finalised unless the program ends. By reusing threads the thread-spawning overhead is amortised across the entire execution.

3.3.4. Memory Model

OpenMP specifies a *relaxed-consistency*, *shared-memory* model. All threads have access to a common memory space to store and retrieve variables, called the *memory*. In addition, each thread is provided with a *temporary view* of the memory. This practically allows the implementation of any hardware/software caching mechanism (e.g. register, caches, etc.) which will result to accessing a copy of the data in memory.

3.3.4.1. Sharing Attributes

Each parallel directive defines two kinds of access to variables which are within the *lexical scope* of the structured block, *private* and *shared*. Each variable referenced within the structured block has an *original variable* which is the variable which is out of current block but has the same name. References to shared variables are translated to references to the original variable. References to private variables translate to references to a private version which is has the type and size of the original. Private versions are allocated in memory for every thread in the team that executed the construct, with the exception of the master thread which can keep the original.

In addition to the private and shared attributes which are mutually exclusive, there are four additional attributes for private variables:

Threadprivate: Global-lifetime variables declared *threadprivate* are replicated and each thread retains its private copy. The values stored in the copy are persistent across parallel regions and in any point of the program in general [4]. In sequential regions the references are made to the copy of the initial thread.

Firstprivate: *Firstprivate* variables have their private copy initialised with the value that the original variable had just before the construct.

Lastprivate: If a variable is defined as *lastprivate*, its original variable gets assigned the value that was assigned to the private copy during the execution of the last iteration of the loop construct or the execution of the lexically last section.

Reduction: Variables participating in a *reduction* clause are a special case of private variables which are initialised with the appropriate *identity element* for the specific reduction operator. In addition, the original variable is updated by applying the operator to each of the private copies.

[4] There are a number of preconditions for this to hold for any thread but the initial one. Nevertheless, in this thesis we are only considering a constant number of threads with no nesting and all these conditions hold.

Copyin: applies to threadprivate variables only. A variable defined *copyin* has its thread-private copies initialised with the value of the master thread before entering the parallel region.

Sharing inference for construct variables The sharing attributes of a variable which is referenced within the lexical scope of a construct can be determined in one of the following ways: (i) predetermined, (ii) implicitly, and (iii) explicitly.

When appearing within the lexical scope of a construct the following sets of variables have *predetermined* sharing attributes :

1. Variables of automatic storage duration declared in the lexical scope of the structured block are private,
2. The loop iteration variable of an OpenMP `for` or `parallel for` construct is private.
3. Heap allocated data are shared.
4. Data with static allocation are shared.

Explicitly determined are the variables which are referenced in one of the sharing clauses of the construct.

Finally, variables which do not belong in one of the former two types are said to have implicitly determined sharing attributes. There are three cases of variables with implicitly determined sharing:

1. In a `parallel` construct which has a `default` clause, in which case they get their sharing attributes from the argument of `default`. Thus in *C* they can only be shared[5].
2. In a `parallel` construct without a `default` clause are implicitly shared, in which case they are determined to be shared.
3. In another construct, in which case they inherit the attributes of the enclosing context.

Sharing inference for non-construct variables Variables which are referenced by a region but are not within the lexical scope of the construct have predetermined sharing attributes in one of the following cases:

Variables referenced in a region but not within the boundaries of a construct [6] have their sharing attributes determined with a more straightforward way:

[5] Or invalid in the case of `default(none)` which demands explicit declaration for not predetermined attributes.

[6] Obviously, this also includes variables which are referenced in the code of routines called within the region.

1. Variables in called routines with static storage duration are shared.
2. Heap allocated data are shared.
3. File-scope or namespace-scope variables are shared unless they are threadprivate.
4. Formal arguments of routines which are passed *by-reference* inherit the attributes of the actual argument passed.
5. All other variables in called routines are private.

3.3.4.2. Coherency

The standard specifies that the following two are unspecified

1. The value stored in a shared variable when multiple threads write to it without appropriate synchronisation.
2. The value read from a shared variable by one or more threads when at least one of them writes to that variable without appropriate synchronisation.

3.3.4.3. Consistency

The memory model defined by OpenMP is clearly a relaxed-consistency model, since each processor is allowed to cache a temporary view of the memory which might not be consistent at all times. In order to enforce the consistency between the temporary view and the memory OpenMP defines a *flush* operation. Flush is always associated with a set of variables, called the *flush-set*. For writes to memory flush is a synchronous operation since it does not complete until the temporary view of the variable has been updated in memory. For reads flush guarantees that the temporary view of the associated variable will be discarded and the next read will be from memory. In addition, flush also serves as a memory barrier since it guarantees that no memory operations will execute from this thread before its completion. It should be clear by now that the OpenMP consistency model diverges from the definition of classic *weak ordering* because synchronisation operations in OpenMP (i.e. flush operations) are guaranteed to be ordered with respect to each other only if they refer to a common variable.

Flush operations are either explicit or implicit. An explicit flush operation is defined using the `flush` directive. An implicit operation related to a specific variable is implied at entry and exit from `atomic` regions, where the flush refers only to the variable that is atomically updated.

If there is no variable list associated with the directive OpenMP assumes that the whole temporary view of the current thread should be flushed. Such a general flush is implied in one of the following cases:

1. In a `barrier` construct.

2. At entry and exit from `parallel`, `critical` and `ordered` regions.

3. At exit from work-sharing constructs unless a `barrier` or `nowait` clause is used.

4. In a `omp_set_lock()` and `omp_unset_lock()` call and in the case of an `omp_test_lock()` that successfully acquires the lock.

3.4. Benchmarks

3.4.1. Overview

For the empirical evaluation of the methods proposed in this thesis we used a very diverse set of benchmarks, ranging from scientific applications to reference implementations of industry-standard multimedia codecs and digital signal processing kernels. This choice was primarily dictated by the diversity of the *forms* of parallelism that is inherent to different types of algorithms and applications. In the subsections that follow we provide a high-level description of each application. Properties relevant to the context of specific parallelisation studies are discussed in greater detail in the evaluation sections of chapters 5 and 6.

Benchmarks					
			Parallelism		**Manual**
Name	**Source**	**LOC**	data	pipeline	**Parallelisation**
BT	NAS-PB 2.3	4058	✓	–	NPB2.3-OMP-C
CG		1275	✓	–	
FT		1608	✓	–	
IS		1063	✓	–	
LU		3952	✓	–	
MG		1651	✓	–	
SP		3402	✓	–	
ammp	SPEC CFP2000	13487	✓	–	SPEC OMP2001
art		1289	✓	–	
equake		1515	✓	–	
bzip2	SPEC CINT2000	4649	–	✓	–
mp3player	EEMBC 2.0	23606	–	✓	–
mpeg2dec		23606	–	✓	–
cjpeg		29371	–	✓	–

Table 3.7.: Summary of the benchmarks used in the evaluation of the parallelisation approaches developed in this thesis.

Table 3.7 summarises the most important benchmarks properties and characteristics.

3.4.2. Scientific Applications

3.4.2.1. NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NAS-PB) [12] is a set of scientific programs which were originally designed from the NASA Advanced Supercomputing Division for the perfor-

mance evaluation of parallel supercomputers. These benchmarks have been widely [7] used by industrial and academic researchers in studies from the fields of parallel architectures, programming models and runtime systems. NAS-PB consists of five kernels and three applications which implement representative algorithms drawn from the field of computational fluid dynamics. The key feature of NAS-PB is that it provides both sequential and parallel implementations of the algorithms. More specifically, NAS-PB ver. 2.3 includes (i) a sequential version in *C/Fortran*, (ii) a parallel implementation using MPI that mainly targets distributed memory parallel systems (e.g. clusters of multiprocessors), and (iii) an OPENMP shared-memory parallel implementation. For the assessment of our parallelisation approach we used both the sequential and OPENMP -parallelised versions, which provided us not only with *absolute speedup* figures but also with an *upper limit* for what manually extracted parallelisation can achieve. Since some of the codes were originally written in Fortran we used the optimised *C* versions from [47]. It is important to clarify that the NAS-PB codes are *manually parallelised* but not *hand-tuned* for a specific target architecture. The designers employ parallelisation patterns that primarily target high-level coarse-grain loop-level parallelism but do not address architecture-specific optimisations like SIMD parallelisation and data placement. Nevertheless, OPENMP 's directive-based approach in addition to the availability of target-specific OPENMP compilers and runtime systems provide the guarantee for highly optimised sequential code-generation and minimum parallel-runtime overhead.

A brief description of the individual kernels and applications follows:

BT A simulated computational fluid dynamics application that uses an implicit algorithm to solve a system of 3-D Navier-Stokes equations [11]. The result of the finite-differences discretisation is a set of three decoupled systems of 5×5 Block-Tridiagonal.

CG A Conjugate Gradient kernel that is used to numerically approximate the largest eigenvalue of a sparse and symmetric positive definite matrix [12].

EP An Embarrassingly Parallel kernel [12]. It generates pairs of Gaussian random deviates and then counts the number of pairs that lie in successive square annuli. This problem is typical of Monte Carlo simulation applications. It provides an upper bound for achievable parallel performance since it features minimal interprocessor communication.

FT A *3D* partial differential equation solver using Fast Fourier Transform [12]. This kernel is the integral part of many *spectral* codes and it features communication-intensive scatter/gather operations.

IS A large Integer Sort kernel implemented using bucket sort. This is an essential component of many *particle method* applications.

[7] At the time these lines are written (July 2010) [12] and [11] have a total number of 1379 and 635 citations respectively based on Google Scholar

LU A simulated computational fluid dynamics application that factors a system of equations in a Lower and an Upper triangular matrix using the successive over-relaxation (SSOR) method. The input is a seven-block-diagonal system that results from the finite-difference discretisation of the Navier-Stokes equations in 3-D.

MG A simplified MultiGrid kernel[12]. It consists of iterations of the V-cycle multigrid algorithm that approximates the solution of a discrete Poisson problem.

SP A simulated computation fluid dynamics application which is similar to BT [11]. It uses a Beam-Warming factorisation which results in a system of Scalar Pentadiagonal bands of linear equations for each of the three dimensions.

3.4.2.2. SPEC CFP2000

In addition to the NAS-PB we used floating-point intensive benchmarks from the SPEC2000 benchmark suite [130]. SPEC2000 is one of the products from the Standard Performance Evaluation Corporation (SPEC) which are the *de-facto* industrial standard for performance evaluation of high-performance computing systems. Due to the lack of a Fortran frontend for the source-to-source toolchain we had restricted our studies to the subset of applications that are coded in *C*, namely *ammp*, *art* and *equake*.

The determining factor for using SPEC2000 instead of the latest SPEC2006 was the availability of manually parallelised versions of the same benchmarks, which are distributed as part of the SPECOMP-2001 suite [8]. The applications were parallelised by expert programmers using OPENMP. Unfortunately, besides the algorithmic modifications that enhance parallelisation, the programmers have also performed some sequential code enhancements (e.g. optimised an inefficient allocation policy in *equake*). As a result, the single-threaded execution of the SPECOMP-2000 codes is significantly faster (2 times faster on average) compared to the sequential code of SPEC2000. Nevertheless, these codes still provide a robust upper bound for the parallelisation potential of these applications.

A brief description of the applications follows:

ammp is computational chemistry application that performs molecular dynamics computations on a protein-inhibitor complex which is embedded in water.

art is an Adaptive Resonance Theory 2 (ART 2) neural network application. It is used to recognise objects in a thermal image. *art* first trains the neural network on the objects and then it is utilising them to identify the objects in a scanfield image.

equake simulates the propagation of seismic wave in large basins. It involves performing computation on an unstructured mesh that locally resolves wavelengths, using a finite element method.

3.4.3. Embedded and Multimedia Programs

This section is presenting a set of applications from the embedded and multimedia domain. We have considered applications from the EEMBC [139] and SPEC2000 benchmarks suites. We focus only on applications which feature rich TLP and thus are amenable to parallelisation that exploits the availability of multiple cores. Most of the benchmarks that are not included in the evaluation can still be parallelised by exploiting word-level parallelism or by performing algorithm-specific modifications. However, these techniques are beyond the scope of the methods proposed in this thesis.

A brief description of the applications follows:

bzip2 is an application from the SPEC2000 benchmark suite that implements a lossless data compression algorithm based on the BurrowsWheeler transform.

mp3player is part of the consumer applications of the EEMBC 2.0 benchmark suite and it implements a decoder for the de-facto standard for digital music compression, MPEG-1 Audio Layer 3.

mpeg2dec is drawn from EEMBC 2.0 benchmarks and implements the widely used international standard for video compression, MPEG-2.

cjpeg is also a consumer applications from EEMBC 2.0 that implements the JPEG image compression algorithm, the dominant standard in digital image photography and the WWW.

3.5. Evaluation Platform

The evaluation of the contributions of this thesis was performed on two machines which represent two vastly different parallel architectures.

The first machine, *M1* in table 3.8, is a shared memory multiprocessor with two *x86_64* quad-core processors and Uniform Access to Memory (UMA) [56]. *M1* is a typical example of a modern high-end general-purpose system, utilising aggressive hardware techniques (e.g. *Out-of-Order* execution, hardware prefetching and multiple instruction fetch) to extract the maximum of the available *Instruction-Level Parallelism* ILP. This guarantees that we present speedups comparing against the *strongest* sequential baseline.

The second machine, *M2*, in our evaluation is a heterogeneous multiprocessor architecture with two *Cell* processors and Non Uniform Access to Memory (NUMA) [120, 23]. Each Cell processor contains a general-purpose core (*Power™ Processing Element*, PPE) and 8 vector accelerators (*Synergistic Processing Element*, SPE). The PPE is a *In-Order* superscalar core with Simultaneous Multi-Threading (SMT) support which is primarily used to execute the Operating System (OS) and orchestrate the execution of threads in the SPEs. SPEs, on the other hand, are Single-Instruction-Multiple-Data (SIMD)

M1: Intel Xeon Server	
Hardware	Dual Socket, Intel Xeon X5450 @ 3.00GHz 2 Quad-cores, 8 cores in total with SSE2, SSE3 and SSE4.1 extensions 32KB I-cache, 32KB D-cache 6MB L2-cache shared/2 cores (12MB/chip) 16GB DDR2 SDRAM
O.S	64-bit Scientific Linux with kernel 2.6.9-55 x86_64

M2: Cell Blade Server	
Hardware	Dual Socket, QS20 Cell Blade 2 × 3.2 GHz IBM Cell processors 1 × PPE/chip: 32KB I-cache, 32KB D-cache, 512KB L2 cache with AVX extesions 8 × SPEs/chip: 128-bit SIMD, 256KB of Local Store 1GB XDRAM
O.S	Fedora Core 7 with Linux kernel 2.6.22 SMP

Table 3.8.: Hardware and software configuration details of the two evaluation platforms.

cores which are designed to accelerate applications with high word-level parallelism (e.g. media and streaming applications). In deep contrast to the homogeneous architecture of *M1* the Cell-based system requires not only compiling for multiple target Instruction Set Architectures (ISA) but also orchestrating the data transfers from/to the SPEs whose Local Storage (LS) memory is globally addressable but not cache-coherent. These rather controversial properties of the Cell BE can be viewed as both features that enable high peak performance but also as extremely challenging hurdles to achievable performance.

3.5.1. OpenMP Overheads

We used the EPCC microbenchmarks *v 2.0* [118] to empirically quantify the overhead of the specific OPENMP implementations in the two evaluation platforms. A summary of the most important metrics are shown in table 3.9. At this point we should clarify that the construct overhead does not include the overhead of the parallel execution itself. The latter overhead is an inherent property of each specific parallel code and architecture and includes, for example, the cost of parallelisation-induced *coherence* and *compulsory* cache-misses. The overheads for the Cell platform are reported using the time of sequential execution in a single PPE as the baseline. It is obvious that platform *M2*, although it features great peak performance, the overhead of data-level parallelism can be dominated by the overhead for thread creation and synchronisation. In fact the overhead of most OPENMP constructs in *M2* is almost two orders of magnitude higher than in *M1*. This observation is used as the primary motivation in chapter 5 for the integration of profile-driven parallelisation with a Machine-Learning based

	Overhead (in μsec)			
	M1		*M2*	
OpenMP construct	μ	$a = 5\%$	μ	$a = 5\%$
Directives				
`parallel`	2.14	$\pm$0.006	620.42	$\pm$0.661
`for`	1.37	$\pm$0.004	622.96	$\pm$0.533
`parallel for`	2.18	$\pm$0.008	622.56	$\pm$0.786
`barrier`	1.36	$\pm$0.006	364.40	$\pm$0.331
`reduction`	2.44	$\pm$0.008	632.00	$\pm$0.618
`single`	1.52	$\pm$0.025	492.20	$\pm$0.626
`critical`	0.44	$\pm$0.052	19.93	$\pm$0.003
`lock/unlock`	0.40	$\pm$0.041	2.54	$\pm$0.019
Privatisation (array of 53{Kb})				
`private`	1.98	$\pm$0.045	622.05	$\pm$0.962
`firstprivate`	41.11	$\pm$0.173	675.71	$\pm$1.321
`copyin`	44.09	$\pm$0.711	1487.00	$\pm$29.848
Scheduling				
`static`	1.69	$\pm$0.063	626.57	$\pm$1.589
`dynamic(1)`	33.36	$\pm$2.317	627.78	$\pm$0.879
`dynamic(16)`	3.02	$\pm$0.011	627.98	$\pm$1.117
`dynamic(64)`	2.30	$\pm$0.019	626.43	$\pm$1.495
`guided(1)`	46.19	$\pm$2.315	626.25	$\pm$1.071
`guided(8)`	27.96	$\pm$2.560	626.29	$\pm$1.048
`guided(16)`	22.40	$\pm$0.079	–	–

Table 3.9.: Overhead of the most important OpenMP constructs. The measurements were taken using the Epcc microbenchmarks *v 2.0* on platforms *M1* and *M2* respectively. The measurements represent mean values (μ) of twenty repetitions using the maximum number of available cores and lie in the relevant confidence intervals with a confidence level of $1-a = 95\%$.

profitability analysis that enables accurate and automatic mapping of parallelism across vastly different architectures.

Chapter 4.

Intermediate Representation Profiling

The primary objective of this thesis is to enhance the static analysis of a traditional parallelising compiler using dynamic dependence information, extracted by means of profiling. The main obstacle in this procedure is correlating the low-level information gathered during program execution to the high-level data and control flow representation of the compiler. In this chapter we present a novel dependence profiling technique that overcomes this issue. Furthermore, we show how this technology can be utilised for the extraction of more complex program properties using *reduction operations* as an illustrative example.

The structure of this chapter is as follows. First, we motivate our work in section 4.1. The concept of IR-profiling and its implementation in the CoSy compiler framework follow in section 4.2. In section 4.3 we describe the details of profile-driven dependence analysis and the construction of the whole-program representation. Section 4.4 presents a brief discussion of possible optimisations that can improve the performance of the current prototype implementation. Finally, we conclude in section 4.5.

4.1. Motivation

4.1.1. Profile-driven Dependence Analysis

Before discussing in more detail the shortcomings of existing dependence profiling methodologies that motivated the introduction of IR-profiling, we present a simple but illustrative example that demonstrates how profiling information can effectively disambiguate statically undecidable data references. In figure 4.1 loop *L1* performs a simple update operation on array A. The index functions of A in *expr1* and *expr2*, however, are not known at compile time since they are indirect and reference elements of arrays R and L respectively. Therefore, static analysis has to pessimistically assume that different iterations of *L1* may access the same elements of A and disallow parallelisation. Profile-driven dependence analysis, on the other hand, inspects the exact data accesses that *expr1* and *expr2* generate when the instrumented code is executed using representative inputs. Therefore, it is able to monitor exactly which range is defined or used by each iteration and thus deduce that there is no true dependence that limits parallelisation for a specific input.

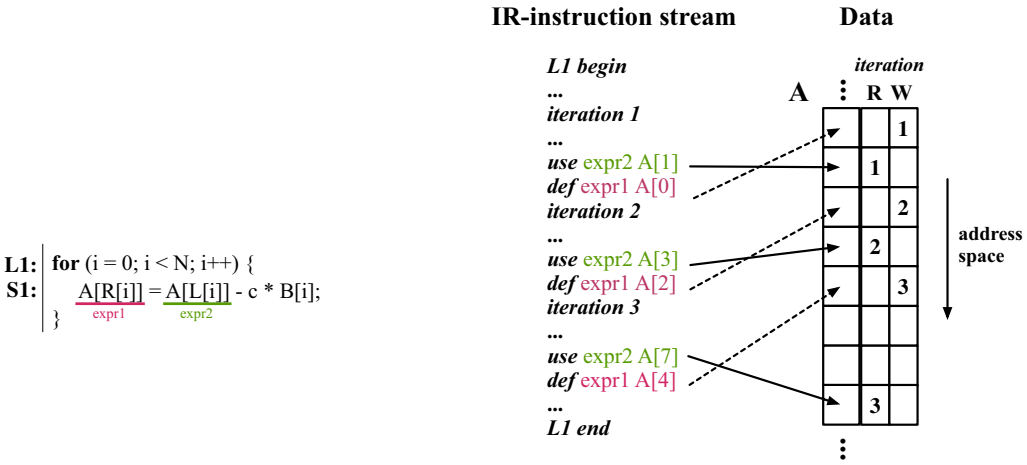


Figure 4.1.: A typical example that contrasts the limitations of static analysis with the strength of profile-driven dependence analysis. The source code on the left poses an insurmountable challenge to static analysis, since the elements of array *A* accessed by expressions *expr1* and *expr2* are statically undecidable. On the right we present a simplified stream of IR-instructions that captures the actual memory accesses of each iteration of loop *L1*. Using this information profile-driven analysis can deduce that for this specific stream there is no flow-dependence (i.e., *use* after *def* dependence) among the iterations of *L1*, and thus parallelise it.

Profile-driven parallelisation despite its strength in disambiguating program behaviour it can not be conclusive, since it is not able to guarantee correctness for a new input dataset. Nevertheless, as we will show in the chapters that follow the majority of scientific and multimedia applications exhibit a stable dependence pattern, thus enabling us to generate correct parallel code and only rely on the user for the final verification. For instance, in this example a user can provide the application-specific knowledge that arrays R and L by virtue of the algorithm or the input data store each index only once, therefore parallelisation is safe for any valid input. Additionally, as we demonstrate in the hierarchical pipeline extraction methodology, which is presented in chapter 6, IR-profiling has an even broader scope. It bears the potential to capture the whole-program data and control-flow, largely removing the barriers that widely-used *C* code features (e.g. powerful pointer arithmetic) impose in static analysis. This way parallelisation is enabled in levels and granularities not reachable before by conventional static techniques. Most, importantly as we will shall see profile-driven approaches are applicable to real-life full implementations, and thus provide a more pragmatic approach to parallelisation of sequential applications.

4.1.2. IR-profiling

Program instrumentation has been long used in research and industry to gain useful insight into the execution behaviour of programs. For instance, *path profiling* [13] determines how many times an acyclic path of the control-flow is executed during a sample run of the program. This information is then utilised by compilers for profile-guided op-

timisation to perform control-flow optimisation in order to improve branch-prediction accuracy. The majority of these tools operates on a low-level representation of the program, most often the final native binary [1]. Instrumentation is then either performed off-line using a binary rewriting tool (e.g. ATOM [131] and DIABLO [152]), or at the execution time using a more powerful technique called dynamic binary instrumentation (e.g. DynamoRIO [136], PIN [82] and Valgrind [97]).

Despite the applicability of these approaches to a variety of problems, ranging from program optimisation to software testing (e.g. heap-memory allocation bugs) and architectural simulation (e.g. fast cache simulation), the information extracted with binary instrumentation is too low-level to be effectively utilised in profile-driven parallelisation. This fundamental problem is analogous to the one often faced by compiler-engineers that have to select in which stage during the IR-lowering should they implement a specific optimisation or transformation. Higher-level representations offer more flexibility but lack the machine-specific specialisation that provides precise modelling of code properties like code-size, register pressure or instruction latency. Therefore, data-flow tools based on binary-level instrumentation (e.g. Redux [96], [121] and [141]) mostly provide information that assists the programmer in program understanding rather than provide the means to drive automatic program transformation. To alleviate this limitation data-flow approaches typically employ *debugging symbol* information, a feature available in most executable file formats (e.g. ELF [143]). Nevertheless, debugging symbols are fundamentally flawed. Even state-of-the-art recovery techniques [137] that reconstruct the program structure can track binary instructions back to the source code with a precision of one line – in the best case. In addition, binary representations are inevitable – even without any optimisation – further obfuscated as a result of low-level features of the target's Instruction Set Architecture (ISA) and Application Binary Interface (ABI) (e.g. calling conventions, alternative addressing modes, register spills, register dependences, etc.).

This thesis aims at automated coarse-grain parallelism extraction that inevitably involves performing high-level data and code transformations such as heap-data privatisation and loop splitting. Considering the limitations of existing binary-level approaches, we believe that such a challenging task requires a more radical and complete solution to these issues. To bridge this *information gap* we perform instrumentation at the IR level of the compiler and more specifically the middle-level IR, called CCMIR (presented in 3.1.2). Effectively our instrumentation creates an *executable*-IR which, when executed generates a trace of IR-instructions. Therefore, further analyses that process this trace can utilise this correspondence to associate the derived information back to the relevant IR data structures of the compiler.

[1] Obviously operating on a binary representation has the extra benefit of being applicable also in programs of which no source is available. However, it should be obvious from the failure of current compiler technology to address the problem of automatic parallelisation under far looser constraints that relying on the availability of the source is a reasonable assumption.

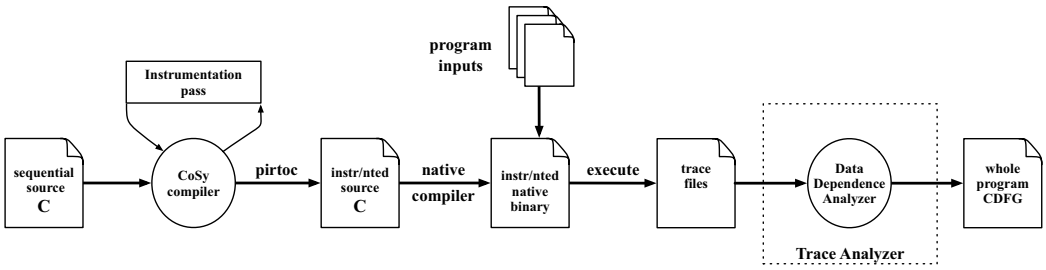


Figure 4.2.: Overview of the Ir-profiling workflow. First, instrumentation calls are inserted in the Ccmir of CoSy. Next, instrumented *C* code is generated utilising *pirtoc* – the Ccmir $\rightarrow C$ pass – which can be compiled with the native compiler of the development platform. Finally, using one or multiple inputs we can collect Ir-instruction traces that are fed into the Trace Analyser. The latter is essentially a generic driver for profile-driven program analysis, such as data dependence analysis.

4.2. Instrumentation Framework

Figure 4.2 gives an overview of the Ir-profiling approach. In a first step, instrumentation calls to the profiling library, which handles the printing of the relevant information, are inserted in the Ccmir of CoSy. Next, we utilise the Ccmir $\rightarrow C$ pass (*pirtoc*) available in CoSy to generate standard *C* code from the instrumented Ir. This instrumented source code can then be compiled with any native *C* compiler to produce an instrumented binary executable. The instrumented program is functionally equivalent to the original code. In addition, when executed using a sample input it will produce – as auxiliary output – a sequence of Ir-instructions that can be either saved in a trace file or processed on-line by a trace analyser[2]. A trace analyser is a driver-program that takes Ir-instructions as input and after resolving their type calls the relevant *callback* routine of the specific analysis.

4.2.1. IR Instrumentation

Our goal is to profile both the control and data-flow of the program. To achieve this we implemented an instrumentation pass in CoSy that operates on Ccmir. It injects instrumentation calls for every Ir construct that either generates a data access observable in Ccmir (definition or use of an Ir object) or corresponds to a high-level control-flow event (function, basic-block, loop entry or exit, etc.). More specifically, for each memory access we emit an Ir-instruction that has the following fields: (i) type of access (*use*/*def*), (ii) the *unique id* of the corresponding Ccmir node, (iii) memory address of the object referenced, and (iv) point-to address (if pointer dereference). For each function, Bb, loop or iteration entry or exit we simply emit an instruction with the (i) type of control-flow event, and (ii) the *unique id* of the corresponding Ccmir node.

[2]For the sake of simplicity in the rest of this thesis we assume an off-line trace analysis.

Instruction	1st arg	2nd arg	3rd arg
Data-flow IR-instructions			
`def`	<*id*>	<*address*>	
`use`	<*id*>	<*address*>	[<*points-to*>]
Control-flow IR-instructions			
`bb`	<*id*>		
`func`	<*id*>		
`loop`	<*id*>		
`iter`	–		
`endfunc`	–		
`endloop`	–		
`enditer`	–		
Allocation pseudo-instructions			
`global`	<*id*>	<*address*>	
`local`	<*id*>	<*address*>	
`alloc`	<*base address*>	<*size*>	
`free`	<*base address*>		
Summarising instructions			
`read`	<*base address*>	<*size*>	
`write`	<*base address*>	<*size*>	

Table 4.1.: Format of IR-instructions.

Finally, we insert pseudo-instructions to reveal the address of variables with static or automatic *storage duration*[3]. Heap allocated objects require explicit instrumentation of the dynamic allocation routines. This can be performed either by the user using a simple set of macros that generate the appropriate IR-instructions or in the common case where the standard library is used fully automatically using the *Memory Allocation Hooks* available in all recent implementations of the GNU *libc* [37]. Pseudo-instructions are necessary to allow precise memory disambiguation in the profile-driven dependence analysis phase.

A similar technique can be used for system or library calls where source code is either not available or not conveniently accessible. Although, our tools are not going to be able to extract parallelism from the routines, they can still be processed as a *black box* that only its external effects are visible to the IR-profile. For this purpose we provide two additional summarising IR-instructions, `read` and `write`, which are equivalent to performing multiple `def` or `use` operations on a continuous range of memory. In addition, we can enforce sequential execution of these routines by using a simple `write`→`read` sequence on a specific reserved address. Effectively this results in a flow-dependence that connects all the invocations of one or more relevant external calls. In fact, we utilise this mechanism to provide simple and efficient instrumentation for all the standard library I/O functions that have OS visible effects and thus should not be reordered. On the contrary, library calls like *malloc* that are commutative[4] thus can

[3] According to the ISO *C* standard specification [57] variables with file, internal or external linkage and variables declared using the keyword `static` have *static* storage duration. Variables declared in procedure scope without the keyword `static` and function formal parameters are assigned *automatic* storage duration.

[4] We are using the term *commutative function* in its more flexible sense. More specifically, commutative functions are sections of code that can be executed in any order without affecting the outcome of the

be freely reordered should not enforce such a dependence to provide greater scope for parallelisation.

Table 4.1 provides a full list of the Ir-instructions and the respective fields. Since Ccmir is fully-typed the type (including its precise size and alignment) of each Ir-instruction can be derived from the corresponding Ccmir node using its *unique id*. Additionally, this choice has the advantage of saving space. The data representation for the Ir-instructions can be either in text (using an Xml schema) or binary format. We have implemented instrumentation libraries and trace analysers for both, but we have primarily used the binary one that produces both more compact traces and is faster in processing. The text representation, however, can be very helpful during the development of the prototypes.

The instrumentation pass presumes that its input Ccmir has been pre-processed by two additional passes that have to be executed in the following strict order:

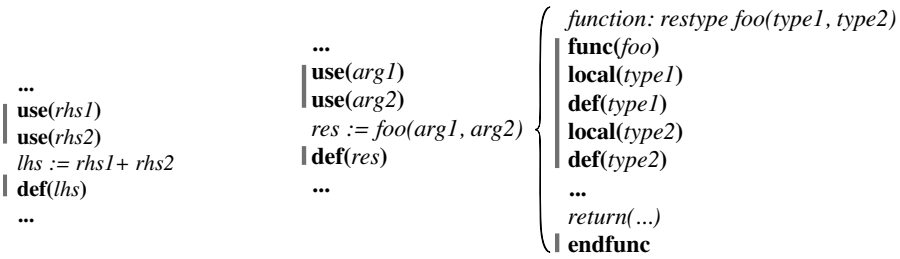
1. *loopdesign:* As we have already discussed in section 3.1.3, Ccmir does not include a high-level loop representation. Instead, this is reconstructed from the *loopdesign* pass based on Ccmir control-flow statements like `mirIf` and `mirGoto`. Subsequent passes can then access this information using the Loop-Markers interface. We have to execute this pass first since statements are moved and new Bb are created to comply with the structure defined in the specification of the Loop-Markers (either *while-do* or *repeat-until*). In addition *loopdesign* generates information about variables with special loop functionality (e.g. induction variables).

2. *splitfunc:* This is a transformation pass that modifies the Cfg so as to ensure that each function call (`mirFuncCall` or `mirCall`) is the only Ccmir statement in a Bb, besides the necessary control statement that is always last. This invariant greatly simplifies the construction of the Cdfg in section 4.3 and removes the need for an Ir-instruction that identifies the calling point of a function.

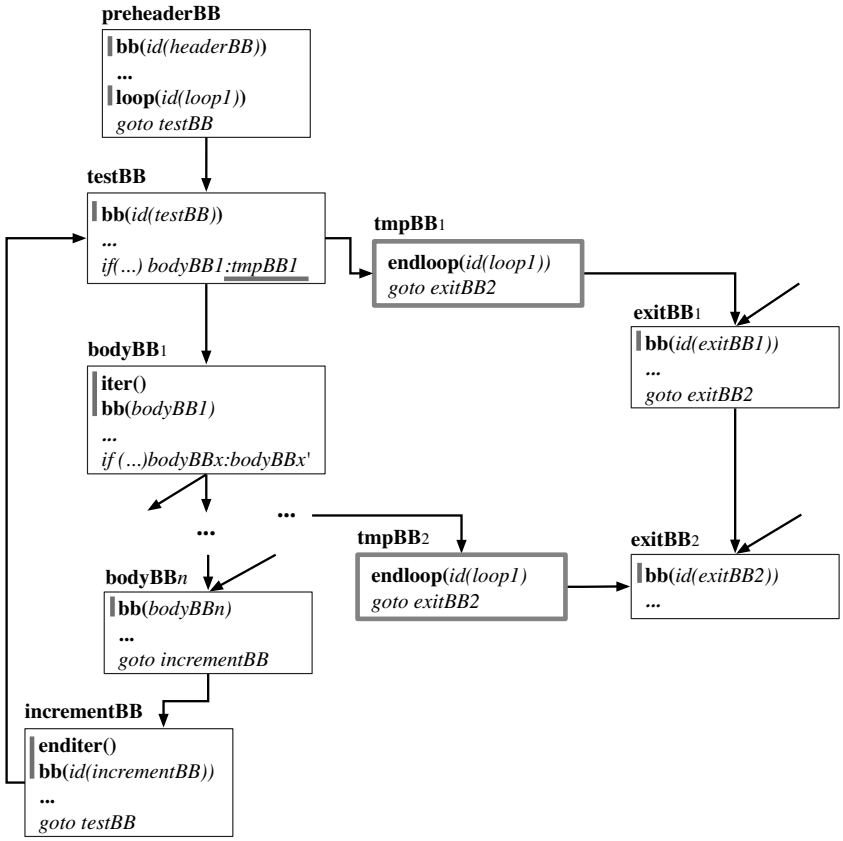
3. *persistentid:* This is a simple pass that we implemented to provide a persistent *unique id* for every Ccmir structure across multiple compiler invocations. This is achieved by using a deterministic pre-order numbering traversal of the Ccmir tree. The property of persistence cannot be guaranteed if the sequence of transformations that are executed before this pass is modified.

It should be clear by now that Ir-profiling does not have to model complex low-level data-flow, such as parameter passing using registers or address calculations, thus it can be relatively simple, compact and efficient. Ir-instructions that correspond to expressions are *side-effect free* and therefore can be inserted in any order as long as they execute before the main side-effect of the statement (if there is any). The Ccmir

application [4]. This is in contrast with more strict definitions that require commutative functions to produce identical memory layout. The latter has been shown to be overly conservative and restrictive for parallelisation [4].



(a) `mirAssign` **(b)** `mirFuncCall`



(c) *while-do* Loop-marker

Figure 4.3.: This figure shows the precise location where IR-instructions have to inserted in order to correctly capture the sequential semantics of the instrumented code. Modifications are highlighted with a thick grey box or line.

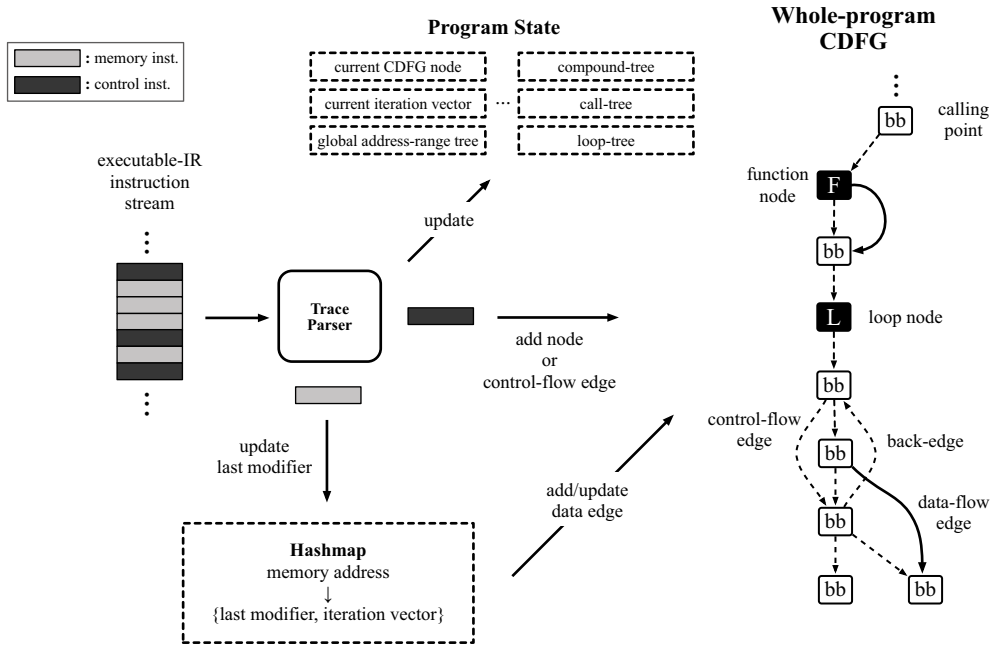


Figure 4.4.: Overview of the Dependence Analyser. The stream of IR-instructions is parsed and processed by algorithm 4.1 that reconstructs the whole-program CDFG depicted on the right. In addition, while executing the Trace Analyser maintains two important data-structures (i) an abstract *Program State*, and (ii) a *Hashmap* that is used to determine the source of data-flow dependences.

statements that we consider have a single side-effect each. To capture the sequential semantics of the original program in the IR-instruction trace statement side-effects have to appear in-order. In addition, control-flow IR-instructions have to be inserted appropriately so that the entry/exit instructions enclose only instructions that execute within the context of the respective structure.

Figure 4.3 illustrates how these properties can be enforced for each one of the data and control-flow instructions under different scenarios. As it is clear from figure 4.3(b) functions calls are instrumented upon entering the function. This way we handle direct and indirect function calls uniformly. Function calls with no return value (i.e., `mirCall`) are handled in a similar fashion but there is no need for a `def` instruction after the call. In addition, formal parameters are instrumented with an additional `def` instruction to make the assignment of the value of actual parameters explicit. Instrumentation of Loop-markers is slightly more complicated. Figure 4.3(c) shows the instrumentation of a *while-do* loop with multiple exits. The `loop` instruction is inserted at the end of the *pre-header* BB, therefore the initialisation of the loop variables is not accounted to be part of the loop. If there is a single *loop-exit* BB and it has no predecessors then the `endloop` instruction can be simply inserted in the beginning of the loop-exit BB. Otherwise, to guarantee that `endloop` executes only once and only after the respective has been executed we have to perform a simple control-flow transformation. We create

an additional Bb for each loop-exit Bb. The only Ccmir statements in these new blocks is the `endloop` followed by a `mirGoto` to the relevant loop-exit Bb. In figure 4.3(c) these are $tmpBB_1$ and $tmpBB_2$ for $exitBB_1$ and $exitBB_2$ respectively. Observe that the additional Bbs are not instrumented themselves and in a later invocation of the compiler (e.g. for performing parallel code-generation) they can be safely ignored. Finally, the `enditer` instruction is inserted at the entry of the *increment* Bb, thus updates of the induction variables are not consider to be a part of any iteration. Also note that there is no `enditer` at the exits of the loop since it can be safely implied based on the `loopend`.

4.3. Profile-Driven Dependence Analyser

When executed the instrumented binary produces a trace of Ir-instructions which is consumed by the trace analyser, figure 4.4. In this section we present the Dependence Analyser that builds on top of the functionality of the trace analyser to extract a whole-program Control and Data-Flow Dependence Graph (Cdfg). Although a large number of Ir-instructions needs to be processed the whole procedure can be simple and fast as no low-level machine details or functionality of the instrumented program needs to be considered.

The remainder of this section is organised as follows. In subsections 4.3.1 and 4.3.2 we describe the construction of the control and data-flow subgraphs of the Cdfg. Data allocation and memory disambiguation issues are discussed in subsections 4.3.3 and 4.3.4 respectively. Subsection 4.3.5 presents an extension to our framework that enables the detection of complex reduction operations. Finally, in subsection 4.3.6 we discuss the limitations of the proposed framework.

4.3.1. Control-flow

Each Ir-instruction is processed using the procedure described in algorithm 4.1. The control-flow handler (lines $4-13$) reconstructs a whole-program Control and Data Flow Graph (Cdfg) of the application. The Cdfg is a hierarchical graph that contains three types of nodes, simple Bb nodes and compound Function (F) or Loop (L) nodes (figure 4.4). F nodes explicitly represent a call of a specific function at a given calling point (a `mirFuncCall` Ccmir node). Therefore, the same function body might appear in the Cdfg multiple times. Although this might lead to a graph of size proportional to the number of distinct calling contexts of each function, in practice for the C programs that we have instrumented in this thesis it does not appear to be cause for serious concern. On the other hand, this feature of the Cdfg allows for precise and context-sensitive dependence analysis. Each F node contains all the Bbs that are executed in its context and do not belong in a nested function or loop. Similarly, L nodes contain the Bbs of the loop-header and body of the loop. Any nested F or L nodes are effectively

Algorithm 4.1: Algorithm for CDFG construction.

Data
- CDFG(V, E_C, E_D): graph with control (E_C) and data-flow (E_D) edges
- $loop\_carried_e[]$: bitset $\forall e \in E_D$: $loop\_carried_e[i] = 1$ if e loop-carried in loop-level i
- $set_e[]$: address-range array $\forall e \in E_D$, indexed by the loop-carried level i
- $it_a[]$: iteration vector of address a
- $M[A, \{V, it_a\}]$: hash table: memory addr. $a \rightarrow \{V, it_a\}$
- $I(k)$: extract field k from instruction I
- GD: global memory address-range tree
- D_f: memory address-range tree of function f
- $it_0[]$: current normalised iteration vector
- $u \in V$: current node
- $f \in V$: current function
- $l \in V$: current loop
- $c \in V$: current component

```
1  Procedure IR_instruction_handler
2  while trace not finished do
3      I ← next instruction;
4      if I is a control instruction then
5          if I(id) ∉ c then
6              create node v for I(id) in c;
7          if edge (u, v) ∉ E_C then
8              add (u, v) in CDFG ;
9          switch I do
10             case bb   u ← v;
11             case func   f ← v;
12             case loop   l ← v;
13             case iter   it_0[depth(l)] ← it_0[depth(l)] + 1;
14     else if I is a memory instruction then
15         a ← I(addr);
16         if I is a def then
17             update last writer of a in M ;
18         else if use then
19             w ← find last-writer of a from M;
20             if u → w edge e ∉ CDFG then
21                 add e in E_D;
22             foreach i : it_a[i] ≠ it_0[i] do
23                 loop_carried_e[i] ← true;
24                 set_e[i] ← set_e[i] ∪ {a};
25             it_a ← it_0;
26     else if I is an allocation instruction then
27         a ← I(addr);
28         if I is local then
29             add {I(id), [a, a + I(size)]} in D_f;
30         else if i is global ∨ alloc then
31             add {I(id), [a, a + I(size)]} in GD_f;
```

pointers to the relevant compound nodes. In addition to the CDFG the control-flow handler dynamically builds three overlay tree structures, one for the function call-tree, one for the loop-hierarchy and one that is effectively the tree of compound (F or L) nodes. The root of the call and compound-tree is function *main*. For the loop-tree we assume a virtual top-level loop which is the parent of all first-level loops of the program. The function call-tree is also utilised as a call-stack using an additional pointer for the current context. Nodes or edges are added to CDFG only when there is a control-flow instruction that references a BB or L node that does not exist yet in the current function or an F node that has not been called before from the exact same point (i.e., the BB containing the `mirFuncCall` or `mirCall`) (lines $5-8$). Finally, utilising the `iter` IR-instructions the analyser maintains a normalised loop-iteration vector at any point of the execution (line 13).

4.3.2. Data-flow

Data-flow IR-instructions are processed in lines $14-25$ of algorithm 4.1. The main data structure utilised is a hash-table M that maps a memory address to a tuple containing (i) the *last-modifier* of this address, and (ii) the normalised iteration vector at that point of the execution. In the current prototype the memory granularity is at a byte-level. Although, this might prevent the identification of parallelism in the presence of bitfield operations, it would primarily concern very fine-grain parallelism. In addition experimental results show that this granularity is sufficient to extract coarse-grain parallelism without sacrificing the performance of the dependence analyser.

Data-dependence information is registered in the form of *data-flow edges* which are inserted as an overlay in the CFG. False-dependences, i.e., output and anti-dependences, are not explicitly represented in the graph, since our primary goal is to record the data-dependences that prevent parallelisation. Instead, we compute them on demand only when parallelisation is possible and based on the form of parallelism (loop-level or pipeline) that is extracted in each case. Each data-flow edge is annotated with (i) the address regions that is communicated, (ii) a bit-vector that records at which levels of the loop-nest this particular edge is loop-carried, and (iii) a vector that records the *maximum* normalised dependence-distance in the loop-levels that the distance is loop-carried. When a `def` (or similarly a summarising `read`) IR-instruction is processed we add or update the relevant entries in the hash-map M (lines $16-17$). In the case of a `use` (or similarly a summarising `write`) IR-instruction, add or appropriate modify the data-flow edge which has as its source the last-modifier and as its target the current BB (lines $18-25$). This mechanism also handles *self-edges*, with the distinction that these are only recorded if they are loop-carried.

As soon as the complete trace has been processed the constructed CDFG with all its associated annotations can be imported back into the CoSy compiler where it is utilised along with the statically derived data and control-flow for parallelism extraction and

parallel-code generation. This is only possible because the dynamic profile is based on the persistent unique ids that were assigned to each IR node at the *persistenid* pass.

4.3.3. Data Allocation

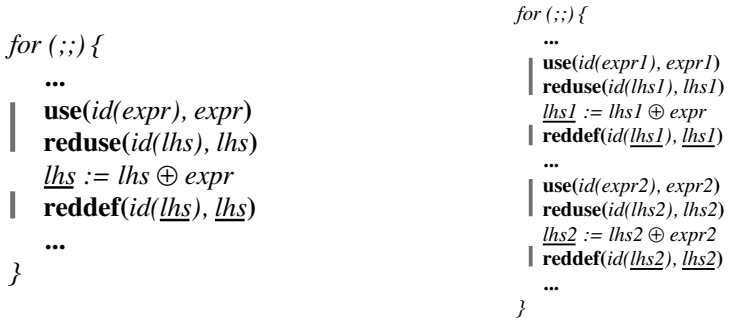
As we have already presented in section 4.2.1 the base address of data with static, automatic or heap storage duration are made explicit using the pseudo IR-instructions `global`, `local` and `alloc` respectively. By convention `global` instructions are executed right after entering the function *main*. The dependence analyser stores the range of addresses that correspond to each global symbol using a tree of non-overlapping address-ranges that is not modified during dependence analysis.

Automatic variables, on the other hand, are allocated upon entering a function, using a similar structure that is associated with each F node. As long as there are no *setjmp* or *longjmp* calls it is safe to assume that the addresses of automatic variables will be constant among the multiple dynamic instances of the same `local` IR-instruction. Handling calls to *alloca* is more complicated though[5]. Typically, most implementations of the *C* library implement this with an inlined compiler-defined function (i.e. intrinsic) that allocates memory on the stack. Since calls to *alloca* take a variable argument this can lead to automatic variables having different dynamic addresses on the stack. We overcome this problem by substituting calls to *alloca* with a pair of calls to *malloc* and *free* with the latter executing upon exit from the current function. This is actually a rare scenario but still we can handle it transparently.

Finally, `alloc` IR-instructions are inserted in the same address-range tree which is used for global data. The only difference is that instead of the global symbol these objects are associated with the calling point of the allocation call. For the sake of simplicity we assume that distinct calls to *malloc* return non-overlapping addresses regions. We can achieve this using either a trivial implementation of *free* that never deallocates memory or a custom memory allocator that actually deallocates memory but utilises OS features (e.g. memory mapped I/O) to allocate heap-memory mapped to distinct address-ranges. Otherwise we should encode the "version" of dynamically allocated memory using an additional field in the memory address object that we are using in the dependence analyser. Although at first this seems like a significant limitation it practically never is because most modern 64-bit machines support address-ranges far greater than what a typical program will allocate.

At this point we should clarify that any modifications in the allocation mechanisms that we described in this section are only necessary in the instrumented program. The parallelised versions are linked with the default standard libraries, so no performance overhead or change in the original functionality will occur due to the instrumentation mechanism.

[5] *alloca* is widely supported in many compilation platforms, however, it is non-standard.



(a) single statement reduction

(b) coupled reduction statements

Figure 4.5.: Examples for Ir-instrumentation in parallel reduction operations.

4.3.4. Memory Disambiguation

It is clear from the description of the profile-driven data-flow analysis that although Ir-profiling facilitates precise annotation of the compiler Ir with dynamically extracted information, it still uses memory address-ranges to handle pointer accesses and other low-level features of *C*. However, automatic parallel-code generation as well as further analyses that we develop in the following chapters necessitate precise memory disambiguation. In this section we describe how we translate memory addresses back to specific source-level symbols of Ccmir and the specific context that they were allocated.

Data with static storage duration are constantly allocated to the same address space (BSS) and thus are trivial to disambiguate based on the information saved in the global address-range tree. Automatic (stack-allocated) variables can be disambiguated using the context of the corresponding reference. More specifically, the symbol table available at each F node in the Cdfg maps the stack address of each automatic variable to the relevant local variable or formal argument. As long as there are no context switches (`setjmp`/`longjmp`) and the *alloca* calls are patched as we specified in the previous section, it is easy to observe that we can derive an 1-1 mapping for all automatic variables moving upwards in the call-tree. Note that when an stack address is referenced on a data-flow edge between two Bbs that belong to different contexts it will definitely be referring to data allocated in the preceding contexts (i.e., deeper in the call-stack). Therefore it is sufficient to look in the contexts preceding the Bb that executed first. Finally, in the case of heap-allocated data it is sufficient to perform a look-up in the global address-range tree to retrieve the precise point and context of its allocation.

4.3.5. Reduction Detection

As we have already seen the profile-driven approach is able to detect true dependences that prohibit parallelisation. However, in many cases flow dependences are an artefact of the sequential programming model rather than a consequence of the sequential se-

mantics of the program. For instance, loop counters and *induction* variables in general result in flow data-dependences but are straightforward to remove them if the induction formula is detected. Another more interesting example which is frequent in both multimedia and scientific applications is that of *reduction* operations (e.g. $A[i] = A[i] + k$). Static analysis is effective enough to address many cases of scalar induction and reduction variables. In the rest of our analysis we are not focusing on these variables which are already identified by means of static analysis. In the presence of pointers or indirect array accesses (e.g. sparse matrices), however, static analysis makes conservative assumptions and thus limits the amount of exploitable parallelism. We enhance the existing static analysis and propose a *hybrid approach* for these complex cases. In a first step we use a simple static analysis pass in our compiler to select reduction statement *candidates* (excluding any statically detected induction and reduction variables). Then, we instrument the relevant statements using two additional IR-instructions (`reddef` and `reduse`) to explicitly denote the operands that are part of a reduction. Finally, the dependence analyser determines which candidates are *valid* reductions and feeds this information back to the compiler to decide about their exploitation. A reduction is valid if and only if all the following statements hold [6]:

1. The reduction operator is both commutative and associative,
2. There is a true self-dependence that denotes the accumulation to the partial result of the reduction,
3. Only the final result of the reduction is used later, i.e., there is no outgoing dependence from the reduction, and
4. No true dependence is fed into the reduction.

The first two properties are determined by means of static analysis. The other two are verified dynamically in the dependence analyser as follows. Let a reduction statement be $lhs := lhs \oplus expr$, where $\oplus$ is a commutative and associative operator. Then we keep record of:

1. the set DEF_{red} of addresses that a reduction candidate is using as partial results (i.e., the addresses it writes to or equivalently the addresses referenced by lhs), and
2. the set $DEF_{other} \cup USE_{other}$ of addresses that are written or read respectively by statements and expressions, including $expr$, but do not belong to the set of *any* of the reduction candidates of the enclosing loop.

At the end of the loop execution[6] we are performing the following test to decide whether a reduction is valid:

$$DEF^i_{red} \cap \left(DEF_{other} \cup USE_{other} \cup \left(\bigcup_{k \neq i} DEF^k_{red} \right) \right) == \varnothing$$

[6]Intermediate checks can be also performed at the end of each or every few iterations to avoid any unnecessary analysis for statements that have been asserted as invalid reductions.

This test effectively determines whether *lhs* is aliased with *expr* or referenced/defined anywhere else within the surrounding loop.

This methodology can be also extended to detect *coupled* reductions, i.e., reductions that span more than one statement, figure 4.5(b). For a set of statements to be a valid reduction there is an additional requirement that the same reduction operator $\oplus$ is used in all the participating statements. Let R be the set of all reduction statements in a given loop and S a set of coupled reduction candidates. Then the parallel reduction test will be:

$$\Big(\bigcup_{i \in S} DEF^{i}_{red}\Big) \cap \Big(DEF_{other} \cup USE_{other} \cup \Big(\bigcup_{k \in R-S} DEF^{k}_{red}\Big)\Big) == \varnothing$$

To perform these tests we have extended the instrumentation framework with two additional IR-instructions: (i) `reddef` that indicates a definition that is also the left-hand-side of a reduction candidate (e.g. $\underline{lhs} := lhs \oplus expr$), and (ii) `reduse` that indicates a use that is also the only use of the left-hand-side expression in the right-hand-side of a reduction candidate. These two instructions are handled by the dependence analyser like regular `def` and `use` in algorithm 4.1 but in addition we utilise them to compute DEF^{i}_{red} and remove the *use* of *lhs* from USE_{other}. Figures 4.5(a) and 4.5(b) show how should the instrumentation calls be inserted in the case of a simple and a coupled reduction respectively.

4.3.6. Limitations

For all its power and useful properties, IR-instrumentation and profile-driven dependence analysis still has limitations. First, as we have already discussed in 4.2.1, IR-instrumentation assumes access not only to the *C* source code of the whole application but also to the source or alternatively appropriate annotated wrappers (i.e. `read`/`write` IR-instructions) of all the external library routines called. Given the potential of the proposed techniques as well as the complexity of the parallelisation task itself, we consider this assumption to be reasonable for the majority of the usage scenarios. For the purpose of the studies of chapters 5 and 6 we annotated a small subset of the standard *C* library, a task that proved to be trivial. In practice, a complete software solution can provide such annotated wrappers for widely-used libraries with minimal development effort.

Second, back-annotation of profile information to the IR and subsequently to the source code requires performing instrumentation at a high-level IR and before applying any IR lowering or optimisations. In fact, common optimisations like redundancy elimination pose the danger of making the association of IR-instructions to source code extremely complex and inaccurate. Besides the implications to the performance of the instrumented code itself (a point discussed in more detail in subsection 4.4.3) this restriction does not influence the extraction of coarse-grain parallelism. Although the

sequential execution time of each of the coarse-grain regions that form a parallel region might vary significantly depending on the applied optimisations, we consider the extraction of TLP to be largely orthogonal to any optimisations that target single-threaded performance. In addition, timing information that is necessary for task partitioning in chapter 6 is extracted using minimal instrumentation (i.e., with data and control-flow instrumentation disabled) and with all the standard optimisations of the native compiler enabled. Thus, the extracted information for the relative weight of each of the scheduled components is accurate. On the other hand, this assumption definitely limits the potential of utilising IR-profiling as a more general profiling technique, since it cannot capture lower-level IR or target-specific code features. However, we consider this to be out of the scope of this thesis. We prefer presenting IR-instrumentation as a solution tailored to the intricate problem of data-flow analysis for program parallelisation.

It is important to clarify that the choice to instrument unoptimised code does not fundamentally preclude optimisations of the instrumented code or of the code after performing the parallelisation itself. Furthermore, the majority of the acyclic code optimisations that improve single-threaded performance are not affected by the parallelising transformations which we present in the subsequent chapters. The only case that we observed such a side-effect is discussed in detail in section 6.2.3.3 where we also present a mechanism that eliminates it. In the case of loop-optimisations, however, parallelisation transformations can be more intrusive and inhibit optimisation like loop unrolling and loop-invariant code motion. Nevertheless, the techniques of this thesis mainly parallelise outer-most loops rather than inner-most loops which are the primary target of such loop optimisations.

4.4. Possible Optimisations

Our current prototype provide the means to instrument and analyse many different embedded and scientific applications of size ranging from small kernels of a few lines up to complicated applications and libraries of 30K lines of code. Although the performance of our toolchain has been adequate so far for the needs of empirical evaluation, there is still room for many optimisations and enhancements that can greatly improve the usability of our tools. In this section we present some of them but since the purpose of this thesis is to exploit this information rather than optimise its extraction, we leave their elaborate study and implementation for future work.

4.4.1. Multi-threading

Multi-cores systems are ubiquitous and the systems used by application developers are no exception. However, the execution of the instrumented binary and also the current implementation of the dependence analyser are sequential. Even if the execution of the instrumented binary is not parallelised itself using advanced slicing techniques like Su-

perPin [156] or Shadow Profiling [94] we can still exploit multiple cores by executing the analyser in parallel to the instrumentation using a pipelining scheme. More specifically, the instrumentation library can be modified to insert the IR-instructions in a SCOs *pipe* or even better in a shared-memory queue[7]. Then the analyser can dequeue and process IR-instruction from this shared queue instead of the trace. Although the analysis typically requires more time than the instrumentation, this scheme can practically overlap the two and completely hide the instrumentation overhead. The buffer itself can be limited down to a few hundred instruction and still this is only to minimise the synchronisation overhead on the shared queue. This is possible because the dependence analyser itself is designed to execute at a single pass over the IR-instruction stream and, thus, there is no need to store the entire stream. In fact, this method has the additional benefit that it eliminates the overhead of formatting (although the binary version has negligible formatting overhead) and performing I/O to store the IR-instructions.

A more ambitious project would aim at parallelising the analysis step itself exploiting the data and pipeline parallelism that it naturally exhibits. For instance `use` instructions as long as there is no intervening `def` instruction can be processed in parallel and commit the changes in the CDFG sequentially in a subsequent step. In fact, this decoupling can create an additional opportunity for optimisation by merging adjacent addresses or removing redundant accesses before modifying the summarising address-range in line 22 of algorithm 4.1. Nevertheless, the task of parallelising such a complex algorithm requires a huge engineering effort – ironically even with the help of tools like the one that we develop in this thesis – and therefore we did not further investigate such a direction.

4.4.2. Strided Accesses

The current implementation of the dependence analyser is using a compact and efficient data-structure to implement address sets. More specifically, it represents an address set as a sorted sequence of continuous ranges of addresses stored in a resizeable array. The size of this structure is proportional to the number of non-continuous address-ranges. *Find* operations that test for membership in a set requires $O(\log_2 n)$ steps on average, where n is the number of continuous ranges. Insertion, however, might require a resize of the array if the inserted item is not adjacent to any range. Tree data structures like binary and splay trees [29] provide faster insertion but increase the space overhead and their extension to support regions of integers can become very complicated. Practically the range sets for the majority of the applications we have examined are relatively small but there is room for improvement especially in the case of accesses that exhibit strided access patterns or nested strided access patterns.

Strided patterns – also called regular sections in array data-flow analysis [45] – are common in scientific application where they typically occur due to affine array index

[7] Reading and writing from a OS pipe requires executing a system call and thus incurs higher overhead than a shared-memory queue which can be synchronised using inter-process mutexes.

functions (e.g. $A[i*c_1+c_2]$, i loop iterator), column-wise traversals of multidimensional arrays and element accesses in arrays of structures. In fact, an extensive study [128] by Shen et al. has shown that about 80% of array accesses in scientific programs exhibit a fixed-stride pattern. Strided access patterns can be represented efficiently since it suffices to store a triplet of (i) the base address, (ii) the stride size, and (iii) the length of the pattern or equivalently the last address of the sequence. Membership test for regular sections can be trivially implemented in constant time. The identification of stride patterns is a well-studied subject primarily in the context of hardware prefetchers that exploit these patterns to reduce the cost of misses in the cache hierarchy [24]. Similarly, recursive strided representations (e.g. Power Regular Section Descriptor PRSD [83, 114]) can additionally capture patterns with affine expressions of more than one variables ($A[i * c_1 + j]$, i, j loop iterators) a typical side-effect of array accesses in nested loops. A complication in the exploitation of strided streams are caused by specific access expressions. Since accesses are summarised for BBs or whole loops in the case of reduction detection, a set of streams – possibly overlapping – is required to be stored. To facilitate efficient membership testing the set should be represented in a sorted tree structure. This fragmentation of the memory access stream, however, might miss the opportunity of summarising adjacent addresses in continuous ranges.

4.4.3. Static Redundancy Elimination

Although in our approach we have opted to instrument all accesses to achieve maximum precision, in many cases data access instrumentation can be avoided altogether just by exploiting local and statically analysable redundancy in computation or accesses (e.g. common sub-expressions). In addition, when targeting specific forms of parallelism like loop-level parallelism the instrumentation of induction variables and automatic variables defined in the loop is not necessary. Furthermore, the granularity of dependence tracking can be extended to include only the control structure which is parallelised (e.g. a whole loop-iteration in the case of loop-level parallelism). Similarly, variables that static analysis indicates that are not part of any "may" dependences can be omitted in the instrumentation pass. This class of optimisations raises the issue of whether the instrumentation pass should precede redundancy elimination optimisation passes or not. The main hurdle in enabling optimisation prior to instrumentation is that most of them, especially the more aggressive global ones introduce new dependences in the data-flow graph and mangle the sequential semantics of the original program. In addition, optimisations ultimately break the $1-1$ correspondence between the source-code and the IR -instructions. Nevertheless, in many cases we can perform a single-pass optimisation that reduces the instrumentation and analysis overhead without compromising precision.

Our current implementation exploits redundancy in expression trees which are side-effect free and thus instrumenting a single instance of an expression does not alter the

dependence graph. Since the CDFG construction is based on BBs we can extend this to remove any redundant references within a BB. This is accomplished by a common expression elimination pass that is an inexpensive standard optimisation implemented in the majority of modern compilers. Global common subexpression elimination on the other hand applies to the whole procedure but it introduces new inter-BB dependences. Closer inspection of the BB data-flow, however, reveals that the instrumentation can be reduced even further. Since the parallelisation methods that we develop in the following chapters target coarse-grain parallelism and in practice never change the order of instructions within a BB, it suffices to instrument only those references that are observable out of the boundaries of the BB. More specifically, this problem can be reduced in the well-known local data-flow problem of *upward exposed uses* (i.e., a $use(expr)$ that is not preceded by a $def(expr)$ in the same BB) and *live definitions* (i.e., a $def(expr)$ that is not killed by a following $def(expr)$ in the same BB). Then the instrumentation can be reduced in instrumenting the upward exposed uses followed by the reaching definitions of the specific BB. Likewise, in loop-level parallelism this analysis can be extended to the boundaries of an iteration. This transformation has also the benefit that the results of data-flow analysis can be propagated to each redundant expression by simply following the results of this local data-flow analysis.

4.5. Conclusion

In this chapter we introduced IR-profiling, a novel methodology that aims at overcoming the limitations of traditional static dependence analysis by utilising profiling information. We presented the design and implementation of a compiler-based tool that instruments the IR of the sequential program and generates control- and data-flow information at profiling time. IR-profiling avoids the complexity of existing low-level instrumentation approaches and provides the means to associate the extracted information with the compiler's internal structures. Furthermore, we introduced the Control- and Data-Flow Graph (CDFG), a powerful whole-program representation that explicitly models the data-dependences which have materialised at profiling time. Finally, we showed how leveraging the technology of IR-profiling enables the identification of complex parallel reduction operations; a common pattern in many sequential applications that existing profiling-based approaches have so far overlooked. In the chapters that follow we demonstrate how this infrastructure facilitates not only the extraction, but also the exploitation of multiple forms of coarse-grain parallelism out of sequential applications.

Chapter 5.

Holistic Approach to Data-level Parallelism Exploitation

The work presented in section 5.2.4 has been conducted in collaboration with Zheng Wang, PhD *student at the University of Edinburgh.*

In this chapter we address the problem of exploiting data-level parallelism from sequential applications written in *C*. We develop a methodology that utilises the IR-profiling infrastructure presented in chapter 4 to overcome the limitations of static analysis in determining loops with independent iterations. Extracted parallelism is then exploited using OpenMP directives. In addition, we replace the traditional target-specific and inflexible mapping heuristics with a machine-learning based prediction mechanism, which results in better mapping decisions while providing more scope for adaptation to different target architectures. Finally, we demonstrate that our approach not only yields significant improvements when compared with state-of-the-art parallelising compilers, but comes close to and sometimes exceeds the performance of manually parallelised codes.

This chapter is structured as follows. We motivate our work based on simple examples in section 5.1.1. This is followed by a presentation of our parallelisation framework in section 5.2. Our experimental methodology and results are discussed in sections 5.3 and 5.4, respectively. Finally, we summarise and conclude in section 5.6.

5.1. Introduction

Parallelisation for multi-processor systems has traditionally followed a *data-level decomposition* in which parallel tasks (e.g. threads) perform the same or similar computation on different elements of aggregate data-structures (e.g. arrays) [76, 77, 159, 132, 107, 14, 6]. Sequential programs typically express that form of computation using loop code constructs (e.g. Do constructs in Fortran) where each iteration operates on a distinct element or dimension of the data. In fact, literature often refers to this paradigm as *loop-level* parallelisation [77, 161, 6]. The abundance of data parallelism in many scientific and embedded algorithms has led to the emergence of many alternative parallel programming models that support the parallelisation of loops using special language

constructs (e.g. `upc_forall` in UPC [50] and `FORALL` in High-Performance Fortran [80]) or annotations (e.g. OPENMP's `#pragma omp` [103]). Nevertheless, compiler support for automatic extraction of data-level parallelism from sequential programs has been limited so far to specific programming languages like Fortran [132, 107] that have found little acceptance in wider application domains. The main reason that auto-parallelisation has not been successful in more general configurations is that static analysis is inherently conservative in languages with more low-level features like pointer arithmetic, dynamic memory allocation and indirect function calls. In addition, current approaches are tuned for fixed target architectures and lack a portable parallelism-mapping methodology. Given that the number and type of processors of a parallel system is likely to change from one generation to the next, finding the right mapping for an application may have to be repeated many times throughout an application's lifetime, hence, making automatic approaches attractive.

5.1.1. Motivation

In this section we provide two simple, but illustrative examples from widely used benchmark applications that demonstrate the significance of the aforementioned limitations in traditional auto-parallelising compilers.

Parallelism Detection Figure 5.1 shows a short excerpt of the *smvp* function from the SPEC *equake* seismic wave propagation benchmark. This function computes a sparse matrix-vector product and takes up more than 60% of the total execution time of the *equake* application on an Intel Xeon architecture. Therefore, it is critical to uncover any available parallelism in it. Unfortunately, automatic parallelisation is bound to fail for both loops. Even if static analysis is able to determine that the assignments to array w might comprise a reduction operation that spans multiple statements, it will fail to disprove that the memory referenced is not accessed at any other point in the respective loop which is a necessary requirement for a reduction to be valid (see section 4.3.5). In fact, w is passed as a pointer argument to function *smvp* and it accesses memory allocated at the same program point with v. In addition, A and w are accessed using indirect index functions (e.g. in expression $w[col]$, col equals $Acol[Anext]$) a common programming technique in applications that use sparse matrix representations which, however, makes most of the static dependence tests inconclusive. On the other hand, IR-profiling instruments every dynamic memory access and thus it provides the necessary information to determine that all the statements that access w are in fact commutative and associative and no other actual flow-dependence materialises for the specific input. Hence the iterations of the inner and outer loop can be executed in parallel given that the partial sums values will be accumulated to w after the execution of the loop. While we still cannot prove absence of data dependences for *every possible* input we can classify both loops as candidates for parallelisation and if profitably parallelisable, present it to the user for approval. For instance, in this example the user can provide

```
1  for (i = 0; i < nodes; i++) {
2    Anext = Aindex[i];     /* The value of Anext is unknown at compile time. */
3                           /* It is used for indexing array A. */
4    Alast = Aindex[i + 1];
5
6    sum0 = A[Anext][0][0]*v[i][0] +
7           A[Anext][0][1]*v[i][1] +
8           A[Anext][0][2]*v[i][2];
9    sum1 = ...
10
11   Anext++;
12   while (Anext < Alast) {
13     col = Acol[Anext];   /* The value of col is unknown at compile time. */
14                          /* It is used for indexing both arrays v and w. */
15
16     sum0 += A[Anext][0][0]*v[col][0] +
17             A[Anext][0][1]*v[col][1] +
18             A[Anext][0][2]*v[col][2];
19     sum1 += ...
20
21     w[col][0] += A[Anext][0][0]*v[i][0] +
22                  A[Anext][1][0]*v[i][1] +
23                  A[Anext][2][0]*v[i][2];
24     w[col][1] += ...
25     Anext++;
26   }
27
28   w[i][0] += sum0;
29   w[i][1] += ...
30 }
```

Figure 5.1.: Static analysis is challenged by sparse array reduction operations and references to arrays that may alias.

the knowledge (and guarantee) that w is not aliased with any other pointer in the loop body.

This example demonstrates that static analysis is overly conservative. Profiling based analysis, on the other hand, can provide accurate dependence information for a *specific* input. When combined we can select candidates for parallelisation based on *empirical evidence* and, hence, can eventually extract more application parallelism than purely static approaches.

```
1 #pragma omp for reduction(+:sum) private(d)
2 for (j=1; j <= lastcol-firstcol-1; j++) {
3     d = x[j] - r[j];
4     sum = sum + d * d;
5 }
```

Figure 5.2.: Despite its simplicity mapping of this parallel loop taken from the NAS CG benchmark is non-trivial and the best-performing scheme varies across platforms.

Mapping In figure 5.2 a parallel reduction loop originating from the parallel NAS Conjugate-Gradient (CG) benchmark is shown. Despite the simplicity of the code, mapping decisions are non-trivial. For example, parallel execution of this loop is not profitable for the Cell BE platform due to high communication costs between processing elements. In fact, parallel execution results in a massive slowdown over the sequential version for the Cell for any number of threads. On the Intel Xeon platform, however, parallelisation can be profitable, but this depends strongly on the specific OPENMP scheduling policy. The best performing scheme (`static`) results in a speedup of 2.3 over the sequential code and performs 115 times better than the worst scheme (`dynamic`) that slows the program down to 2% of its original, sequential performance.

This example illustrates that selecting the correct mapping scheme has a significant impact on performance. However, the mapping scheme varies not only from program to program, but also from architecture to architecture. Therefore, we need an automatic and portable solution for parallelism mapping.

5.1.2. Overview

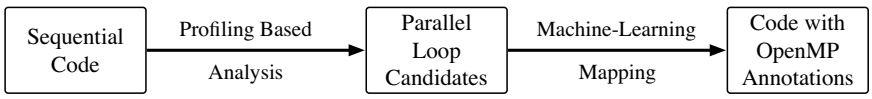


Figure 5.3.: A two-staged parallelisation approach combining profile-driven parallelism detection and machine-learning based mapping to generate OPENMP annotated parallel programs.

In this chapter we follow an approach that integrates profile-driven parallelism detection and machine-learning based mapping in a single framework. We use profiling information to extract only those data dependences that materialise at execution time and enhance the corresponding static analyses with dynamic information. Subsequently, we apply a previously trained machine-learning based prediction mechanism to each parallel loop candidate and decide if and how the parallel mapping should be performed. Finally, we generate parallel code using standard OPENMP annotations. Our approach is semi-automated, i.e., we only expect the user to finally approve those loops where parallelisation is likely to be beneficial, but correctness cannot be proven conclusively.

5.1.3. Contributions

We have evaluated our parallelisation strategy against the NAS and SPEC OMP benchmarks and two different multicore platforms (dual quad-core Intel Xeon SMP and dual-socket QS20 Cell blade). We demonstrate that our approach not only yields significant improvements when compared with state-of-the-art parallelising compilers, but comes close to and sometimes exceeds the performance of manually parallelised codes. We

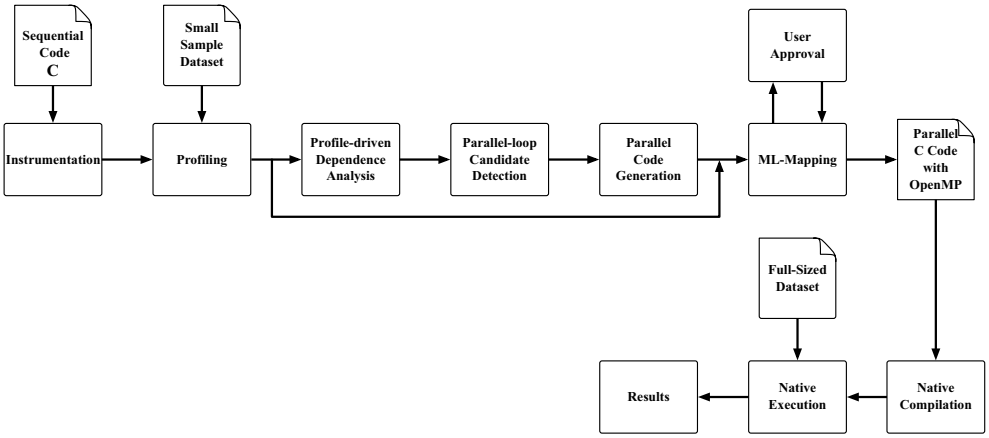


Figure 5.4.: Our parallelisation framework comprises IR-level instrumentation and profiling stages, followed by static and dynamic dependence analyses driving loop-level parallelisation and a machine-learning based mapping stage where the user may be asked for final approval before parallel OpenMP code is generated. Platform-specific code-generation is performed by the native OpenMP-enabled *C* compiler.

show that profile-driven analyses can detect more parallel loops than static techniques. A surprising result is that all loops classified as parallel by our technique are correctly identified as such, despite the fact that only a single, small data input is considered for parallelism detection. Furthermore, we show that parallelism detection in isolation is not sufficient to achieve high performance, and neither are conventional mapping heuristics. Our machine-learning based mapping approach provides the adaptivity across platforms that is required for a genuinely portable parallelisation strategy. On average, our methodology achieves 96% of the performance of the hand-tuned OpenMP NAS and SPEC parallel benchmarks on the Intel Xeon platform, and a significant speedup for the Cell platform, demonstrating the potential of profile-guided machine-learning based auto-parallelisation for complex multicore platforms.

5.2. Parallelisation Framework

In this section we present a parallelisation framework that targets loop-level parallelism. It builds on top of the IR-profiling infrastructure introduced in chapter 4 to determine the parallelisable loops of an application and then generate parallel code using OpenMP directives.

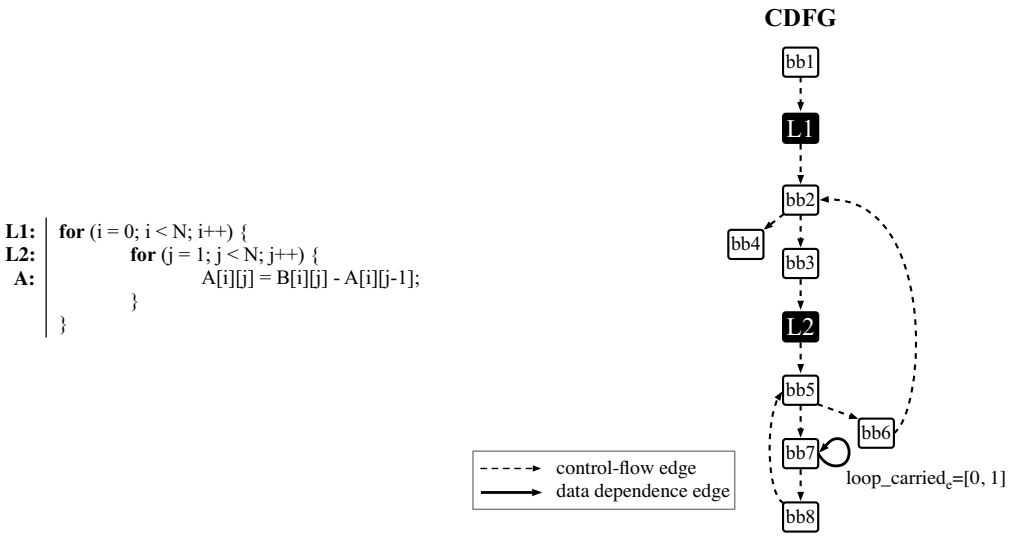
The structure of this section is as follows. First, we provide an overview of the workflow of the parallelisation framework in subsection 5.2.1. A description of the parallel loop detection follows in subsection 5.2.2. Then, in subsection 5.2.3 we present the parallel-code generation methodology and show how data privatisation and reduction operations are handled. The Machine-learning based mapping stage is explained in subsection 5.2.4. Finally, safety issues regarding profile-driven parallelism detection are discussed in subsection 5.2.5.

5.2.1. Workflow

In this subsection we provide an overview of the parallelisation framework workflow. As shown in figure 5.4, after instrumentation and profiling of the sequential *C* code, parallelism candidate detection is performed. For these stages we are using a small sample dataset as input. In this step every loop that is found to be parallel by means of static or profiling-based dependence analysis is marked as parallel using OpenMP annotations. In addition, data scoping for shared, private and reduction data also takes place at this stage. Next, the machine-learning based mapper predicts the performance of each candidate for the specific target machine and selects only those which are profitable. Then, only these selected loops are presented to the user for approval. At the next step parallel *C* code with OpenMP annotations, including loop-scheduling clauses, is generated only for those loops that the user guaranteed correctness. Finally, the parallel code is compiled with the native OpenMP-enabled compiler of the target platform. A complete overview of tool-chain workflow is shown in figure 5.4.

Ir-instrumentation and profile-driven dependence analysis have already been described in detail in chapter 4. Parallel loop detection and code-generation are discussed in the sections that follow.

5.2.2. Parallel Loop Detection

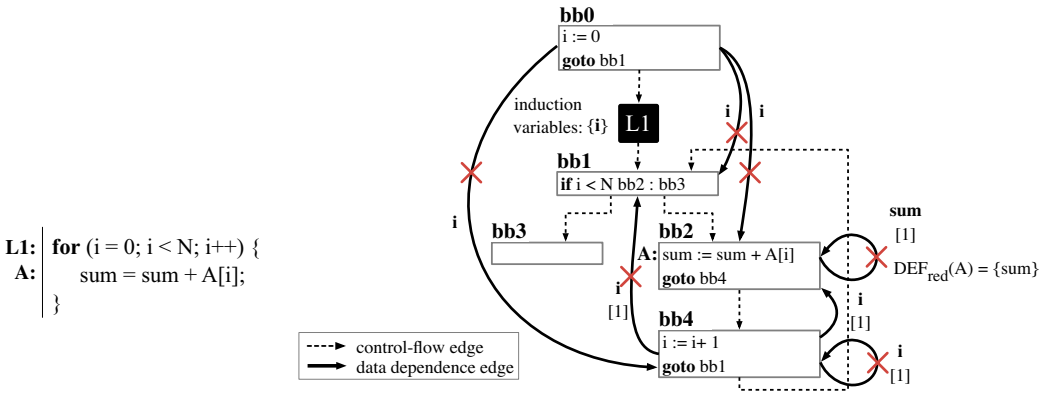


(a) source code **(b)** CDFG

Figure 5.5.: An example of two nested loops where a data-dependence is carried only by the innermost loop (*L2*). The respective Cdfg that is extracted using the profile-driven dependence analysis is shown on the right. Note that the information stored in $loop\_carried_e$ for the self dependence edge on *bb7* allows parallel-loop detection to deduce that *L1* is parallelisable but *L2* is not.

After the Ir-trace processing has been completed, parallel loop candidates are detected based on the dependence information of the Cdfg. More specifically, the loop

hierarchy tree, which is extracted at the CDFG construction stage, is traversed in a top-down fashion. For each loop all the data-dependence edges that flow between nodes of the specific loop are processed. Each dependence edge is annotated with a bit vector that specifies the loop-levels for which it is a *loop-carried* dependence. Based on this information and the loop-level of the current loop we can determine whether this particular edge prohibits parallelisation or otherwise we proceed with the next edge. For instance, in figure 5.5 statement A results in a self flow-dependence but this is carried only from the inner loop $L2$. The outer loop ($L1$), however, modifies each line of the array A independently. In the CDFG this is represented by a self data-dependence edge for BB $bb7$ with its bit-vector $loop\_carried_e$ set only only at the position corresponding to loop $L2$ ($loop\_carried_e$ is computed during the profile-driven dependence analysis, line 23 of algorithm 4.1).



(a) source code **(b)** CDFG

Figure 5.6.: An example where the removal of flow-dependences which are induced by induction and reduction variables enables parallelisation. In the CDFG data-dependence edges are annotated with the set of variables which are carried by the dependence and the *loop_carried*[] vector. Statement A is a valid reduction and thus the address range that is written as a result of the reduction (i.e. the address range accessed by the lhs of A, $DEF_{red}(A)$) can be subtracted by the self-dependence edge on *bb2*. Similarly, i can be removed by all the data-dependence edges of *L1*.

There are two cases, however, that although there is a loop-carried true dependence, parallelisation is still permitted without breaking the sequential semantics. These are flow-dependences regarding (i) loop *induction* variables, and (ii) variables that appear as the left-hand expression of valid *reduction* variables. The addresses of these variables are removed from the respective address sets and if the resulting range is empty we can mark this loop as parallel. Figure 5.6 shows a simple illustrative example where the removal of loop-carried flow-dependences that relate to induction and reduction variables enables parallelisation. Induction variables are statically determined by the compiler in the *loopanalysis* pass and can be accessed at compile-time using the Loop-markers infrastructure. In fact, when targeting loop-level parallelism we can selectively remove the instrumentation for induction variables before the profiling stage to avoid

any redundant computation. Reduction variables can be ignored in parallel loop detection only if these are marked as *valid* from the dependence analyser. Reduction variable analysis has been discussed in detail in section 4.3.5. In this work we are only considering single loop-level parallelisation and more specifically we heuristically select always the outermost loop which is parallel. Thus, parallel loop detection for a specific loop-nest can stop as soon as one level is found to be parallel. All the loops that have been eventually marked as parallel are subsequently fed to the parallel-code generation stage.

5.2.3. Parallel-Code Generation

We use OpenMP for parallel code-generation due to the low complexity of generating the required code annotations and the widespread availability of native OpenMP compilers. In addition, the availability of well studied and widely used sequential benchmarks (Nas-Pb and Spec2000 Fp/Spec Omp2001) that are *hand-parallelised* using OpenMP provides a *strong* and *realistic* upper limit for the speedup that can be achieved with parallelisation.

Currently, our framework is targeting only *C* `for` loops. OpenMP provides two alternatives for the parallelisation of `for` loops: (i) using a parallel worksharing construct, i.e., `omp parallel for`, and (ii) a simple worksharing construct `omp for` nested in a parallel region. For simplicity reasons we primarily use the first construct unless otherwise stated. A simple example from the NAS benchmarks Bt is shown in figure 5.7.

```
1 #pragma omp parallel for
2 for (i3 = 1; i3 < n3-1; i3++) {
3     for (i1 = 0; i1 < n1; i1++) {
4         u[i3][n2-1][i1] = u[i3][1][i1];
5         u[i3][0][i1] = u[i3][n2-2][i1];
6     }
7 }
```

Figure 5.7.: An example from the Nas Mg benchmark where code-generation is inserting the worksharing construct `omp parallel for` to parallelise an individual Doall loop.

Additionally, we have to check that the for loop statement complies with OpenMP standard requirements (§ 2.5.1 in [103]). These restrictions essentially provide the guarantee that the loop variable in the `for` construct is also a valid induction variable of the form $k = k \pm cexpr$, where *cexpr* is a loop invariant expression [95]. In addition, the loop termination test should be an expression that contains only loop-invariant expressions of the form $k \oplus cexpr$, where *oplus* is one of the following operators {"<", "<=", ">" or ">="}. The standard *loopanalysis* pass of CoSy provides detailed information for induction already and thus the test can be trivially implemented on the statically identified increment and termination test expressions. These tests are necessary to allow the OpenMP runtime to compute the number of iterations on entry to the loop.

For the applications that we consider in the experimental evaluation we did not find any case of a profitable loop where parallelisation was prohibited due to this limitation. Nevertheless, loops that do not comply can be still easily parallelised using alternative methods. For instance, the latest OpenMP standard (ver. 3.0) [104] supports the `task` construct which can be used in the case of *uncounted* loops. We did not further investigate this option since we provide a more general and powerful code-generation solution to handle more complex loops in chapter 6.

5.2.3.1. Privatisation

Cdfg provides the necessary information to determine whether a loop is parallelisable or not. Nevertheless, parallel code-generation requires determining all those variables that cause loop-carried anti- and output-dependences in order to privatise them. Variables that should be privatised are computed using an additional analysis of the Ir-instruction stream. This can be performed either in parallel to the Cdfg construction or more efficiently as a post-pass analysis after parallel loop detection. Our current prototype follows the latter approach to avoid computing false-dependences for loops that are not parallelisable. In addition, we can perform the analysis only for a single loop-level, i.e., the outermost parallel loop-level. To compute the data that carry a false dependence it suffices to determine the data that are read or written in an iteration and then written in a subsequent iteration. More specifically, for each memory location a and a specific parallel loop l we record:

1. R_{iter_a}: the normalised iteration number of l when the first `use` for location a was executed in l. Otherwise R_a is negative.
2. W_{iter_a}: the normalised iteration number of l when the first `def` for location a was executed in l. Otherwise it is negative.

Then we determine if there is a loop-carried false dependence for a memory location a at analysis time as follows. If Ir -instruction `def`(a) is executed in iteration k, then there are two non-trivial cases:

1. If $(W_{iter_a} > 0 \wedge W_{iter_a} \neq k)$ then there is a loop-carried output-dependence (Waw) for memory location a in loop l.
2. If $(R_{iter_a} > 0 \wedge R_{iter_a} \neq k)$ then there is a loop-carried anti-dependence (War) for memory location a in loop l.

At the end of the analysis we can determine if a variable requires privatisation by checking whether an anti- or output-dependence was detected in any memory location in its address range.

Finally, we have to check that that the variables requiring privatisation are indeed privatisable.

> A variable x is *privatisable* within a loop if and only if every path from the beginning to the loop body to a *use* of x passes from a *definition* of x before the *use* [6].

Observe that since a loop is parallel there are no flow dependences between two distinct iterations. If a variable is used before its definition and this definition is executed in a different iteration we would have detected a flow-dependence. Therefore, to determine if a variable is privatisable it suffices to check if there are any incoming or outgoing data-dependence edges for this specific variable in the CDFG.

Still, there is a case where privatisation can be valid in the presence of incoming and outgoing data-dependences. More specifically, if there is a data-dependence edge from one loop to a subsequent loop that appears as non loop-carried[1], then we can deduce that every value that is defined in the first loop is used in the second loop in the iteration with the exact same normalised iteration number. In fact this feature is supported by OPENMP using global lifetime private copies, i.e. `threadprivate` data (section 3.3.2). The only restriction that OPENMP imposes is that the respective loops should be scheduled with STATIC loop-scheduling policy and that the same number of threads executes both loops. Since we execute OPENMP with a constant thread number – set with the environmental variable `OMP_THREAD_NUM` – and we can enforce a loop scheduling policy to the machine-learning mapper both requirements are met.

After determining that a variable requires privatisation and it is permitted to be privatised, we add a special `private` OPENMP clause with the list of these variables at the end of the parallel loop directive (clause `private(`*`var1, var2, ...`*`)`). In the case of a global variable, however, there are two cases that require different handling. If there is no function called within the loop body that accesses this variable we can still use the `private` clause. Otherwise, we add a `threadprivate` construct after its definition to make this variable globally private. An example demonstrating this transformation is shown in figure 5.8. In the latter case, if the thread-private global variable is not privatisable in all the parallel loops that it is accessed it should be renamed in this loop and any functions that are accessed within the loop.

Privatisation of heap-allocated objects is more complicated though. For the simple but common case that an array is allocated using a single call to `malloc()` we retrieve the IR statement using the information stored in the global address-range tree constructed during the IR -instruction processing. Based on this statement we create `threadprivate` copies for each thread which are all accessed using the same symbol name inside the parallel loop. In more complicated cases where a multidimensional array is allocated as an array of pointers, we present to the user with the exact point of the source where the allocation is performed and we ask the user to provide the symbolic or constant dimensions of the array. Based on this information we can perform the allocation for `threadprivate` data as in the previous case.

[1]Since iteration numbers are normalised a dependence between two loops appears as non loop-carried in the CDFG if a value that is defined in the first loop in iteration k is used in the second loop in iteration k too.

```
1  double A[N][N];
2  double B[N];
3  int counter;
4  #pragma omp threadprivate(counter);
5
6  int f(int r)
7  {
8      int j;
9
10     for (j = 0; j < N; j++) {
11         if (A[r][j] > 0.0 && A[r][j] < 1.0) {
12             counter++; /* Access out of the lexical scope of the parallel
13                         * construct but within the parallel region.    */
14         }
15     }
16     return counter;
17 }
18
19 void g()
20 {
21     int i;
22
23 #pragma omp parallel for
24     for (i = 0; i < N; i++) {
25         counter = 0; /* Access within the parallel construct */
26         B[i] = f(i);
27     }
28 }
```

Figure 5.8.: This example demonstrates the privatisation of a global scope variable that is accessed out of the lexical scope of the OpenMP parallel construct. In this case it is necessary to use a global-lifetime private copy of this variable using the `threadprivate` OpenMP construct.

5.2.3.2. Reduction Operations

Reduction detection is performed by the dependence analyser as described in section 4.3.5. Note that a reduction operation is not necessary if dependence analysis shows that the loop is parallel without removing any reduction related true dependences (the reduction is still valid but unnecessary). Therefore, in this case we can avoid the reduction overhead altogether. OpenMP handles scalar reductions automatically. A `reduction` clause is inserted at the end of the parallel loop directive specifying the commutative and associative operator $\oplus$ and the scalar variable name x (clause `reduction(`$\oplus$`:x)`).

In the case of an array reduction we insert a template reduction *prologue* and *epilogue* around the loop to be parallelised. Figure 5.9 shows an example template for a two-dimensional array of dimensions $dim1 \times dim2$. Instead of a parallel worksharing construct `omp parallel for` we enclose the array in a parallel region (`omp parallel`) and the loop itself in a nested simple worksharing construct (`omp for`). This way we perform the reduction itself in parallel after the loop execution, avoiding the creation of multiple parallel regions. In the reduction prologue the private copies of the array are initialised with the *identity* element of the specific operator and of the specific data

```
1  #pragma omp parallel
2  {
3      int t, i1, i2;
4      int tid = omp_get_thread_num();
5
6  /* Reduction Prologue */
7      for(i1 = 0; i1 < <dim1>; i1++) {
8          for(i2 = 0; i2 < <dim2>; i2++) {
9              A_priv[tid][i1][i2] = <identity_value>;
10         }
11     }
12
13 #pragma omp for
14     for (i = ...) {
15         for (j = ...) {
16             A_priv[tid][i][j] = A_priv[tid][i][j] ⊕ ...;
17         }
18     }
19
20 /* Reduction Epilogue */
21 #pragma omp for
22     for(i1 = 0; i1 < <dim1>; i1++) {
23         for (t = 0; t < omp_get_num_threads(); t++) {
24             for(i2 = 0; i2 < <dim2>; i2++) {
25                 A[i1][i2] = A[i1][i2] + A_priv[tid][i1][i2] ⊕ ...;
26             }
27         }
28     }
29
30 } /* end of parallel region */
```

Figure 5.9.: An example of the parallel array reduction template for a two-dimensional array. The epilogue is distributed among the threads. Each one performs the accumulation of the partial results to an independent region of the original array.

type (e.g. `0.0f` for "+" on floats). The prologue is performed in parallel by each thread. Each thread is initialising its own copy so that subsequent references do not cause coherency misses. In the reduction epilogue we distribute the reduction computation among all threads. Entry in the epilogue is always safe for all the threads since there is an implied barrier at the end of the `omp for` region. Each thread is accumulating the partial results to the initial variable. Finally note that we extend the private array by one dimension to enable each thread accessing each own copy but also the copies of the other threads. This is necessary to avoid performing the reduction computation sequentially or expensive critical sections.

5.2.3.3. Limitations

Our approach to code-generation is relatively simple and, essentially, relies on OpenMP code annotations alongside minor code transformations. We do not yet perform high-level code restructuring which might help expose more parallelism or improve data locality. While OpenMP is a compiler-friendly target for code-generation it imposes a number of limitations. For instance, we do not yet exploit more flexible parallelisation

structures like pipelines that enable the exploitation of parallelism in partially sequential loops. In fact, this is one of the primary motivations behind the more general parallelism extraction approach which will be presented in chapter 6 and features a more general and effective code-generation methodology.

5.2.4. Machine Learning Based Parallelism Mapping

The responsibilities of the parallelism mapping stage are to decide if a parallel loop candidate is *profitable* to parallelise and, if so, to select a scheduling policy from the four options offered by OpenMP : `cyclic`, `dynamic`, `guided`, and `static`. As the example in figure 5.2 demonstrates, this is a non-trivial task and the optimal solution depends on both the particular properties of the loop under consideration *and* the target platform. To provide a portable, but automated mapping approach we use a machine learning technique to construct a predictor that, after some initial training, will replace the highly platform-specific and often inflexible mapping heuristics of traditional parallelisation frameworks.

5.2.4.1. Predictive Modelling

Separating profitably parallelizable loops from those that are not is a challenging task. Incorrect classification will result in missed opportunities for profitable parallel execution or even in a slow-down due to an excessive synchronization overhead. Traditional parallelising compilers such as Suif-1 employ simple heuristics based on the iteration count and the number of operations in the loop body to decide on whether or not a particular parallel loop candidate should be executed in parallel.

Our data – as shown in figure 5.10 – suggests that such a naïve scheme is likely to fail and that misclassification occurs frequently. A simple work based scheme would attempt to separate the profitably parallelizable loops by a diagonal line as indicated in the diagram in figure 5.10. Independent of where exactly the line is drawn there will always be loops misclassified and, hence, potential performance benefits wasted. What is needed is a scheme that (i) takes into account a richer set of – possibly dynamic – loop features, (ii) is capable of non-linear classification, and (iii) can be easily adapted to a new platform.

We propose a *predictive modelling* approach based on machine-learning classification. In particular, we use *Support Vector Machines (*Svm*)* [15] to decide (a) whether or not to parallelise a loop candidate and (b) how it should be scheduled. The Svm classifier is used to construct hyper-planes in the multi-dimensional space of *program features* – as discussed in the following paragraph – to identify profitably parallelizable loops. The classifier implements a multi-class Svm model with a *Radial Basis Function* (Rbf) kernel capable of handling both linear and non-linear classification problems [15]. The details of our Svm classifier are provided in figure 5.11.

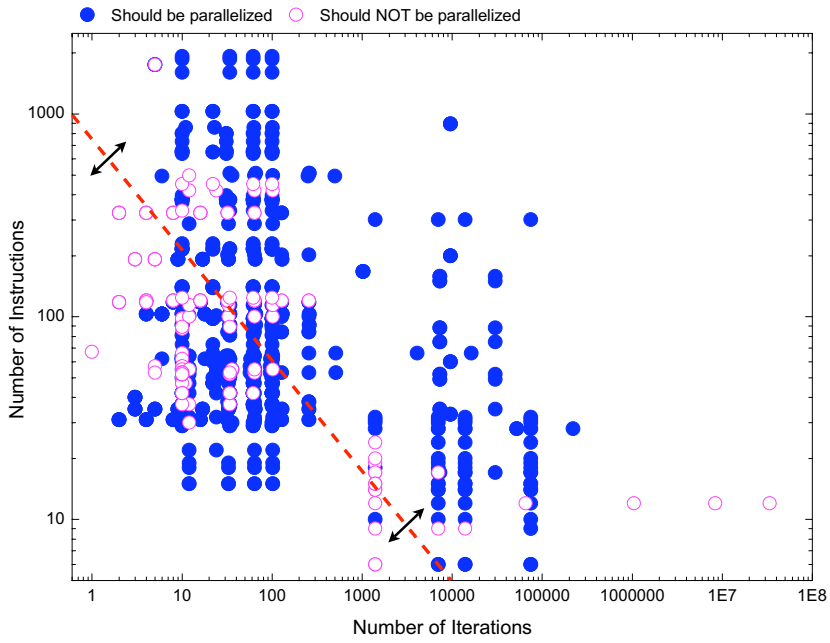


Figure 5.10.: This diagrams shows the optimal classification (sequential/parallel execution) of all parallel loop candidates considered in our experiments for the Intel Xeon machine. Linear models and static features such as the iteration count and size of the loop body in terms of IR statements are not suitable for separating profitably parallelizable loops from those that are not.

5.2.4.2. Program Features

We extract characteristic *program features* that sufficiently describe the relevant aspects of a program and present it to the SVM classifier. An overview of these features is given in table 5.1. The *static features* are derived from CoSy's internal code representation. Essentially, these features characterize the amount of work carried out in the parallel loop similar to e.g. [162]. The *dynamic features* capture the dynamic data access and control flow patterns of the *sequential* program and are obtained from the same profiling execution that has been used for parallelism detection.

Static features	IR Instruction Count IR Load/Store Count IR Branch Count Loop Iteration Count
Dynamic features	Data Access Count Instruction Count Branch Count

Table 5.1.: Features characterizing each parallelizable loop.

It is rather surprising that none of the features in table 5.1 directly expresses the imbalance of the work performed by each iteration of a loop. Still these feature suffice to effectively select the loop-scheduling policy that yields the highest performance. Note, however, that a non-linear machine-learning model is often able to combine the given

1. **Baseline SVM for classification**
   a) Training data:
   $\mathcal{D} = \{(\boldsymbol{x}_i, c_i) | \boldsymbol{x}_i \in \mathbb{R}^p, c_i \in \{-1, 1\}\}_{i=1}^n$
   b) Maximum-margin hyperplane formulation:
   $c_i(\boldsymbol{w} \cdot \boldsymbol{x}_i - b) \geqslant 1$, for all $1 \leqslant i \leqslant n$.
   c) Determine parameters by minimization of $||\boldsymbol{w}||$ (in $\boldsymbol{w}$, b) subject to 1.(b).
2. **Extensions for non-linear multiclass classification**
   a) Non-linear classification:
   Replace dot product in 1.(b) by a kernel function, e.g. the following *radial basis function*:
   $k(\boldsymbol{x}, \boldsymbol{x'}) = exp(-\gamma||\boldsymbol{x} - \boldsymbol{x'}||^2)$, for $\gamma > 0$.
   b) Multiclass SVM:
   Reduce single multiclass problem into multiple binary problems. Each classifier distinguishes between one of the labels and the rest.

Figure 5.11.: Support vector machines for non-linear classification.

features in non-trivial ways and thus its findings are not open to direct interpretation. In fact, experimentation with more features showed that there is no room for further improvement of the scheduling policy prediction. This is also due to the fact that for the specific programs and target platforms that we have studied only in a few cases it is more profitable to select a scheduling policy other than `static`. In fact, this is also the default scheduling policy for both evaluation platforms that we studied and therefore the benefit of correctly selecting the scheduling policy does not improve the performance over manual or heuristic-based mapping. Additionally, in some cases the benefit is negligible. Nevertheless, machine-learning has the advantage that it offers a solution that is both automatic and portable. Therefore, the proposed mechanism, even if it requires the extraction of features like the variation of instructions executed by each iteration, can be also applied in programs and platforms that exploit less biased behaviour and benefit from scheduling policies other than `static`.

5.2.4.3. Training Summary

We use an *off-line supervised learning* scheme whereby we present the machine-learning component with pairs of program features and desired mapping decisions. These are generated from a library of known parallelizable loops through repeated, timed execution of the sequential and parallel code with the different available scheduling options and recording the actual performance on the target platform. Once the prediction model has been built using all the available training data, no further learning takes place.

5.2.4.4. Deployment

For a new, previously *unseen* application with parallel annotations the following steps need to be carried out:

1. *Feature extraction.* This involves collecting the features shown in table 5.1 from the *sequential* version of the program and is accomplished in the profiling stage already used for parallelism detection.

2. *Prediction.* For each parallel loop candidate the corresponding feature set is presented to the SVM predictor and it returns a classification indicating if parallel execution is profitable and which scheduling policy to choose. For a loop nest we start with the outermost loop ensuring that we settle for the most coarse-grained piece of work.

3. *User Interaction.* If parallelisation *appears* to possible (according to the initial profiling) and profitable (according to the previous prediction step), but correctness cannot be proven by static analysis, we ask the user for his/her final approval.

4. *Code Generation.* In this step, we extend the existing OpenMP annotation with the appropriate scheduling clause, or delete the annotation if parallelisation does not promise any performance improvement or has been rejected by the user.

5.2.5. Safety

Unlike static analysis, profile-guided parallelisation cannot conclusively guarantee the absence of control and data dependences for *every possible* input. One simple approach regarding the selection of the "representative" inputs is based on control-flow coverage analysis. This is driven by the empirical observation that for the vast majority of the cases the profile-driven approach might have a *false positive* (i.e., there is a flow-dependence but the tool suggests the contrary) is due to a control-flow path that the data input set did not cover. This also gives a fast way to select representative workloads (in terms of data-dependences) just by executing the applications natively and recording the resulting code coverage. Of course, there are many counter-examples where an input dependent data-dependence appears with no difference in the control-flow. The latter can be verified by the user.

For this current work, we have chosen a "worst-case scenario" and used the *smallest* data set associated with each benchmark for profiling, but evaluated against the *largest* of the available data sets. Surprisingly, we have found that this naive scheme has detected almost all parallelisable loops in the NAS and SPEC OMP benchmarks while not misclassifying any loop as parallelisable when it is not.

Furthermore, with the help of our tools we have been able to identify three incorrectly shared variables in the original NAS benchmarks that should in fact be privatised.

This illustrates that manual parallelisation is prone to errors and that automating this process contributes to program correctness.

5.3. Experimental Methodology

In this section we summarise our experimental methodology and provide an overview of the multicore platforms and benchmarks used throughout the evaluation.

5.3.1. Platforms

For the empirical evaluation we have selected two different architectures that also represent the two extremes of the multi-core design space. The first machine, *M1* in table 3.8, is a shared memory multiprocessor with two *x86_64* quad-core processors and Uniform Access to Memory (UMA). The second machine, *M2*, is a heterogeneous multiprocessor architecture with two *Cell* processors and Non Uniform Access to Memory (NUMA). Each Cell processor contains a general-purpose core (PPE) and 8 vector accelerators (SPE) each with a software-managed private memory. For a more detailed description of the architectural and system configuration of the two machines, as well as an empirical analysis of the overhead for the OPENMP constructs used in code-generation the reader can refer to section 3.5.

5.3.2. Benchmarks

For our evaluation we have selected benchmarks (NAS and SPEC OMP) where both sequential and manually parallelised OPENMP versions are available. This has enabled us to directly compare our parallelisation strategy against parallel implementations from independent expert programmers.

More specifically, we have used the NAS NPB (sequential ver. 2.3) and NPB (OMP ver. 2.3) codes alongside the SPEC CPU2000 benchmarks and their corresponding SPEC OMP2001 counterparts. However, it should be noted that the sequential and parallel SPEC codes are not immediately comparable due to some amount of restructuring of the "official" parallel codes, resulting in a performance advantage of the SPEC OMP codes over the sequential ones, even on a single processor system.

Each program has been executed using multiple different input data sets (shown in table 5.2), however, for parallelism detection and mapping we have only used the *smallest* of the available data sets[2]. The resulting parallel programs have then been evaluated against the larger inputs to investigate the impact of *worst-case input* on the safety of our parallelisation scheme.

[2]Some of the larger data sets could not be evaluated on the Cell due to memory constraints.

Program	Suite	Data Sets/Xeon	Data Sets/Cell
BT	NPB2.3-OMP-C	S, W, A, B	NA
CG	NPB2.3-OMP-C	S, W, A, B	S, W, A
EP	NPB2.3-OMP-C	S, W, A, B	S, W, A
FT	NPB2.3-OMP-C	S, W, A, B	S, W, A
IS	NPB2.3-OMP-C	S, W, A, B	S, W, A
MG	NPB2.3-OMP-C	S, W, A, B	S, W, A
SP	NPB2.3-OMP-C	S, W, A, B	S, W, A
LU	NPB2.3-OMP-C	S, W, A, B	S, W, A
art	SPEC CFP2000	test, train, ref	test,train, ref
ammp	SPEC CFP2000	test, train, ref	test,train, ref
equake	SPEC CFP2000	test, train, ref	test,train, ref

Table 5.2.: Benchmark applications and data sets.

5.3.3. Methodology

We have evaluated three different parallelisation approaches: *manual*, *auto-parallelisation* using the Intel ICC compiler (just for the Intel platform), and our *profile-driven* approach.

For native code-generation all programs (both sequential and parallel OPENMP) have been compiled using the Intel ICC and IBM XLC compilers for the Intel Xeon and IBM Cell platforms, respectively.

Furthermore, we use *leave-one-out cross-validation* to evaluate our machine-learning based mapping technique. This means that for K programs, we remove one, train a model on the remaining $K-1$ programs and predict the K^{th} program with the previously trained model. We repeat this procedure for each program in turn.

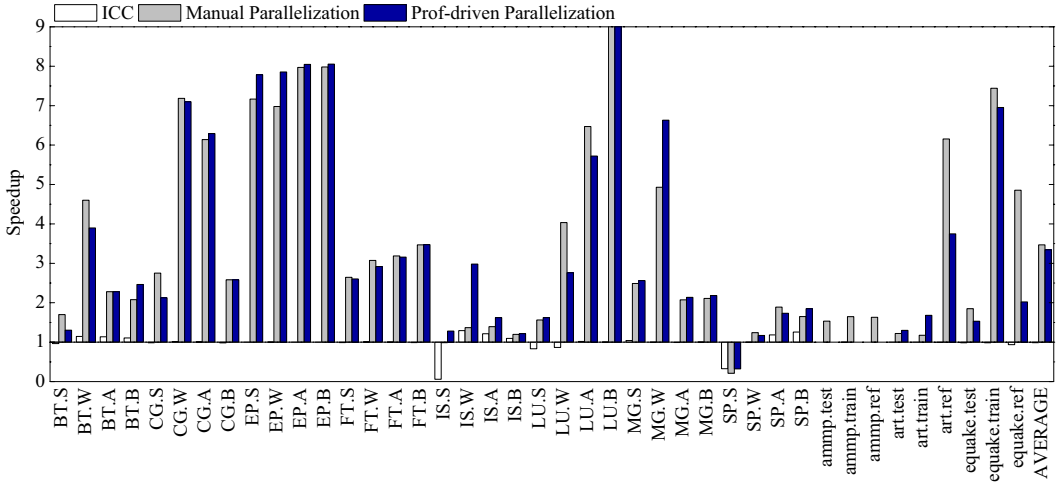
For the Cell platform we report parallel speedup over sequential code running on the general-purpose PPE rather than a single SPE. In all cases the sequential performance of the PPE exceeds that of a single SPE, ensuring we report improvements over the *strongest* baseline available.

All execution time and speedup figures presented in the following paragraphs have been computed as the arithmetic mean over 10 executions. Machines *M1* and *M2* were practically idle before the experiments start and besides the absolutely necessary system services no other programs were executing during the experiments. Additionally, any I/O performed has been exclusively to local filesystems in order to reduce variation introduced by the cluster's network filesystem. Performance figures do not include standard deviation bars since the relative standard error of the mean was less that 0.5% for all the experiments. This rather surprising fact can be primarily attributed to static work partitioning that is selected for the majority of the parallelised loops. Furthermore, binding threads to specific cores by utilising the thread-to-core affinity feature of ICC's runtime system did not only result to the highest performance for each individual parallelisation approach, but also practically eliminated any variation introduced due to OS scheduling decisions.

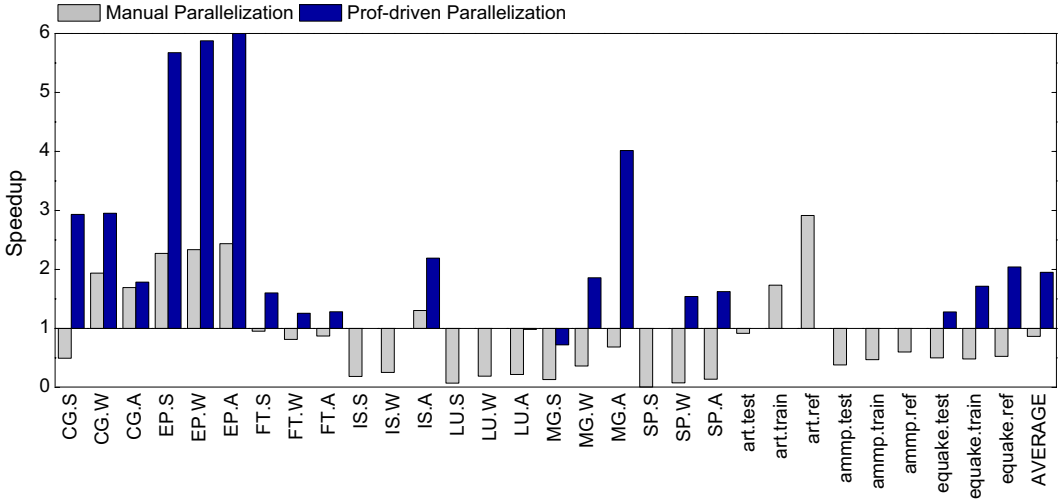
5.4. Empirical Evaluation

In this section we present and discuss the results of the empirical evaluation.

5.4.1. Overall Results



(a) Speedup over sequential codes achieved by Icc auto-parallelisation, manual parallelisation and profile-driven parallelisation for the Xeon platform.



(b) Speedup over sequential code achieved by manual parallelisation and profile-driven parallelisation for the dual Cell platform.

Figure 5.12.: Speedup of the different parallelisation schemes on the two evaluation platforms.

Figures 5.12(a) and 5.12(b) summarise our performance results for both the Intel Xeon and Ibm Cell platforms.

5.4.1.1. Intel Xeon

The most striking result is that the Intel auto-parallelising compiler fails to exploit any usable levels of parallelism across the whole range of benchmarks and data set sizes. In

fact, auto-parallelisation results in a slow-down of the BT and LU benchmarks for the smallest and for most data set sizes, respectively. ICC gains a modest speedup only for the larger data sets of the IS and SP benchmarks. The reason for this disappointing performance of the Intel ICC compiler is that it is typically parallelising at inner-most loop level where significant fork/join overhead negates the potential benefit from parallelisation.

The manually parallelised OPENMP programs achieve an average speedup of 3.5 across the benchmarks and data sizes. In the case of EP, a speedup of 8 was achieved for large data sizes. This is not surprising since this is an embarrassingly parallel program. More surprisingly, LU was able to achieve super-linear speedup (9×) due to improved caching [39]. Some programs (BT, MG and CG) exhibit lower speedups with larger data sets (A and B in comparison to W) on the Intel machine. This is a well-known and documented scalability issue of these specific benchmarks [39].

For most NAS benchmarks our profile-driven parallelisation achieves performance levels close to those of the manually parallelised versions, and sometimes outperforms them (EP, IS and MG). This surprising performance gain can be attributed to three important factors. Firstly, our approach parallelises outer loops whereas the manually parallelised codes have parallel inner loops. Secondly, our approach exploits reduction operations on array locations and, finally, the machine learning based mapping is more accurate in eliminating non-profitable loops from parallelisation and selecting the best scheduling policy.

The situation is slightly different for the SPEC benchmarks. While profile-driven parallelisation still outperforms the static auto-paralleliser we do not reach the performance level of the manually parallelised codes. Investigations into the causes of this behaviour have revealed that the SPEC OMP codes are not equivalent to the sequential SPEC programs, but have been manually restructured [7]. For example, data structures have been altered (e.g. from *list* to *vector*) and standard memory allocation (excessive use of *malloc*) has been replaced with a more efficient scheme. Obviously, these changes are beyond what an auto-paralleliser is capable of performing. In fact, we were able to confirm that the sequential performance of the SPEC OPENMP codes is on average about 2 times (and up to 3.34 for *art*) above that of their original SPEC counterparts. We have verified that our approach parallelises the same critical loops for both *equake* and *art* as SPEC OMP. For *art* we achieve a speedup of 4, whereas the SPEC OMP version is 6 times faster than the sequential SPEC FP version, of which more than 50% is due to sequential code optimisations. We also measured the performance of the profile-driven parallelised *equake* version using the same code modifications and achieved a comparable speedup of 5.95.

Overall, the results demonstrate that our profile-driven parallelisation scheme significantly improves on the state-of-the-art Intel auto-parallelising compiler. In fact, our approach delivers performance levels close to or exceeding those of manually parallelised codes and, on average, we achieve 96% of the performance of hand-tuned parallel

	Profile-driven			ICC	Manual
Application	**#loops (%cov)**	**FP**	**FN**	**#loops (%cov)**	**#loops (%cov)**
BT	205 (99.9%)	0	0	72 (18.6%)	54 (99.9%)
CG	28 (93.1%)	0	0	16 (1.1%)	22 (93.1%)
EP	8 (99.9%)	0	0	6 (<1%)	1 (99.9%)
FT	37 (88.2%)	0	0	3 (<1%)	6 (88.2%)
IS	9 (28.5%)	0	0	8 (29.4%)	1 (27.3%)
LU	154 (99.7%)	0	0	88 (65.9%)	29 (81.5%)
MG	48 (77.7%)	0	3	9 (4.7%)	12 (77.7%)
SP	287 (99.6%)	0	0	178 (88.0%)	70 (61.8%)
equake_SEQ	69 (98.1%)	0	0	29 (23.8%)	11 (98.0%)
art_SEQ	31 (85.6%)	0	0	16 (30.0%)	5 (65.0%)
ammp_SEQ	21 (1.4%)	0	1	43 (<1%)	7 (84.4%)

Table 5.3.: Number of parallelised loops and their respective coverage of the sequential execution time.

OpenMP codes, resulting in an average speedup of 3.34 across all benchmarks.

5.4.1.2. IBM Cell

Figure 5.12(b) shows the performance resulting from manual and profile-driven parallelisation for the dual-Cell platform.

Unlike the Intel platform, the Cell platform does not deliver a high performance on the manually parallelised OpenMP programs. On average, these codes result in an overall slowdown. For some programs such as CG and EP small performance gains could be observed, however, for most other programs the performance degradation is disappointing. Given that these are hand-parallelised programs this is perhaps surprising and there are essentially two reasons why the Cell's performance potential could not be exploited. Firstly, it is clear that the OpenMP codes have not been developed specifically for the Cell. The programmers have not considered the communication costs for a distributed memory machine. Secondly, in absence of specific scheduling directives the OpenMP runtime library resorts to its default behaviour, which leads to poor overall performance. Given that the manually parallelised programs deliver high performance levels on the Xeon platform, the results for the Cell demonstrate that parallelism detection in isolation is not sufficient, but mapping must be regarded as equally important.

In contrast to the "default" manual parallelisation scheme, our integrated parallelisation strategy is able to successfully exploit significant levels of parallelism, resulting in average speedup of 2.0 over the sequential code and up to 6.2 for individual programs (EP). This success can largely be attributed to the improved mapping of parallelism resulting from our machine-learning based approach.

5.4.2. Parallelism Detection and Safety

Our approach relies on dynamic profiling information to discover parallelism. This has the obvious drawback that it may classify a loop as potentially parallel when there exists another data set which would highlight a dependence preventing correct parallelisation. This is a fundamental limit of dynamic analysis and the reason for requesting the user to confirm uncertain parallelisation decisions. It is worthwhile, therefore, to examine to what extent our approach suffers from *false positives* (i.e., loop is incorrectly classified as parallelisable). Clearly, an approach that suffers from high numbers of such false positives will be of limited use to programmers.

Column 2 in table 5.4.2 shows the number of loops our approach detects as potentially parallel. The column labelled Fp (False Positive) shows how many of these were in fact sequential. The surprising result is that none of the loops we considered potentially parallel turned out to be genuinely sequential. Certainly, this result does not prove that dynamic analysis is always correct. Still, it indicates that profile-based dependence analysis may be more accurate than generally considered, even for profiles generated from small data sets. Clearly, this encouraging result will need further validation on more complex programs before we can draw any final conclusions.

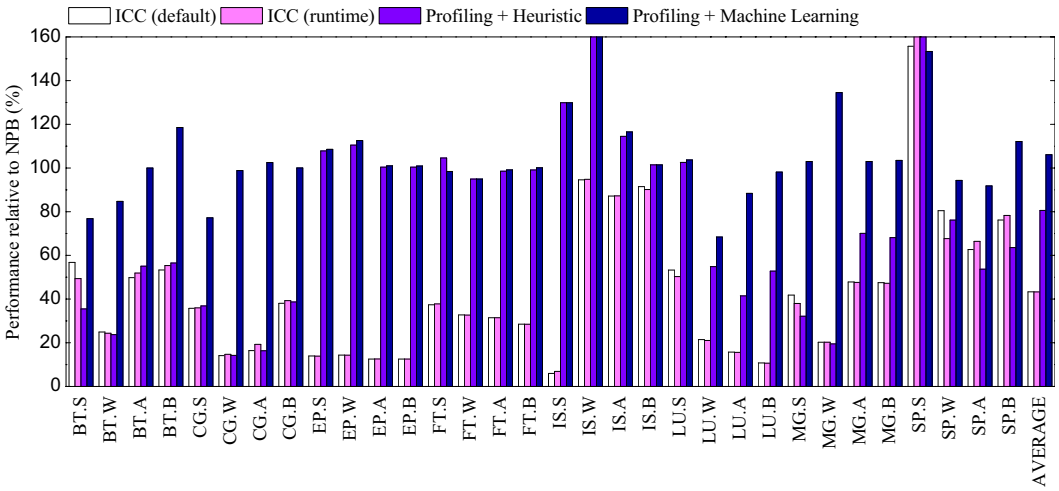
Column 3 in table 5.4.2 lists the number of loops parallelisable by Icc. In some applications, the Icc compiler is able to detect a considerable number of parallel loops. In addition, if we examine the coverage (shown in parentheses) we see that in many cases this covers a considerable part of the program. Therefore we conclude that it is less a matter of the parallelism detection that causes Icc to perform so poorly, but rather how it exploits and maps the detected parallelism (see section 5.4.3).

The final column in table 5.4.2 eventually shows the number of loops parallelised in the hand-coded applications. As before, the percentage of sequential coverage is shown in parentheses. Far fewer loops than theoretically possible are actually parallelised because the programmer have obviously decided only to parallelise those loops they considered "hot" and "profitable". These loops cover a significant part of the sequential time and effective parallelisation leads to good performance as can be seen for the Xeon platform.

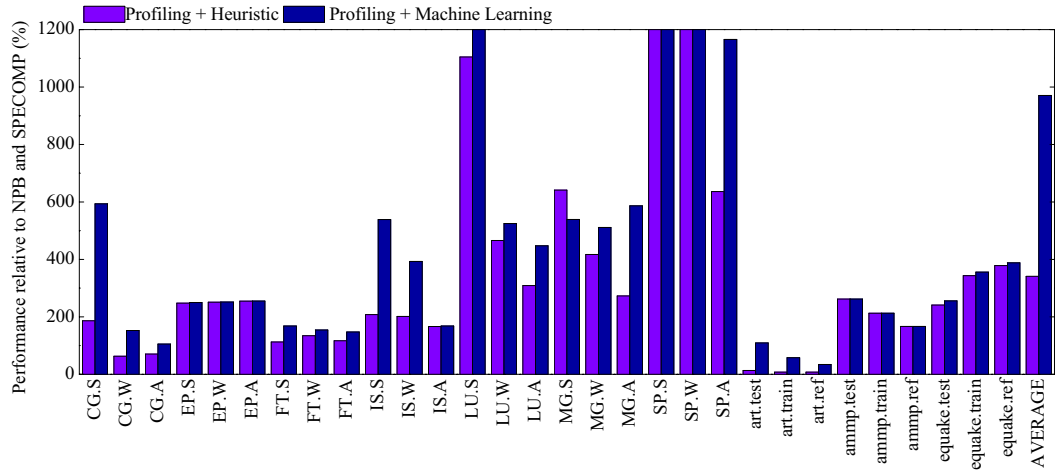
In total there are four *false negatives* (column Fn in table 5.4.2), i.e., loops not identified as parallel although safely parallelisable. Three false negatives are contained in the *MG* benchmark, and two of these are due to loops which have zero iteration counts for all data sets and, therefore, are never profiled. The third one is a Max reduction, which is contained inside a loop that our machine-learning classifier has decided not to parallelise.

5.4.3. Parallelism Mapping

In this section we examine the effectiveness of three mapping schemes (manual, heuristic-based, and machine-learning using profiling information) for the two platforms.



(a) NAS benchmarks on the Intel Xeon platform.



(b) NAS and SPEC FP benchmarks on the IBM Cell platform.

Figure 5.13.: Impact of different mapping approaches (100% = manually parallelised OPENMP code).

5.4.3.1. Intel Xeon

Figure 5.13(a) compares the performance of ICC and our approach to that of the hand-parallelised OPENMP programs. In the case of ICC we show the performance of two different mapping approaches. By default, ICC employs a compile-time profitability check while the second approach performs a runtime check using a dynamic profitability threshold.

For some cases (BT.B and SP.B) the runtime checks provide a marginal improvement over the static mapping scheme while the static scheme is better for IS.B. Overall, both schemes are equally poor and deliver less than half of the speedup levels of the hand-parallelised benchmarks. The disappointing performance appears to be largely due to non-optimal mapping decisions, i.e., to parallelise inner loops rather than outer ones.

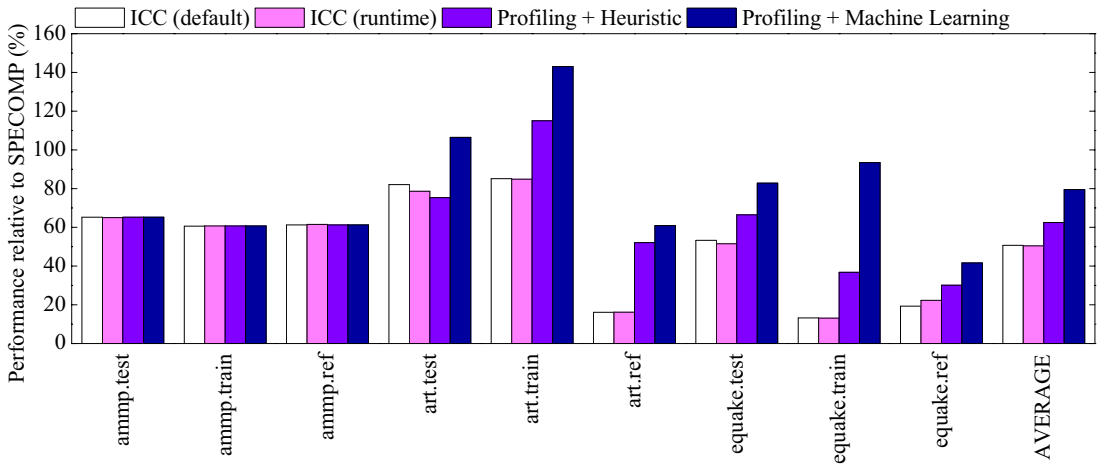


Figure 5.14.: Impact of different mapping approaches for the SPEC benchmarks (100% = manually parallelised OPENMP code).

In the same figure we compare our machine-learning based mapping approach against a scheme which uses the same profiling information, but employs a fixed, work-based *heuristic* similar to the one implemented in the SUIF-1 parallelising compiler. This heuristic considers the product of the iteration count and the number of instructions contained in the loop body and decides against a static threshold. While our machine-learning approach delivers nearly the performance of the hand-parallelised codes and, in some cases, is able to outperform them, the static heuristic performs poorly and is unable to obtain more than 85% of the performance of the hand-parallelised code. This translates into an average speedup of 2.5 rather than 3.7 for the NAS benchmarks. The main reason for this performance loss is that the default scheme using only static code features and a linear work model is unable to accurately determine whether a loop should be parallelised or not.

In figure 5.14 we compare the performance resulting from the different automated mapping approaches to that of the hand-parallelised SPEC OMP codes. Again, our machine-learning based approach outperforms ICC and the fixed heuristic. On average, our approach delivers 88% of the performance of the hand-parallelised code, while ICC and the fixed heuristic approach achieve performance levels of 45% and 65%, respectively. The lower performance gains for the SPEC benchmarks are mainly due to a better starting point of the hand-parallelised SPEC OMP benchmarks (see section 5.4.1.1).

5.4.3.2. IBM Cell

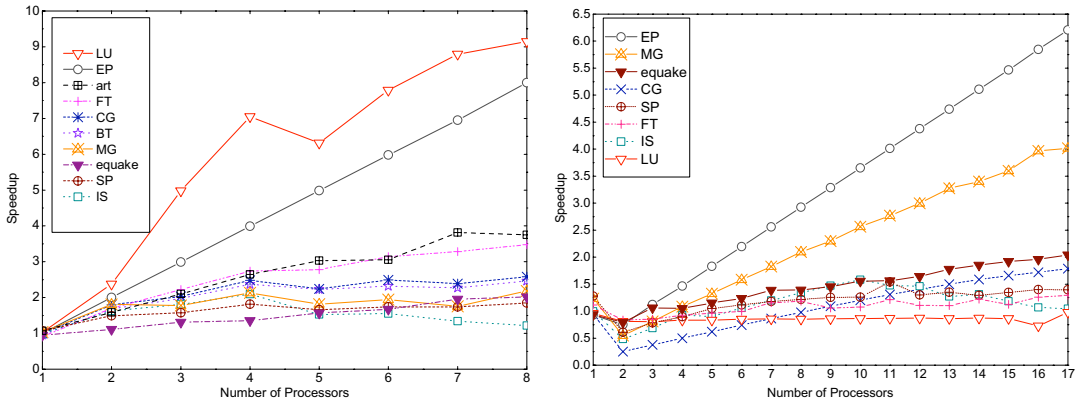
The diagram in figure 5.13(b) shows the speedup of our machine-learning based mapping approach over the hand-parallelised code on the Cell platform. As before, we compare our approach against a scheme which uses the profiling information, but employs a fixed mapping heuristic.

The manually parallelised OpenMP programs are not specifically "tuned" for the Cell platform and perform poorly. As a consequence, the profile-based mapping approaches show high performance gains over this baseline, in particular, for the small input data sets. Still, the combination of profiling and machine-learning outperforms the fixed heuristic counterpart by far and, on average, results in a speedup of 9.7 over the hand-parallelised OpenMP programs across all data sets.

5.4.3.3. Summary

The combined profiling and machine-learning approach to mapping comes within reach of the performance of hand-parallelised code on the Intel Xeon platform and in some cases outperforms it. Fixed heuristics are not strong enough to separate profitably parallelisable loops from those that are not and perform poorly. Typically, static mapping heuristics result in performance levels of less than 60% of the machine learning approach. This is because the default scheme is unable to accurately determine whether a loop should be parallelised or not. The situation is exacerbated on the Intel Cell platform where accurate mapping decisions are key enablers to high performance. Existing ("generic") manually parallelised OpenMP codes fail to deliver any reasonable performance and heuristics, even if based on profiling data, are unable to match the performance of our machine-learning based scheme.

5.4.4. Scalability



(a) Intel Xeon platform **(b)** IBM Cell platform

Figure 5.15.: Scalability of all the applications using the largest data set on the two platforms.

For the Xeon platform the Lu and Ep benchmarks scale well with the number of processors (see figure 5.15(a)). In fact, a super-linear speedup due to more cache memory in total can be observed for the Lu application. For other benchmarks scalability is more limited and often saturation effects occur for four or more processors. This scalability issue of the Nas benchmarks is well-known and in line with other research publications [39]. Figure 5.15(b) shows a performance drop for the step from one to

two processors on the Cell platform. This is due to the fact that we use the generally more powerful PPE to measure single processor performance, but then use the multiple SPEs for parallel performance measurements. The diagram reveals that in the best case it takes about three SPEs to achieve the original performance of the PPE. Some of the more scalable benchmarks such as EP and MG follow a linear trend as the number of processors increases, however, most of the remaining benchmarks saturate at a low level.

5.5. Data-parallelism in Embedded Applications

So far our evaluation has focused on scientific applications. Still, profile-driven parallelism extraction is generally applicable to all applications that exhibit data-level parallelism. Many embedded and multimedia kernels consist of parallelisable DOALL loops. The majority of them, however, operates on small datasets and thus parallelisation is often not effective, especially on modern powerful embedded or general-purpose processors. These kernels are typically utilised as constituting components in more complex algorithms where they are arranged so as to form a processing *pipeline* – a parallelisation structure that we address in the chapter that follows. These observations are in agreement with recent studies [62, 68] that reach the conclusion that the potential of loop-level parallelism in embedded applications is limited and thus other forms of parallelism should be explored. Nevertheless, for the sake of completeness we include the experimental results of a study for data-parallelism in appendix A. Note that although our approach successfully detects parallelism in these kernels, the input datasets used in the evaluation had to be scaled up. For typical inputs these programs have very short execution time and thus parallelisation leads to limited speedup if not slowdown. This leads us to the conclusion that in the case of embedded applications the rather limited impact on overall performance is due to inherent features of the applications and the typical datasets rather than a limitation of the proposed methodology.

5.6. Conclusion

In this chapter we have developed a platform-agnostic, profiling-based parallelism exploitation methodology that enhances static data dependence analyses with profiling information, resulting in larger amounts of parallelism uncovered from sequential applications. We have also shown that parallelism detection in isolation is not sufficient to achieve high performance, but requires close interaction with an adaptive mapping scheme to unfold the full potential of parallel execution across programs and architectures. Results obtained on two complex multicore platforms (Intel Xeon and IBM Cell) and two sets of benchmarks (NAS and SPEC) confirm that our method is more

aggressive in parallelisation and more portable than existing static auto-parallelisation and achieves performance levels close to manually parallelised codes.

Chapter 6.

Semi-Automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism

In the past parallelising compiler technology targeted mainly scientific applications with an abundance of data-level parallelism and, hence, the scope was largely restricted to the detection and mapping of this particular kind of parallelism. On the other hand, popular multimedia – and many other streaming applications – typically comprise multiple levels of parallelism. The advent of general-purpose chip multi-processors (CMPs) as well as the proliferation of multimedia consumer electronics has underlined the importance of tools that take a broader approach and target higher-level parallelism. At the same time the increased complexity of modern multimedia algorithms necessitates the exploitation of not only the inner-most vectorizable loops – which also characterised traditional digital signal processing (DSP) applications – but also coarse-grain parallelism [108]. In this chapter we address the problem of extraction and exploitation of *pipeline parallelism* which is central to most multimedia and stream processing applications [108]. This does not exclude the exploitation of other levels of parallelism and, in fact, the approach presented is orthogonal to e.g. data-level parallelisation *within* individual pipeline stages.

This chapter is structured as follows. We present motivational examples and an overview of our approach in section 6.1. The methodology for pipeline partitioning and parallel-code generation follows in section 6.2. Finally we demonstrate the applicability of our approach in section 6.3, before we summarise and conclude in section 6.4.

6.1. Introduction

Parallelism detection is tightly coupled to static data and control-flow analyses that provide essential information about dependence relationships that a parallelisation scheme must obey in order to guarantee *correctness*. Unfortunately, dependence analysis is often statically *undecidable* and *conservative* approximations need to be made which in turn limit the success of automatic parallelisation. In the previous chapters we introduced a profiling-based parallelisation approach which demonstrated that many

potential dependences do not materialise in real-world applications. Therefore, more aggressive approaches can deliver performance improvements matching those of manual parallelisation while only requiring minimal user interaction [145]. We build on top of this work and present an *optimistic* pipeline extraction methodology based on *intermediate representation profiling* (IR-profiling) – introduced in chapter 4 – and a *hierarchical whole program representation*. Since profiling for data and control dependences is inherently *unsafe* our approach does not *guarantee* correctness. We therefore provide the user with a graphical representation of the extracted pipeline and highlight the critical items for *manual verification*.

Traditionally, there has been an *information gap* between profiling and compilation. This is due to binary profiling tools that track dependences on machine instruction level, but are unable to *back annotate* this information and make it usable within the parallelising compiler. We bridge this gap and show how instrumentation of the compiler *intermediate representation* (IR) enables us to make the profiling data readily available in the compiler. Furthermore, we develop a uniform program representation based on the *Program Dependence Graph* (PDG) that is used throughout the entire parallelisation process including profiling, transformation, partitioning, mapping and eventually code-generation. We introduce a *hierarchical top-down* pipeline extraction methodology that splits functions and multi-level loops *on demand*, for example, to achieve a better balance between pipeline stages. Individual pipeline stages that form a performance bottleneck, but can be shown to operate on independent data items, are *replicated* and, effectively, create the opportunity for *Out-of-Order* (OoO) execution.

Our approach is complete and comprises a code-generation stage that automatically forms processing pipelines and builds on top of a lightweight, retargetable runtime system. Our current code generator targets readily available commodity systems and does not require special hardware (HW) or operating system (OS) support. We have evaluated our approach on a number of popular multimedia and stream processing applications taken from the EEMBC and SPEC benchmark suites. These applications contain complex, idiosyncratic programming constructs that are typical of many real-world applications. We demonstrate how we perform pipeline extraction in the presence of multi-level loops, dynamic memory management, and interleaved I/O and achieve speedups of up to 4.7 on a eight-core Intel Xeon machine.

6.1.1. Motivating Examples

Multimedia applications typically comprise pipelined computations and operate on a stream of data of different granularity. This is illustrated in the code excerpt in figure 6.1 representing a simplified MP3 decoder. The outer-most loop operates on audio frames, *III_stereo* operates on both stereo channels, and *III_hufman_decode* and *III_antialias* operate on a single audio channel at each invocation. It is clear that any approach that forms pipelines by partitioning the code on a single level (e.g. [141, 20, 142])

```
1  while(end) {
2    /* ...input... */
3    decode_info(&bs, &fr_ps);
4    /* ...input... */                                  level 1
5    III_get_side_info(&bs, &III_side_info, &fr_ps);
6
7    for (gr = 0; gr < max_gr; gr++) {
8      for (ch = 0; ch < stereo; ch++) {
9        III_get_scale_factors(gr, ch, ...);
10       III_hufman_decode(gr, ch, ...);                level 3
11       III_dequantize_sample(gr, ch, ...);
12     } /* ch */
13
14     III_stereo(gr, ...);                             level 2
15
16     for (ch = 0; ch < stereo; ch++) {
17       III_reorder(ch, gr, ...);
18       III_antialias(ch, gr, ...);
19
20       for (sb = 0; sb < SBLIMIT; sb++) {
21         III_hybrid(sb, ch, ...);
22       } /* ss */
23
24       for (ss = 0; ss < SSLIMIT; ss++) {             level 3
25         for (sb = 0; sb < SBLIMIT; sb++) {
26           if ((ss % 2) && (sb % 2))
27             polyPhaseIn[sb] = -hybridOut[sb][ss];
28           else
29             polyPhaseIn[sb] = hybridOut[sb][ss];
30         } /* sb */
31         clip += SubBandSynthesis(ch, ss, ...);
32       } /* ss */
33     } /* ch */
34   } /* gr */
35
36   out_fifo(*pcm_sample, ...);                        level 1
37 } /* while */
```

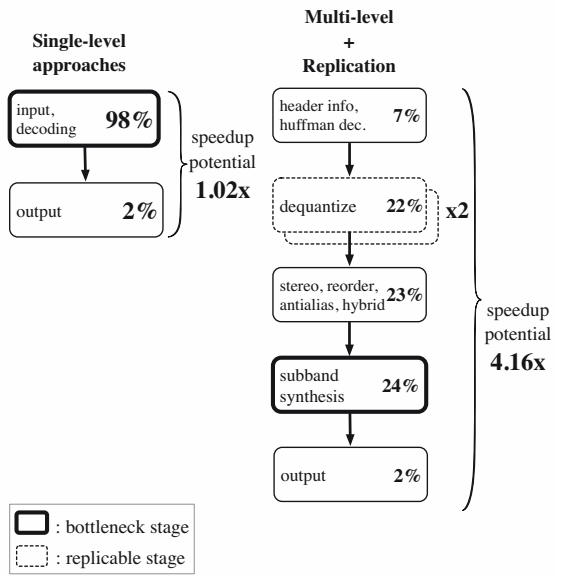
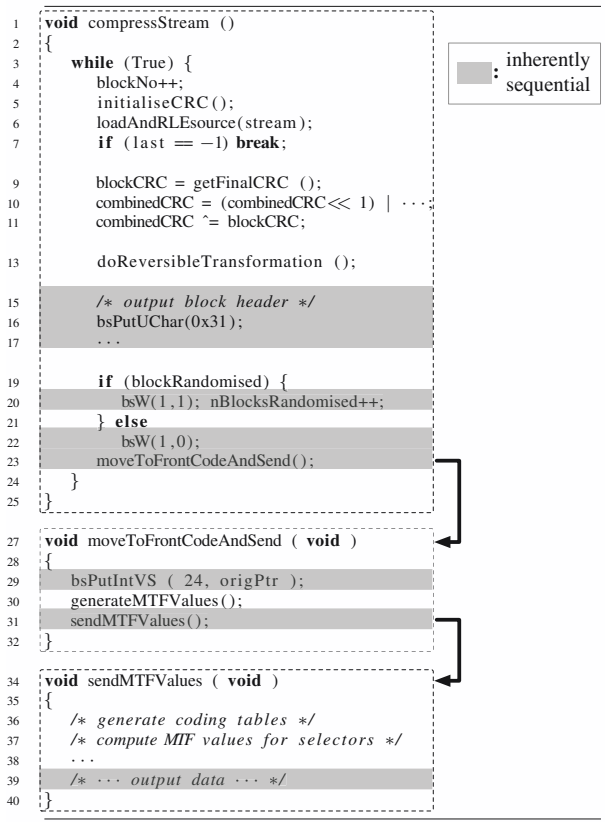


Figure 6.1.: Multimedia applications typically consist of computation pipelines which operate on a stream of data of different granularity. This figure shows a code excerpt from the EEMBC 2.0 *MP3 decoding* application.

is always bound by the execution time of the slowest pipeline stage. For example, partitioning on level 1 can only form a pipeline of two highly imbalanced stages, input & decoding (98% of total execution time) and output (2%), which in theory can lead to a minuscule 2% improvement over the sequential execution, but in practice will lead to slowdown due to communication overheads. Similar to top-level partitioning neither does partitioning at levels 2 or 3 alone result in a balanced pipeline. *Decoupled Software Pipelining* (DSWP) [106], for example, operates on a representation too low level to perform the necessary data privatisation required for its parallel implementation. Our approach, on the other hand, is not restricted to a single loop level from which to extract pipeline stages and constructs the more balanced pipeline on the right side of figure 6.1. Using a hierarchical whole-program representation we identify the pipeline stage that represents the bottleneck (input & decoding) and selectively expose inner levels, creating a pipeline that spans multiple levels of the loop nest. The more equally balanced pipeline has a speedup potential of 4.16.

In general, multi-level loop partitioning can expose more parallelisation opportunities and, thus, provide more flexibility to the partitioner. For instance, the *III_hufman_decode* function that is inherently sequential can be decoupled from *III_dequantize_sample*. Then the latter, which is in fact a parallel stage and can operate on two audio channels independently, can be replicated as shown in the rightmost pipeline in figure 6.1.

Another common scenario is that parallel operations are interleaved with sequential operations such as I/O. Although, program representations with explicit dependences (control or data) can overcome such limitations, effectively separating the parallelism from code syntax, the problem still holds for code with function calls. Figure 6.2 shows




```
1  void compressStream ()
2  {
3      while (True) {
4          blockNo++;
5          initialiseCRC();
6          loadAndRLEsource(stream);
7          if (last == -1) break;
9          blockCRC = getFinalCRC ();
10         combinedCRC = (combinedCRC<< 1) | ...;
11         combinedCRC ^= blockCRC;
13         doReversibleTransformation ();
15         /* output block header */
16         bsPutUChar(0x31);
17         ...
19         if (blockRandomised) {
20             bsW(1,1); nBlocksRandomised++;
21         } else
22             bsW(1,0);
23         moveToFrontCodeAndSend();
24     }
25 }
27 void moveToFrontCodeAndSend ( void )
28 {
29     bsPutIntVS ( 24, origPtr );
30     generateMTFValues();
31     sendMTFValues();
32 }
34 void sendMTFValues ( void )
35 {
36     /* generate coding tables */
37     /* compute MTF values for selectors */
38     ...
39     /* ... output data ... */
40 }
```

Figure 6.2.: Sequential programs are usually factored using the *procedural* programming paradigm. Nevertheless, programmers most often aim at following software engineering related properties of the code, such as modularity, rather than expressing parallelism. This figure shows a code excerpt from the SPEC *bzip2 compression* application. Inherently sequential code, I/O in this case, is intertwined in nested function calls, preventing the identification of parallelism in the outer-most loop.

the main loop of the *bzip2* compression algorithm. On an algorithmic level *bzip2* exhibits a three stage pipeline, namely *input*, *compression*, and *output*. However, as the shaded areas in figure 6.2 show the parts of the code that have to be executed sequentially are coupled with *compression*. This enforces nested functions like *generateMTFValues*, which can be executed in parallel for each data block, to execute sequentially. In addition, the *moveToFrontCodeAndSend* function that accounts for up to 36% of the total execution time, contains a significant amount of parallelism which belongs to the top loop-level of the *compressStream* function. Although aggressive function inlining could possibly resolve these issues, it leads to code bloat and other detrimental effects [25]. In contrast, our analyses operate on a partially folded whole-program representation that enables us to hierarchically and on-demand discover function calls which – if inlined – can lead to more profitable parallelisation. For the code in figure 6.2, *moveToFrontCodeAndSend* will be inlined only if it turns out to be the dominating stage of the pipeline. Similar to MP3 decoding example, this will uncover further parallelisation opportunities. For example, the call to function *generateMTFValues* can

be fused with *doReversibleTransformation* in a single scalable data-parallel stage.

In summary, our approach extracts a pipeline structure that a programmer would construct using a data-flow or streaming programming paradigm. For instance, *replication* of pipeline stages corresponds to *stateless filters* and *split-join* operations in the StreamIt programming language. Similarly, multi-level loop partitioning corresponds to multi-rate signal processing where filters operate at different input and output rates.

6.1.2. Contributions

Among the contributions of the work presented in this chapter are:

- We demonstrate that Ir-profiling is a powerful tool for the extraction and exploitation of parallelisation beyond loop level. Ir-profiling bridges the *information gap* between the low-level execution profile and the IR maintained within the parallelising compiler. This simple, yet effective *back annotation* capability distinguishes our approach from most other profiling methodologies.

- We develop a top-down approach for the extraction of processing pipelines from sequential applications. We exploit the power of a whole-program Ir, but avoid exposing overly detailed (and possibly redundant) dependence information to the pipeline extraction pass. Using an iterative, *selective unfolding* strategy we specifically target the levels of computationally intensive code regions that will yield exploitable parallelism.

- We extend conventional linear-pipeline parallelisation with two concepts borrowed from streaming languages, namely *multi-level pipelines* and *stage replication*. Additionally, we demonstrate how their combination can uncover further parallelisation opportunities.

6.2. Methodology

The parallelisation work-flow is illustrated in figure 6.3. The *sequential source program* written in C is processed by the CoSy *C* compiler and its *IR* is instrumented. The resulting *executable* is executed with one or more representative *input files*. This results in a set of *trace files* that is presented to our *trace analyser* for dependence analysis. The generated program dependence graph is passed on to the *partitioner* that will produce a *pipeline specification* with annotations relating to the original Ir. The CoSy compiler then processes the source program a second time. It generates parallel C code with calls to our *runtime system* according to the *pipeline specification*. Eventually this code is compiled for the target platform using the Gcc compiler and linked against the *pipeline runtime library*.

In subsection 6.2.1 we explain how we generate the program dependence graph from the trace files. The hierarchical top-down partitioning algorithm is presented in section

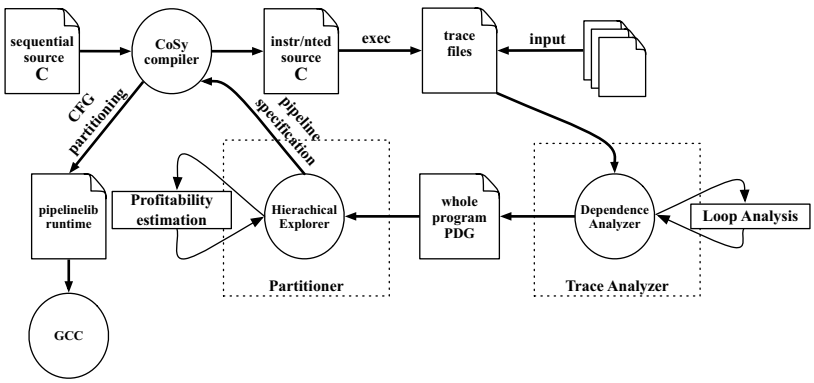


Figure 6.3.: Overview of the parallelisation work-flow.

6.2.2. Code-generation is covered in section 6.2.3. Pipeline stage *replication* and the generation of *multi-level pipelines* are part of this algorithm and are introduced in sections 6.2.4 and 6.2.5, respectively.

6.2.1. Program Dependence Graph

The Ir which is used to perform the hierarchical parallelism extraction is based on the *Program Dependence Graph* (Pdg) [35] with its explicit representation of both *control* and *data-dependences*. In contrast to control-flow representations which are tied to a fixed sequencing enforced mainly by syntactic properties of the source rather than the necessary ordering of operations enforced by the sequential model of computation. This increased expressiveness of the Pdg representation enables the partitioning algorithms to be simple and efficient. In our case the Pdg is derived from the middle-level control-flow representation which is augmented with data-dependence edges as computed by the *Dependence Analyser* (algorithm 4.1) based on profiling information. Pdgs, typically focus on lower-level Irs where each instruction or operand/operator form a separate node. Our choice to operate on a middle-level representation is motivated by the following factors: (i) it provides rich high-level information about the loop constructs and the data organisation, facilitating aggressive code and data transformations, (ii) code partitioning transformations can remain simple since we can still rely on the full set of compiler optimisation passes and (iii) the existing basic-block structure avoids overly aggressive coarsening during partitioning without compromising load-balancing.

Each node of the Pdg represents a *single-entry-multiple-exit* (Seme) list of Ir statements. In addition, using a preprocessing pass we isolate function calls and place them in separate nodes (pass *splitfunc* is presented in section 4.2.1). We define *compound* nodes to encapsulate the hierarchical structure of a program as any of the following: (i) the basic block (Bb) containing a function call site in addition to the maximum recursively defined set of Bbs that the specific function call includes, (ii) the set of Bbs belonging to a loop (loop-structure information is based on a variant of standard

control-flow graph analysis performed before the instrumentation pass) (iii) a *Strongly Connected Component* (Scc) containing other compound nodes or single Bbs.

The Pdg is initially computed for the whole program and then compound nodes are formed in a postorder fashion based on a whole program function/loop tree. At this point compound nodes consist of either functions or loops. The current implementation is based on the simple iterative algorithm from [6]. Each data-dependence edge in the Pdg is annotated with the following fields: (i) loop-carried bit mask that designates the loop-levels which carry the relevant dependences, (ii) loop-independent dependence bit mask, and (iii) mean size of the data communicated between the adjacent nodes per iteration at each level. A data-dependence is loop-independent when both the relative `def` and `use` have the same iteration vector. The Cdfg construction algorithm in 4.1 can be trivially extended to compute these since it already records the iteration vectors to compute loop-carried dependences.

In addition to true data-dependences that we have considered so far, pipeline parallelism requires handling some cases of false dependences that are not amenable to privatisation. This is due to the fact that Pdg partitioning might reorder nodes of the loop body and thus we have to ensure that producers do not get hoisted over a preceding producer of the same variable. At the same time we have to avoid introducing dependences between operations that operate on the same variable but have no producer/consumer relation. To distinguish these two cases we introduce a new term, that of *relevancy set*. More specifically:

Relevancy set Two nodes u and w are said to belong to the same *relevancy set* of a variable a if there is an undirected path of data-flow edge between u and w that carries a.

Effectively a relevancy set of a contains all the nodes that belong to the same undirected connected component in the data-flow graph of variable a. Nodes which do not belong to the same relevancy set of variable a are called non-relevant nodes regarding a. In abstract terms, the relevancy set represents the set of `def`/`use` operations that use the same variable a to communicate. Likewise, non-relevant nodes might operate on the same variable but do not communicate, and therefore we can later use privatisation to decouple them. Therefore, our goal is to enforce the original sequential order only between producers that belong in the same relevancy set.

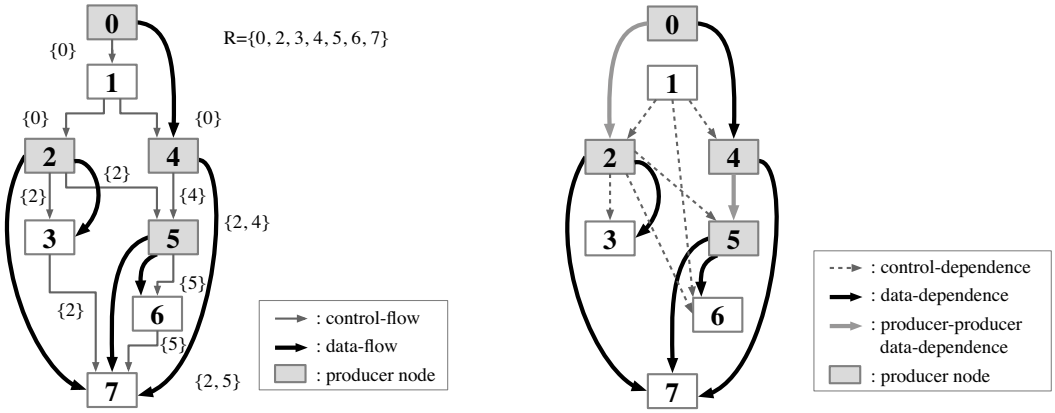
We achieve this by inserting additional data-dependence edges between producer nodes (i.e. nodes with an outgoing data-flow edge) of the same relevancy set when there is a path that connects them in the Cdfg. More specifically, we determine the edges that have to be added using a single-pass forward data-flow problem formulation. We present the formulation for a single variable and relevancy set but it can be easily extended to handle multiple variables in parallel. For each node u of the Cdfg and a given relevancy set R of variable a we first compute the set:

- $W[u] = u$, if there is a data-flow edge for a with its source in u, otherwise $W[u] = \varnothing$.

Using a reverse postorder traversal of the CDFG we compute the following data-flow equations for each node u:

- $\text{IMPR}_{in}[u] = \bigcup_{v \in pred[u]} \text{IMPR}_{out}[v]$.
- $\text{IMPR}_{out}[u] = if\ W[u] \neq \varnothing,\ then\ W[u],\ else\ \text{IMPR}_{in}[u]$

Finally, for all the producer nodes u in the relevancy set we add a data-flow edge from all the nodes in $\text{IMPR}_{in}[u]$ to u if there is no such data- or control-dependence edge. Figure 6.4 shows the computed equations for a sample CDFG graph and the resulting data-flow graph with the additional dependence edges[1].



(a) CDFG showing only the data-flow edges for a specific variable.

(b) PDG with the necessary producer-producer data-flow edges that prevent hoisting.

Figure 6.4.: An example showing how additional data-flow edges are added in the data-flow graph to enforce producer nodes of the same relevancy set R to execute in the order appearing in the sequential CFG. The sets in brackets show the progressive computation of the data-flow equations IMPR_in and IMPR_out from node 0 to node 7.

Note that the CDFG like the PDG consists of coarse-grain compound nodes and thus it is compact. Additionally, any unfolded loops can be considered a single folded node and thus can be processed independently since we never hoist any computation from nested loops out of their loop-body. In practice, these edges do not limit pipeline parallelism but just enforce the ordering of dependent writes that can be still assigned in different stages. Pipeline parallelism is primarily limited by loop-carried flow dependences that form SCCs which have to be scheduled in the same stage and not false dependences. Non-relevant reads and writes are handled using privatisation which is presented in section 6.2.3.2. In addition, observe that we did not limit producer nodes being hoisted over nodes that are only consumers in the same relevancy set. We show how to handle this case with a similar technique also in section 6.2.3.2.

[1]Note that a graph like the one in figure 6.4(a) is valid because in the case of a compound variable (e.g. array) two producers might write disjoint subsets of the variable, in which case the nodes have no direct producer/consumer relationship. However, the two producers might have the same consumer in the case that the latter reads from both subsets of the compound variable.

6.2.2. Top-Down Hierarchical Pipeline Stage Partitioning

Algorithm 6.1: Top-down parallelism selection.

Input
- L, F: loop and function set respectively
- $CTREE$: tree of compound nodes
- L_0: *virtual* top-most loop
- L_{doall}: profitable DOALL loops
- W_i : $\forall i \in CTREE$, profiled weight of i
- np: # of available cores

Result
- P_{doall}: selected DOALL loops
- P_{pipe}: selected pipelined loops

Data
- Q: work queue

```
1  Procedure top_down_parallelise
2  Q ← {L_0} ;
3  while Q ≠ ∅ do
4      c ← Q.poll();
5      if (c ∈ L_doall) ∧ (W_c > threshold_doall) then
6          add c in P_doall;
7      else if (c ∈ L) ∧ (W_c > threshold_pipe) then
8          (P, W_pipe) ←partition_loop(c, W_c, np);
9          if P ≠ ∅ ∧ W_c/W_pipe > threshold_speedup then
10             add c in P_pipe;
11         else
12             Q.add(children of c in CTREE);
13     else if (c ∈ F) ∧ (W_c > threshold) then
14         Q.add(children of c in CTREE);
```

Most applications that exhibit pipeline parallelism will only have a small number of dominating stages. When targeting CMPs with high a number of cores this factor will eventually limit the maximum attainable speedup and lead to poor utilisation of the available resources. This necessitates the exploitation of multiple domains of parallelism. More specifically, we consider (i) data-level parallelism as well as (ii) pipelines that span multiple loop-levels, and (iii) replication of *stateless* stages.

The parallelism selection algorithm operates on a partially folded PDG. Initially, the whole program is folded into a single function node. Algorithm 6.1 describes the iterative traversal of the IR and unfolding of compound nodes which results in the selection of the loops that contain profitable parallelism. At this stage, the algorithm unfolds only the compound nodes (loops or function calls) that are likely to expose additional exploitable parallelism. Currently, we use a simple heuristic based on fixed thresholds to determine profitability. Future work, however, can investigate more advanced and possibly machine-learning based strategies.

For sequential loops we use pipelining (line 8). The effectiveness of pipeline parallelisation is mainly determined by the execution time of the longest stage. Therefore, the *primary objective* of a pipeline partitioning algorithm is to *balance the load* across

Algorithm 6.2: Pipelined-loop partitioning.

Input
· l: the target loop
· W_l: profiled weight of loop l
· np: # of available cores
Result
· P: the set of partitions
· $W_{max} : \max_{P}\{PW_i\}$
Data
· Q: sorted descending list, sorted by $W_v, v \in PDG$
· P_i: set of nodes in PDG assigned to partition i
· PW_i: aggregate weight of partition P_i
· $RS[v]$: $v \rightarrow [1, np], v \in PDG$, > 1 if v is replicated

Procedure *partition_loop*(l, W_l, np)
$(DAG, W_{max}) \leftarrow$ preprocess_PDG();
$k \leftarrow 0$;
Q.add(roots of DAG);
while $(Q \neq \varnothing) \wedge (k < np)$ **do**
  $v \leftarrow$first in $Q : PW_k + W_v < W_{max} + \theta$;
  if $v == \varnothing$ **then**
    $k \leftarrow k + 1$;
    $PW_k \leftarrow 0$;
    continue ;
  else if $RS(v) > 1$ **then**
    add v in P_{k+1} .. $P_{k+RS(v)}$;
    $k \leftarrow k + RS(v) + 1$;
  else
    add v in P_k;
    $PW_k \leftarrow PW_k + W_v$;
  Q.add(ready children of v in DAG);
add remaining nodes of PDG in P_{np-1};
return $(\bigcup_{i \in [0,k-1]} P_i,\ max\{PW\})$;

Procedure *preprocess_PDG*(l);
set $RS(u) \leftarrow 1, \quad \forall u \in l$;
repeat
  insert_IMPR_dependences(PDG_l);
  $DAG \leftarrow$ compute_SCCs(PDG_l);
  augment_replicable_stages();
  peek $v \in DAG : W_v$ is max;
  update $W_v, \forall v \in DAG$;
  if v *is replicable* $\wedge$ $RS(u) < np - 2$ **then**
    $RS(v) \leftarrow RS(v) + 1$;
    $W_v \leftarrow W_v \cdot \frac{RS(v)-1}{RS(v)}$;
  else if $v \in F \bigcup L$ **then**
    PDG_l.unfold(v);
  else if v *is SCC* **then**
    peek $w \in SCC : W_w$ is max;
    PDG_l.unfold(w);
  else break ;
until $W_v \leqslant (W_l \div np) + \theta \ \vee \ ldepth(u) > threshold_{depth}$;
return (DAG, W_v);

the available cores. Our key observation is that balancing can be effective only if the granularity of each node is small relatively to the stage size of a theoretically optimally balanced pipeline that uses the maximum of the available cores, i.e., $W_l \div \#P$ (W_l the profiled weight of the loop). Unfortunately, the size of the schedulable nodes is inherently limited by the code structure (e.g. function, loop nests) and the dependences among them. We address this problem through (i) partitioning of multi-level loops, (ii) stage replication and (iii) function-splitting. Therefore, partitioning is performed in two steps. First, we preprocess the PDG performing the aforementioned transformations when applicable (*preprocess_PDG()* in algorithm 6.2). Next, we partition the resulting *Directed Acyclic Graph* (DAG) using a heuristic that effectively tries to balance the load given the number of available resources (*partition_loop()*).

The preprocessing step iterates over the PDG of the loop attempting to break the slowest compound node in finer-grain blocks of computation. At the beginning of each iteration, the additional dependence edges that enforce the sequential order between the producers of the same relevancy set are inserted using the analysis presented in the previous section (*insert_IMPR_dependences()*, line 23). Next, SCCs are collapsed, resulting in a DAG. This ensures that the algorithm processes chunks of computation that can be actually executed in parallel. In the case of a dominating stage that is replicable we eagerly augment it with nodes that preserve its replication property, i.e., without introducing new cycles in the PDG. This augmented stage is *replicated* multiple times till it is no longer the dominating stage of the pipeline (lines 28-30). Alternatively, if a loop or function dominates the loop body we opt to unfold it (lines 31-32) in order to generate finer-grain components using *multi-level loop partitioning* and *function-splitting*. Finally, in the case of a *Strongly Connected Components* (SCC) we seek to uncover a computation which is not part of the dependence cycle. Again, we select to unfold the compound node of the SCC with the highest weight (lines 33-35). This iterative process is repeated until the dominating stage is under the threshold $W_l \div \#P$ (line 37), since further partitioning will not increase the attainable speedup. In addition, we bound the unfolding of loops up to a depth $threshold_{depth}$ since partitioning deeply nested loops is limited by the synchronisation/communication overhead. For the applications covered in this thesis a loop depth of 5 was adequate to uncover all available exploitable parallelism.

In the second step, *partition_loop* partitions the PDG of a given loop in successive stages ($P_0..P_{np-1}$) by effectively following a topological ordering of the graph. Consequently, any inter-stage data dependences will be pointing to a successive stage, preserving the sequential semantics of loop execution. Nodes are processed using a descending sorted list, sorted by the profile weight of each node. In case of a tie, the node that minimises communication with the next stage is selected. A node is included in the current stage only if its addition will not result in exceeding the threshold W_{max} (line 6). The threshold is not $W_l \div \#P$ as before, but it is given as a parameter that equals the maximum node weight that derived from preprocessing. Thus, we avoid

eager partitioning – which subsequently leads to higher communication and context switching cost – when it does not decrease the execution time of the dominating stage. In addition, we use an slack factor θ to avoid excessive number of under-filled partitions.

Loop partitioning, along with the set of partitions P, also returns the weight of the dominating stage, which assists hierarchical parallelisation to determine whether it should add the current loop to the set of loops that can be effectively parallelised P_{pipe} (line 9 in algorithm 6.1) or proceed to inner-levels of the component hierarchy (line 12).

6.2.3. Parallel Code Generation and Runtime System

The code-generation process (figure 6.5) takes as input the partitioned pipeline specification and the original procedure-level CFGs. In order to handle any unfolded function nodes which are descendants of the loop to be partitioned, we utilise the inline transformation already present in our compiler. Parallel code-generation is then performed on the CFG for each procedure at the middle-level IR in the CoSy compiler.

Additionally, although the resulting partitioned CFGs are not optimal in any case (in terms of number of nodes, control instructions, complexity of data access expressions etc.), parallel-code generation is part of the middle-level passes of the compiler and thus it enables our approach to remain relatively simple and rely on subsequent intra-procedural optimisations to eliminate any inefficiencies.

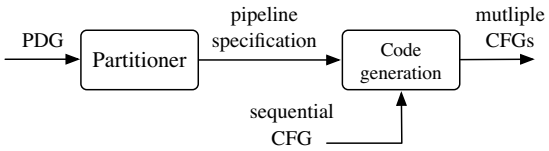


Figure 6.5.: Overview of the code-generation process.

The description of the code-generation process that follows is based on the example code in figure 6.6. On the left hand, the annotations refer to the contribution (in %) of each statement in relation to the whole program execution time. The initial CFG which is used for the profiling stage as well as parallelisation is shown in figure 6.7(a). Basic blocks with just one unconditional control statement (e.g. *bb4*, *bb5*) are place holders for loop control structures which are not present (e.g. test and increment block respectively) and are inserted by the initial control-flow analysis pass.

6.2.3.1. Control-Flow

First, algorithm 6.3 computes the set of BBs (V_s^a) of the original CFG which include the control instructions that determine the execution of the BBs in the current pipeline stage s (line 3). This *control-replication* set of BBs can be computed as the union of

Algorithm 6.3: Pipeline parallel-code generation.

Input
- CFG: original control-flow graph
- $PDOMT$: post-dominance tree of CFG
- PDG: program dependence graph
- N: number of pipeline stages
- $P_{desc}[N]$: pipeline stage descriptors
- V_s^a: set of BBs assigned to stage s

Result
- $CFG_{/\!/}[N]$: a (V_s, E_s) graph $\forall$ stage $s \in P_{desc}$

Data
- V_s^r: control-replication BBs of stage s
- $outctrl_s$: predicate variables assigned in stage s

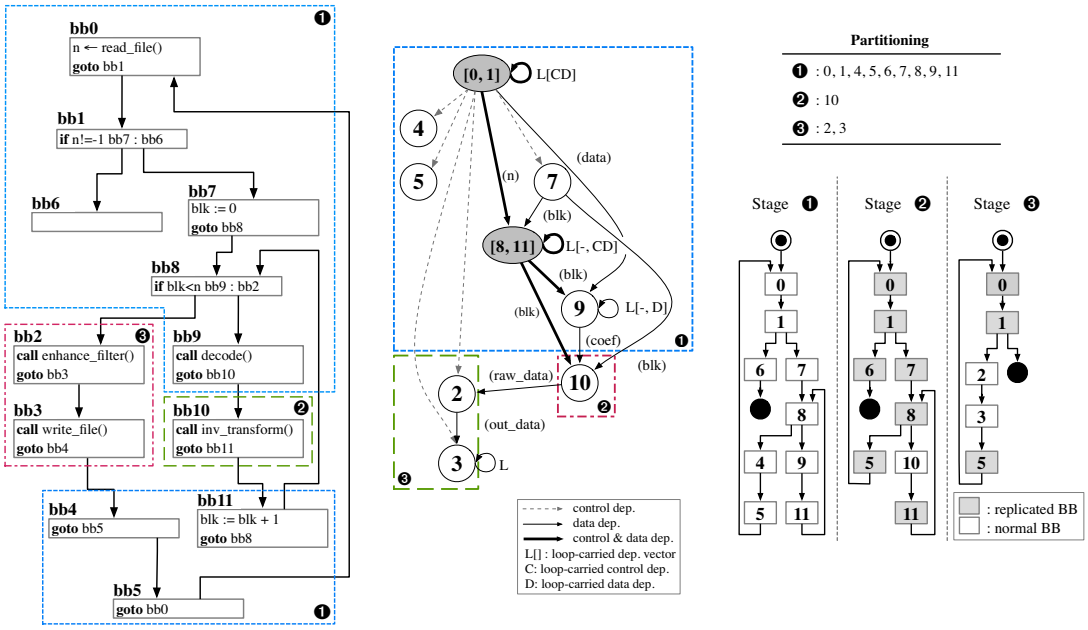
```
1  Procedure pipeline_loop:
2  foreach s ∈ P_desc[N] do
3      V_s^r = compute_control_replication_BB(PDG, s);
4      V_s = V_s^r ⋃ V_s^a;
5      E_s = {e(u,v) : u,v ∈ V_s};
6      foreach e(u,v) : u ∈ V_s ∧ v ∉ V_s do
7          v' = deepest ancestor of v in PDOMT : v' ∈ V_s;
8          E_s = E_s ⋃ {e'(u,v')};
9      CFG_//[s] = outline_CFG_subgraph(CFG, V_s, E_s);
10     outctrl_s = capture_outgoing_control_flow(CFG_//[s]);
11     fix_local_data_references(CFG_//[s], s);
12     fix_global_data_references(CFG_//[s], s);
13     insert_pipeline_dataflow(CFG_//[s], outctrl_s, s);
14     init_copyin_data(CFG_//[s], outctrl_s, s);
15     init_copyout_data(CFG_//[s], outctrl_s, s);
16 insert_pipeline_initialisation_finalisation(CFG);
```

the vertices in Pdg that are backwards-reachable from any vertex in s using control-dependence edges only. In figure 6.7, for stage 2 this is $V_2^r\{bb1, bb8\}$. Then, we create a new function with a subgraph of the original Cfg that includes both the blocks assigned to stage s by the partitioner V_s^a and the control-replication blocks V_s^r(line 10). We also replicate the *pre-header*, *loop-header* and the *loop-increment* blocks (e.g. $bb7$, $bb0$ and $bb5$ respectively) which are later used to inject the pipeline communication code. Initially, replicated Bbs contain only the control instruction of the original Bb. The subgraph includes every edge connecting vertices in V_s, but also edges that substitute dangling outgoing edges. We redirect these edges to the deepest ancestor of the missing vertex in the post-dominance tree of the original graph (lines 6-8). For instance, in stage 3 of the previous example the edge $(bb1, bb7)$ is replaced by edge $(bb1, bb2)$ as shown in figure 6.7(c).

The final step is to add Ir statements that capture the outcome of the conditional-control statements of Bbs in V_s^a which are *control-replication* Bbs for any of the subsequent stages (line 11). For each such conditional-control statement we define a new local variable, *predicate*, which is assigned the value of the conditional expression. For example, this is shown for the predicate variables *pred1* and *pred2* in figure 6.11. This transformation enables handling control-dependences with a single, uniform pipeline

```
 5%  1  while((n = read_file(inf, data)) != EOF) {
     2    for (blk=0; blk<n; blk++) {
20%  3      coef[blk] = decode(data, blk);
50%  4      raw_data[blk] = inv_transform(coef, blk);
     5    }
20%  6    out_data = enhance_filter(raw_data);
 5%  7    write_file(outf, out_data);
     8  } /* while */
```

Figure 6.6.: Source code for the example in figure 6.7



(a) sequential CFG **(b)** PDG with folded SCCs **(c)** partitioned CFGs

Figure 6.7.: A simplified example based on the source of figure 6.6, demonstrating the intermediate representations used for partitioning and parallel-code generation.

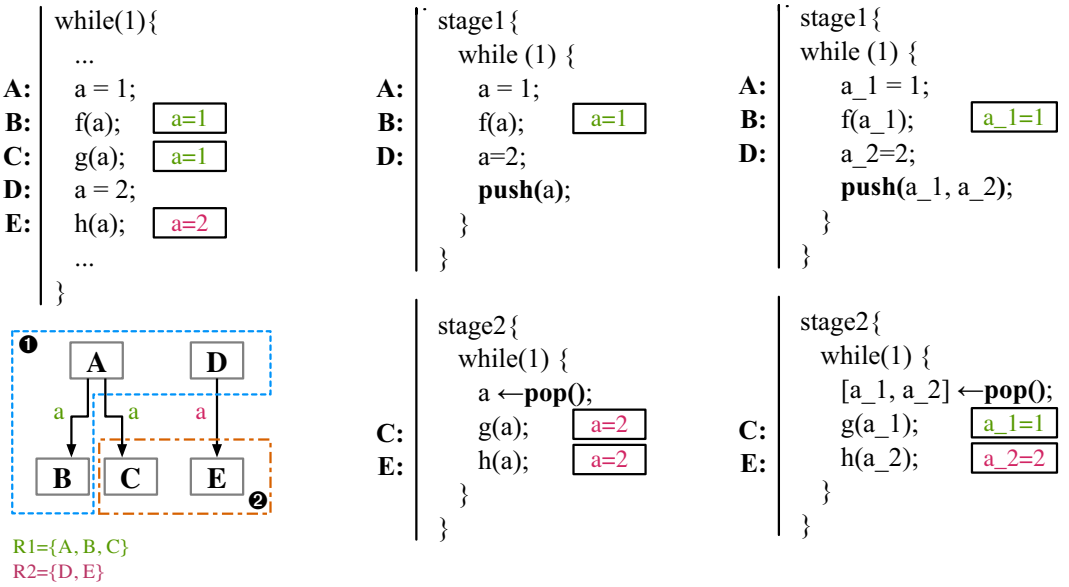
data-flow mechanism (line 14).

6.2.3.2. False Dependences

Flow-dependences are derived from the data-dependence edges of the PDG that connect two nodes belonging in different pipeline stages. The data-flow mechanism that enforces these dependences is based on explicit communication and is described in detail in section 6.2.3.4. Besides flow-dependences, pipeline code-generation should satisfy any false dependences too. These can be grouped based on their properties in the following classes:

i. data intra-stage anti- and output-dependences in a sequential stage,
ii. data inter-stage anti- and output-dependences, and
iii. data loop-carried intra-stage anti- and output-dependences in a replicated stage.

Intra-stage false dependences when the stage is not replicated, class (i), are implicitly satisfied by virtue of the CFG partitioning algorithm. More specifically, the construction of the partitioned CFGs (procedure *outline_CFG_subgraph()*) follows the control-flow of the original sequential CFG. Therefore, an intra-stage anti-dependence will never be inverted in the resulting CFGs (figure 6.7(c)).



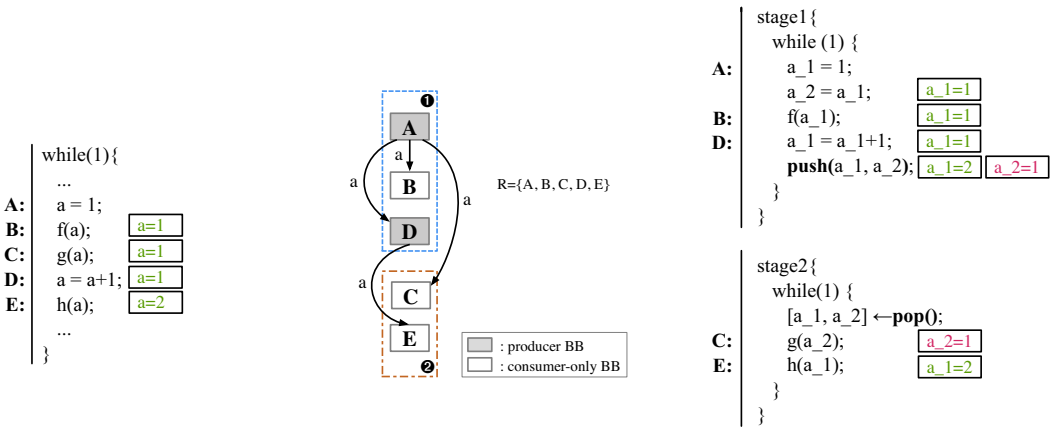
(a) Source code and a valid partitioning of the resulting PDG.

(b) Wrong value communication due to interfering relevancy sets $R1$ and $R2$.

(c) Code-generation using separate thread-private variables.

Figure 6.8.: An example demonstrating the necessity of handling interfering relevancy sets and the technique of *renaming* that overcomes this problem. The boxes show the value of the variable read in the respective BB.

In section 6.2.2 we presented how the sequential order of false dependences between producers in the same *relevancy set* can be preserved across multiple stages by inserting explicit dependence edges. Non-relevant operations in the same variable, on the other hand, can be handled with privatisation since they represent independent computation streams. This is better demonstrated by the example in figure 6.8. It is clear that if statement (C) is placed in a stage after the stage of statement (D) like in figure 6.8(b), the data flow mechanism will communicate the updated value rather than the one from statement (A). This is an artefact of the communication mechanism that transfers data only at the beginning and the end of an iteration as we will see in section 6.2.3.4. Transferring data after the source and before the sink of the dependence resolves this problem but it leads to fine-grain communication and therefore we did not follow this approach. Instead we create separate thread-private variables which are used in the two disjoint relevancy sets, $R_1 = A, B, C$ and $R_2 = D, E$. This is necessary only if (i) two or more nodes of a single pipeline stage belong to vertex-disjoint relevancy sets of the same variable, and (ii) one of the nodes in these sets is scheduled in another stage. The latter is required because otherwise there is no need for duplicating variables in a



(a) Source code.

(b) A valid partitioning of the PDG. Producer node D is hoisted over node C.

(c) Code-generation using an intermediate buffer.

Figure 6.9.: An example of using an intermediate buffer to handle the case when a producer node is hoisted over a consumer-only node in the same relevancy set. The boxes show the value of the variables read in the respective BB.

single stage where the control-flow is identical to the sequential one (by virtue of the CFG partitioning algorithm).

In fact, this technique resembles *register renaming* in OoO microprocessors where it is utilised to resolve false dependences in architectural registers and thus increase the extracted ILP. Similarly, compilers compute intersected Def-Use chains (*webs*) that represent connected live ranges and use this representation to perform register allocation. Figure 6.8(c) illustrates the transformed program with the renamed variables.

Finally, we create an additional thread-private variable for those stages that have nodes with no exposed reads or writes, i.e., no incoming or outgoing data-flow edges for the specific variable. This is necessary because these nodes can be freely reordered and might perform write operations that conflict with writes of a relevancy set that is scheduled in the same stage. The mechanism used for privatisation is presented in detail in section 6.2.3.3.

The second case that requires privatisation is when a producer or consumer node is hoisted over a consumer-only node of the same relevancy set. Remember that we have only enforced the original ordering for producer nodes. To handle this case we create a additional thread-private instance of this variable only for the specific consumer node. This is necessary only if (i) a producer or consumer node w in a relevancy set R is scheduled in a stage before a consumer-only node u also in R, and (ii) u is a predecessor of w in the CFG. The difference with the previous case of non-relevant nodes is that we have to save the intermediate result at the end of the producer node. Then its value will be communicated using the same uniform data-flow mechanism that we use for all variables. Also observe that this applies for a hoisted consumer node because it might still perform conflicting write operations that are not communicated to any

further nodes.

Figure 6.9 demonstrates this technique for an example of a single variable. In the PDG partitioning in figure 6.9(b) producer D is hoisted over C which is legal since C is a consumer-only node and thus we have not inserted a data-flow edge from C to D. Since C is a predecessor of D in the control-flow the result of producer A is saved after its execution in the separate thread-private variable a_2. Finally, node C in stage 2 is consuming the communicated value of a_2. Node E – the other node of the relevancy set R which is in stage 2 – is consuming the communicated value from the default thread-private variable a_1 which is assigned to every other node of R in stage 1. Although this analysis is necessary to ensure correctness under aggressive reordering, in practice producer/consumer nodes in the same relevancy set form a sequential pipeline and the creation of additional buffers was not necessary in any of the program that we have considered.

Besides the false dependences that involve data communicated from/to another stage, the rest require simple privatisation. We determine these dependences using a post-pass on the IR-instruction stream. This process can be performed efficiently given that it is performed when the code partitioning is fixed and tracking dependences in stage granularity is sufficient. More specifically, for false dependences in class (ii) for each memory location a we record:

1. $W_a[]$ bitset of size N where $W_a[s]$ is set if stage s has executed a `def` for memory location a (N the number of pipeline stages).
2. $R_a[]$ bitset of size N where $R_a[s]$ is set if stage s has executed a `use` for memory location a.

Then we determine if there is an inter-stage false dependence for a memory location at analysis time as follows. If a `def`(a) instruction is executed in stage s, there are two non-trivial cases:

1. If $(W_a[:] \neq 0 \wedge W_a[:] \neq (1 \ll s))$ then there is an inter-stage output-dependence (WAW) between s and the stages that correspond to the non-zero bits of the value W_a.
2. If $(W_a[:] \neq 0 \wedge R_a[:] \neq (1 \ll s))$ then there is an inter-stage anti-dependence (WAR) between s and the stages that correspond to the non-zero bits of the value R_a.

In practice we can relax both tests and perform the following test at the end of the analysis.

1. If W_a has more than one bit set then there is an inter-stage output-dependence (WAW) between the stages that correspond to the non-zero bits of W_a.
2. Let $RW[:] = R_a[:] \mid W_a[:]$. If RW has more than one bits set then there is an inter-stage anti-dependence (WAR) between the stages that correspond to the non-zero bits of RW.

This might result to false-positive anti-dependences which are in fact flow-dependences or output-dependences. Since we are privatising these variables anyway this overestimation is sufficient. An alternative approach is to compute the read and write address

range for each stage but the technique presented was significantly more efficient for some programs that we have instrumented. In addition note that we do not have to distinguish between loop-carried and loop-independent dependences or determine their direction.

Dependences in class (iii) apply only in pipeline stages that are replicated. Let these stages be $M \in [0, N]$ in number. For each memory location a we record:

1. $R_{iter_a}[]$ array of size M where $R_{iter_a}[s]$ is the normalised iteration number of the pipelined-loop when the first `use` for location a was executed in stage s. Otherwise it is negative.
2. $W_{iter_a}[]$ array of size M where $W_{iter_a}[s]$ is the normalised iteration number of the pipelined-loop when the first `def` for location a was executed in stage s. Otherwise it is negative.

Then we determine if there is a loop-carried intra-stage false dependence for a memory location as follows. If a $\texttt{def}(a)$ instruction is executed in stage s at iteration number k, there are two non-trivial cases:

1. If $(W_{iter_a}[s] > 0 \wedge W_{iter_a}[s] \neq k)$ then there is a loop-carried intra-stage output-dependence (WAW) in stage s.
2. If $(R_{iter_a}[s] > 0 \wedge R_{iter_a}[s] \neq k)$ then there is a loop-carried inter-stage anti-dependence (WAR) in stage s.

Since we are always privatising whole objects we can stop the analysis of class (iii) for the respective address ranges as soon as the first false dependence is observed for an object.

6.2.3.3. Privatisation

Based on the analyses of section 6.2.3.2 we determine which variables should be privatised to preserve correctness. The paragraphs that follow describe the privatisation mechanism for the case of local and global variables.

Local Variables Local variables are privatised by default since they execute in the context of a new thread stack. In line 11 of algorithm 6.3 any references to local[2] variables or arguments of the original function are modified to reference thread-private data. For every thread-private variable referenced in the body of an outlined stage function a new local, stack-allocated variable is defined. Scalars are allocated with the same type of the original variable. Aggregate-typed variables (e.g. structures and arrays) are transformed to *pointer-to-original-type* variables in order to facilitate *copy-by-reference* initialisation and communication. Initialisation by reference can reuse the original object of the sequential thread but this is only legal in the following two cases:

[2]Terminology regarding variable or object visibility, storage, etc. is based on the language-portable IR of the compiler and not the one of ISO *C*. Nevertheless, the usage of terms should be clear from the context.

Function name	% reduction in accesses loads	stores
qSort3	20.82	3.59
sortIt	50.41	20.88
simpleSort	15.91	4.77
fullGtu	1.21	3.59
Total for compression thread	24.35	2.52

Table 6.1.: Comparison of using *thread-local storage* for privatising global and static variables, compared to a *thread-id indexed array* scheme. The table shows the reduction in dynamic memory accesses for the dominating functions of *bzip2* compression algorithm.

(i) a shared aggregate local variable, or (ii) a thread-private copy of the first pipeline thread that accesses a private aggregate object.

Global Variables References to global variables (line 12) are more complicated to handle since all threads share the same global naming scope. For instance, if a function is called from more than one stages, changing the accesses of the global variable to reference the thread-private copy is not an option. Likewise, supporting multiple thread-private instances of a variable to avoid conflicting relevancy sets is not possible. We first discuss the case of aggregate types. Global aggregate variables are substituted with *pointer-to-original-type* variables. This mechanism has the benefit that it avoids allocating and initialising memory in Bss for threads that do not access a specific variable. Therefore the only wasted space is of the size of a pointer and not that of the original type. Accessing thread-private data using the same global symbol from all threads can be implemented in two alternative ways. The first is following the popular parallel programming paradigm of using an *array-of-original-type*, indexed by the unique thread ID. A second option is to use *Thread-Local Storage* variables (i.e. variables declared with the qualifier __thread) [146]. This feature implements a platform-specific and optimised mechanism to transparently access thread-private data using the same symbol name. Evaluation of these two options led to the conclusion to adopt the latter, although not purely portable, because thread ID indexing results in significant performance degradation. This was particularly significant in applications like *bzip2* (see table 6.1) that access global variables repetitively in computationally intensive code.

Further investigation revealed that this was due to a combination of poor common subexpression elimination and increased register pressure. The latter is in fact overemphasised in architectures like *x86_64* which have a limited amount of architectural registers. Thread-local storage, on the other hand, does not suffer from this side-effect because it utilises a free segment register and fixed offsets to perform each access to private data. Nevertheless, we retain thread-id indexed arrays in order to provide a

mechanism to access private data before/after thread creation/termination. Finally, any local variables of *static* storage duration are handled similarly to globals.

For global scalar variables we utilise thread-local storage too. The difference with aggregate variables is that we preserve the original type. This way we avoid using indirection and switching between multiple thread-private instances can be performed with a single scalar assignment of the value stored in the thread-id indexed array.

6.2.3.4. Data-flow

Pipeline data-flow is based on the communication primitives of the runtime system (*libpipeline*). Each pipeline thread is connected to each immediate predecessor and successor using two *single-producer-single-consumer* (SPSC) queues, one outgoing for produced data and one incoming for recycling consumed buffers. Queue operations (*push()*, *pop()*) communicate fixed-sized pointers, which effectively point to structures that contain the communication data.

Before continuing, we introduce a few clarifying definitions:

1. *Loop backedge* (x, y) is an edge where y dominates x.
2. *loop header* is a node y that dominates all the nodes in the loop.
3. *Body BBs* are all the predecessors of x that can reach y without traversing (x, y) and do not belong to test or increment BBs do not belong to.
4. *Test BBs* is a *single-entry-multiple-exit* (SEME) subgraph containing all BBs with an exit to either a loop exit or a body BB.
5. *Increment BBs* is a *single-entry-single-exit* (SESE) subgraph which exits to test BB only and it contains any loop-variable update.

Body, test and increment subgraphs are vertex disjoint. These blocks are identified or created by an early loop analysis performed after the front-end. The header of the loop might be either a body or test BB (*repeat−until* or *while−do* loops respectively).

Inter-stage dependences are satisfied at the loop iteration boundaries. Stages with a predecessor pop a context buffer and restore its contents at the beginning of the *loop-header* (e.g. *bb*0 for the outer loop in figure 6.7(a)). Similarly, stages with a successor store the outgoing data in a buffer structure and push them in the outgoing queue at the end of the *loop-increment* SESE (e.g. *bb*11 for the inner loop). Scalar variables are copied-by-value in the context buffer. Aggregate-typed and dynamically allocated objects are at the producer side using memory copying (e.g. `memcpy()`). When data-flow analysis shows that there is no intra-stage loop-carried dependence for an aggregate variable copy-by-reference can be used. However, in the case that data are pushed further to subsequent stages we have to establish additional pointer recycling queues between non-adjacent stages. In addition, in multi-core architectures that support cache-bypassing stores this migratory access pattern does not necessarily lead to better cache behaviour than memory copying. Therefore, we opted to not implement this scheme in our compiler. At the consumer side scalar variables are updated using the

value stored in the buffer. Aggregates variables have been substituted by pointer-to-original-type (section 6.2.3.3 and just require updating the respective pointer to refer to the object of the buffer.

Memory copying can exploit streaming SIMD extensions, available to all modern multi-cores (e.g. Intel's SSE and IBM's ALTIVEC) to minimise the copying delay. Additionally, since the producer is not going to access these data soon we evaluated the use of non-temporal store instructions (e.g. *movntdq* for Intel's MMX). These hints can effectively minimise cache pollution, avoiding allocation of cache lines for data not in the cache and bypassing the cache-hierarchy for memory stores. In addition, these instructions are implemented using *write combining* memory operations, leading to more efficient bus bandwidth utilisation. Nevertheless, this might not be beneficial in the case of communicating threads that share higher-cache levels.

6.2.3.5. "Copy in" and "Copy out" Data

In addition to data communicated through inter-stage dependences, code-generation has to handle the initialisation/finalisation of *"copy in"* and *"copy out"* variables, respectively (lines 14-15). We qualify as *"copy in"* (similar in semantics to OPENMP's *copyin* clause) any thread-private variable which needs to be initialised with the value of the *sequential* thread. *"Copy in"* variables are identified by incoming data-dependence edges that have their source outside the loop body. A variable that is *"copy in"* in multiple stages, but it is also part of an inter-stage dependence is initialised only in the earliest stage. Pipeline data-flow ensures that later stages will receive the initialised copies.

On the other hand, thread-private variables are qualified as *"copy out"* if there is a definition in the loop body that reaches an external use. Handling "copy out" variables is more complicated, though. Variables that have been initialised with the copy of the sequential thread by reference are implicitly copied out and no further action is necessary. Variables which are private to more than one stage are finalised by the sequential thread using the array of private copies where each thread has stored a reference to the buffer it used before terminating. If the first stage has many BBs pointing to a loop-exit, the last modifying stage has to be identified at runtime. In order to determine the loop-exit that terminated the loop we use a special variable private to each thread and a mechanism analogous to the one described earlier for control-replication blocks.

6.2.3.6. Pointer Disambiguation

The allocation of thread-private pointer variables is handled just like any other variable, scalar or aggregate. Initialisation of pointers that might point to thread-private data, though, is more complicated. Each thread is consuming multiple incoming buffers which in turn contain privatised copies of data. Pointers to these data must be initialised

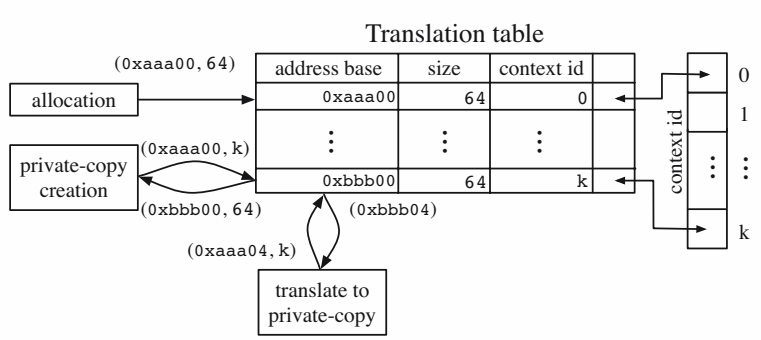


Figure 6.10.: To disambiguate pointers that point to privatised data we utilise a translation table like the one depicted here. The original objects of the sequential thread is stored at address `0xaaa00` and is associated with *context-id* 0. Thread-private copies are registered when they are created in the pipeline initialisation section. Each one gets assigned a unique *context-id* `k`. Address `0xaaa04` is translated to the corresponding address for the private-copy of context `k`, which is `0xbbb00`+`0x04`=`0xbbb04`.

appropriately. In principle, in the presence of recursive data-structures (e.g. linked lists) this requires either scanning and updating pointers through the whole structure or patching each dynamic pointer access. *Copy-or-discard* proposed by Tian et al. [142] is following the latter approach which introduces a significant overhead at every pointer access[3]. Fortunately, in practice the majority of streaming applications use simpler structures. Therefore, we opted for a simple, yet effective mechanism that dynamically disambiguates and translates pointers to *thread-private-data*.

For each privatised object we register its original address in the context of the sequential thread in a *translation table*[4]. In the case of objects with static or automatic storage duration we emit explicit calls at the pipeline initialisation section. For heap-allocated objects, we intercept any memory allocation calls (*malloc()*, etc.). At inter-stage buffer allocation each buffer is assigned a sequence number, *context-id*, and each object copy is associated with this ID and its (dynamic or static) size in bytes (figure 6.10). Effectively, this establishes a notion of alternative thread-private contexts. Each object as accessed by the initial context of the sequential thread is associated with the special *zero* context-id. Then, at pipeline runtime when a buffer is popped out of the queue every pointer to *"copy in"* or *thread-private* data is updated based on its current value and the context-id of the incoming buffer. The table in figure 6.10 stores both the original copy of the sequential thread and the thread-private copies used in the buffers. Let a translation query request the equivalent of address $addr_0$ that corresponds to the copy with context-id *id* (a tuple $(addr_0, id)$). The table is looked up with $addr_0$ and returns the line that contains the base of $addr_0$, $base_0$. The *offset* is computed by subtracting the base from *addr* ($offset=addr_0-base_0$). Then using the pointer table that

[3] In *C* programs arrays are passed-by-reference, i.e., translated to pointers. This results to most of the accesses in nested functions to be pointer rather than array accesses.
[4] The translation table is practically implemented using a search tree rather than a table.

is indexed by the context-ids – depicted on the right side of figure 6.10 – the base $base_{id}$ corresponding to context *id* is retrieved. Finally, the requested pointer is computed as $base_{id}+offset$.

6.2.4. Stage Replication

Pipeline stages that have no loop-carried intra-stage dependences are *stateless* and, thus, can be *replicated* in order to further reduce the critical path of the pipeline. This kind of parallelism is highly beneficial for applications that contain a single dominating stage that forms a performance bottleneck.

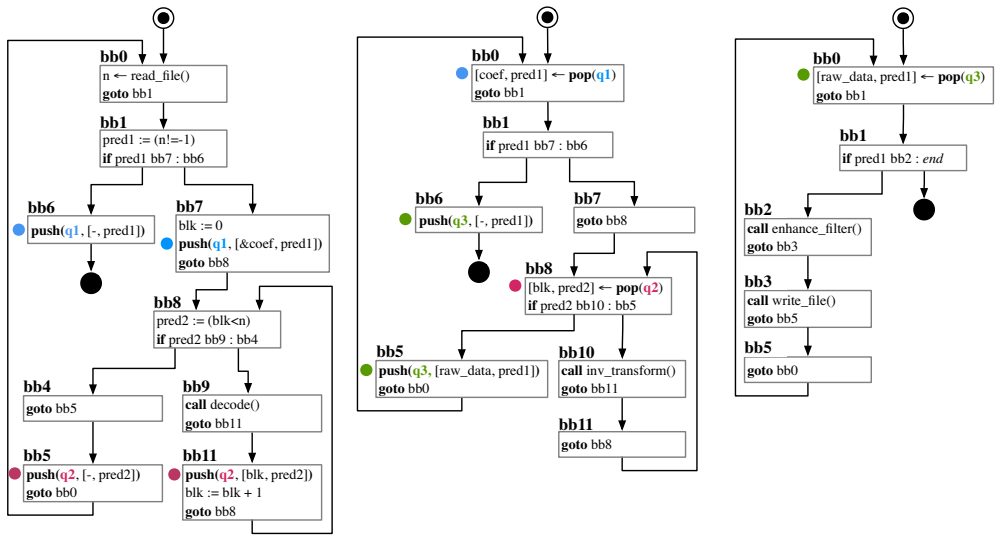
In order to process multiple incoming buffers in parallel our framework spawns multiple threads. Data that carry any loop-carried false dependences within the replicated stage have to be privatised to facilitate independent execution. Incoming buffers can be processed *Out-of-Order* (OoO) by the thread pool since its output is either completely independent or it will be explicitly pushed to the next stage through the outgoing buffers. It is important to stress at this point that the OoO property creates further opportunities regarding the work distribution policy, i.e., the policy that specifies which thread will consume each incoming buffer. Thus, in the case that different iterations of a stage show great variation in processing time we have the flexibility to employ a load balancing scheme. In this study we employ a simple *round-robin* policy that has the benefit of *lock-free single-producer-single-consumer* queues.

Detecting parallel stages using the PDG is relatively straightforward. Using the loop-carried dependence bit sets precomputed in the profiling stage we can mask out data dependences that manifest only across inner loop levels. Computing the additional data that have to be privatised to eliminate any false dependences can be deferred until the code-generation phase. Stage replication, however, is invalid in the case of stages with loop exits.

Although stage replication essentially diverges from a purely linear pipeline paradigm it is straightforward to extend our code-generation and runtime strategy to handle it. We extended *libpipeline* runtime system with the following primitives: (i) *push* a buffer to one of the multiple out-going queues, the selection is based on a callback (current implementation supports only static balancing), and (ii) *pop* a buffer from on of the multiple queues of the previous stage.

6.2.5. Multi-Level Pipelines

We have extended our analyses to handle partitions that span more that one loop-level. Effectively, this coalesces multiple pipelines that operate at different rates into a single linear pipeline. It is important to note that the choice of a PDG based representation, as opposed to a control-flow based one, enables us to handle the rather complicated control of nested loops in a uniform and transparent way. Therefore, the partitioning algorithm can proceed and distribute the internal nodes of an unfolded inner loop level to different



(a) stage ❶ **(b)** stage ❷ **(c)** stage ❸

Figure 6.11.: This figure demonstrates the insertion of data and control-dependence communication using the primitives of *libpipeline*. The partitioned CFGs are from the example in figure 6.7(c). Matching pairs of *push()* and *pop()* calls in adjacent stages are highlighted with the same color. Communication calls that recycle the consumed buffers are omitted.

partitions and largely disregards the code-generation intricacies. Then, as part of the $PDG \rightarrow CFG$ transformation, the unfolded loops are processed in a bottom up order. The replication of the additional control dependences that determine the execution of the nodes of nested loops result in the replication of the control-flow of the inner-loop to multiple threads, similar to single-level partitioning. Data communication is handled in the same fashion, too.

For instance, in figure 6.7 we process the inner FOR loop first. In this case *bb*9, *bb*10 are control dependent to the header of the nested loop *bb*8. This BB is included in the resulting partitioned CFGs of both stage 1 and 2, figure 6.7(c). The result of the *if* conditional is captured in the predicate variable *pred2* which is communicated either in the increment block of the inner loop (*bb*11). Data-communication code which is relevant to the inner-loop only (e.g. *blk*) is injected in the beginning of *bb*11 for stage 1 (outgoing) and *bb*8 for stage 2 (incoming), figure 6.11. Next, the outer loop is processed. Communication regarding dependences of the outer loop should be inserted at the end (before the control statement) of the pre-header of the inner loop (*bb7*). This is necessary to enable the execution of stage 2 to proceed to the header of the inner loop. In the example of figure 6.11 the predicate *pred1* is pushed in *bb7* which is associated with the *if* conditional in *bb1*. The matching communication is inserted like in the single-loop case in the header of the outer loop in stage 2 which is *bb0*. Stage 3 receives only the predicate *pred1* since none of its nodes is control-dependent to *bb8*. This is pushed in queue *q3* along with the data-dependences (*raw_data*) at the end of the iteration in stage 2 which is the beginning of the loop-increment BB *bb5*. Note that

this way we preserve any data dependences that are due to `def` statements in the rest of the loop body. In stage 1 the communication was necessary to be inserted in the pre-header of the inner loop to allow the execution of the inner loop in stage 2. In the case that there were any dependences with their source after the inner loop in stage 1 it would be necessary to communicate them at the end of the outer loop, i.e., in *bb5*. Another interesting point is that the variable *coef* is communicated out of the loop instead of being communicated at each iteration. This is permitted only if there are no intra-stage loop-carried false dependences for a variable, hence privatisation is not required. Therefore, we only communicate the reference to the variable and the threads modify it in parallel. Although this is rarely true for the outermost loop, in nested loops it is common and we exploit this to reduce the communication cost. Finally, note that the predicate values have to be communicated in both the left and right path of the conditional to ensure termination of the relevant loops. So, for instance, in *pred1* is pushed in stage 2 both in *bb5* and *bb6*.

6.2.6. Safety

The use of profiling for dependence analysis *cannot guarantee safety*. Unlike static analysis that reasons about *all possible* program execution paths profile-driven analysis is limited to a small number of paths and, hence, might miss data or control dependences. We therefore expect the user to perform the final verification of the suggested partitioning scheme.

In this work we have taken the following steps to verify the correctness of the generated parallel code. First, we have run a number of tests with both the sequential and the parallel versions of the programs on different input data sets and compared their outputs. Second, we have manually inspected the generated code. In this second step we have been guided by our tools that generate an additional, graphical pipeline diagram highlighting the data items and code regions where static and dynamic dependence information differs.

We are fully aware that such a manual verification process is not scalable and envisage a scenario whereby dynamic checks for dependence violations are inserted into the generated code. These checks may be implemented in the underlying pipeline library or be based on additional hardware [148]. This is, however, beyond the scope of this thesis.

6.3. Empirical Evaluation

We evaluated our methodology on *M1*, a shared memory system comprising two quad-core Intel Xeon processors. The configuration of the target platform is given in table 3.8.

Benchmark	LOC	dataset	Pipeline techniques			#cores	speedup
			replicate	multi-level	func. split		
mp3	20K	128Kb/s cbr stereo	✓	✓	–	6	3.52x
bzip2	5K	64MB program	✓	–	✓	8	4.70x
mpeg2dec	23K	375 704x576 frames	–	✓	✓	3	2.68x
cjpeg	22K	4096x4096x24bit bmp	–	✓	✓	2	1.47x

Table 6.2.: List of the benchmarks used for evaluation and their main characteristics. For all the benchmarks more than one of the techniques that we introduced were necessary to achieve reasonable parallel speedups. The last column reports the attainable speedup in *M1* (table 3.8) which is equipped with 8 cores in total. Note that some applications utilise less than 8 cores since further partitioning would only increase the communication overhead without reducing the execution time of the slowest stage.

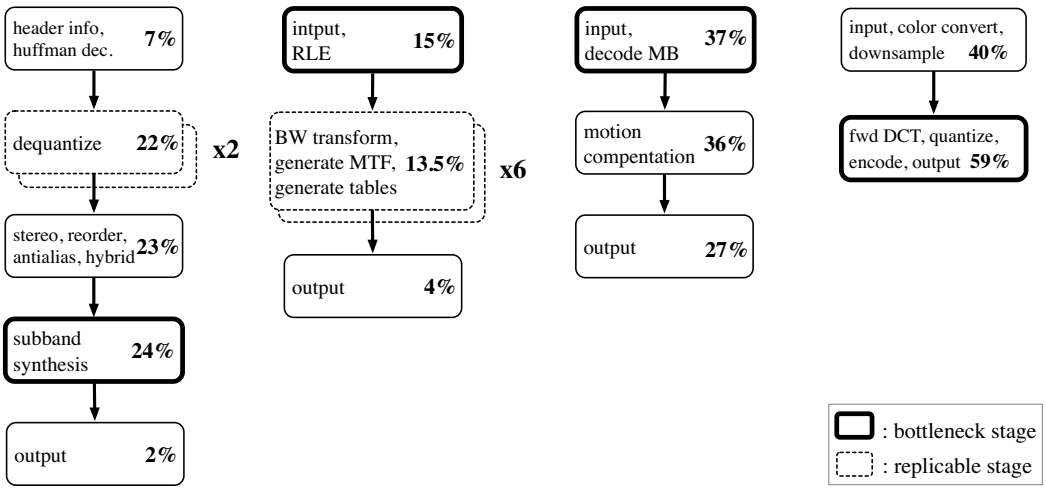
The speedup figures presented in the following paragraphs have been computed as the arithmetic mean over 10 executions. Machine *M1* was idle before the experiments start and besides the absolutely necessary system services no other programs were executing during the experiments. Additionally, any I/O performed has been exclusively to local filesystems in order to reduce variation introduced by the cluster's network filesystem. The relative standard error of the mean was less that 0.5% for all the experiments. This is primarily due to fact that the proposed parallelisation strategy yields static work partitioning and scheduling, i.e., it results in exactly the same data buffers being processed by the same threads in the same order[5]. Also note that replicated stages although in principle can process incoming buffers Out-of-Order the current prototype is using a simple *round-robin* scheduling policy to simplify the runtime system and minimise the synchronisation overhead (see section 6.2.4). Finally, application threads were bound to a single-core each and without Cpu time-sharing among them[6].

6.3.1. Performance

For our evaluation we have chosen four non-trivial benchmark applications from the Eembc and Spec benchmark suites with up to 23,000 lines of code. The details of these benchmarks are shown in table 6.2. While there are more programs in the two benchmark suites that are amendable to pipeline extraction we are mainly interested in multimedia and stream processing and, hence, we have restricted ourselves to those programs which are most representative of this application domain. In the following paragraphs we provide an in-depth discussion of the four benchmark applications, their parallelisation and the resulting performance.

[5]Static parallelisation or work partitioning does not necessarily result to deterministic parallel execution. The execution of the application threads might still result to significantly different overlapping of computation/communication, unavoidably leading to further variation in the architectural events (e.g. consistency protocol events) seen by each thread.

[6]This was implemented by utilising the thread-to-core affinity feature available in Linux and more specifically the `sched_setaffinity()` interface.



(a) MP3 decoding **(b)** bzip2 compression **(c)** MPEG-2 decoding **(d)** JPEG compression

Figure 6.12.: Flow graph of the extracted pipeline for each application in table 6.2

6.3.1.1. MP3 Decoding (EEMBC 2.0)

This benchmark implements a decoder for the de-facto standard for digital music compression, MPEG-1 Audio Layer 3. As shown in the motivating example in figure 6.1, the decoding pipeline comprises multiple kernels that process the encoded data stream at various levels of granularity, ranging from whole audio frames down to frequency sub-bands.

The key challenge in parallelising this application is in exposing sufficient work spread over multiple loop levels to the PDG partitioner in order to facilitate the extraction of a well-balanced pipeline. The MP3 decoder makes use of idiosyncratic programming idioms that typically evade static analysis such as returning function values through buffers passed into functions by pointers, deeply nested function calls and the extensive use of dynamically allocated buffers. Existing approaches either do not address this issue at all [20], or rely on manual code transformations (e.g. function inlining, full loop unrolling or loop distribution) [141]. The latter is both an error-prone process, but most importantly, if it is not guided and selective, it can lead to suboptimal results.

Using multi-level loop distribution and stage replication we achieve a speedup of 3.52 on 6 cores over the sequential baseline. The extracted pipeline structure for this benchmark is shown in figure 6.12(a) and, in fact, this structure resembles the one contained in the explicitly parallel streaming StreamIt benchmarks [140]. This result is encouraging and suggests that our pipeline extraction methodology is capable of "imitating" the coarse-grain algorithmic parallelisation that previously was alone in the hand of the expert programmer.

6.3.1.2. Bzip2 Compression (SPEC2000)

This benchmark implements a lossless, block-sorting data compressor. It exhibits a typical pipeline structure which operates on constant size data blocks. It consists roughly of the following stages: (i) input and *Run-Length Encoding* (RLE) which are both inherently sequential, (ii) independently compression of each block by performing a Burrows-Wheeler transform, (iii) MTF transform and (iv) output.

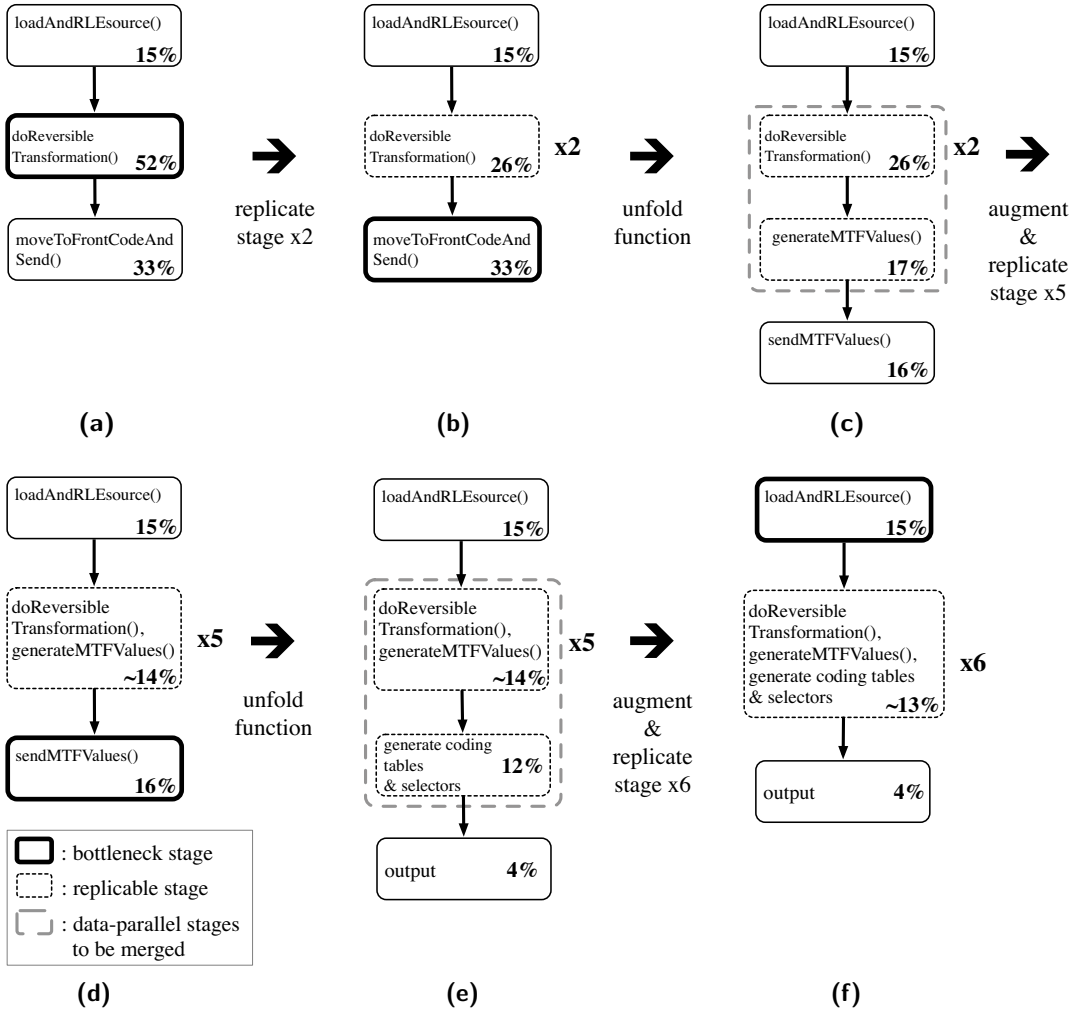


Figure 6.13.: The main steps of the partitioning algorithm for *bzip2 compression*. Note that selective function unfolding and function splitting creates new opportunities for stage replication. In addition, it is clear that the form of the pipeline is changing as the number of utilised cores increases.

Figure 6.12(b) depicts the extracted pipeline structure for the *bzip2* benchmark. Processing of constant sized blocks can be accomplished in parallel in a replicated stage. The extensive use of dynamic memory allocation, pointer arithmetic and pointer aliasing (e.g. pointers of different types are used to access the same allocated buffer) have been a particular challenge to our tools, still an overall speedup of 4.7 on eight processors could be achieved. This is an example where a purely static approach would have

failed completely to detect the parallelisation opportunities, but profiling accurately and correctly identifies the independence of individual blocks, even if this has to be ultimately verified by the user.

Figure 6.13 illustrates the main steps of the partitioning algorithm (algorithm 6.2). Selective unfolding and splitting of function nodes successfully decouples interleaved I/O operations and computation nested in lower-level functions. For instance, in step 6.13(c)→(d) function *moveToFrontCodeAndSend()* is unfolded and thus *generateMTFValues()* and *sendMTFValues()* are decoupled, forming two new stages that can be executed in parallel and reduce the length of the dominating stage. In addition, this split creates an additional parallelisation opportunity. Function *generateMTFValues()* is a replicable stage since it does not perform any I/O and thus it can be merged with the preceding stage of *doReversibleTransformation()* without breaking the data-parallel property (line 24 in algorithm 6.13). This way a coarser replicable stage is formed which can scale up to 5 cores, further reducing the execution time of the slowest stage. Similarly, in step 6.13(d)→(e) function *sendMTFValues()* is unfolded to enable the decoupling of intertwined I/O. This results to a pipeline where the dominating stage is the one that reads the input file and performs the Run-Length encoding, a process which is inherently sequential and exhibits only fine-grain parallelism. Finally, note that there is no fixed partitioning that performs better for a given total number of utilised cores.

An alternative to function splitting would have been to use special link-time hooks that bypass output calls and save the output in intermediate buffers which are later flushed *in-order* [112, 142]. This, however, would assume the use of specific I/O libraries and bzip2 is using memory mapped I/O instead of the standard library. Therefore, these approaches presume *manually* modified sources.

6.3.1.3. MPEG-2 Video Decoding (EEMBC 2.0)

This benchmark implements the widely used international standard for video compression. At an algorithmic level, MPEG-2 decoding features multiple processing stages (e.g. coefficient decoding, saturation control, motion compensation) which successively operate on the encoded input stream of frames on different levels of granularity (e.g. frames, slices, components, macro-blocks).

Using our tools we extract the multi-level pipeline structure shown in figure 6.12(c) that results in a speedup of 2.68 utilising three processors. Given that we solely rely on pipeline parallelism and do not attempt to exploit further fine-grain ILP or SIMD-style short vector data parallelism this result is impressive. For this application the speedup is eventually restricted by an unbalanced distribution of work between stages. This can be seen in figure 6.12(c) where next to the extracted pipeline and relative time spent in each pipeline stage is shown. Integrated approaches targeting ILP and short vector

instructions inside pipeline stages may eventually contribute to further speedups, but are outside the scope of this thesis.

6.3.1.4. JPEG Compression (EEMBC 2.0)

This benchmark implements the JPEG image compression algorithm, the dominant standard in digital image photography and the WWW. Using our tools we managed to extract a two-stage pipeline (see figure 6.12(d)) which resulted to a moderate speedup of 47%. This is mainly due to the fact that although JPEG features abundant fine-grain parallelism, even on an algorithmic level most components are inter-dependent. In order to parallelise this application we annotated the memory allocation routines before analyzing the application with the IR-profiler. This was necessary because the reference implementation uses a custom memory manager that utilises OS features like memory mapped I/O. Nevertheless, this is a relatively straightforward process compared to process-based systems like ([112, 141]) that claim transparent privatisation but in this case would most probably require major modification to the source of the memory manager.

6.3.2. Safety

We have tested each of the benchmarks using an extensive set of additional, previously unseen input data sets. For all benchmarks and inputs this has resulted in identical outputs for the sequential and the parallelised codes. In addition, manual code inspection did not reveal any violation of dependences. This result is initially surprising. However, for the static pipelines we are targeting in this work complex, but regular dependence patterns can be expected.

6.4. Conclusion

In this chapter we have presented a semi-automatic, profile-driven methodology for the extraction of pipeline parallelism from sequential codes. We improve on existing work in that we do not rely on manual code annotation, but only involve the user for final approval. We consider our work to be a feasibility study into the limits of parallelism detection using profile information. Our approach covers multi-level loops, hierarchical pipelines and pipeline stage replication in a uniform framework. It, thus, avoids the performance bottlenecks resulting from imbalanced pipeline stages that existing pipeline parallelisation approaches suffer from. We have demonstrated that our methodology can successfully exploit pipeline parallelism in real-world multimedia and streaming applications featuring idiosyncratic programming constructs. Speedups of up to 4.7 for an eight-core Intel Xeon machine are promising and demonstrate the potential of profile-driven, semi-automatic parallelisation approaches that target parallelism beyond the loop level. Future work will focus on the extraction and exploitation of dy-

namic parallelism, target heterogeneous architectures, improve our safety mechanisms and consider the use of machine learning for the construction of improved partitioning heuristics.

Chapter 7.

Conclusions

This thesis investigated the case of leveraging profile-driven dependence analysis to enhance coarse-grain parallelism extraction from sequential applications. We presented an instrumentation framework that enables the integration of precise dependence information in the Intermediate Representation of the compiler. Our first study aimed at demonstrating the applicability but also the effectiveness of profile-driven approaches for the exploitation of loop-level parallelism in scientific and embedded applications. Next, in a more ambitious study, we presented how this methodology can be extended to address more powerful parallelisation schemes. To accomplish this we developed a hierarchical partitioning algorithm and a flexible code-generation infrastructure that automatically transforms a sequential application in a multi-threaded pipeline.

The structure of this chapter is as follows. In section 7.1 we present a summary of the contributions and experimental results of this thesis. Section 7.2 provides a critical analysis of issues that we have not addressed in this thesis and discusses possible directions for future research.

7.1. Contributions

7.1.1. Intermediate-Representation Profiling

Auto-parallelisation despite its success in uncovering parallelism from programs written in languages like Fortran has so far failed to reach its potential in a more general setup. This thesis was primarily focused in the exploitation of *coarse-grain* parallelism from applications written in the *C* programming language; including pointers, dynamic memory allocation, non-affine index expressions and other language features or idioms that make static dependence analysis conservative. To remove these obstacles in chapter 4 we introduced a code instrumentation approach that facilitates powerful and precise dependence analysis on a whole-program level.

Established dependence profiling techniques utilise binary instrumentation mechanisms and hence rely on inaccurate debugging information to establish a connection of the information back to the source code. To bridge this information gap, we followed an approach that operates directly on the Intermediate Representation (IR) of the compiler. Additionally, it provided the means to identify opportunities for more aggressive

transformations like *privatisation* and *reduction operations*; two transformations that are commonly used in static compilation of array-based programs but have limited applicability in languages like *C*. In fact, in the case of parallel reductions this was only made possible with a novel bidirectional analysis that identifies candidate reduction operations at compile-time, verifies their semantics at profile-time and finally annotates the Ir of the compiler to enable their exploitation in parallel-code generation.

7.1.2. Data-parallelism

Chapter 5 presented a parallelisation methodology for data-parallel loops that builds on top of profile-driven dependence analysis. To generate parallel code we used OpenMP directives. OpenMP offers compiler and runtime support for the majority of modern parallel architectures and thus allowed us to target multiple architecture avoiding any cumbersome code-generation issues. Additionally, in this study we integrated a powerful Machine-Learning profitability analysis to select the loops to parallelise as well as the most suitable loop-scheduling policy.

For the evaluation of this scheme we used an Intel Xeon architecture with eight cores in total and an extensive set of benchmarks, namely Nas Pb and Spec Fp2000. Besides the sequential code these benchmarks also include OpenMP versions manually parallelised by expert programmers. The ML-based scheme improved the attainable speedup over the default heuristic-based 48% on average. More surprisingly, the combined scheme achieved on average 98% of the hand-parallelised code performance, resulting in an average speedup of 3.5. Furthermore, profile-driven parallelisation was proven to be effective for loops that although sequential, can still be parallelised if transformations like privatisation or reduction is applied. Loops parallelised using this approach often span multiple files, and include deeply nested function calls, dynamically allocated objects and pointers addressing resizeable arrays. These results look definitely promising given that profile-driven parallelisation can actually match the performance attainable till now only by experienced developers with domain-specific knowledge of both the algorithms and the implementation. Finally, in an additional experiment we investigated the case of utilising Ml-mapping to target a non-conventional architecture, the Ibm Cell Be. Unlike manually tuned and hard-wired heuristics that lack any portability across architectures, ML-mapping not only successfully selected the loops which are profitable, but also managed to significantly outperform the hand-parallelised codes which were also compiled with the native OpenMP compiler.

7.1.3. Pipeline-parallelism

Despite the ample data-parallelism available in scientific applications and embedded kernels, simple iteration-level decomposition is not sufficient for more complex applications. For instance, multimedia applications typically feature loops that execute multiple algorithms and data transformations – some of them data-parallel – intertwined

with I/O operations, all in a single iteration. To exploit available parallelism under this scenario, programmers typically employ a parallelisation paradigm inspired by the *pipelined* organisation of modern processors. More specifically, the loop body is divided in stages, each one operating in turn on successive packets of data, effectively forming a functional pipeline that processes a stream of data. Nevertheless, this comes at the cost of increased design and programming effort; requiring non-trivial manual transformations like outlining the thread code for each pipeline stage, privatising and initialising local, global or heap-allocated structures, inserting communication and synchronisation calls. And this procedure has to be repeated for the exploration of alternative partitionings or adjusted for new target architectures or system configurations.

Contrary to loop-level parallelisation where entire loop iterations are executed in parallel, pipeline partitioning required a more flexible approach. To achieve this in chapter 6 we introduced a whole-program representation based on the Program Dependence Graph where both data and control dependences are explicitly represented. Based on this IR we developed a top-down partitioning strategy that aims at balancing the load across the available resources while keeping communication overhead low. In addition, we enhanced the traditional pipeline skeleton with two powerful concepts borrowed from streaming languages, namely *stage replication* and *multi-rate* filters. Finally, we developed a powerful code-generation infrastructure and an efficient runtime system to enable their automatic exploitation. Empirical evaluation using real-life multimedia codec implementations confirmed the effectiveness of this approach, demonstrating speedups of up to 4.7 (3.1 on average). These results provide evidence that IR-profiling can not only overcome many of the hurdles imposed by widely used low-level features of *C*, but can also extend the reach of automatic parallelisation methods to more complex forms of parallelism.

As software development costs are rising rapidly during this unprecedented transition of the microprocessor design industry to multi-cores we expect next-generation development tools to adopt profile-driven techniques to increase productivity and turn parallelism from a problem to an opportunity for higher performance.

7.2. Future Directions

In this section we discuss related issues that have not been thoroughly addressed in this thesis along with promising directions for future research.

7.2.1. Intermediate-Representation Profiling

Optimisations In section 4.4 we discussed how the current implementation of IR-profiling can be further optimised for performance without losing its precision. Furthermore, there are techniques which can trade-off between performance and the precision of the profiling information. Information about deeply nested loops and functions,

for instance, can be collapsed in a single compound node where control-flow is not instrumented and data accesses are summarised in a single address range. This can be particularly advantageous for short loops with irregular accesses, where address signatures cannot be effectively compressed. This way we avoid the overhead of tracking loop-carried dependences within a loop that is highly unlikely to be parallelised and we just process it as a single *black box*. A possible technique to achieve this is to leverage a Machine-Learning approach similar to the one presented in chapter 5 that predicts the maximum loop-depth that parallelisation will be profitable. Another, more ambitious technique would be to dynamically remove the instrumentation for loops that have already been determined to be sequential. Nevertheless, this entails the integration of a complex virtualisation layer, possibly utilising dynamic binary translation.

Trace indexing Another implementation issue that we hope to address in future research is the storage of the trace itself. Instead of storing a flat sequence of IR-instructions we can devise a hierarchical structure that enables fast seek operations in specific parts of the execution. This can be achieved either by using an index that records the offset of specific control-flow operations (e.g. function or loop entry) in the instruction stream or an even more radical relational representation that can be stored in a database management system. The latter option also has the benefit of transparently utilising the powerful and efficient file-backed data structures (e.g. *B+* trees) implemented in modern RDBMSs. Building on top of this feature the analysers but also the graphical interface can extract additional metrics (e.g. size and granularity of data communication) about a specific code region on-demand, in contrast with an *a priori* global detailed analysis.

7.2.2. Data-parallelism

Loop transformations Our methodology for extracting data-parallelism in chapter 5 uses the heuristic of selecting for parallelisation the outermost parallel loop of a loop nest. This choice was based on the simplistic assumption that the coarser the iteration granularity the more effective its parallelisation. This technique performed reasonably well as it is clear from the empirical evaluation, however, there is still room for improvement. The problem of determining the optimal loop-depth for parallelisation is quite complicated since it involves modelling the access patterns of each loop-level in relation to the memory hierarchy of the target architecture. The problem gets further complicated if we consider unimodular (e.g. loop-interchange, -reversal and skewing) or more complex loop transformations (e.g. loop-fusion, unrolling and tiling) that can improve parallelisation, data-locality and vectorisation opportunities. The problem of finding the optimal sequence of such transformations has been heavily studied, especially in the context of the polyhedral model, employing advanced iterative or Machine-Learning approaches to counter the combinatorial explosion of the optimisation space [158, 160, 110, 111]. Nevertheless, the majority of these studies has only addressed

these issues to fully analysable linear algebra kernels and thus presumes detailed and accurate dependence information. Despite the fact that complex locality optimisations like loop-tiling are difficult to apply in the presence of ambiguous dependence information, simpler transformations like loop interchange or loop-fusion can still be performed using profiling-based dependence information.

Nested parallelism Another opportunity for future research is the exploration of multi-level data-parallelism. Our approach so far selected a single loop-level to parallelise (the outer most one). The next step will be to investigate the parallelisation of multiple nested loops. Despite the additional threading overhead, this technique can provide better load-balancing and data locality compared to single-level. In fact, recent studies [151, 9, 63] on Computational Fluid Dynamics applications have demonstrated improved scalability especially for large distributed memory machines with hundreds of processors. As the number of cores per chip has already started following an exponential trend – analogous to the trend predicted by G. Moore for the number of transistors in a chip – such multi-level approaches will become the norm for many applications that feature big datasets. The challenges for auto-parallelisation in this case will be 1. uncovering such multi-level parallelism from sequential applications, 2. finding a partitioning/mapping (i.e. which loops in a nest should be parallelised) that best matches the performance characteristics (e.g. threading overhead, communication latency) of the targeted architecture, and 3. determining the loop scheduling policy for each level that achieves the optimal trade-off between effective load-balancing and synchronisation cost.

7.2.3. Pipeline-parallelism

Partitioning and scheduling The pipeline partitioning algorithm presented in chapter 6 had proven to be a simple, yet effective approach at least for the applications that we have considered so far. Nevertheless, it is based on the simplifying assumption that the load of each pipeline stage is evenly distributed in the program execution time. In addition we presumed a fixed target architecture and number of resources. Finally, we optimised parallelisation having only maximum throughput as an objective. However, there are many cases where these conjectures may be proven to be insufficient. Future research should account for these sources of variation and more specifically:

Input variation Variation in the execution behaviour due to characteristics inherent in the input stream may lead to profile-driven partitionings that are suboptimal for a new input. For instance in MPEG-2 motion compensation is only performed for I-frames. A dynamic scheme can exploit this property to schedule other threads of the application that might be co-scheduled due to resource limitations.

OS scheduling Our approach is binding threads to cores using a static round-robin scheduling scheme. In a multi-programmed system this might not be preferable or

even feasible[1]. The Os can reduce the number of cores available to a program/user or use time-sharing for some cores based on the current workload and system scheduling policy.

Power and thermal constraints The proliferation of portable consumer devices in addition to technology constraints often force designers to optimise for power-efficiency rather than just performance. Furthermore, voltage and frequency of individual cores has to be dynamically scaled due to thermal or other constraints that regard the operation stability of the chip. The mechanism controlling these parameters are currently fixed on the hardware or offer limited control to the Os. However, recent developments like the Intel Single-chip Cloud Computer [55] research prototype that features Os-level control for the multiple voltage domains of the chip definitely show that this restrictions are soon to be waived.

Letting the Operating System (Os) or the compiler address this adaptation problem independently will unavoidably lead to suboptimal solutions. The compiler even if we assume precise profiling information, has little flexibility to address input-dependent or hard to predict load variation scenarios. On the other hand the Os, can utilise advanced monitoring techniques like performance counter sampling but still has limited knowledge for the interaction between the application's threads and should keep the runtime overhead to a minimum. Purely runtime solutions on the other hand incur significant overheads and lack the flexibility of more elaborate approaches. Our view is that the problem of runtime adaptation and the variation of the application load require a multi-layered *synergistic* approach that involves all the critical components: compiler, runtime and operating system. Such a scheme can provide both high flexibility and reasonable overhead by delegating expensive scheduling decisions to the compiler and only perform minimal adaptation on the runtime utilising Os-based policies and runtime-system mechanisms.

Only recently, Hormati et al. proposed Flextream [49] a novel approach that combines static compilation and dynamic adaptation for streaming applications. This approach although promising is only focusing on variation in the parameters of the target architecture (e.g. on-chip memory size, number of cores). In addition it addresses only programs with explicit stream graph representations (e.g. StreamIt) and explicitly managed memory communication (e.g. similar to Cell Be local storage). These programs, however, are relatively simple and thus feature minimum inherent variation. Compared to stream programs automatic parallelisation of sequential application offers less control over the granularity of the extract pipeline but is applicable to more complex program and thus requires performance modelling that exceed the scope of

[1] Most probably, future many-core systems will probably not be utilised as time-shared resources not only due to the abundance of available cores but also due to thermal and power constraints. For instance, *hotspots* can be prevented or mitigated by spreading the computation across many cores operating at a relatively low-frequency. Still these are scenarios where an application should not assume exclusive access to cores.

Flextram (e.g. interaction with the cache hierarchy). Nevertheless, this is one of the first approaches that identifies the key problem of existing approaches, i.e. assuming a fixed pipeline structure.

Suleman et al. introduce in [135] an adaptive scheme for scheduling pipeline programs on homogeneous multi-cores. They control the number of threads executing each stage and their mapping to the available cores with the objective of optimising either power or performance. The main shortcoming of this approach is that it explores the space of different mapping decisions on the runtime. In addition, in order to minimise the instrumentation and adaptation overhead they only monitor the execution time of each pipeline stage. We believe that such run-time schemes can be further enhanced and remain minimally obtrusive at the same time even. First, by explicitly monitoring the memory behaviour of each thread using hardware performance counters (e.g. L2 cache misses and coherency invalidations). Using these explicit metrics compared to the implicit metric of the execution time the scheduler can more accurately model 1. the interaction of co-scheduled threads regarding the size of their working-set, and 2. the communication pattern/dependence between threads scheduled in different cores. Second by utilising off-line analysis performed at compile/profile time and precise knowledge of the underlying cache-hierarchy on the runtime. For instance, adjacent stages can exploit shared higher cache-levels to minimise communication time. Similarly, stages with large working-sets can be co-scheduled with stages that are not that memory intensive to avoid cache-thrashing. Similar monitor techniques can also be applied for inter-core, inter-chip and memory bandwidth.

Another related monitoring technique is presented in [5]. The authors use taint analysis to determine key control points in streaming applications that might lead to execution time variation. Then, this information is used to build an on-line execution time estimator for each stage of a set of manually parallelised applications. Contrary to our original assumption, their experimental analysis presumes more application threads than available cores, which are therefore dynamically scheduled using the produced estimations. Under this scenario, this estimation mechanism can be utilised to pro-actively balance the load in a way that favours the current (but time variable) dominating stage. This study is definitely paving the way for more aggressive compiler-based techniques to predict variability at runtime accurately and ahead-of-time. Its main shortcoming, however, is that the load-balancing heuristic is not accounting for runtime overhead caused by cache sharing, thread migration and context switching. This is also suggested by the parallelisation methodology that the authors followed, since they eagerly partitioned the code in parallel stages without accounting for the aforementioned overheads [2]

[2] Although direct comparison is not possible, we observed that the speedup attained for the only benchmark that we have also parallelised (MPEG-2 decoding) is lower than the one reported in section 6.3, even after dynamic load-balancing is applied.

Heterogeneous cores Most of today's multi-core systems consist of homogeneous processing elements – IBM Cell BE being a notable exception. Heterogeneous configurations, on the other hand, have been proposed as a solution to the escalating problem of power efficiency. For embedded systems in particular heterogeneity can be considered as another design option in the hardware-software co-design process. As is obvious from figure 6.12 each pipeline stage comprises one or more coarse-grain algorithmic blocks. As a consequence each thread of the pipeline exhibits better instruction locality and therefore is more power and performance efficient. More interestingly, pipeline partitioning by separating the application in its functional components creates the opportunity for finer tuning of the architectural parameters of each core. For instance, pipeline stages rich in word-level parallelism can be executed more efficiently on VLIW cores or in-order cores with powerful SIMD units. Similarly, the designers have the option to specialise the instruction set extensions for each pipeline stage rather than the whole program. This technique can result in more efficient utilisation of the limited configurable-hardware budget available on each embedded core. Still, pipeline partitioning can be advantageous in general-purpose systems too. The main difference is that core specialisation is transformed in its dual problem, mapping stages to cores while trying to maximise an objective function like throughput or power.

Our view is that future research that will focus on these topics should incorporate the pipeline partitioner in its design methodology as a dynamic component rather than assume a fixed pipeline configuration. Finer-grain partitioning, for instance, provides the designer with more opportunities for core specialisation but this comes at the cost of increased communication overhead.

Architectural extensions The primary overhead of pipeline parallelisation is due to the inter-thread communication cost, either explicit or implicit. Explicit communication occurs while copying the contents of thread-private memory to the pipeline buffers. Implicit communication, on the other hand, occurs when the consumer thread accesses the contents of the incoming buffer and stalls in a coherency miss till that cache-line is fetched either from main memory or one of the cache-levels of the core executing the producer thread. In our prototype we utilised SIMD extensions for streaming loads and stores a feature already available from the majority of modern CMPs to mitigate these effects. Streaming instructions avoid *cache pollution* because they bypass the cache-hierarchy writing directly to memory, and have the extra benefit of effectively utilising the memory bandwidth using *write-combining* operations. Another technique to reduce the implicit communication cost is *cache-aware scheduling* (i.e., scheduling threads with high communication in cores that share a higher cache-level). Nevertheless, since modern CMPS typically share caches in groups of two (e.g. L2 in Intel Core2) or four (e.g. L3 in Intel Nehalem) and cache sharing does not scale well beyond eight cores anyway, there is a limit to this approach.

This thesis intentionally did not utilise any hardware features not readily available in commodity hardware. However, it would be very interesting to investigate the potential of a simple and generic hardware mechanisms that can address the communication problem that we have just reviewed. For instance, an interesting approach for this problem would be the addition of a special instruction that not only invalidates lines in the producer side but also starts pushing the data to the consumer cache. The problem with existing prefetching hints is that they have to be initiated on the consumer's side and therefore most probably prefetching will be untimely. Therefore, future research should aim for *pro-active* schemes.

A promising proposal towards that direction, called Data Marshalling, was only recently published in [134]. It is based on a profiling stage that aims at identifying the load and store instructions where inter-thread communication occurs both incoming (first-load) or outgoing (last-write). Then, based on this information it annotates the binary using two special instructions. On the runtime these instructions trigger explicit cache-line communication which is implemented using a very small buffer (16-entries) for storing the relevant addresses and a cache-to-cache network. The main shortcoming of this approach is that it requires profiling for detecting a pattern that can be easily monitored on the runtime using a simple Program Counter (PC) predictor. Furthermore, their profiling scheme does not disambiguate among communicating and non-communicating dynamic instances of the same load/store instruction. In fact, this is also obvious from the very low accuracy, less than 60% and 50% on average, of the predictions. Previous work on the related subject of cache-line self-invalidation has actually shown that far better accuracies are possible if a *trace-based predictor* [3] is utilised [75, 74].

Another problem with Data Marshalling is that it only exploits very short communication sequences (up to 16 cache-lines long). This is sufficient for the majority of the applications that the authors used that were manually parallelised and use fine-grain communication. On the contrary, the applications that we consider in chapter 6 feature more coarse-grain communication (e.g. bzip2 processes a buffer of 100-900KB). Therefore a more general approach is required, most probably similar to pre-invalidation that starts evicting cache-lines from the lower cache-levels towards larger shared cache-levels. Such a scheme will have the additional benefit of reducing cache-pollution in the producer core too since evictions will start evicting data that are not going to be accessed again from the current core and thus will improve cache utilisation.

[3] Trace-based predictors for cache-line self-invalidation predict the cache-lines that will be invalidated by the coherency protocol before being accessed again (i.e. last-write). In addition, such a scheme does not depend on off-line profiling. They are based on the same principle with *path-based* branch-predictors that exploit the history of branch target addresses to predict the outcome of conditional jump.

7.2.4. User Interface Enhancements

The user interface that visualises the whole-program dependence graph and the extracted pipeline flow-graph was designed having as a primary objective to assist us in the empirical evaluation. Nevertheless, it lacks some features that will allow our approach to increase the productivity of developers in a realistic usage-scenario. A short list of potential enhancement follows:

- Incorporate the dependence graph visualisation in a widely-used Integrated Development Environments (IDE) like Eclipse [138] or Microsoft® visual studio [89]. This will allow developers to inspect the original source code in parallel with the extracted pipeline.
- The current visualisation depicts all the dependence arcs at the same time. A simple filtering mechanism that will select dependence arcs that manifest only on specific levels of the code or relate to a subset of all the communicated data will provide a more focused and clear view of the data-flow.
- The validation procedure can be more systematic. More specifically, the IDE can provide a list of validation steps that the user should follow in order to confirm the correctness of the proposed parallelisation scheme.

Appendix A.

Data-level Parallelism in Embedded Applications

In this appendix we present an empirical evaluation of the data-level parallelism extraction method presented in chapter 5 focused on embedded programs. The purpose of this study is to demonstrate that profile-driven parallelism detection is applicable to a wider domain of applications rather than just scientific applications. However, as we discussed in section 5.5, data-parallelism is often exploitable only in individual stages of a full-scale embedded application. Exploitation on whole-program scale requires more powerful schemes like pipeline parallelisation, a method which is covered in chapter 6. For the purpose of this empirical evaluation we use a collection of embedded kernels that contain exploitable loop-level parallelism as well as a full-scale implementation of the JPEG-2000 image-coding standard.

The structure of this appendix is as follows. In section A.1 an overview of JPEG-2000 and its parallelisation is given. The experimental methodology and the empirical evaluation follows in section A.2.3. Finally, we conclude in section A.3.

A.1. Case Study: JPEG-2000 Still Image Compression

JPEG-2000 [84, 26] is the most recent standard produced by the JPEG committee for still image compression and is widely used in digital photography, medical imaging and the digital film industry. The main stages of the wavelet-based coding process as defined in the JPEG-2000 standard are shown in the diagram in figure A.1.

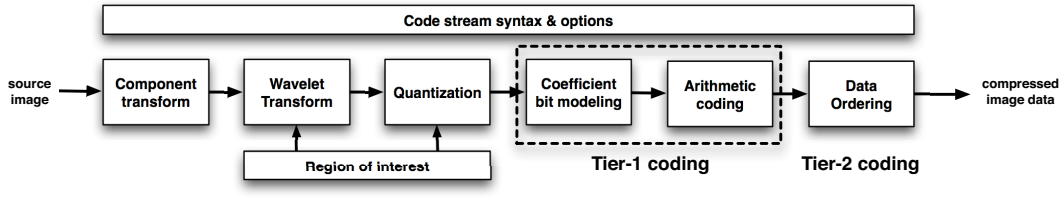


Figure A.1.: Overview of the JPEG-2000 coding process.

In the *component transformation stage* images have to be transformed from the RGB colour space to another colour space, leading to three components that are handled separately. Furthermore, the image is partitioned into non-overlapping rectangles, called *tiles*. The tile size can be chosen arbitrarily. Each tile, with the exception of the border-line tiles, are equally sized and are encoded *independently*, enabling the use of JPEG-2000 on memory constrained platforms. However, tiling is also known to cause a degradation of the rate-distortion performance as well as introducing visible blocking artefacts when used in *lossy mode*. In the *wavelet transformation stage* each tile is processed by either an irreversible CDF 9/7 floating point wavelet transform or a reversible, biorthogonal CDF 5/3 integer wavelet transform, corresponding to lossy or lossless coding modes respectively. In the subsequent *quantisation stage* the coefficients computed in the earlier wavelet transformation stage are quantised to reduce the number of bits used to represent them. In addition, sub-bands resulting from the previous stage are further divided into *code-blocks*. These are rectangular arrays of coefficients that can be extracted *independently*. In the following passes of the coding pipeline entropy encoding is applied *separately* to each code-block (*Tier-1 encoding*). Finally, *Tier-2* reorders and packs the code-block bit-streams into the final JPEG-2000 bit-stream.

A.1.1. Detection of Parallelism

JPEG-2000 exhibits multiple levels of data-parallelism (e.g. tiles, components, code-blocks). Even without tiling most of the coding stages can be performed independently on each component. Furthermore, during entropy bit coding, an additional level of parallelism can be exploited using a code-block decomposition. However, extracting all these levels of parallelism from a sequential implementation in *C* is challenging.

A.1.2. Static Analysis of DOALL Loops

Initially, we evaluated the amount and type of parallelism a production auto-parallelising compiler can extract. We used Intel ICC version 10.1 (section 3.2). Enabling the most powerful optimisations available in the Intel compiler (e.g. multi-file interprocedural analysis, auto-parallelisation and auto-vectorisation) had no impact on the overall performance of the application for both compression and decompression. The compiler vectorised 17 and 11 loops, respectively, but could not parallelise any loops. Setting the profitability threshold to its minimum value resulted in 31 and 20 vectorised loops for compression and decompression, respectively, and 54 and 27 parallelised loops. This more aggressive parallelisation scheme, however, resulted in a slow-down of 8 compared to the default setting. Similarly to the scientific applications, this is due to static analysis being successful in extracting parallelism only from deeply nested loops which are too fine-grain to be profitably parallelised.

Block name	Function	% seq. time
DWT	dwt_encode	53.3
Tier-1 coding	t1_encode_cblks	41.2
Tier-2 coding	t2_encode_packets	< 1

Table A.1.: Execution times for each stage of the JPEG-2000 algorithm (compression in lossless mode).

A.1.3. Hot Spot Detection

In a first step we identify the computational intensive parts of the application. Table A.1 shows the execution times for each of the algorithmic stages of the JPEG-2000 application are shown. Clearly, Tier-1 encoding and discrete wavelet transform are the most computationally intensive stages. Tier-2 coding, on the other hand, contributes less than 1% of the total encoding time. Therefore, in the subsequent steps we primarily focus on extracting coarse-grain loop-level parallelism from functions *dwt_encode* and *t1_encode_cblks*.

A.1.4. Profile-Driven Dependence Analysis

Following hot spot detection we use our analysis tool described in section 5.2 to extract parallelism at any level of the call hierarchy under *dwt_encode* and *t1_encode_cblks*.

The main loop of the discrete wavelet transform can be easily parallelised as a DOALL loop at the colour-component level. Tier-1 coding can be either parallelised in the colour-component level or at a code-block level. Although both are coarse-grain loops the main difference is that there are only three colour-component in each image, thus the scalability of the former is expected to be limited. Parallelisation of the loops in *t1_encode_cblks* was more complicated. A loop-carried dependence prevents parallelisation of the original sequential code. Using the automatic reduction detection feature of the Dependence Analyser, however, we can automatically detect that this dependence is due to a valid parallel reduction operation. Indeed, manual code inspection revealed that this is due to the update of an accumulation variable which is not used before later stages of the encoding process.

Decoding We repeated the same process for JPEG-2000 decoding where we parallelised the corresponding stages of the codec. To achieve this we had to repeat the profile-driven analysis using appropriate input data and command line flags. The code is almost symmetrical, however the execution-time of the parallelisable stages (Tier-1 and DWT) represents less than 90% of the total decoding time.

Tiling As we have already mentioned, another interesting opportunity resides in the tiling option of JPEG-2000. Tiling is inherently parallel. However, analysing the OPENJPEG implementation using the profile-driven dependence analyser revealed that

although the decoding process is trivially parallelizable the same does not hold for encoding. This is due to a true dependence caused by the call to an I/O function in the tile-level loop. Once again we observe that arbitrary decisions of the programmer might hamper parallelisation. In fact, parallelising this form of partially sequentially loops is of the main motivations of the study on pipeline parallelisation in chapter 6.

A.2. Empirical Evaluation

A.2.1. Experimental Methodology

We evaluated the achievable speedup of the parallelised benchmark on an Intel Xeon machine with 8 cores in total (*M1* in table 3.8). Since most applications feature only a few coarse-grain loops we followed a manual approach for selecting them rather than the integrated machine-learning based mapping scheme used in chapter 5. Both the sequential and OpenMP versions of the code were compiled using Intel Icc compiler (*C1* in table 3.3). The reported speedups are over the unmodified sequential code and include automatic vectorisation using Sse Simd extensions, thus ensuring the strongest available sequential baseline.

We have selected applications from multiple embedded benchmark suites: MiBench [42], MediaBench [79], and Utdsp [78] based on their suitability for parallelisation and compatibility with our tools. The benchmarks and their main characteristics are summarised in table A.2. For all the applications we are using properly scaled input datasets so that the execution time of the sequential kernel is over 300*msec* on the target platform.

Benchmarks					
			Parallelism		**Manual**
Name	**Source**	**LOC**	data	pipeline	**Parallelisation**
j2kenc	OpenJpeg	19934	✓	✓	[86, 101]
j2kdec		18975	✓	✓	
epic	MediaBench	3533	✓	–	–
susan	MiBench	2130	✓	–	
stringsearch		524	✓	–	–
fft	Utdsp	86	✓	–	–
edge_detect		208	✓	–	–
compress		181	✓	–	–
histogram		75	✓	–	–
spectral		240	✓	–	–

Table A.2.: Summary of the embedded benchmarks used for evaluation.

A.2.2. JPEG-2000

Performance results for both encoding and decoding using either parallelisation at colour-component or code-block level of *t1_encode_cblks* are shown in figure A.2. In

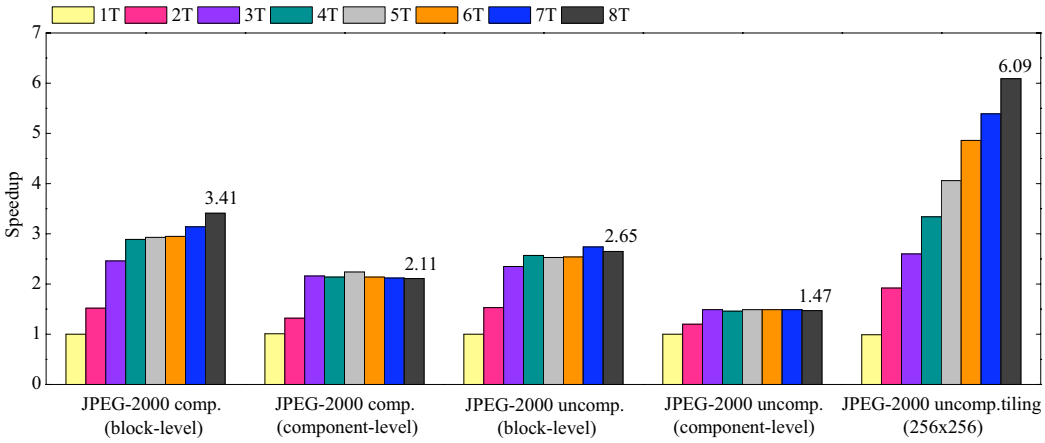


Figure A.2.: Speedups achieved for various functional modes of the JPEG-2000 codec using alternative parallelisation schemes.

the case of compression we achieve promising speedups of up to 3.41 using the block-level parallelisation scheme. Comparing the performance of the component-level and the block-level decomposition, it is clear that the latter is more scalable since as we already mentioned the maximum number of components in an image is three. This also explains why the performance of the component-level parallelisation on more than three cores is almost identical to the one on three cores. Finally, note that only Tier-1 coding – which accounts for 41% of the total encoding time – is parallelisable at the block-level. In fact, this is also clear in the results of figure A.2. We observe that increasing the number of threads to more than three improves the overall speedup, but the rate of improvement is significantly reduced.

In the case of JPEG-2000 decompression, parallelisation results in speedups of up to 2.65. This improvement is lower than the one in the compression scenario but this is to be expected if we take into account that Tier-1 and DWT decoding represent a smaller fraction of the total coding time.

Image Tiling Parallelising JPEG-2000 decompression at a tile-level achieved far better scalability than the one-tile parallelisation schemes, reaching a speedup of 6.09 on eight cores and 256×256 tiles[1]. In fact, these results are in line with those obtained by manual parallelisation by expert programmers [86, 101]. The results in this case should be interpreted as a manifestation of the fact that our approach can exploit multiple levels of inherent parallelism available on different inputs, rather than a more scalable parallelisation strategy of the non-tiled scenario.

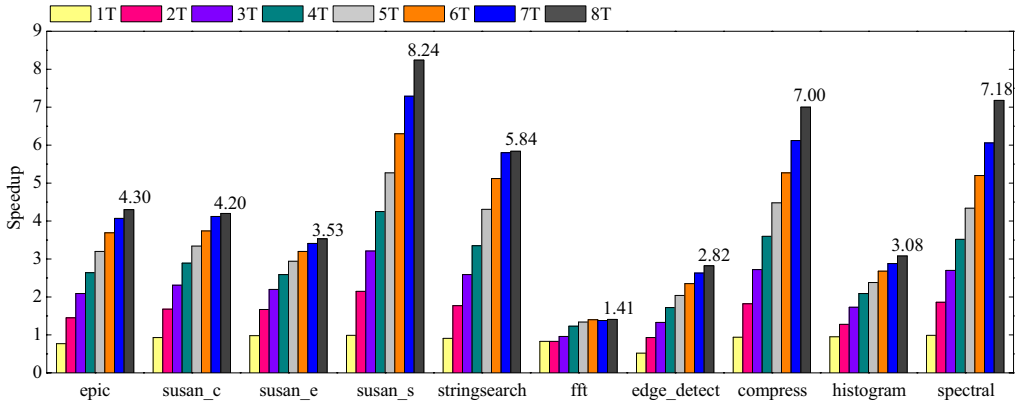


Figure A.3.: Speedups achieved by the parallelised implementations of the benchmarks on *M1* utilising 1 to 8 cores.

A.2.3. Broader Evaluation

In this subsection we present performance results for the remaining embedded benchmarks of table A.2. Figure A.3 shows the maximum attainable speedup. The average speedup using 8 threads is 4.76. Applications with ample coarse-grain DOALL parallelism in outer loops like *epic*, *susan*, *compress*, and *spectral* achieve relatively higher speedups and demonstrate good scalability. In fact, in the case of *susan_s* the parallelised version achieves a super-linear speedup. This is mainly due to better cache-utilisation. On the other hand applications, like *fft*, and *histogram* exhibit either large sequential parts, or are only parallelizable in lower-levels of nested-loops. Therefore, these applications are not scaling so well, attaining relatively lower speedups. In the case of *edge_detect*, performance of the OPENMP version with a single thread is almost two times slower than the original sequential version. This is primarily due to high-level optimisations (e.g. loop unrolling) which are applied by ICC in the sequential code but not in the OPENMP version[2]. Similar phenomena have also been reported by Rul et al. in [123]. Still, the performance of *edge_detect* improves when more threads are added, attaining a maximum speedup of 2.82 on 8 cores. Finally, manual inspection of the source code revealed that profile-driven parallelisation was able to uncover loop-level parallelism even in cases where static analysis fails. In fact, ICC was able to improve performance merely by means of auto-vectorisation rather than parallelisation of coarse-grain loops. As in the case of scientific applications this was due to the extended use of pointers, nested function calls and non-affine array index functions.

[1]Tile size in decompression is fixed for a given input and not user settable. Thus, 256 should is an indicative example rather than a parameter that our parallelising framework selects.

[2]The information about the transformations that are applied to each loop were extracted utilising the verbose optimisation report of ICC (flag `-opt-report`).

A.3. Conclusions

In this appendix we presented an empirical evaluation of profile-driven data-parallelism extraction for embedded programs. The results show that our approach is able to effectively parallelise kernels and applications that feature exploitable, coarse-grain data-parallelism. However, in many cases loop-level parallelism is of very fine granularity or it is constrained by sequential operations and I/O. In addition, in some cases parallel loops account only for a small fraction of the total execution time and thus parallel speedup has low potential for enhancing the overall application performance. In these cases alternative means for enhancing performance should be investigated. More specifically, the performance of fine-grain parallel loops can be improved using more aggressive auto-vectorisation. However, this is a direction which is beyond the scope of this thesis. For the case of partially sequential loops these results are actually the main motivation point of profile-driven pipeline parallelism extraction which is presented in chapter 6.

Bibliography

[1] ACE Associated Compiler Experts b.v. Cosy compiler development system, 2007. `http://www.ace.nl/compiler/cosy.html`.

[2] ACE Associated Compiler Experts b.v. *CCMIR Definition: Specification in fSDL, Description and Rationale*, Feb 2008.

[3] ACE Associated Compiler Experts b.v. *Loop Markers: Design and Implementation*, Feb 2008.

[4] F. Aleen and N. Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *ASPLOS-XIV: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 241–252, 2009.

[5] F. Aleen, M. Sharif, and S. Pande. Input-driven dynamic execution prediction of streaming applications. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 315–324, 2010.

[6] J. Allen and K. Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2001.

[7] V. Aslot, M. Domeika, R. Eigenmann, and G. Gaertner. SPEComp: A new benchmark suite for measuring parallel computer performance. *Lecture Notes in Computer Science*, Jan 2001.

[8] V. Aslot and R. Eigenmann. Performance characteristics of the SPEC OMP2001 benchmarks. *ACM SIGARCH Computer Architecture News*, 29(5):31–40, 2001.

[9] E. Ayguade, M. Gonzalez, X. Martorell, and G. Jost. Employing nested OpenMP for the parallelization of multi-zone computational fluid dynamics applications. *Journal of Parallel and Distributed Computing, IPDPS '04 Special Issue*, 66(5): 686 – 697, 2006.

[10] R. Baert, E. Brockmeyer, S. Wuytack, and T. Ashby. Exploring parallelizations of applications for MPSoC platforms using MPA. In *DATE '09: Proceedings of Design, Automation Test in Europe Conference Exhibition*, pages 1148 –1153, 2009.

[11] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, and A. Woo. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec 1995.

[12] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon,

V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks–summary and preliminary results. In *SC '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165, 1991.

[13] T. Ball and J. Larus. Efficient path profiling. In *MICRO-29: Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46 –57, 1996.

[14] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers Norwell, MA, USA, 1993.

[15] E. B. Bernhard, M. G. Isabelle, and N. V. Vladimir. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, 1992.

[16] T. Brandes, S. Chaumette, M. C. Counilh, J. Roman, A. Darte, F. Desprez, and J. C. Mignot. HPFIT: A set of integrated tools for the parallelization of applications using High Performance Fortran. part I: HPFIT and the TransTOOL environment. *Parallel Computing*, 23(1-2):71 – 87, 1997.

[17] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *MICRO-40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69 – 84, Nov 2007.

[18] Z. Budimlic, A. Chandramowlishwaran, K. Knobe, G. Lowney, V. Sarkar, and L. Treggiari. Multi-core implementations of the concurrent collections programming model. In *CPC '09: 14th International Workshop on Compilers for Parallel Computers*, 2009.

[19] M. G. Burke and R. K. Cytron. Interprocedural dependence analysis and parallelization. *ACM SIGPLAN Notices*, 39(4):139–154, 2004.

[20] J. Ceng, J. Castrillon, W. Sheng, H. Scharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. MAPS: an integrated framework for MPSoC application parallelization. In *DAC 2008: Proceedings of the 45th ACM/IEEE Design Automation Conference*, pages 754–759, 2008.

[21] S. C. Chan, G. R. Gao, B. Chapman, T. Linthicum, and A. Dasgupta. Open64 compiler infrastructure for emerging multicore/manycore architecture all symposium tutorial. In *IPDPS '08: Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[22] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A High-Performance Sparc CMT Processor. *Micro, IEEE*, 29(2):6 –16, 2009.

[23] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: A performance view. *IBM Journal of Research and Development*, 51(5):559 –572, Sep 2007.

[24] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 51–61, 1992.

[25] W. Chen, P. Chang, T. Conte, and W. Hwu. The effect of code expanding optimizations on instruction cache design. *IEEE Transactions on Computers*, 42: 1045–1057, 1993.

[26] C. Christopoulos, A. Skodras, and T. Ebrahimi. The JPEG2000 still image coding: An overview. *IEEE Transactions on Consumer Electronics*, 46(4):1103–1127, Nov 2000.

[27] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. *SIGARCH Computer Architecture News*, 29(2):14–25, 2001.

[28] D. Cordes, P. Marwedel, and A. Mallik. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In *Proceedings of the 8th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, 2010.

[29] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT press, 2nd edition, 2001.

[30] B. Creusillet and F. Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, 1996.

[31] CriticalBlue. Prism® : Evaluating complex code for parallelism. `http://www.criticalblue.com/prism/eval/whitepapers/eval_complex_code/eval_complex_code.htm`.

[32] J. Dai, B. Huang, L. Li, and L. Harrison. Automatically partitioning packet processing applications for pipelined architectures. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, Jun 2005.

[33] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 223–234, 2007.

[34] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[35] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[36] J. Finkle. Reuters Special Report: Can that guy in Ironman 2 whip IBM in real life? `http://www.reuters.com/article/idUSTRE64B5YX20100512`.

[37] Free Software Foundation. *The GNU C Library: §3.2.2.10 Memory Allocation Hooks*. `http://www.gnu.org/s/libc/manual/html_node/Hooks-for-Malloc.html`.

[38] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, 1998.

[39] R. E. Grant and A. Afsahi. A comprehensive analysis of OpenMP applications on dual-core Intel Xeon SMPs. In *IPDPS '07: Proceedings of the International Parallel and Distributed Processing Symposium*, pages 1 – 8, Feb 2007.

[40] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. *SIGPLAN Notices*, 42(6):256–265, 2007.

[41] J. Guo, G. Bikshandi, D. Hoeflinger, and G. Almasi. Hierarchically tiled arrays for parallelism and locality. In *IPDPS '06: Proceedings of the 20th International Parallel and Distributed Processing Symposium*, Jan 2006.

[42] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, and et al. Mibench: A free, commercially representative embedded benchmark suite. *International Workshop on Workload Characterization*, Jan 2001.

[43] M. Haghighat and C. Polychronopoulos. Symbolic Analysis: A Basis for Parallelization, Optimization, and Scheduling of Programs. In *LCPC '93: Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 567–585. Springer-Verlag, 1993.

[44] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, and E. Bu. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84 –89, Dec 1996.

[45] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, pages 350–360, 1991.

[46] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):557–567, 1997.

[47] High Performance Computing System (HPCS) Laboratory. NAS Parallel Benchmarks in OpenMP. `http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html`.

[48] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, 2001.

[49] A. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *PACT '09: Proceedings of 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 214 –223, 2009.

[50] P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC compiler. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 63–73, 2003.

[51] IBM. IBM XL C/C++ for Multicore Acceleration for Linux. `http://www.ibm.com/software/awdtools/xlcpp/multicore/`.

[52] Intel® Corporation. Intel® Parallel Advisor Lite. `http://software.intel.com/en-us/articles/intel-parallel-advisor-lite/`.

[53] Intel® Corporation. Getting Started with the Intel® Parallel Amplifier. `http://software.intel.com/sites/products/documentation/studio/amplifier/en-us/2009/start/getting_started_amplifier.pdf`.

[54] Intel® Corporation. Getting Started with the Intel® Parallel Inspector. `http://software.intel.com/sites/products/documentation/studio/inspector/en-us/2009/start/getting_started_inspector.pdf`.

[55] Intel® Corporation. Single-chip cloud computer. `http://techresearch.intel.com/articles/Tera-Scale/1826.htm`.

[56] Intel® Corporation. *Quad-Core Intel® Xeon® Processor 5400 Series, Datasheet*. Intel® Corporation, Aug 2008.

[57] International Organization for Standardization. ISO/IEC 9899:1999 Specification: International Standard - Programming Languages - C. Apr 1999.

[58] International Roadmap Committee. The international technology roadmap for semiconductors (IRTS), 2007. `http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_Design.pdf`.

[59] International Roadmap Committee. The international technology roadmap for semiconductors (IRTS), 2009. `http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_Design.pdf`.

[60] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, pages 244–251, 1991.

[61] M. Ishihara, H. Honda, and M. Sato. Development and implementation of an interactive parallelization assistance tool for OpenMP: iPat/OMP. *IEICE - Trans. Inf. Syst.*, E89-D(2):399–407, 2006.

[62] M. Islam. On the limitations of compilers to exploit thread-level parallelism in embedded applications. *Computer and Information Science*, Jan 2007.

[63] H. Jin and R. F. V. der Wijngaart. Performance characteristics of the multi-zone nas parallel benchmarks. In *IPDPS '04: Proceedings of the 18th International Parallel and Distributed Processing Symposium*, volume 66, pages 674 – 685, 2006.

[64] I. Karkowski and H. Corporaal. Design of heterogenous multi-processor embedded systems: Applying functional pipelining. In *PACT '97: Proceedings of the Sixth International Conference on Parallel Architectures and Compilation Techniques*, 1997.

[65] I. Karkowski and H. Corporaal. Fp-map-an approach to the functional pipelining of embedded programs. In *HiPC '97: Proceedings of the fourth International Conference on High-Performance Computing*, 1997.

[66] I. Karkowski and H. Corporaal. Overcoming the limitations of the traditional loop parallelization. In B. Hertzberger and P. Sloot, editors, *High-Performance*

Computing and Networking, volume 1225 of *Lecture Notes in Computer Science*, pages 898–907. Springer Berlin / Heidelberg, 1997. http://dx.doi.org/10.1007/BFb0031661. 10.1007/BFb0031661.

[67] A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. On the performance potential of different types of speculative thread-level parallelism. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, page 24, 2006.

[68] A. Kejariwal, A. V. Veidenbaum, A. Nicolau, M. Girkarmark, X. Tian, and H. Saito. Challenges in exploitation of loop parallelism in embedded applications. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 173–180, 2006.

[69] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using spec cpu 2006. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 215–225, Jan 2007.

[70] K. Kennedy, K. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329 – 341, Jul 1991.

[71] Khronos OpenCL Working Group. OpenCL 1.1 Specification, 2009. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[72] V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 85–92, 1998.

[73] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, 2007.

[74] A. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. *ACM SIGARCH Computer Architecture News*, 28(2):139–148, 2000.

[75] A. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. *ACM SIGARCH Computer Architecture News*, 29(2):144–154, 2001.

[76] L. Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2):83–93, 1974.

[77] J. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):812 –826, Jul 1993.

[78] C. Lee. UTDSP benchmark suite, 1998.

[79] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *MICRO-30:*

Proceedings of the 30th annual international symposium on Microarchitecture, Jan 1997.

[80] D. Loveman. High performance Fortran. *IEEE Parallel Distributed Technology: Systems Applications*, 1(1):25 –42, Feb 1993.

[81] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. *SIGARCH Comput. Archit. News*, 29 (2):40–51, 2001.

[82] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[83] J. Marathe, F. Mueller, T. Mohan, S. Mckee, and et al. METRIC: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Jan 2007.

[84] M. W. Marcellin and M. J. Gormish. *The JPEG-2000 Standard*. Kluwer Academic Publishers, Norwell, MA, 1995.

[85] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 365–372, 1999.

[86] P. Meerwald, R. Norcen, and A. Uhl. Parallel JPEG2000 image coding on multiprocessors. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, Jan 2002.

[87] M. Mehrara, J. Hao, P. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 166–176, 2009.

[88] W. mei Hwu, S. Ryoo, S.-Z. Ueng, J. Kelm, I. Gelado, S. Stone, R. Kidd, S. Baghsorkhi, A. Mahesri, S. Tsao, N. Navarro, S. Lumetta, M. Frank, and S. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, Jun 2007.

[89] Microsoft Corporation. Microsft Visual Studio. `http://www.microsoft.com/visualstudio/en-us/`.

[90] S. Moon, B. So, M. Hall, and B. Murphy. A Case for Combining Compile-Time and Run-Time Parallelization. In *Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, page 106. Springer-Verlag, 1998.

[91] S. Moon and M. W. Hall. Evaluation of predicated array data-flow analysis for automatic parallelization. *ACM SIGPLAN Notices*, 34(8):84–95, 1999.

[92] S. Moon, M. W. Hall, and B. R. Murphy. Predicated array data-flow analysis for run-time parallelization. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 204–211, 1998.

[93] G. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, Jan 1998.

[94] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 198–208, 2007.

[95] S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1998.

[96] N. Nethercote and A. Mycroft. Redux a dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science*, Jan 2003.

[97] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, 2007.

[98] C. Newburn and J. Shen. Automatic partitioning of signal processing programs for symmetric multiprocessors. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, pages 269–280, 1996.

[99] C. Newburn, A. Huang, and J. Shen. Balancing fine-and medium-grained parallelism in scheduling loops for the XIMD architecture. In *Proceedings of the IFIP WG10*, volume 3, pages 39–52. Citeseer, 1993.

[100] C. Newburn, D. Noonburg, and J. Shen. A PDG-based tool and its use in analyzing program control dependences. In *Proceedings of the IFIP WG10. 3 Working Conference on Parallel Architectures and Compilation Techniques*, pages 157–168. North-Holland Publishing Co. Amsterdam, The Netherlands, The Netherlands, 1994.

[101] R. Norcen and A. Uhl. High performance JPEG 2000 and MPEG-4 VTC on SMPs using OpenMP. *Parallel Computing*, Jan 2005.

[102] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on Cell. In *IWOMP '07: Proceedings of the 3rd international workshop on OpenMP*, pages 65–76, 2008.

[103] OpenMP Architecture Review Board. *OpenMP Application Programming Interface v2.5*, May 2005.

[104] OpenMP Architecture Review Board. *OpenMP Application Programming Interface v3.0*, May 2008.

[105] B. O'Rourke. Based on the Q&A session with Barry O'Rourke that followed the presentation of CriticalBlue Prism at ICSA, 16th November 2009.

[106] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO-38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, volume 0, pages 105–118, 2005.

[107] D. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin. Polaris: A new-generation parallelizing compiler for MPPs. *In CSRD Rept. No. 1306. Univ. of Illinois at Urbana-Champaign*, 1993.

[108] H. Park, Y. Park, and S. Mahlke. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *MICRO-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 370–380, 2009.

[109] S. Pawlowski. Exascale science: the next frontier in high performance computing. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, page 1. ACM, 2010.

[110] L. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *CGO '07: Proceedings of the fifth annual IEEE/ACM international symposium on Code generation and optimization*, Jan 2007.

[111] L. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: part ii, multidimensional time. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, Jan 2008.

[112] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *ASPLOS XV: Proceedings of the 15th international conference on Architectural support for programming languages and operating systems*, pages 65–76, Mar 2010.

[113] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *CGO '08: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 114–123, 2008.

[114] R. Ramaseshan and F. Mueller. Toward thread-level speculation for coarse-grained parallelism of regular access patterns. *Workshop on Programmability Issues for Multi-Core Computers*, page 12, Feb 2008.

[115] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August. Decoupled software pipelining with the synchronization array. In *PACT '04: Proceedings of the 13th international conference on Parallel architectures and compilation techniques*, number 177 – 188, Jan 2004.

[116] L. Rauchwerger and D. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *ACM SIGPLAN Notices*, 30(6):218–232, 1995.

[117] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard templates adaptive parallel library. In *LCR '98: Proceedings of the 4th International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, pages 402–409, 1998.

[118] F. Reid and J. Bull. OpenMP Microbenchmarks Version 2.0. In *EWOMP 2004*, page 63, 2004.

[119] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with out-of-order spawn in TLS chip multiprocessors: microarchitecture and compilation. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 179–188, 2005.

[120] M. W. Riley, J. D. Warnock, and D. F. Wendel. Cell broadband engine processor: Design and implementation. *IBM Journal of Research and Development*, 51(5): 545 –557, Sep 2007.

[121] S. Rul, H. Vandierendonck, and K. Bosschere. Extracting coarse-grain parallelism in general-purpose programs. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, Feb 2008.

[122] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: Static & dynamic memory reference analysis. *International Journal of Parallel Programming*, Jan 2003.

[123] S. Rus, M. Pennings, and L. Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 263–273, 2007.

[124] J. H. Salz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, 1991.

[125] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In *ASPLOS-XV: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 311–322, 2010.

[126] V. A. Saraswat, V. Sarkar, and C. von Praun. X10: concurrent programming for modern architectures. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 271–271, 2007.

[127] M. Shah, J. Barreh, J. Brooks, R. Golla, and G. Grohoski. UltraSPARC T2: A highly-treaded, power-efficient, SPARC SOC. *Solid-State Circuits Conference*, Jan 2007.

[128] Z. Shen, Z. Li, and P.-C. Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356 –364, Jul 1990.

[129] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425, 1995.

[130] SPEC: Standard Performance Evaluation Corporation. SPEC CPU2000, 2001. `http://www.spec.org/cpu2000/docs/`.

[131] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, 1994.

[132] Stanford SUIF Compiler Group. SUIF: A Parallelizing & Optimizing Research Compiler. Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University, 1994.

[133] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA '08: Proceedings of Fourth International Symposium on High-Performance Computer Architecture*, pages 2 –13, 1998.

[134] A. M. Suleman, O. Mutlu, J. A. Joao, Khubaib, and Y. N. Patt. Data marshaling for multi-core architectures. In *ISCA '10: Proceedings of the 37th annual international Symposium on Computer Architecture*, pages 441–450, 2010.

[135] M. Suleman, M. Qureshi, Khubaib, and Y. Patt. Feedback Driven Pipelining. In *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 147–156, 2010.

[136] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 50–57, 2003.

[137] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 441–452, 2009.

[138] The Eclipse Foundation. Eclipse. `http://www.eclipse.org/projects/project_summary.php?projectid=eclipse`.

[139] The Embedded Microprocessor Benchmark Consortium (EEMBC). EEMBC 2.0. `http://www.eembc.org`.

[140] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. *Lecture Notes in Computer Science*, 2304:179–??, 2002.

[141] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, 2007.

[142] C. Tian, M. Feng, N. Vijay, and G. Rajiv. Copy or discard execution model for speculative parallelization on multicores. In *MICRO-41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–341, 2008.

[143] TIS Committee. *Executable and Linking Format (ELF) Specification Version 1.2*, 1995.

[144] G. Tournavitis and B. Franke. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *PACT '10: Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques*, pages 377–388, Sep 2010.

[145] G. Tournavitis, Z. Wang, B. F. Franke, and M. F. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 177–187, 2009.

[146] R. H. I. Ulrich Drepper. ELF Handling for Thread-Local Storage, Version 0.20, 2005.

[147] S. Unger and F. Mueller. Handling irreducible loops: optimized node splitting versus DJ-graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):299–333, 2002.

[148] N. Vachharajani. *Intelligent Speculation for Pipelined Multithreading*. PhD thesis, Princeton University, 2008.

[149] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59, 2007.

[150] K. Vallerio and N. Jha. Task graph extraction for embedded system synthesis. In *Proceedings of the 16th International Conference on VLSI Design*, pages 480 – 486, 2003.

[151] R. Van der Wijngaart and H. Jin. NAS Parallel Benchmarks, Multi-Zone Versions. 2003.

[152] L. Van Put, D. Chanet, B. De Bus, B. De Sutler, and K. De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology*, pages 7 –12, 2005.

[153] H. Vandierendonck, S. Rul, and K. D. Bosschere. The paralax infrastructure: Automatic parallelization with a helping hand. In *PACT '10: Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques*, 2010.

[154] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28tflops network-on-chip in 65nm cmos. *ISSCC 2007: Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, pages 98 – 589, Jan 2007.

[155] Vector Fabrics. Using vfAnalyst® to understand program behavior. `http://www.vectorfabrics.com/images/uploads/products/vfAnalyst_Whitepaper_A4.pdf`.

[156] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *CGO '07: Proceedings of the fifth annual IEEE/ACM international symposium on Code generation and optimization*, pages 209–217, Mar 2007.

[157] S. wei Liao, A. Diwan, R. Bosch, Jr, A. Ghuloum, and M. Lam. Suif explorer: an interactive and interprocedural parallelizer. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, Aug 1999.

[158] R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SC '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.

[159] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, USA, 1990.

[160] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, 2005.

[161] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *HPCA '08: 14th IEEE International Symposium on High Performance Computer Architecture*, Jan 2008.

[162] H. Ziegler and M. Hall. Evaluating heuristics in automatically mapping multi-loop applications to fpgas. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 184–195, 2005.

[163] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 2–13, 2001.