THE UNIVERSITY *of* EDINBURGH

Automated Detection of Structured Coarse-Grained Parallelism in Sequential Legacy Applications

Tobias J.K. Edler von Koch

Doctor of Philosophy

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2014

Abstract

The efficient execution of sequential legacy applications on modern, parallel computer architectures is one of today's most pressing problems. Automatic parallelization has been investigated as a potential solution for several decades but its success generally remains restricted to small niches of regular, array-based applications. This thesis investigates two techniques that have the potential to overcome these limitations.

Beginning at the lowest level of abstraction, the binary executable, it presents a study of the limits of *Dynamic Binary Parallelization* (DBP), a recently proposed technique that takes advantage of an underlying multicore host to transparently parallelize a sequential binary executable. While still in its infancy, DBP has received broad interest within the research community. This thesis seeks to gain an understanding of the factors contributing to the limits of DBP and the costs and overheads of its implementation. An extensive evaluation using a parameterizable DBP system targeting a CMP with light-weight architectural TLS support is presented. The results show that there is room for a significant reduction of up to 54% in the number of instructions on the critical paths of legacy SPEC CPU2006 benchmarks, but that it is much harder to translate these savings into actual performance improvements, with a *realistic* hardware-supported implementation achieving a speedup of 1.09 on average.

While automatically parallelizing compilers have traditionally focused on data parallelism, additional parallelism exists in a plethora of other shapes such as task farms, divide & conquer, map/reduce and many more. These *algorithmic skeletons*, i.e. high-level abstractions for commonly used patterns of parallel computation, differ substantially from data parallel loops. Unfortunately, algorithmic skeletons are largely informal programming abstractions and are lacking a formal characterization in terms of established compiler concepts. This thesis develops compiler-friendly characterizations of popular algorithmic skeletons using a novel notion of commutativity based on liveness. A hybrid static/dynamic analysis framework for the context-sensitive detection of skeletons in legacy code that overcomes limitations of static analysis by complementing it with profiling information is described. A proof-of-concept implementation of this framework in the LLVM compiler infrastructure is evaluated against SPEC CPU2006 benchmarks for the detection of a

typical skeleton. The results illustrate that skeletons are often context-sensitive in nature.

Like the two approaches presented in this thesis, many dynamic parallelization techniques exploit the fact that some statically detected data and control flow dependences do not manifest themselves in every possible program execution (*may*-dependences) but occur only infrequently, e.g. for some corner cases, or not at all for any legal program input. While the effectiveness of dynamic parallelization techniques critically depends on the absence of such dependences, not much is known about their nature. This thesis presents an empirical analysis and characterization of the variability of both data dependences and control flow across program runs. The cBench benchmark suite is run with 100 randomly chosen input data sets to generate whole-program control and data flow graphs (Cdfgs) for each run, which are then compared to obtain a measure of the variance in the observed control and data flow. The results show that, on average, the cumulative profile information gathered with at least 55, and up to 100, different input data sets is needed to achieve full coverage of the data flow observed across all runs. For control flow, the figure stands at 46 and 100 data sets, respectively. This suggests that profile-guided parallelization needs to be applied with utmost care, as misclassification of sequential loops as parallel was observed even when up to 94 input data sets are used.

Acknowledgements

First and foremost I would like to thank my *Doktorvater*, Björn Franke, who supported and inspired me throughout this PhD. I could not have wished for a better supervisor. I would like to thank all of my colleagues past and present in the PASTA office and CArD research group who made the last four years in the School of Informatics such an enjoyable experience. I'm equally grateful to my wonderful friends in the Edinburgh CSU, the chaplains at St Albert's, and the members of the Rotaract Club of Edinburgh.

I would like to thank my thesis examiners, Murray Cole and Sebastian Hack, for their inspiring questions and helpful suggestions. Jeremy Singer and Tom Spink read drafts of this thesis and gave valuable feedback.

There are many people who selflessly supported me on the journey that ultimately led to this PhD thesis. It all began with Wilhelm Kaltenstadler, who one day declared that "this boy needs a computer" and gave me an Intel 8088 when I was 8 years old. In my school years, at a time when the internet was still a novelty to most, I found a welcoming community of hackers in the Bürgernetz Pfaffenhofen computer club that gave me the opportunity to learn, develop, and apply my skills. Hans-Georg Haehnel and Martin Niemer were irreplaceable mentors during this time and beyond. Later, my university studies were generously financially supported by the people of Scotland, the benefactors of the University of Edinburgh, Intel Corporation and Barclays Capital. Qualcomm Innovation Center and Freescale Semiconductor gave me the opportunity to experience industrial research in practice during two internships.

I owe everything to my parents, to whom this thesis is dedicated, who have always supported and encouraged me at every step of the way. And to Lorena, who has brought more joy to my life than I can ever thank her for.

AMDG

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.

(*Tobias J.K. Edler von Koch*)

To my parents.

Publications

The following refereed conference papers (in reverse chronological order) have been published during the course of this PhD. These form the basis for parts of this thesis as indicated.

- **Tobias J.K. Edler von Koch**, Björn Franke, Pranav Bhandarkar, and Anshuman Dasgupta. "Exploiting Function Similarity for Code Size Reduction." In: *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'14),* Edinburgh, UK, June 2014.

- **Tobias J.K. Edler von Koch** and Björn Franke. "Variability of Data Dependences and Control Flow." In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14),* Monterey, CA, March 2014
  — Chapter 5 is based on this publication.

- **Tobias J.K. Edler von Koch** and Björn Franke. "Limits of Region-Based Dynamic Binary Parallelization." In: *Proceedings of the 9th Annual ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'13),* Houston, TX, March 2013
  — Chapter 4 is based on this publication.

- Oscar Almer, Igor Böhm, **Tobias J.K. Edler von Koch**, Björn Franke, Stephen Kyle, Volker Seeker, Christopher Thompson, and Nigel Topham. "Scalable Multi-Core Simulation Using Parallel Dynamic Binary Translation." In: *Proceedings of the 11th IEEE International Conference on Embedded Computer Systems: Architectures, Modelling, and Simulation (SAMOS'11),* Samos, Greece, July 2011
  — Section 3.2 contains background material from this publication.

- Igor Böhm, **Tobias J.K. Edler von Koch**, Stephen Kyle, Björn Franke and Nigel Topham. "Generalized Just-In-Time Trace Compilation using a Parallel Task Farm in a Dynamic Binary Translator." In: *Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation (PLDI'11),* San Jose, CA, June 2011
  — Section 3.2 contains background material from this publication.

- **Tobias J.K. Edler von Koch**, Igor Böhm, and Björn Franke. "Integrated Instruction Selection and Register Allocation for Compact Code Generation Exploiting Freeform Mixing of 16- and 32-bit Instructions." In: *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10),* Toronto, Canada, 2010.

Contents

Chapter 1

Introduction

"The free lunch is over" concluded a seminal article by Herb Sutter [119] in 2005. Until then, software developers could rely on every new generation of processors delivering ever-increasing performance without changing the underlying sequential model of computation. Computer architects were able to leverage the exponential growth of transistors per die (referred to as Moore's Law [111]) to improve performance through better microarchitectures, better caches, and above all higher clock speeds. Existing software would in turn simply run faster on newer processors with every generation (see SPECint performance 1990-2005 in Figure 1.1).

Around the beginning of the last decade, it became apparent that this 'free' growth of performance would come to an end. As can be seen in Figure 1.1, Moore's Law continued to be in effect after 2000 but a number of physical limitations put a stop to attempts at scaling clock speeds beyond 4GHz. Computer architects faced rapidly growing transistor power leakage and unacceptably high thermal output [13] and ultimately had to abandon existing microarchitectural designs, such as Intel's NETBURST microarchitecture [115].

This crisis led to a complete paradigm shift in the industry. Rather than delivering ever-increasing performance by improving sequential processing speed, the focus shifted to exploiting parallelism as the new driver for performance. The replication of processing elements – either within a processor core for simultaneous multi-threading (SMT), or in multiple cores on the same die in the form of a chip multiprocessor (CMP) – allowed the beneficial use of increased transistor counts while avoiding the problems encountered with previous, single-threaded designs.

Since then, the shift to multi-core architectures has become so pervasive that it affects all computing domains from high performance computing to embedded

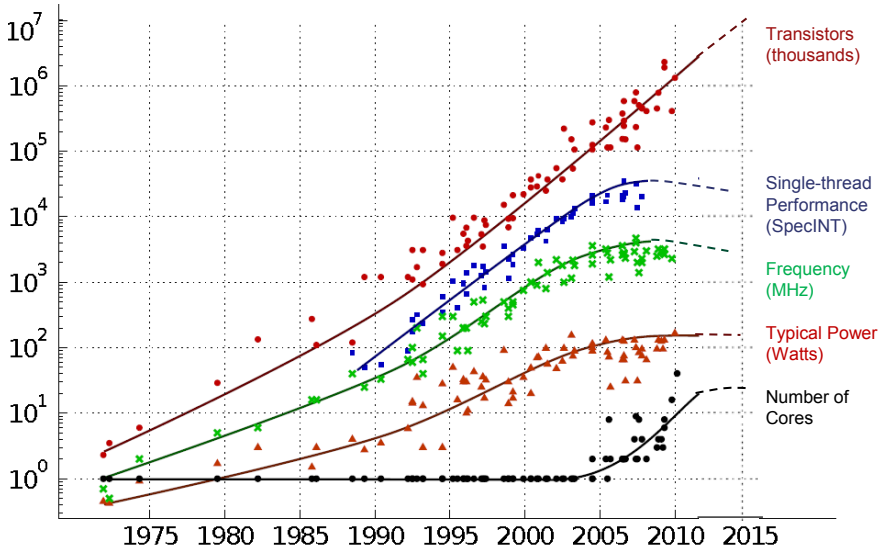


Figure 1.1: Microprocessor trend data for the years 1970-2010. The chart shows the exponential growth of transistors per die (Moore's Law) and the flattening of single-thread performance, clock frequency, and power consumption trends after 2005. This coincides with a notable increase in cores per die. Source of illustration: C. Moore (AMD) [83].

systems. High-end mobile phones now commonly offer four or eight processor cores. Architectures with a thousand cores are on the horizon [20].

1.1 The Problem: Legacy Code in the Multi-Core Era

The advent of the multi-core era spurred a flurry of research into new parallel programming models, languages, and architectures (see our discussion of related work in Chapter 2). The considerable amount of resources invested into this endeavor by both industry and academia might give an impression of firm control over this revolution. One might expect that given those new techniques the performance of applications would continue to increase exponentially as it did during the uniprocessor era.

Reality paints a very different picture. The amount of **legacy software** still in use is considerable [38]. Most of it was written for sequential uniprocessor architectures and hence cannot benefit from current advances in multi-processor technology. The reasons for the prevalence of legacy software lie both in the complexity and importance of current software. It is often too costly to rewrite existing

code. The original authors may not be available to carry out this work. New code needs to be tested extensively and cannot afford to introduce bugs. Hence, legacy code continues to be reused even in new projects (e.g. recent reports of COBOL applications being moved to cloud computing platforms [32]). Virtualization technology, which allows legacy systems to be run in virtual containers on more recent hardware, exacerbates this trend [87]. Consequently, there is often a desire to handle ever-increasing amounts of data with existing codes and certainly a need to maintain current performance levels on future hardware. The latter point is not self-evident, since *single-core* performance on future architectures may well stagnate or even decrease while *multi-core* performance continues to increase (see Figure 1.1).

Even where code is already written for parallel execution, the continuous evolution of parallel architectures and the resulting lack of portability of hand-tuned codes leads to a related problem of **parallel legacy code**. If a program is parallelized by hand, this is usually done with a particular target architecture and programming model in mind. Consequently, such code may perform poorly on future architectures that may require a different granularity or type of parallelism. For instance, message passing (MPI) served as the dominant parallel programming model across the high-performance computing industry for many years; now the use of GPUs in heterogeneous architectures has become prevalent, but these require entirely different programming concepts (OPENCL, CUDA). While it raises primarily the question of finding the best *mapping* of parallel programs to particular architectures, any mapping strategy nonetheless presupposes the availability of sufficient amounts of parallelism at various levels.

In summary, we are faced with two conflicting trends: the paradigm shift of computer architecture towards exploiting parallelism as the main driver of performance; and the continuing prevalence of legacy software that is ill-suited to leverage the full capabilities of such architectures.

1.2 The Solution: Automatic Parallelization

An obvious solution to this problem would be a system that takes an existing sequential application as its input and automatically transforms it into an equivalent application that exploits the parallel capabilities of the target architecture wherever this improves the application's performance. This process is commonly

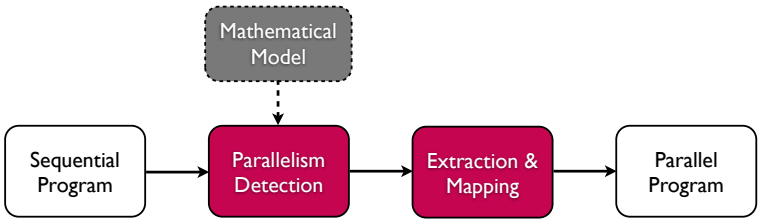
known as **automatic parallelization** and has been under active research for several decades [70]. This section gives a necessarily brief overview of what is in fact a vast research area.

It is important to note that the process of automatic parallelization actually comprises two distinct steps: the **detection** of parallelism and its **extraction and mapping** onto a specific target architecture.

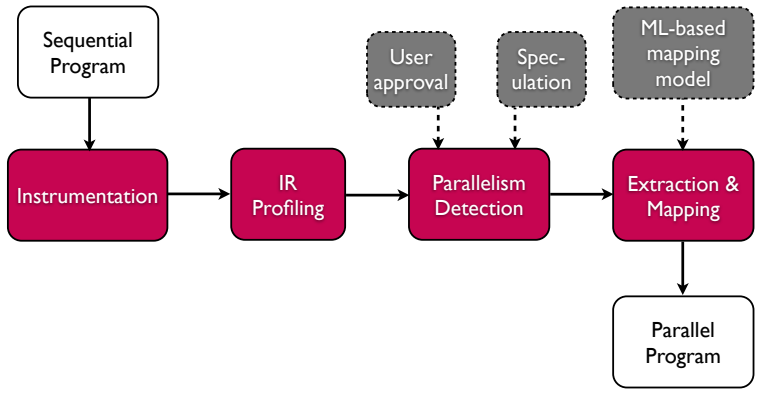
The **detection stage** aims to identify sections of code that can be executed in parallel. These sections could come in various shapes, such as loops, regions, or more complex parallel patterns. The **granularity** of parallelism is an important factor in this context. It refers to the size of the individual work units extracted for parallel execution. Architectures like graphical processing units (GPUs) generally require fine-grained work units with a high degree of parallelism to deliver best performance, while conventional chip multiprocessors (CMP) deliver better results with coarser granularity. This is a result of the costs of parallel execution, such as thread creation, synchronization, and communication overheads, which vary significantly between different architectural concepts.

In the subsequent **extraction and mapping stage**, semantics-preserving transformations are first applied to prepare the detected parallel sections for actual parallel execution. Examples for such transformations are the privatization of variables or the insertion of synchronization primitives. The extracted parallelism is then *mapped* onto the target architecture by assigning the various parallel sections to the parallel processing elements of the target architecture. This can be done statically (at compilation time) or dynamically (at runtime). Mapping has both a spatial (where to execute) and a temporal (in which order) dimension. It is a complex process in itself since it is generally impossible to find an optimal mapping in polynomial time. Various heuristics, profiling information, and machine learning techniques are commonly employed to discover the best mapping [125].

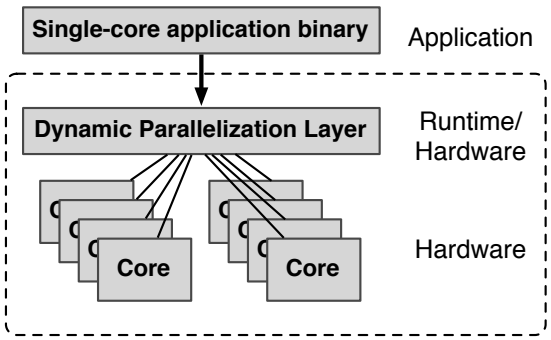
While often described as a compiler technique, automatic parallelization can be implemented at any point in the spectrum that comprises hardware, low-level firmwares, compilers, and runtime systems. It can be applied **statically** (before the application is executed) [50] or **dynamically** at application runtime [34]. It may be **profile-guided** in that it could rely on knowledge about program behavior gained from actual application runs [125]. **Speculation** may be used in cases where it is impossible to determine ahead of time whether a parallelization strategy is legal, i.e. preserving original program behavior [99]. Speculation is often employed



(a) Static parallelization.



(b) Profile-guided parallelization.



(c) Dynamic binary parallelization.

Figure 1.2: Three typical approaches to automatic parallelization, representing systems like [50, 125, 53].

in combination with **transactional memory** to roll back the program state to a known 'good' state should mis-speculation occur [51]. Alternatively, **user input** in the form of code annotations [129] or an interactive parallelization approach [74] may fill the gaps in analysis.

To illustrate the design of automatic parallelization systems more concretely, we show three examples for typical parallelization workflows in Figure 1.2. (a) shows the classic, static approach where the source code is analyzed and transformed in the compiler, like in [50]; a mathematical model of dependences, such as the

polyhedral model, is used to detect parallelism. In (b), an instrumented run of the application provides profiling information to the parallelism detection stage; the discovered parallelism is then extracted and mapped to the target architecture using a model obtained by machine learning. This is the approach taken, e.g. in [125]. Finally, (c) is a dynamic binary parallelization scheme; an unmodified application binary runs on top of a transparent parallelization layer which discovers parallelism on-the-fly and maps it onto a multi-core architecture during execution. An example for this type of system is [53] and our study in Chapter 4.

1.3 The Problem with The Solution

As much as it sounds like a panacea to solve the problem of porting legacy code to multi-core architectures, the history of automatic parallelization has turned out to be a history of broken promises, documented in e.g. [3, 82, 125]. Up to this day and despite decades of intensive research automatically parallelizing compilers have *failed* to deliver except for small niches of applications involving regular array-based computations [90], where techniques based on mathematical models (e.g. the polyhedral model [14]) have been successfully applied.

There are a number of reasons for this:

1. **Theoretical limitations:** The underlying problem of proving the absence of parallelism-inhibiting dependences is generally undecidable [71, 103].

2. **Static vs. dynamic analysis:** Much of the prior work on automatic parallelization relies purely on static analyses. Such analyses can only compute rough approximations to the actual dependence relations and are necessarily conservative to maintain correctness. Information about actual program behavior obtained at runtime would complement such analyses and compensate for their shortcomings.

3. **Structure and granularity of parallelism:** Existing parallelizing compilers generally focus solely on data-parallel loops, ignoring other patterns of parallelism.

4. **Disconnection from software engineering practice:** Current techniques are driven by mathematical models of dependences which aim to respect all

dependences in the sequential code even if those are merely implementation artifacts. In contrast, programmers do not use mathematical models to parallelize sequential code. Instead, manual parallelization is usually driven by algorithmic insights and knowledge of the the data structures used by the application. The programmer's goal is to maintain correctness of output rather than meticulous replication of intermediate program behavior.

1.4 Contributions

In response to these shortcomings, the present thesis makes a number of distinct but closely related contributions to the field of automatic parallelization. We take a vertical approach in our study. Beginning at a very low level of abstraction, we first establish the limits of dynamic binary parallelization, which aims to extract parallelism from sequential binaries at runtime, and illustrate benefits of dynamic information. This leads to a study of the variability of data dependences and control flow which are key factors in the profile sensitivity of dynamic parallelization approaches. We conclude at a much higher level of abstraction with a novel characterization of algorithmic skeletons in terms of well-known compiler concepts for their automated detection in application source code.

Among the main contributions of this thesis are:

1. The introduction of a generic cost-adjustable dynamic binary parallelization (DBP) scheme that allows the modelling of a variety of possible implementations (Chapter 4),

2. the study of the limits of DBP using this model, demonstrating the difficulty of a practical implementation of the technique, the importance of dynamic information for parallelization, and the need for higher-level approaches to address its shortcomings (Chapter 4),

3. the development of a simple, yet powerful profiling-based analysis to capture data dependences and control flow for program executions with different input data sets and its use to analyze the variability of dependences and control flow in a benchmark suite given a large number of input datasets (Chapter 5),

4. the introduction of a new notion of commutativity for regions of code based on liveness and its use to characterize a number of popular algorithmic skeletons (Chapter 6), and

5. the combination of static analysis and profiling into a practical methodology for detecting parallelizable algorithmic skeletons in real-world legacy applications (Chapter 6).

1.5 Thesis Structure

The remainder of this thesis is structured as follows.

Chapter 2 presents the related work. We introduce and discuss significant prior work in the fields of dynamic binary parallelization, the variability of data dependences and control flow, algorithmic skeletons, and the detection of patterns in code for parallelization.

Chapter 3 provides background on the infrastructure used for our research. This includes the ArcSim simulation environment, the Llvm compiler toolchain, various benchmark suites, and our source code instrumentation and profiling techniques.

Chapter 4 investigates the limits of dynamic binary parallelization (Dbp), an automatic parallelization technique targeting existing application binaries. We perform an extensive evaluation using a parameterizable Dbp system targeting a Cmp with light-weight architectural Tls support and discuss the implications of the results for the practical feasibility of the technique. This chapter is based on work published in [34].

Chapter 5 analyzes the variability of data dependences and control flow in typical benchmarks and discusses the implications of such variations on profile-guided automatic parallelization. We run a suite of benchmarks with 100 randomly chosen input data sets and record complete control and data flow traces. Based on these traces, we build a whole-program control and data flow graph (Cdfg) for each run and compare the resulting graphs to obtain a measure of the variance in the observed control and data flow. The results provide an empirical foundation for existing work in profile-guided automatic parallelization. This chapter is based on work published in [35].

Chapter 6 proposes a compiler-friendly characterization of algorithmic skeletons using a novel notion of commutativity based on liveness. We demonstrate how this can be applied to characterize a number of popular skeletons. We also introduce a hybrid static/dynamic analysis that serves as a framework to identify potential candidate skeletons in benchmark code.

Chapter 7 concludes this thesis by summarizing the main contributions, providing a critical analysis of our work and discussing future directions.

1.6 Summary

This chapter has introduced the thesis, outlining the key problems and challenges. It discusses why automatic parallelization techniques have failed to live up to expectations and proposes the use of dynamic information as well as a focus on the detection of coarser-grained patterns as a future direction. The key contributions of this work have been listed and an outline of the thesis described. The next chapter discusses the related work in the area of automatic parallelization.

Chapter 2

Related Work

2.1 Introduction

This chapter presents a critical review of prior work most relevant to this thesis. The primary objective of this chapter is to compare and contrast the approaches and techniques we propose in this thesis to the existing body of work. The literature on automatic parallelization is vast and goes back several decades [70]. In line with the structure of this thesis, we take a vertical approach in the discussion of this work.

Beginning at a very low level of abstraction, the binary executable, we discuss prior work on dynamic binary parallelization in Section 2.2. This forms the background for Chapter 4. This is followed by a review of the literature on the sensitivity of profiling information to input data in Section 2.3, which is relevant to our study in Chapter 5. Finally, we discuss the related work on algorithmic skeletons, pattern detection in source code, and commutativity analysis in Section 2.4. This serves as the basis for our work in Chapter 6.

2.2 Dynamic Binary Parallelization

Unlike its more mature siblings *Dynamic Binary Instrumentation* (DBI) [76] and *Dynamic Binary Translation* (DBT) [5], *Dynamic Binary Parallelization* is an emerging field and has only recently found attention [53, 29, 137, 139, 134] within the academic community.

The theoretical availability of significant amounts of parallelism in single-threaded binaries is known from studies such as Vachharajani et al. [127], which

examines the scalability potential for exploiting the parallelism in single-threaded applications on chip multi-processor (CMP) platforms. The paper explores the total available parallelism in unmodified sequential applications and then examines the viability of exploiting this parallelism on CMP machines. It shows that it is possible to achieve as much as a 200% speedup by parallelizing unmodified applications, but 8-16 cores will be needed to unlock this parallelism. The paper does not address the limits of DBP, though.

Thread-level speculation (TLS) has been proposed as a way of extracting parallelism speculatively from legacy applications, but generally relies on complex additional hardware and the availability of source code for recompilation using a TLS-aware compiler. Its limits are evaluated in [58]. While targeting a similar problem, TLS is more concerned with the parallelization of loops and functions that are mostly independent, but occasionally exhibit sequentializing data dependences. The MULTISCALAR architecture [113] represents an early implementation. JRPM [24] implements TLS-based dynamic parallelization for the Java virtual machine. Dou and Cintra [31] present a compile-time cost model for TLS that aims to predict the expected speedup for a given candidate speculative section. A study involving manual parallelization of the SPECCPU2006 benchmarks for a TLS platform has been conducted in [89]. This study confirms the "conventional wisdom" that these benchmarks are difficult to parallelize using traditional parallelizing compilers (Intel's ICC compiler in this case).

Current DBP approaches span the entire range from solutions implemented in hardware to those relying entirely on software runtime systems.

Marcuello and González [77] propose the Clustered Speculative Multithreaded Microarchitecture, a pure hardware implementation of DBP, which extracts speculative threads from single-threaded applications at runtime to use otherwise idle resources of the processor. The system speculates on control flow, dependences between threads, and data values using additional hardware on top of a TLS platform. The paper reports an average of 1.6 parallel threads during the execution of the SPECINT95 benchmark suite.

Wang et al. [132] introduce a DBP approach based on program slicing. The speculative slicing algorithm and various other program transformations to expose parallelism are implemented in a runtime system. However, additional hardware for parallel slice execution is required on top of the already complex extensions for speculation support.

Krishnan et al. [68] use an initial offline stage to identify and annotate parallel work units and dependences between them. The modified binaries are executed on a CMP with hardware speculation support implementing a complex register communication scheme based on synchronizing scoreboards. Kotha et al. [67] and Pradelle et al. [95] rely entirely on off-line preprocessing of binaries to extract parallel loops. Zilles and Sohi [144] use a 'program distiller' to produce an approximate, but faster variant of the binary. This distilled program is used for the prediction of input values for tasks during speculative execution. Our study differs in that it does not rely on a static transformation stage which would be limited in its ability to extract control and data flow information.

RASP [53] is a runtime system based on a dynamic binary translator from x86 to RISC. It leverages idle cores in a CMP to analyze, optimize, and speculatively parallelize sequential programs at runtime and thus enables a collection of simpler cores to achieve sequential performance on par with a significantly more complex core without any need for recompilation or hardware support beyond transactional memory. We compare this approach directly to ours in Section 4.4.6 of Chapter 4.

DeVuyst et al. [29] propose a similar runtime system, but their study does not account for the possibility of mis-speculation and the technique is limited to loop parallelization. A recent paper by Yang et al. [137] re-evaluates the feasibility of DBP using a trace parallelization process. This work remains on a proof-of-concept level and neither does it develop a realistic implementation nor a sound limit study.

Yardımcı and Franz [139] dynamically recompile binaries to generate parallelized code for frequently executed code regions. The system is entirely software-based and relies on a prior static analysis stage. It remains open whether there are limits to this approach and how it scales beyond dual-core configurations.

Wentzlaff and Agarwal [134] present a DBT-based simulator capable of emulating a superscalar x86 processor on the tile-based RAW architecture. Parallelization is achieved by implementing the superscalar pipeline stages on distinct parallel processing elements of the tiled architecture. The authors report an overall slowdown for the SPECINT2000 benchmarks.

All of these approaches are specific to a particular target system and focus on improvements resulting from the application of an isolated technique within the context of a distinct implementation. In Chapter 4 of this thesis, we instead seek to gain an understanding of the general limits of DBP, independent of any particular implementation.

2.3 Sensitivity of Profiling to Input Data

In recent years, the use of dynamic information about actual application behavior has become increasingly prevalent in automatic parallelization as it provides a means of overcoming the limitations of static analysis.

Kim et al. [65] give an overview of profiling-based parallelization approaches. DBP, discussed in the previous section, is another technique motivated by the possibility of extracting greater amounts of parallelism by leveraging dynamic information.

Early work in dynamic dependency analysis by Austin et al. [10] dates back to the 1990s. The primary goal of that work was to characterize the amount and nature of instruction-level parallelism in ordinary programs and to guide the design of processors taking advantage of this type of parallelism.

Despite the widespread use of dynamic data dependence and control flow data, their variability caused by different input data sets and its impact on parallelization has not yet been sufficiently investigated.

Faxén et al. [37] analyze the variability of data dependences in the `403.gcc` benchmark. The study is limited to the one benchmark and does not evaluate the impact of dependence variability on parallelization outcomes.

Liebig et al. [75] present variability-aware analyses, which exploit similarities of software products that belong to a common product line. This, however, is more a problem of scalability of analyses, rather than input sensitivity of dependences.

Fisher and Freudenberger [39] studied the use of branch profiles from previous runs of a program (using different input data sets) as predictors for branch directions in future runs. While they conclude that branches generally take the same directions in different runs of a program, some runs were observed to exercise entirely different program paths.

Wall [131] defines abstract measures that indicate how well profiles, such as basic block counts or number of references to global variables, predict the behavior of applications in future runs. The study compares the accuracy of 'perfect' profiling information - obtained from a previous run with the same input data set - to profiling information obtained with different input data sets. In some cases, profiles obtained with different data sets reach only half of the accuracy of the 'perfect' profile. This implies that there can be significant variations in program behavior depending on the input data set.

Hunter et al. [55] study code coverage and input variability of telecommunication applications, suggesting that input data sets provided with standard benchmarks do not sufficiently exercise control flow paths.

Eeckhout et al. [36] describe methods of exploring the workload space of benchmarks using statistical techniques to select representative input data sets. This provides a method for investigating the impact of input data sets on benchmark behavior, but the paper does not consider the variability of dependences or the effect of different workloads on dynamic parallelization.

A poster abstract by Bhattacharyya [17] presents an outline of a study investigating the impact of inputs on *may*-dependences in the context of TLS, but the work remains at the conceptual level.

The notion of control and data flow coverage is of central importance in the fields of software reliability and testing. An early technique by Rapps et al. [98] for selecting test data using data flow techniques dates back to the 1980s. This is related to dynamic program slicing [66], which is concerned with determining a dynamic slice that contains all statements that actually affect the value of a variable at a program point for a particular execution of the program. While many coverage analyses for program testing have been developed, these do not consider the statistical nature of dependences and control flow.

Berube et al. [16] present a methodology to produce statistically sound combined profiles from multiple runs of a program. This is important for feedback-directed optimization, where branch frequencies determine optimization strategies. In our study in Chapter 5, this is less relevant as we are concerned with dependences preventing parallelization, where a single dependence is enough to enforce sequential execution.

Tallam et al. [120] describe a methodology for the generation and efficient compression of an extended whole program path representation. Their approach allows profile information to be compressed to about 4% original size, but comes with an additional runtime overhead of 20%. SD3 by Kim et al. [65] achieves a significant reduction of profiling overheads by parallelizing dependence profiling. In Chapter 5, we are more concerned with the analysis of profiling information rather than its efficient collection and storage.

2.4 Automated Detection of Algorithmic Skeletons

Our work on the detection of algorithmic skeletons in Chapter 6 draws on prior research in five distinct areas: the concept of algorithmic skeletons, pattern recognition in source code for automatic parallelization, commutativity analysis, liveness of variables, and hybrid static/dynamic analysis techniques. Liveness analysis is a well-known concept in compiler engineering and is discussed in textbooks such as Muchnik [84]. In the following sections, we will review related work in the remaining areas.

2.4.1 Algorithmic Skeletons

The concept of *algorithmic skeletons* was introduced by Cole [26] in the 1980s to abstract commonly-used patterns of parallel computation, communication, and interaction. Cole envisaged the development of libraries that would provide parameterizable implementations – templates – of such patterns. The developer of a parallel program then essentially selects the most appropriate skeleton for the task at hand and inserts the user code into the template. As the orchestration and synchronization of parallel tasks is implicitly defined by the skeleton, programmers are not burdened with such complex and target-dependent aspects of parallelization. The idea has been widely adopted in the parallel programming community, for example, in the shape of Intel's *Threading Building Blocks* (TBB) [102], Google's MAPREDUCE [28], and in a large number of dedicated skeleton frameworks. A detailed survey is given in [47].

More recently, algorithmic skeletons have inspired textbook literature on parallel programming using *parallel design patterns*. Books by Ortega-Arjona [88] and McCool et al. [81] present catalogues of architecture and communication patterns for parallel applications in much the same way as the well-known "Gang of Four" book [43] did for object-oriented software design patterns.

The term *skeleton* is at times also used for the unrelated idea of 'stripping' a program of some of its code to expose an essential core which can be analyzed more easily. It has been used with this meaning in the context of interactive parallelization [1] and benchmark generation [114], but those techniques are not relevant to our approach in this thesis.

Algorithmic skeletons are commonly defined using a graphical representation, a verbal description, pseudo-code notation, or concrete code examples [47]. How-

ever, this is not sufficient to develop compiler analyses capable of detecting algorithmic skeletons in sequential code. In Chapter 6, we therefore introduce a *formal* characterization in terms of concepts defined in compiler theory and commonly available in compiler frameworks.

While no such unified characterization currently exists, isolated techniques for pattern recognition and detection of specific skeletons have been proposed in the literature. The next section gives an overview.

2.4.2 Pattern Recognition for Automatic Parallelization

PARAMAT [62] is an algorithmic pattern recognition framework. Using an hierarchical approach, it discovers instances of frequently occurring programming constructs in source code. The patterns are described using a domain-specific language and matched over the program's abstract syntax tree (AST). The framework was later used by Sarvestani et al. [112] to detect common algorithmic patterns, such as matrix operations or FIR filters, in DSP codes. The analysis is purely static and patterns are expressed at a low, syntactic level without taking into account other compiler analyses, such as alias or commutativity analysis. In practice, this restricts the PARAMAT approach to a pointer-free subset of input languages and leads to a narrow focus on mathematical codes, for which other parallelization techniques have also proven to be effective. Unlike our work, it aims to detect particular idioms, such as matrix multiplication, rather than the more abstract algorithmic skeletons. Nonetheless, this work is somewhat orthogonal to our approach as it could be used as a building block for the static skeleton candidate detection stage of our hybrid analysis infrastructure.

Di Martino and Iannello [79] present the PAP Recognizer, another tool for the detection of algorithmic patterns in code. A detailed comparison between PAP and PARAMAT was published in [80]. Both use a hierarchical detection approach, but differ in significant other aspects, such as the intermediate program representation used and whether pattern recognition is deterministic. PAP uses a plan-based recognizer on the program dependence graph (PDG) and is implemented in Prolog. It is geared towards interactive use and presents the user with all possible matches, some of which may overlap, while in PARAMAT a node can only be matched by one pattern. The PAP approach was later proposed specifically as a method for skeleton recognition in [30] with Divide and Conquer given as an example. The

paper, however, is a mere sketch of a possible implementation and does not provide an empirical evaluation.

XARK [8] is a framework for the detection of computational kernels in source code. It uses Gated Single Assignment (GSA) form as its intermediate representation and matches kernels to strongly connected components (SCCs) in the IR, which can in turn be composed hierarchically to form more complex kernels.

Besides focusing on the detection of specific algorithms rather than generic structures, all of these techniques have in common that they use dependence relations to characterize algorithmic patterns and rely purely on static analysis. In our work in Chapter 6, we show that commutativity is a more powerful concept for the characterization of skeletons and that shortcomings in static analysis can be overcome by taking dynamic information into account.

In the world of functional languages, Scaife et al. [110] describe an auto-parallelizing compiler for Standard ML that exploits the close correspondence between higher-order functions, such as `map` and `fold`, and algorithmic skeletons for automatic parallelization. Functional languages, however, inherently expose a large degree of parallelism so the bigger question addressed by the article is how such parallelism can be exploited profitably. Our approach targets legacy code which is commonly implemented in imperative rather than functional languages.

In the space of general purpose GPU computing, Samadi et al. [108] describe a technique to recognize common patterns (map, scatter/gather, reduction, stencil, etc.) in parallel OPENCL or CUDA code and to transform them into approximate implementations of the same idioms. BONES [85] introduces a C-to-CUDA compiler that extracts skeletons from sequential code based on user annotations. The user manually selects the regions to be parallelized and classifies the algorithm expressed in the code using a complex domain-specific language. In contrast, our approach does not require such annotations.

In the following sections, we review techniques for the detection of specific skeletons.

Task Parallelism

PARADIGM [12] has limited support for the simultaneous exploitation of data parallelism (within functions) and functional parallelism (between functions) for some applications. Such applications can be viewed as a graph composed of a set of data parallel tasks with precedence relationships which describe the functional

parallelism that exists among the tasks. The restricted focus on the function level is a serious limitation, which together with necessarily conservative precedence relationships almost entirely inhibit success in detecting functional parallelism in real-world applications.

MAPS [22] is a tool for profile-guided detection of task parallelism in sequential C programs. In this framework tasks are unusual hybrids of basic blocks and traces. An empirical heuristic is used for clustering blocks considering profitability of the resulting task partitioning. MAPS targets static embedded kernels and a proprietary, low-latency MPSoC platform.

Dünnweber et al. [33] employ conventional loop parallelization and then map the resulting parallelism to a grid-based task farm implementation.

Pipeline Parallelism

The pipeline skeleton applies the well-known computer architecture concept of pipelining to software parallelization. Sequences of code regions represent the stages of a pipeline; parallelism is uncovered by removing stages from the critical path where dependences permit.

Thies et al. [121] describe a framework for the parallelization of applications that exhibit a streaming computation pattern. Their approach requires the manual annotation of pipeline stages.

Speculative decoupled software pipelining (SPECDSWP) [128] parallelizes loops by partitioning their body into fine-grained pipeline stages, some of which may run in parallel in the absence of dependences. Infrequent or easily predictable dependences are speculated. The approach requires support for memory versioning in the underlying hardware.

Tournavitis and Franke [124] describe a profile-guided pipeline extraction technique. Pipeline stages are extracted at multiple levels of loop nests and can be replicated to prevent bottlenecks.

Cordes et al. [27] investigate the extraction of pipeline parallelism in conjunction with mapping to a heterogeneous target architecture.

Divide and Conquer

Freisleben and Kielmann [40] propose a semi-automated technique to parallelize divide-and-conquer algorithms. It requires code annotations by the user to indi-

cate parallelizable functions, synchronization points, input and output parameters, array dimensions, and the problem size. The system maps the annotated code to a task farm implemented on a message-passing parallel architecture.

Rugina and Rinard [106] and Gupta et al. [48] also target divide-and-conquer algorithms. Their techniques are more sophisticated since they do not require user annotations but instead rely on interprocedural pointer analysis and symbolic memory bounds analysis to identify independent function calls in the recursive divide-and-conquer call graph. Both approaches do not exploit commutativity and are limited to algorithms operating over array-like data structures.

2.4.3 Commutativity Analysis

Commutativity as a distinct and more general concept than parallelism was explored by Bernstein as far back as 1966 [15]. The precise definition of commutativity, however, varies between authors.

Two different notions of commutativity supporting parallelization have been suggested, namely *operations-based commutativity*, where commutativity of code regions is based on the commutativity of the operations performed in those regions, and *effects-based commutativity*, where two regions are considered commutative if their effect on the visible program state is identical for any order of execution.

An early notion of *operations-based commutativity* and a suitable analysis was developed by Rinard and Diniz [104, 105]. This analysis views computation as composed of separable operations on objects. It analyzes the program at this granularity to discover when operations commute, i.e. generate the same final result regardless of the order in which they execute. If all of the operations required to perform two given computations A and B commute, then A and B are commutative. In particular, A and B are said to be commutative if *memory contents match exactly* for the candidate functions executed in any order. While this notion of commutativity is powerful, it has significant drawbacks. It is *conservative* as its restriction to exact matches of memory contents rules out valid parallelization opportunities resulting from non-matched memory contents if these are not relevant to the output of the computation. Our use of liveness analysis overcomes this problem. Relying purely on static analysis, Rinard and Diniz require *extensive support for symbolic computation and transformation* in the compiler, which is not currently available in most compiler frameworks. While we exploit available static analyses in our

framework, dynamic information is used to compensate for any limitations. Rinard and Diniz define commutativity as a *context-insensitive* property. In contrast, we are able to detect commutativity even in cases where it depends on the execution context. Finally, the implementation in [104, 105] is restricted to a subset of the C++ programming language and requires the program to be written in strict object-oriented style.

Aleen and Clark [2] propose an alternative definition of *output-based commutativity*. This definition of commutativity is less conservative as it requires *outputs* of a function to be checked only if they are used later on in the program. In fact, it does not require an exact match of the memory contents (between two different execution orders), but classifies a single function as commutative if the candidate function can be executed in an arbitrary order with respect to itself and a later computation, which uses the function output and delivers indistinguishable results either way. Clearly, this analysis is more aggressive as it focuses on values that matter (=outputs). On the surface, this may seem similar to our approach applied to the detection of a single skeleton (commutative calls). However, there are significant differences: (1) We define commutativity as a property of two code regions which may or may not be the same region. The approach in [2] limits it to self-commutativity of a function. This is a serious restriction as it prohibits task parallelism, where parallel tasks can be *any* computation. (2) Aleen and Clark's definition of commutativity as an inherent property of a function is context-insensitive. Some operations commute only under certain conditions; the context-sensitive nature of our approach takes this into account. Context-insensitivity also requires full view of the program as it cannot handle commutative and non-commutative calling contexts of the same function. (3) The results are of limited use for the stated goal of automatic parallelization, because a function may be determined to be commutative, but the program may not actually contain two consecutive independent calls to it. Our technique only considers commutativity that is actually present in the application code. (4) They use an approximate form of static analysis, random interpretation, which has an – albeit quantifiable – margin of error. Reducing this error leads to increased complexity and exacerbates the well-known disadvantages of symbolic execution; our hybrid static/dynamic commutativity analysis approach does not suffer from this problem and can guarantee correctness at least for given program inputs.

Kim and Rinard [63] present a technique to verify commutativity conditions, which are logical formulae that characterize when operations on a linked data structure commute. The conditions, such as "*HashMap.contains*(k_1) commutes with *HashMap.insert*(k_2, v) if $k_1 \neq k_2$", must be provided by the user and are then verified automatically. Exploiting commutativity conditions proven in this way would enable our approach to detect algorithmic skeletons involving operations on linked data structures. While this is not yet part of our implementation, it could be integrated into the detection framework easily. The more difficult aspect, not covered by the paper, is the detection of linked data structures and operations on them in source code in the first place. Furthermore, Kim and Rinard require the data structure implementation to be fully verified which could be envisaged for standard libraries but is unlikely for custom implementations in application code.

ALTER [126] is a framework for loop parallelization using code annotations. Unlike traditional loop parallelization approaches, it allows data dependences to be violated where this does not affect the outcome of the loop. Loop iterations can be annotated as commutative or tolerating stale reads. The annotations can also be inferred using automated testing on input data. The framework relies on a speculative runtime system that implements a variant of software transactional memory. This requires loops to be instrumented in the final parallel program; our approach only uses instrumentation for the analysis stage. We also target more general algorithmic skeletons instead of only DOALL loops.

Code annotations for commutative functions are also proposed in parallelization frameworks by Bridges et al. [21], as well as in GALOIS [69] and PARALAX [129].

A generalized semantic commutativity based programming extension, called *Commutative Set* (COMMSET), and its associated compiler technology are presented in [94]. COMMSET supports pipeline and data parallelism, but not task parallelism. Code annotations enable the programmer to specify commutativity relations between groups of arbitrary structured code blocks. Using this construct, serializing constraints that inhibit parallelization can be relaxed. The key differences to our work are that COMMSET relies on user annotations and provides little guidance to the user where to look for commutativity. It presents all statically discovered dependences, possibly overloading the user with irrelevant information. COMMSET has no built-in notion of algorithmic skeletons, but is restricted to pipelines and traditional data parallelism.

2.4.4 Hybrid Static/Dynamic Approaches for Parallelization

The problem of proving the absence of parallelism-inhibiting dependences is generally undecidable [71, 103]. In order to preserve correctness, static dependence analysis techniques hence often present so-called *may*-dependences, which may or may not manifest themselves during program execution [96]. Similarly, the undecidability of associativity and commutativity analysis in general was proven in [23]. These results establish important theoretical limits on static analyses for parallelization and motivate the use of profiling information in our approach described in Chapter 6.

The use of dynamic information to overcome the limitations of static analyses has not yet been investigated in the context of *commutativity* analysis, but has been the subject of several studies relying on *dependence* analysis for parallelization. In this section, we give a brief overview of such techniques.

Inspector/Executor techniques, e.g. [107, 100], generate inspector code that first analyzes cross-iteration dependences in loops and scheduler/executor code that then schedules and executes the loop iterations in parallel (where possible) using the dependence information extracted by the inspector at runtime.

Multi-versioning approaches, e.g. [86, 97], generate both a sequential and a parallel version for a parallel loop candidate. At run-time, the evaluation of a guarding predicate determines whether the parallel or sequential version of the code is executed. Generation of a suitable predicate can be hard and its evaluation at run-time costly.

Speculative schemes, such as TLS (see above) [99], execute a parallel loop candidate in parallel while simultaneously collecting a (partial) trace of its memory references which could not be conclusively analyzed by the compiler. After the parallel execution, the trace is analyzed and if the optimistic parallel execution is found to be invalid (potential data dependences are uncovered) the loop is re-executed sequentially. The run-time overhead for keeping track of memory references and checking for dependence violations is often prohibitive and, thus, specialized hardware support has been proposed.

Profile-guided approaches, e.g. [10, 37, 143, 125, 65, 64, 60, 130, 109, 142, 141] combine dependence profiling information with static dependence analyses to gain additional information on *may*-dependences. While run-time overhead is avoided, this approach requires an additional profiling step at compile-time and

does not provide safety guarantees, requiring either additional, manual validation or automatically inserted, but costly run-time checks.

SAMBAMBA [116, 117] combines a number of these approaches into a unified framework for automatic parallelization, which includes both the detection and extraction/mapping of parallelism. The approach is centered around a runtime system which dynamically profiles, analyzes, and recompiles the program during execution. Multiple versions of functions, e.g. parallel, sequential, or specialized for a specific execution context, are compiled just-in-time and selected depending on the execution context and profiling information to maximize performance. The framework uses speculative execution to overcome the limitations of the analyses used. The current implementation only supports functional parallelism and does not exploit commutativity. The use of software transactional memory to support speculation in [117] removes the need for specialized hardware, but comes with significant performance penalties. Code annotations are proposed to mitigate these issues. The proposed approach is orthogonal to our work in the sense that skeleton-based parallelism detection could serve as an initial off-line stage that can uncover parallelism at various granularities which can then be exploited profitably by an adaptive runtime system like SAMBAMBA.

2.5 Conclusion

In this chapter, we have presented a detailed overview of prior work relevant to this thesis and contrasted it with our work. We briefly summarize the key differences between our work and previous approaches here.

Dynamic binary parallelization was presented as a promising mechanism for the automatic parallelization of legacy applications in a multitude of previous studies. However, these studies had a narrow focus on particular implementations and did not sufficiently characterize the limits of the technique. In Chapter 4, we overcome these shortcomings by conducting a limit study using a parameterizable cost model of a generic DBP system capable of covering the spectrum from hardware to software-based implementations.

Like DBP, many recent parallelization techniques rely on dynamic data dependence and control flow information. However, there is insufficient empirical evidence on the variability of such information given different input data sets and

its impact on parallelization. The study of the variability of data dependences and control flow in Chapter 5 fills this gap.

Finally, previous work on algorithmic skeletons focused on their definition and the provision of libraries to develop skeleton-based parallel code. Frameworks for generic algorithmic pattern matching and isolated techniques for the detection of individual skeletons have been proposed in the literature. However, there is no unified approach to the detection of algorithmic skeletons in legacy code. In Chapter 6, we introduce a novel characterization of algorithmic skeletons using commutativity and liveness and propose a hybrid static/dynamic framework for their context-sensitive detection. Commutativity analysis has been used for automatic parallelization before, but current approaches rely on symbolic execution and are context-insensitive.

Before presenting these contributions in detail, we discuss some of the infrastructure and tools used in this thesis in the next chapter.

Chapter 3

Infrastructure

3.1 Introduction

In this chapter, we describe the infrastructure and tools used in the remainder of this thesis. These include an instruction set simulator for microarchitectural simulation, code instrumentation techniques to capture program behavior, and two benchmark suites for empirical evaluation.

3.2 ARCSIM Instruction Set Simulator

While the experiments in Chapters 5 and 6 are conducted with native code, our limit study of dynamic binary parallelization in Chapter 4 requires microarchitectural simulation of a hardware model. This section gives a brief overview of the ARCSIM simulation environment that was used for these experiments. A number of contributions to the simulator were made in the course of this work and were published in [19, 4].

ARCSIM is a target adaptable simulator with extensive support of the ARCompact instruction set architecture (ISA) [7], a typical RISC ISA comparable e.g. to ARM. It is a full-system simulator, implementing multiple processors, the memory subsystem (including MMU), and sufficient interrupt-driven peripherals to simulate the boot-up and interactive operation of a complete Linux-based system. The simulator has a very fast and highly-optimized interpreted simulation mode [123]. By using a parallel just-in-time compiler (JIT) to translate frequently interpreted instructions into native code, ARCSIM is capable of simulating applications at speeds approaching or even exceeding that of a silicon implementation while faithfully

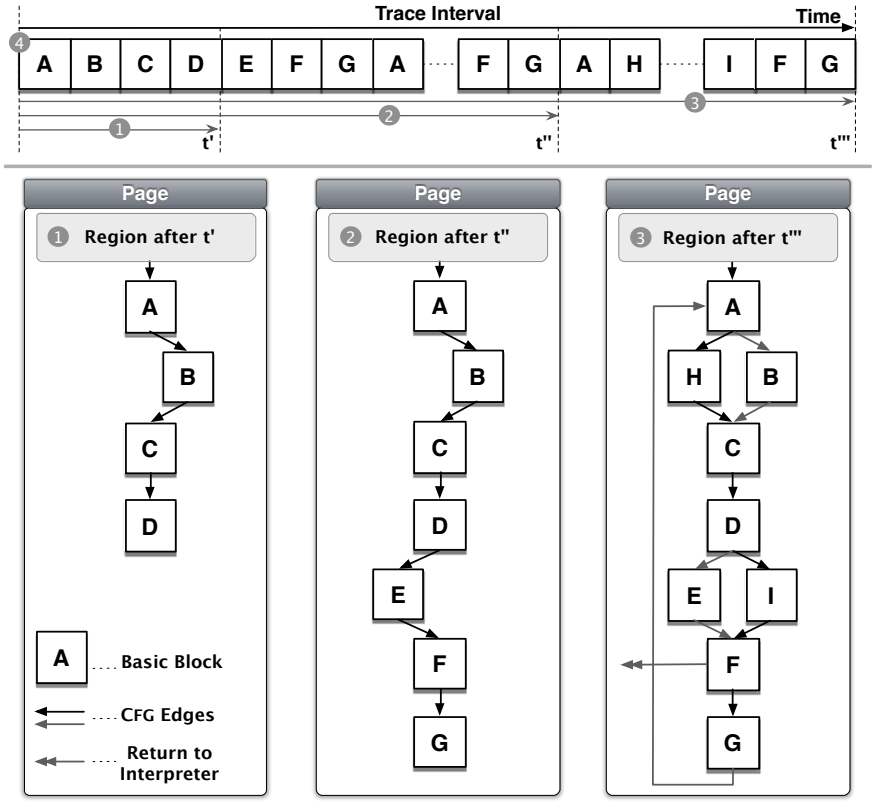


Figure 3.1: Incremental ①②③ control flow graph from sequence of interpreted basic blocks ④ during one trace interval.

modeling the processor's architectural state [19]. ARCSIM simulates the ENCORE microprocessor [122] developed at the University of Edinburgh. ARCSIM's microarchitectural processor model of ENCORE has been verified against a synthesizable RTL implementation. ENCORE has been successfully taped out to silicon, and both ARCSIM and ENCORE have been commercially licensed.

The just-in-time compiler in ARCSIM works at region granularity [19]. Interpreted simulation time is partitioned into trace intervals whose length is determined by a user-defined number of interpreted instructions. In the course of interpreted simulation, ARCSIM incrementally constructs a dynamic control flow graph for each code page (see Figure 3.1). At the end of each trace interval, the regions selected for compilation are dispatched to a decoupled, parallel JIT compiler farm. ARCSIM must maintain an accurate memory model to preserve full architectural observability. Region traces generated during a trace interval are therefore separated at page boundaries, where a page can contain up to 8 KB of target instructions.

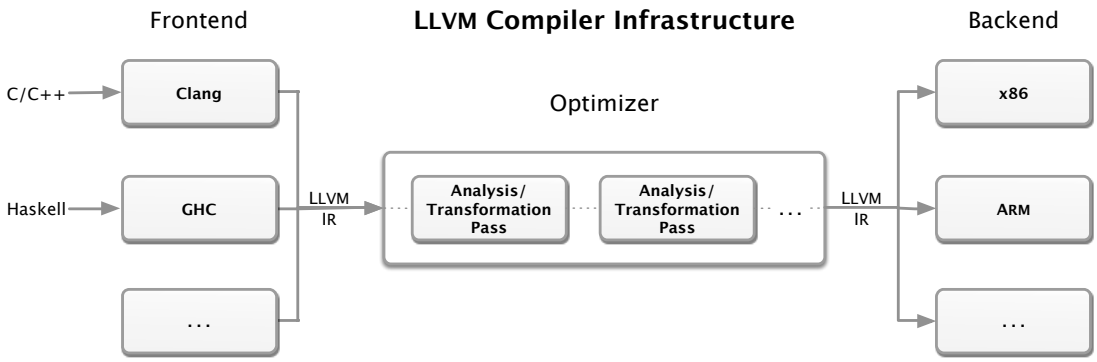


Figure 3.2: Schematic overview of the LLVM compiler infrastructure showing the clear separation of frontend, target-independent analysis and transformation passes, and the backend. LLVM IR is used as a target-independent program representation throughout the optimization stage.

We extended and repurposed this incremental CFG construction infrastructure to build a parameterizable model for region-based dynamic binary parallelization (DBP). Instead of dispatching them for JIT compilation, regions are dynamically partitioned into smaller units which are then speculatively executed in parallel. Our system allows the modelling of thread management overheads (spawning, setup, commit, and invalidation) using parameterized unit costs on top of ARCSIM's multiprocessor simulation capabilities [4]. The DBP model is described in detail in Chapter 4.

3.3 LLVM Compiler Infrastructure

Throughout this thesis, we make extensive use of the LLVM compiler infrastructure as the implementation framework for our compiler-based instrumentation and analyses. The LLVM compiler infrastructure [72] began its life as part of a Master's thesis by Chris Lattner at the University of Illinois at Urbana-Champaign. It has since developed into a mature compiler framework highly regarded for its modularity, extensibility, and permissive open-source license. LLVM is widely used both in academia and industry, where it now serves as the default compiler for all of Apple's products and the Sony PlayStation 4, among others.

Figure 3.2 gives an overview of its architecture. Notable features include the clear separation between the language frontends, the target-independent optimizer, and the target-specific backends; the implementation of the optimizer as

a sequence of transformation and analysis passes; as well as the use of *one single* target-independent program representation, LLVM intermediate representation (IR), throughout the optimization stage.

Optimization passes – regardless of whether they are analyses or transformations – are implemented as C++ classes and can target individual basic blocks, loops, functions, or the entire module. They are scheduled by a pass manager based on dependences expressed in their implementation. For instance, the *Global Value Numbering* transformation requires the results of the *Dominator Tree* and *Alias Analysis* passes. Optimization passes can also be compiled as shared libraries and can be loaded into the compiler dynamically.

Optimization passes operate on a target-independent intermediate representation (LLVM IR). The IR is a strongly typed, RISC-like set of instructions with associated metadata (e.g. for debug information). It represents memory accesses with `load/store` instructions and uses *static single assignment form* (SSA) for register values. Listing 3.3 shows a C program and its corresponding LLVM IR at `-O1` optimization level.

Automatically parallelizing compilers have traditionally been implemented as source-to-source compilers (e.g. SUIF [50]) and therefore operated at a relatively high level of representation. LLVM IR may seem low-level in comparison. Nonetheless, its use has distinct advantages over a source-level representation. It allows us to leverage the rich set of analyses and transformations already implemented in the LLVM infrastructure. Dataflow through SSA registers is made explicit and greatly facilitates analysis. At the same time, crucial information such as types, typed pointer arithmetic, and control flow is retained. By adding additional metadata to instructions in the frontend, it is possible to link IR instructions back to program statements and hence present analysis results in terms of source-level concepts.

3.4 Code Instrumentation Framework

In Chapter 1, we discussed the value of exploiting knowledge about dynamic program behavior in the automatic parallelization workflow. Such knowledge can be gained from actual runs of an application by **instrumenting** the code to record information like control and data flow. This can be done in two ways: either using **dynamic instrumentation** at runtime or **static instrumentation** at compile-time.

```c
#include <math.h>

void map_sqrt(float *A, int N) {
  for (int i = 0; i < N; ++i) {
    A[i] = sqrtf(A[i]);
  }
}
```

(a) C source code of a function

```
define void @map_sqrt(float* %A, i32 %N) {
entry:
  %earlyexit = icmp sgt i32 %N, 0, !dbg !20
  br i1 %earlyexit, label %for.body, label %for.end, !dbg !20

for.body:
  %i = phi i32 [ %inc, %for.body ], [ 0, %entry ]
  %arrayidx = getelementptr inbounds float* %A, i32 %i, !dbg !21
  %arrayval = load float* %arrayidx, align 4, !dbg !21
  %result = call float @sqrtf(float %arrayval), !dbg !21
  store float %result, float* %arrayidx, align 4, !dbg !21
  %inc = add nsw i32 %i, 1, !dbg !20
  %exitcond = icmp eq i32 %inc, %N, !dbg !20
  br i1 %exitcond, label %for.end, label %for.body, !dbg !20

for.end:
  ret void
}

...
!20 = metadata !{i32 4, i32 0, ...}
...
```

(b) LLVM IR after compilation with CLANG -O1 -g (simplified)

Figure 3.3: Comparison of C source code and LLVM IR illustrating the SSA form used for register values and `load`/`store` instructions for pointer-based memory accesses. Corresponding source code statements and IR instructions are shown in the same color. Debug information (indicated by `!dbg`) is attached to the instructions as metadata.

The circumstances of the intended use of instrumentation dictate which of the two forms can be used. It is important to note that instrumentation itself does not affect the behavior of the original code; its sole aim is to gather information, which can then be used for analysis purposes e.g. in automatic parallelization.

Dynamic binary parallelization (Chapter 4) uses dynamic instrumentation. The binary is run in a virtual execution environment (VEE) which tracks all memory

references and control flow and exploits this information directly for automatic parallelization. It would not be possible to use source instrumentation in this context, as the source code is not available during execution and DBP by its nature must remain transparent to the application being parallelized. The binary represents a very low level of abstraction and would generally not be statically analyzable. However, heavy optimizations which obscure original program structures make even dynamic analysis a challenging problem.

Static instrumentation, where applicable, does not suffer from these issues. Code is instrumented during compilation by inserting calls to a runtime library into source code locations where relevant events need to be recorded. This allows the results to be related immediately to the source code. Chapters 5 and 6 make use of static instrumentation.

We have developed a static instrumentation framework based on LLVM. It is implemented as a module pass at LLVM IR level and is capable of instrumenting control flow (function calls, loop iterations, and other branches) as well as memory references (reads, writes, and allocations). Code is instrumented by traversing the IR of a module and marking relevant IR nodes with function calls to a runtime library. Pertinent information, e.g. the address read or written, the basic block branched to, or the function being called, and – crucially – a reference to the LLVM IR node are passed as arguments to these calls. The node references can in turn be linked back to the source code. It is thus possible to later exploit the resulting dynamic information both directly in the compiler and in interactive environments, such as an IDE. Figure 3.4 shows the instrumentation process in detail.

We have created two runtime libraries that implement the instrumentation hooks:

- CDFGPROFILER produces a trace of dynamic control flow and memory references to construct a complete control and data flow graph (CDFG) of the application. This methodology and its use for a large-scale study on the variability of data dependences and control flow is discussed in detail in Chapter 5. Figure 3.4 gives an example of its use.

- COMMPROFILER uses the instrumentation to profile liveness of memory locations and to determine commutativity of regions in a candidate skeleton instance. The approach is described in detail in Chapter 6. In comparison

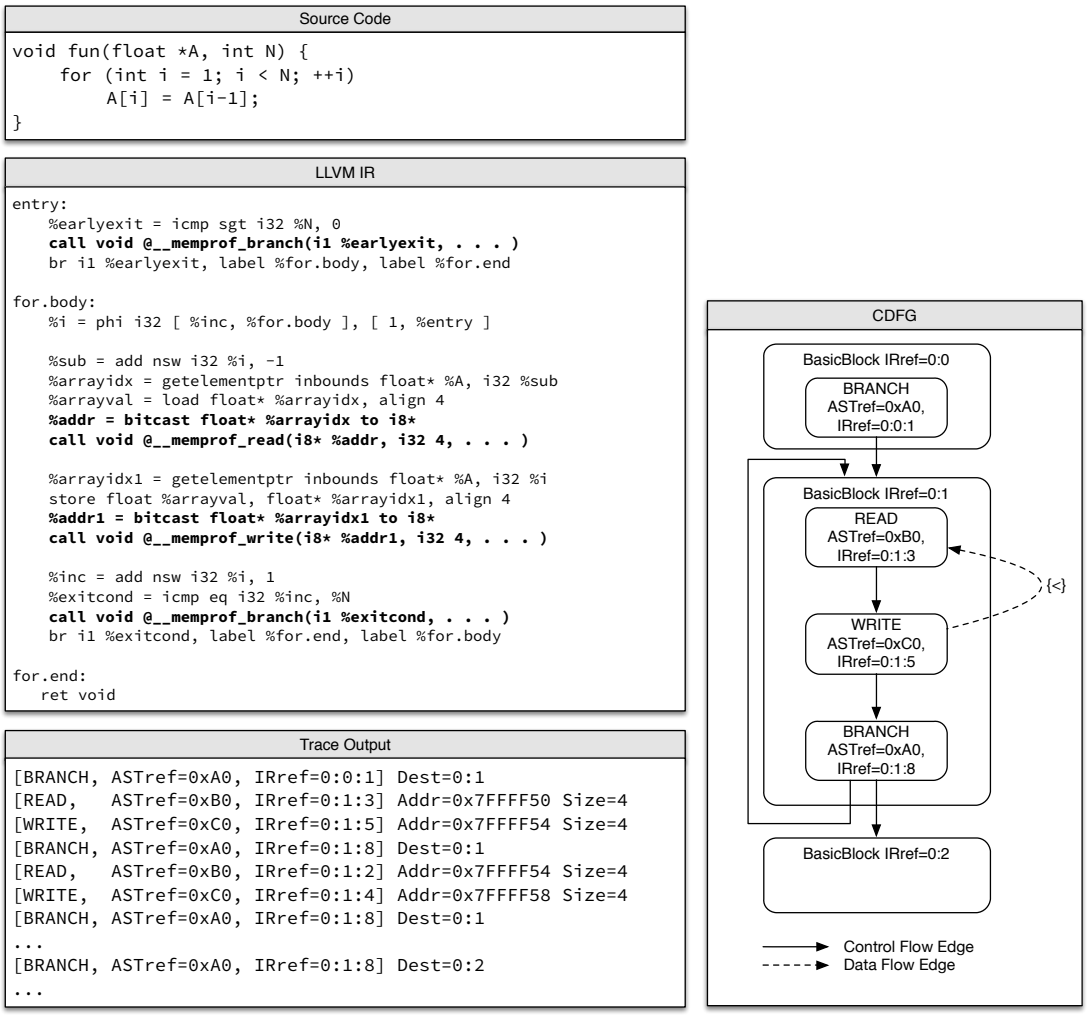


Figure 3.4: Static instrumentation. The source code is compiled to LLVM IR which in turn is instrumented by the insertion of calls to a runtime library (shown in **bold**; the arguments relating calls to AST and IR nodes have been omitted for brevity). If the benchmark is linked against the CDFGPROFILER library and run, the trace shown is produced. Based on this trace, a CDFG can be generated; the loop-carried flow dependence with its direction vector [135] is shown by the dashed line.

to CDFGPROFILER, additional instrumentation is used to mark candidate skeletons.

3.5 Benchmarks

We use a number of standard benchmark applications for the empirical evaluation of our techniques. This has a two-fold purpose. Firstly, the use of these benchmarks

Benchmark	Language	Subset	Application Area
400.perlbench	C	INT	Programming Language
401.bzip2	C	INT	Compression
403.gcc	C	INT	C Compiler
429.mcf	C	INT	Combinatorial Optimization
445.gobmk	C	INT	Artificial Intelligence: Go
456.hmmer	C	INT	Search Gene Sequence
458.sjeng	C	INT	Artificial Intelligence: chess
462.libquantum	C	INT	Physics / Quantum Computing
464.h264ref	C	INT	Video Compression
471.omnetpp	C++	INT	Discrete Event Simulation
473.astar	C++	INT	Path-finding Algorithms
483.xalancbmk	C++	INT	XML Processing
410.bwaves	Fortran	FP	Fluid Dynamics
416.gamess	Fortran	FP	Quantum Chemistry
433.milc	C	FP	Physics / Quantum Chromodynamics
434.zeusmp	Fortran	FP	Physics / CFD
435.gromacs	C, Fortran	FP	Biochemistry / Molecular Dynamics
436.cactusADM	C, Fortran	FP	Physics / General Relativity
437.leslie3d	Fortran	FP	Fluid Dynamics
444.namd	C++	FP	Biology / Molecular Dynamics
447.dealII	C++	FP	Finite Element Analysis
450.soplex	C++	FP	Linear Programming, Optimization
453.povray	C++	FP	Image Ray-tracing
454.calculix	C, Fortran	FP	Structural Mechanics
459.GemsFDTD	Fortran	FP	Computational Electromagnetics
465.tonto	Fortran	FP	Quantum Chemistry
470.lbm	C	FP	Fluid Dynamics
481.wrf	C, Fortran	FP	Weather
482.sphinx3	C	FP	Speech recognition

Table 3.1: The SPEC CPU2006 benchmark suite.

demonstrates the practical applicability and relevance of the proposed research. Secondly, it becomes possible to relate the results to those of prior and future work. In this section, we describe the two benchmark suites used in the following chapters.

3.5.1 SPEC CPU2006 Suite

The SPEC CPU2006 benchmark [52] is an industry-standard CPU-intensive benchmark suite. It includes 29 individual benchmark applications that are subdivided into 12 integer (SPECint) and 17 floating point (SPECfp) applications (see Table 3.1). The benchmarks are implemented in C, C++, and Fortran. They were developed from real user applications that are in wide use across industry and academia and are therefore considered 'typical' for the workloads a general purpose com-

puter system has to handle. Size metrics have only been published for the C++ benchmarks [136], but these indicate binary sizes ranging from 142KB to 17MB and line counts from just over 5,000 to 550,000.

The benchmark suite was designed primarily to provide a comparative measure of compute-intensive performance across different hardware platforms but is equally suited to demonstrate the relative impact of optimizations on the same platform.

The SPEC CPU2006 benchmarks are sequential applications and have been described as difficult to parallelize using existing automated compiler-based methods [89]. As the same paper shows, this is not due to an inherent lack of parallelism but due to the restrictions of existing static automatic parallelization methods that are unable to analyze the complex general purpose code in these benchmarks.

The SPEC CPU2006 benchmark suite comes with three input data sets, called `test`, `train`, and `ref`, of increasing size. SPEC requires a commercial license and cannot be freely distributed.

3.5.2 cBENCH Suite

While SPEC CPU2006 is regarded as the industry standard, the large size of its constituent benchmark applications and the small number of input data sets does not make it an ideal candidate for large-scale evaluations. For this reason, we use an alternative benchmark suite, cBENCH, for our study of the variability of data dependences and control flow in Chapter 5.

The cBENCH suite contains 32 benchmark applications and is originally based on the MIBENCH [49] suite. While primarily an embedded benchmark suite, it comprises codes that cover the entire spectrum from small algorithmic kernels to complex applications; from `crc32` with 130 lines of code to `ghostscript` with over 99,000. Table 3.2 gives an overview of the benchmarks.

A collection of over 1,000 input data sets for the cBENCH benchmarks is available in the form of the KDATASETS collection [25]. This makes it an ideal subject for large-scale studies like the one presented in Chapter 5.

Unlike SPEC CPU2006, the cBENCH suite and KDATASETS are freely available and do not require a commercial license.

Program (# source lines)	Data set file size	Data set description
bitcount (460)	-	Numbers: randomly generated integers
qsort1 (154)	32K-1.8M	3D coordinates: randomly generated integers
dijkstra (163)	0.06k-4.3M	Adjacency matrix: varied matrix size, content, percentage of disconnected vertices (random)
patricia (290)	0.6K-1.9M	IP and mask pairs: varied mask range to control insertion rate (random)
jpeg_d (13501)	3.6K-1.5M	JPEG image: varied size, scenery, compression ratio, color depth
jpeg_c (14014)	16k-137M	PPM image: output of jpeg_c (converted)
tiff_2bw (15477), _2rgba (15424), _dither (15399), _median (15870)	9K-137M	TIFF image: from JPEG images by ImageMagick converter (converted)
susan_c,_e,_s (each 1376)	12K-46M	PGM image: from jpeg images by ImageMagick converter (converted)
mad (2358)	28K-27M	MP3 audio: varied length, styles (ringtone, speech, music)
lame (14491), adpcm_c (210)	167K-36M	WAVE audio: output of mad (converted)
adpcm_d (211)	21K-8.8M	ADPCM audio: output of adpcm_c (converted)
gsm (3806)	83K-18M	Sun/NeXT audio: from MP3 audios by mad (converted)
ghostscript (99869)	11K-43M	Postscript file: varied page number, contents (slides, notes, papers, magazines, manuals, etc.)
ispell (6522), rsynth (4111), stringsearch1 (338)	0.1K-42M	Text file: varied size, contents (novel, prose, poem, technical writings, etc.)
blowfish_e (863)	0.6K-35M	Any file: a mix of text, image, audio, generated files
blowfish_d (863)	0.6K-35M	Encrypted file: output of blowfish_e
pgp_e (19575)	0.6K-35M	Any file: a mix of text, image, audio, generated files
pgp_d (19575)	0.4K-18M	Encrypted file: output of pgp_e
rijndael_e (952)	0.6K-35M	Any file: a mix of text, image, audio, generated files
rijndael_d (952)	0.7K-35M	Encrypted file: output of rijndael_d
sha (197)	0.6K-35M	Any file: a mix of text, image, audio, generated files
CRC32 (130)	0.6K-35M	Any file: a mix of text, image, audio, generated files
bzip2e (5125)	0.7K-57M	Any file: a mix of above text, image, audio, generated files, and other files like program binary, source code
bzip2d (5125)	0.2K-25M	Compressed file: output of bzip2e

Table 3.2: The cBench benchmark suite and kDataSets (from [25]).

3.6 Summary

In this chapter, we have introduced the infrastructure used throughout this thesis. We presented the ArcSim instruction set simulator and the Llvm compiler infrastructure. We discussed the concept of code instrumentation to gain knowledge

about runtime behavior of an application and its implementation in both dynamic and static instrumentation frameworks. The two benchmark suites used for our empirical evaluation, SPEC CPU2006 and CBENCH, and their relative merits were presented. In the next chapter, we begin our study of automatic parallelization with the concept of dynamic binary parallelization (DBP).

Chapter 4

Dynamic Binary Parallelization and its Limits

We begin our vertical approach to the study of parallelization by considering automatic parallelization at the binary level. The key advantage of this type of parallelization is that it remains entirely transparent to the user; no recompilation or access to the source code of legacy applications is required. Before considering higher-level techniques, we therefore first investigate a key question: ***Is it possible to extract parallelism from legacy applications when only the binary executable is available and what are the limits of this approach?***

In this chapter, we investigate *Dynamic Binary Parallelization* (DBP), a recently proposed technique [53, 29, 137] that aims to address this issue. In establishing its limits, we seek to gain an understanding of the factors contributing to these limits and the costs and overheads of its implementation. We perform an extensive evaluation using a parameterizable DBP model targeting a chip multi-processor (CMP) with light-weight architectural thread-level speculation (TLS) support.

The chapter is structured as follows. We introduce the background and motivation for our study in Section 4.1. The architecture of our dynamic binary parallelization model is presented in Section 4.2. We evaluate the sensitivity of this model to changes in various parameters. Our experimental setup is described in Section 4.3, followed by the results of our evaluation in Section 4.4. Finally, we summarize and conclude in Section 4.5.

4.1 Introduction

While the current generation of multi-core processors still comprises relatively few, but powerful cores [18] it is anticipated that in the foreseeable future processor manufacturers will provide us with chip multi-processors (CMPs) containing more, but possibly less powerful cores [101]. A current example of such an architecture is the Intel Single-Chip Cloud Computer [57], where each SCC chip contains 48 P54C Pentium cores. How to run sequential legacy applications on such many-core architectures without incurring an unacceptable performance penalty is a key research challenge.

One obvious solution to this problem would be to include one or more ILP-rich cores specifically dedicated to the execution of sequential workloads in an *Asymmetric Multicore Processor* (AMP). While this is an entirely feasible approach [54], it creates a number of new problems related to the design and software control of such a heterogeneous processor [45]. An alternative approach to executing legacy code on CMPs is to employ *Dynamic Binary Parallelization* (DBP) [53, 29, 137] to dynamically remap a sequential application to a *Symmetric Multicore Processor* (SMP) in a *Virtual Execution Environment* (VEE).

Despite the flexibility DBP offers, its main drawback is that the extraction of parallelism from a binary executable compiled and optimized for a particular single-core machine is a hard problem. This is not only due to the idiosyncrasies of the *Instruction Set Architecture* (ISA) the application has been compiled for, but also due to the low-level nature of the binary representation which obfuscates much of the higher-level code structure so vital to traditional control and data flow dependence analyses supporting parallelization. Recently proposed DBP approaches, e.g. [53, 29], seek to overcome these difficulties by incorporating *Thread-level Speculation* (TLS), *Dynamic Binary Rewriting* (DBR), and a wealth of behavioral information collected during the execution of a program. Such proposals often narrowly concentrate on a specific design point, though, and thus give only a limited insight into the constrains on the design space for DBP implementations.

In this chapter, we attempt to remedy some of the shortcomings of previous DBP studies. Our work is the first to pursue a characterization approach that is, as much as possible, independent of specific DBP systems and architecture configurations. In particular, we explicitly make no assumptions on whether TLS is implemented in software or hardware. High-level DBP with minimal, light-weight TLS architectural

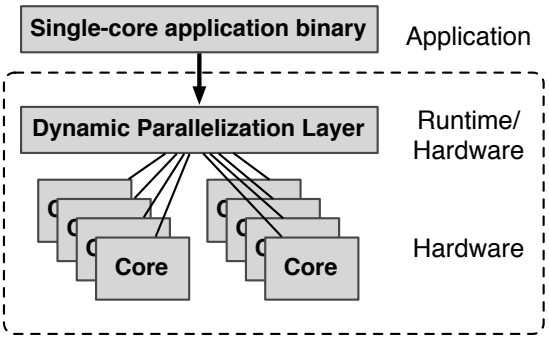


Figure 4.1: Overview of the general architecture of a dynamic binary parallelization system.

support is explored in *one common, parameterizable framework*. The model employs a generic region-based tracing approach, which represents the state-of-the-art in dynamic instruction tracing [19], and subsumes various other, more fine-grained, thread identification algorithms proposed in previous work (see Section 2.2). In this way, a more accurate upper bound on the performance potential of DBP is obtained (as opposed to some particular implementation) and, moreover, relative performance sensitivity can be related to specific high-level system parameters.

4.1.1 Motivation

Consider the generic dynamic parallelization system in Figure 4.1. An unmodified single-core application binary is run on top of a multi-core architecture which parallelizes the application at runtime to improve its performance while remaining entirely transparent to the original application. DBP can be realized in a number of different ways ranging from runtime systems fully implemented in software, low-level firmwares relying on a limited degree of hardware support, to entirely hardware-based solutions.

The key idea behind such systems is to reduce the number of *critical path instructions* by overlapping the execution of subsequent segments of the instruction stream. The parallel execution has to respect true dependences between code segments. While this may seem similar to the exploitation of instruction level parallelism (ILP) in superscalar processors, the goal is usually to parallelize much larger sections of binary code.

To motivate the study presented in this chapter, we have implemented a speculative region-based DBP system. We assume an unlimited number of RISC cores

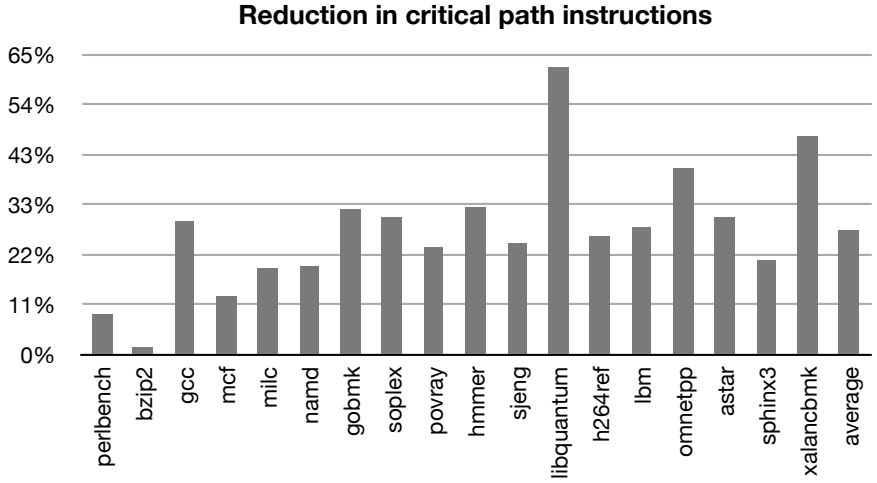


Figure 4.2: Reduction in critical path instructions when using region-based dynamic binary parallelization on a speculative CMP architecture with unlimited cores.

and write-back caches to support memory transactions. The system identifies control flow regions in the instruction stream at runtime and speculatively overlaps segements where there are no dependences or where these can be trivially speculated (see Section 4.2 for more architectural details). We run a number of SPEC CPU2006 benchmarks and identify the percentage of total instructions that can be taken off the critical path using this technique in functional simulation without regard for the overhead of speculation. In this sense, the results illustrate the scope for any concrete implementation of a region-based DBP.

The results, shown in Figure 4.2, are encouraging. For all benchmarks, DBP reduces the number of instructions on the critical path. In the case of highly data-parallel benchmarks, such as `libquantum`, we achieve a reduction of up to 62%. Even a more task-parallel application, such as `gcc`, shows a 29% decrease. On the other hand, in the case of `bzip2`, the high degree of dependences due to the use of shared data structures leads to a rather poor overlap of only 2%.

These results show that there is significant scope for a dynamic binary parallelization system for a wide range of applications. The difficult question, however, is how to translate this into actual performance gains when considering the overheads incurred by a DBP system. Many of these overheads are adjustable – they depend on the particular implementation of the system and especially the amount of hardware support required. Feasibility and design considerations have to be guided by a thorough understanding of these factors. In the remainder of this

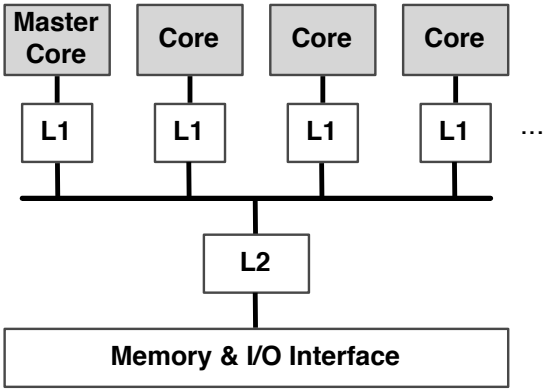


Figure 4.3: Architecture Overview

chapter, we will address precisely this question by analyzing the sensitivity of a DBP architecture to changes in various parameters.

4.2 Architecture for DBP

A dynamic binary parallelization system extracts parallelism from single-core applications at runtime to improve performance while remaining entirely transparent to the application. It operates directly on the level of binary code and therefore requires no access to the application source code. As an on-line method, DBP has the advantage of being able to exploit information about the actual runtime behavior of the application which may be difficult to obtain by static analysis of the source code. This allows it to target frequently executed code portions specifically.

On the other hand, DBP poses a number of challenges. The units of parallelism detected at runtime may be fairly small so low overheads become critical to achieving performance gains. Modern compilers produce dense and highly optimized code so the degree of dependences between code sections can be high and it is often impossible to determine such dependences ahead of execution

The consensus in existing research [53, 29, 137] in the area of DBP is therefore:

- Parallelization needs to be speculative, i.e. there must be a mechanism to recover from attempted parallelizations which turn out to violate dependences or prove incorrect due to the execution taking a different path in the application code.

- A degree of hardware support, such as transactional memory buffers, is required to achieve a performance benefit from DBP.

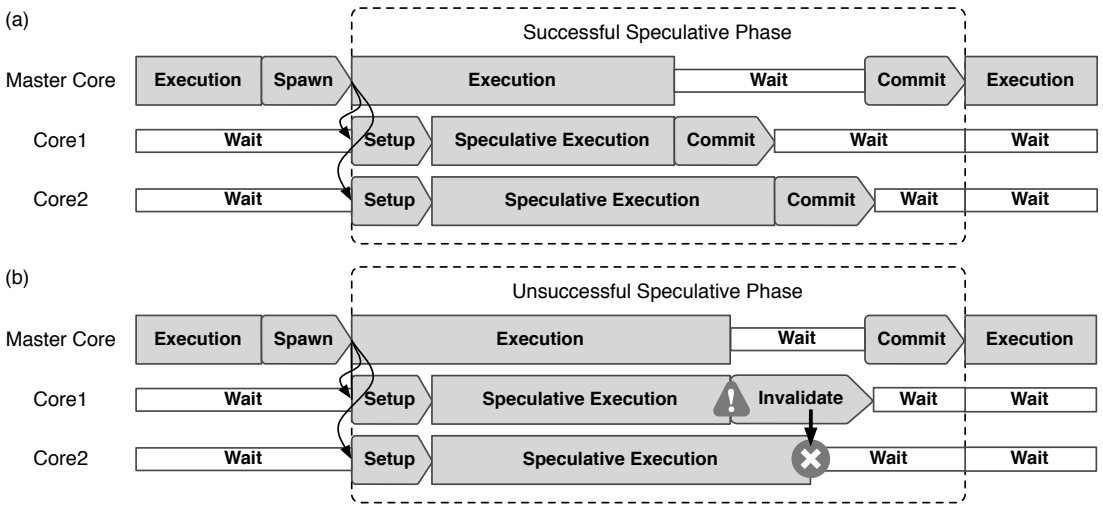


Figure 4.4: Example of two speculative parallel execution epochs on a 3-core system. In both cases, the *master* core spawns two speculative threads. In (a), the parallel execution completes successfully. In (b), core 1 detects a violation and enters the invalidation phase, rolling back memory transactions and canceling subsequent threads.

Based on this experience, we have designed a cost-adjustable dynamic binary parallelization scheme. Our design is relatively conservative and provides the ability to vary the costs of various operations. This will allow us to gain insights on a wide variety of possible hard- and software-based DBP implementations.

4.2.1 Overview

We assume a CMP architecture with a number of cores with private L1 caches and a shared L2 cache connected to a shared bus (Figure 4.3). This ensures fast and low-latency communication between cores which is critical for efficient parallelization. Transactional memory support is implemented on top of write-back L1 caches which allow speculative data to be stored locally.

The available cores are divided into a *master core* and several *speculative cores*. Any application begins executing on the master core. At appropriate points, the master core launches speculative threads on the remaining cores. The code executed on the master core is always non-speculative and never reverted, while code running on speculative cores may turn out to violate dependences or not be on the execution path of the program at all.

In Figure 4.4, we show examples of (a) successful and (b) unsuccessful speculation phases. The system launches two threads which speculatively execute code blocks further ahead in the predicted flow of execution, while the master core continues at the current program counter. Once its share of the code has finished executing, the master core waits for the speculative cores to either complete successfully or to abort execution if the speculation proves to be incorrect.

This epoch-based implementation of speculation is rather conservative, since idle cores could continue to speculate ahead as soon as they finish executing a section of code instead of waiting for the current parallelization phase to end. This would, however, make the speculation layer significantly more complex without leading to performance gains for the most common case of speculative execution – the overlapping of data parallel loop iterations – when most threads will be comparable in length.

4.2.2 Trace-based Thread Identification

A key feature of any DBP system is the manner in which it identifies sections of the instruction stream that can be run as parallel threads. In contrast to most TLS schemes, where compiler-based analysis identifies threads ahead of execution, a DBP system can only rely on information obtained from execution counters and analysis of the binary code of the application. In line with traditional off-line automatic parallelization, most systems target elementary loops as the basic unit of parallelism [53, 29, 77, 139]. However, the use of more general structures such as *traces* [137] or *program slices* [132] has also been proposed.

We implement a lightweight region trace-based approach, a technique successfully employed in high-performance dynamic binary translation (DBT) systems [19]. At runtime, we construct control-flow graphs of the application on a per-page basis. Tracing is lightweight because we only record basic block entry points (i.e. memory addresses) as nodes, and pairs of source and target entry points as edges in the CFG traces. Edges are annotated with execution frequencies.

Execution time is partitioned into *trace intervals* (see Figure 4.5) whose *length* is determined by a user-defined number of executed instructions. After each trace interval, the DBP system enters an analysis phase – which can be decoupled from execution and performed on an idle or dedicated core – to identify *cutting points* in the CFG traces. Based on this information, we can predict likely future control flow

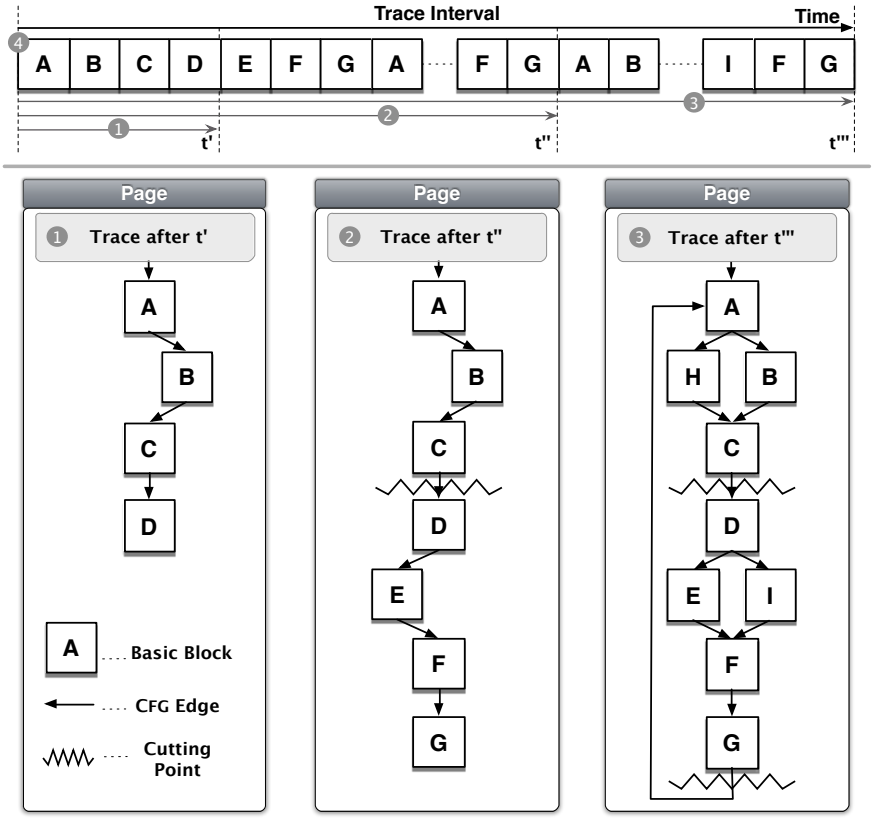


Figure 4.5: Incremental construction (①, ②, ③) of region traces from the sequence of basic blocks ④ executed during one trace interval. Cutting points separate segments of the trace whose execution is speculatively overlapped.

and speculatively overlap execution of segments of the predicted control path at cutting points when control reaches the same region again. We employ a number of heuristics (loop boundaries, backward branches, and a threshold on the maximum number of instructions) to identify such points in traces. Once speculation has been attempted for a given region, the outcome can be used to vary the location of cutting points as well as the likelihood of further attempts at speculation (*dynamic adaptation*). The maximum length of a region trace is bounded by the page size. This limitation is an architectural requirement, as the underlying system architecture includes a memory management unit (MMU). In order to support generic code, the DBP system must allow for the possibility of self-modifying code and needs to be compatible with memory paging by the operating system.

The region trace-based approach is sufficiently generic to subsume other, more fine-grained, thread identification algorithms proposed in related work (see Section 2.2). It can express loop, task, and data parallelism at various granularities and

hence derive results indicative of the upper bound performance that more targeted algorithms would exhibit. Some of the limitations of this approach are discussed in Chapter 7.

Regardless of their shape, the identification of suitable code sections for parallelization incurs an initial overhead and potentially ongoing costs. Unlike other parallelization costs, these overheads manifest themselves in delays until parallelization can be attempted but do not cause a potential slowdown compared to single-core execution. This is because any such overheads can be offloaded into a free hardware context (e.g. an idle core) where thread identification can take place concurrently with the execution of the binary.

4.2.3 Speculative Execution Costs

Once threads have been identified, speculative parallel execution involves a number of different steps. Figure 4.4 shows the four stages of thread spawning, setup, commit, and invalidation, all of which incur costs that are dictated by the particular implementation of the DBP system. We will now discuss them in more detail.

Thread spawning. After the identification of suitable sections for parallel execution and when execution has reached a point where such a section begins, the DBP system needs to *spawn* threads to begin parallel execution. This requires at least the sending of the current program counter (PC) to the various cores participating in parallel execution. It may also involve the transfer of *live-in* registers either through memory or dedicated communication lines. Costs for this stage can range from very few cycles (e.g. in the Multiscalar architecture [113]) to several hundreds in software-based solutions.

Thread setup. Before a newly spawned thread can begin execution, some setup work may be required. This can happen simultaneously on all cores running parallel threads and may involve value prediction for registers or memory locations, or the copying of state from the spawning core. Both [29] and [53] implement these features in software although the respective costs are not explained in detail.

Thread commit. At the end of a successful parallel execution, each core needs to commit its state. If execution was speculative, this includes checking whether there were violations of dependences or mis-speculated values and making memory or register updates non-speculative. It may also involve waiting for predecessor threads to complete and the forwarding of data to successor threads. In RASP [53],

for instance, speculative stores are marked non-speculative in constant time and become architecturally visible immediately, while value speculation is verified in software after waiting for predecessor threads to complete.

Thread invalidate. Speculative execution may fail for two reasons: (a) we may mispredict future control flow and hence execute code that is not on the actual execution path; or (b) we may discover a violation of dependences or an incorrect value speculation upon reaching the commit phase. In both cases, the speculative parallel execution has to be aborted and its possible effects reverted ('squashed'). The costs resulting from mis-speculation are twofold. Firstly, execution cycles are 'wasted' on code which later has to be squashed. This element of the cost is determined by the length of the speculative threads. In our model, it is represented by cycle-accurate simulation of the 'wasted' execution. Secondly, additional overheads may be incurred in invalidating the speculative execution phase, such as canceling a memory transaction and stopping successor threads. We allow these additional overheads to be modeled separately by a parameterizable cost.

4.2.4 Data Dependence Speculation

We have so far described a basic model for DBP. A number of optimizations are commonly employed in order to improve performance. Of particular importance are those which aim to reduce data dependences between threads.

A data dependence between two threads can be classified as either a *flow*, an *anti*, or an *output* dependence. Anti and output dependences can be eliminated using memory and register *renaming*. This is easily achieved through small modifications to the memory hierarchy which carry little overhead [53]. Flow dependences, on the other hand, actually prevent parallelization.

Value prediction is one way of alleviating this problem. DBP implementations, such as [53], use it to predict values of memory locations and registers based on historic observations or typical code patterns, like spilling and reloading across calls, and thus speculatively break flow dependences between threads. This is especially important for loop induction variables where future values need to be predicted in order to execute several loop iterations in parallel.

In our study, we use a less complex approach. We resolve anti and output dependences using renaming. Flow dependences on memory locations are always treated as violations; those on registers are speculated using a *last value* predictor.

CMP	8-core RISC
Pipeline	3-Stage
Execution Order	In-Order
Branch Prediction	Static (BTFN)
ISA	ARCompact
Floating-Point	Hardware
Memory System	
L1 I-Cache	32k/4-way
L1 D-Cache	32k/4-way
L2 Unified Cache (shared)	1M/4-way
L2 Latency	10 cycles
Cache Replacement Policy	Pseudo-random
Bus Width/Latency/Clock Divisor	32-bit/16 cycles/2
Simulation	
Simulator	Full-system, cycle-accurate
I/O & System Calls	Emulated
Speculative parallelization costs	
Spawning, Setup, Commit, Invalidate	0 ... 1000 cycles

Table 4.1: Simulator Configuration and Setup.

Whenever a read-after-write violation on a register occurs, we save the last value written to that register by the 'writer' thread in the metadata for the 'reader' thread. Every time the 'reader' thread is executed in the future, the register is speculatively set to this value on thread entry. Loop induction variables are detected and handled separately.

4.3 Experimental Setup

Before discussing the results of our DBP study, we briefly describe our experimental setup. All experiments were conducted using a fast, cycle-accurate, full-system instruction set simulator (ISS) of the target architecture. Its micro-architectural processor model has been verified against a synthesizable RTL implementation. We faithfully model the actual hardware execution of parallel threads, including waiting times for thread completion, wasted execution cycles on invalidation, and cache/memory effects of speculative execution. Thread management overheads

(spawning, setup, commit, and invalidation) are represented using parameterized unit costs that are added to the cycle count whenever a given event occurs in simulation.

The speedups reported in the results section are based on *cycle count* measurements obtained from this simulation environment.

4.3.1 Processor Model

We simulate an 8-core CMP architecture of RISC processors using the ARCSIM simulator described in Chapter 3. The simulator faithfully models each core's 3-stage pipeline, mixed-mode 16/32-bit ARCompact instruction set, zero overhead loops, static branch prediction, branch delay slots, and four-way set associative data and instruction caches. The cores share a unified L2 cache. Cycle penalties for memory transactions and thread management are reflected in the parameterizable costs for the various DBP stages discussed in previous sections.

4.3.2 Benchmarks

Our evaluation focuses on the SPEC CPU2006 benchmark suite. We use all of the benchmarks implemented in C/C++, both integer and floating point codes[1]. As we noted in Chapter 3, these are sequential applications and have been described as difficult to parallelize using existing automated compiler-based methods [89]. The benchmarks were compiled using `arc-gcc-4.2.1` with `-O2` optimization level.

We simulate complete runs of the benchmarks. In order to reduce simulation times, we use smaller data sets provided with the SPEC benchmark suite. The benchmarks execute an average of 10 billion RISC instructions each per run.

4.4 Results

We begin our evaluation by establishing the theoretical maximum speedup achievable using region-based DBP in the absence of costs for speculative parallel execution. We then analyze the impact of the thread spawn, setup, and commit costs. Initially, we do this under the assumption that traces which our system would try

[1] `447.dealII` was excluded due to issues in adjusting the data set size.

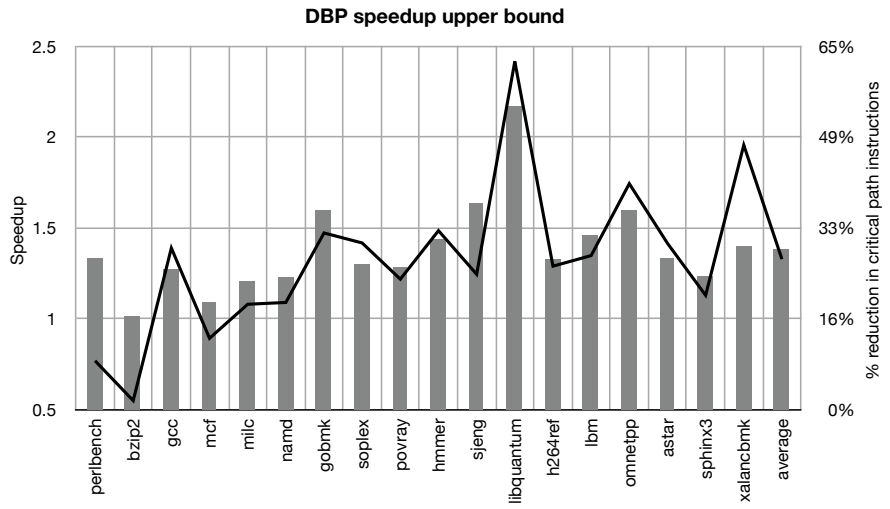


Figure 4.6: Upper bound to speedup if all DBP costs are set to zero (bars); and reduction in critical path instructions from Figure 4.2 (line).

to parallelize but fails to achieve a speedup with have already been identified and rejected prior to the measurement interval. In this way, we can illustrate the actual impact of successful parallel execution irrespective of the particular predictor used to decide whether parallelization should be attempted, and thus determine an upper bound for the performance under the given parameters.

Following on from these experiments, we continue with a more realistic model that does allow for the occurrence of mis-speculation. We investigate the impact of varying mis-speculation penalties and finally show the performance achieved with a realistic set of parameters determined in the course of our experiments.

4.4.1 Upper Bounds

Before investigating the impact of individual DBP parameters, it is important to understand the theoretical maximum speedup achievable using the technique. This upper bound is reached if all costs are set to zero, i.e. if speculative parallelization does not incur any cycle penalties. Figure 4.6 shows the results.

We achieve an average speedup of `1.43x` over single-core execution. The `libquantum` benchmark performs particularly well with a speedup of `2.17x`. Only one benchmark, `bzip2`, fails to show any significant improvement from DBP.

Our work was motivated by the insight that a speculative DBP system can overlap sizable portions of the instruction stream for parallel execution. In Figure 4.6, we again show the possible reduction in critical path instructions, which we

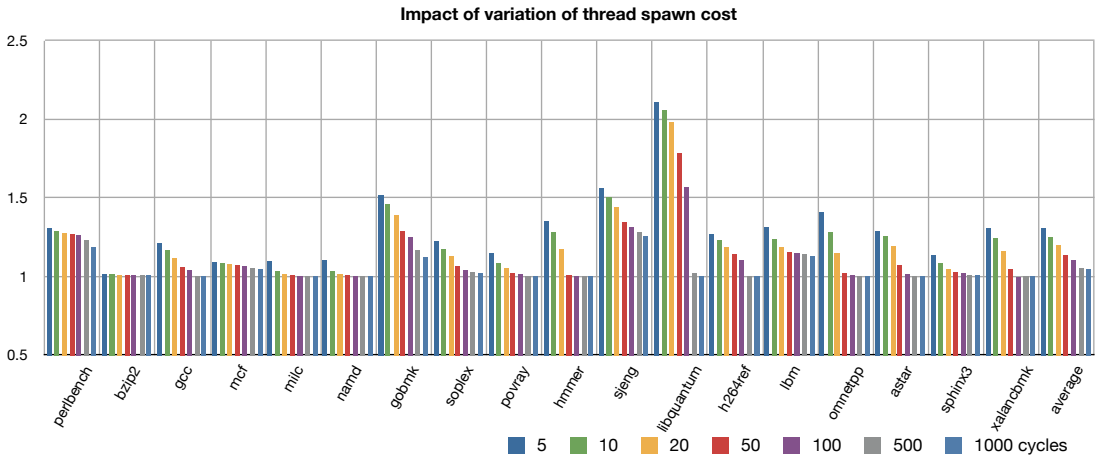


Figure 4.7: DBP speedup achieved with varying costs for thread spawning; all other costs are set to zero.

discussed earlier in Section 4.1.1. A comparison with the DBP speedup figures indicates broadly similar trends, namely that a reduction in critical path instructions also translates into a corresponding reduction in the number of cycles executed.

However, the `perlbench` and `xalancbmk` benchmarks diverge significantly[2]. In the case of `xalancbmk`, we would expect a higher speedup given that there is significant parallelism on the instruction level. Conversely, for `perlbench`, a relatively low degree of instruction overlap should lead to similarly moderate performance gains. Further analysis of the two benchmarks reveals that in both cases the behavior is caused by cache effects resulting from the execution of code sections on separate cores. If each thread accesses different memory locations, parallel execution will benefit from separate caches in each core; this is the case with `perlbmk`. If the same memory locations are accessed by all threads, the data needs to be replicated in each core's L1-cache; this leads to additional memory traffic and can cause slowdowns, as is the case with `xalancbmk`.

4.4.2 Thread Spawn Cost

The cost of spawning threads is seen as a key parameter in TLS systems and consequently much effort has been spent on optimizing this step in previous work, e.g. [78]. We thus begin by exploring the impact of the thread spawn cost parameter in a range from 5 to 1000 cycles, with all other costs (setup, commit, and invalidate) initially set to zero. This range of costs is inspired by previous work on

[2] `bzip2` has little instruction overlap and consequently does not show any speedup.

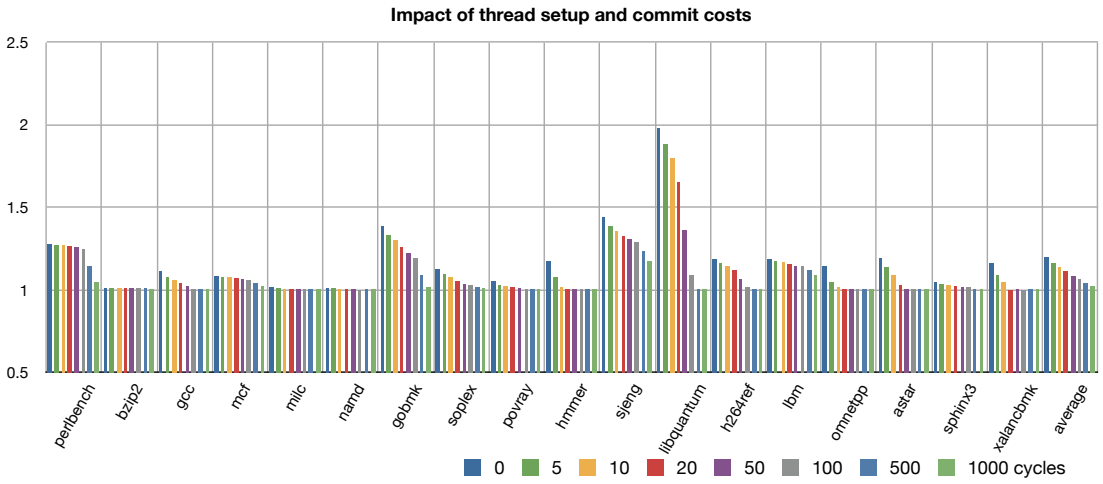


Figure 4.8: DBP speedup achieved with varying setup and commit costs and a fixed thread spawning cost of 20 cycles.

TLS and DBP systems and covers the spectrum from very aggressive hardware implementations (e.g. [44]) to highly optimized software-based schemes (e.g. [139]). The results are shown in Figure 4.7.

The average speedup over single-core execution ranges from `1.30x` with a spawn cost of 5 cycles to `1.04x` for 1000 cycles. In the more realistic range for optimized hardware implementations, we achieve `1.20x` (20 cycles) and `1.13x` (50 cycles).

The `libquantum` benchmark performs exceptionally well up to a cost of 100 cycles where we still observe a speedup of `1.57x`. The most consistent performance is obtained with `perlbench` which shows a speedup of `1.18x` even with 1000 cycles thread spawn cost. Among the remaining benchmarks there are some that gain little benefit from DBP at any cost level while the rest generally exhibit speedups at least up to a 50 cycle thread spawn cost.

4.4.3 Thread Setup and Commit Costs

In the next step, we fix the thread spawn cost at 20 cycles and analyze the impact of varying thread setup and commit costs. A cost of 20 cycles for thread spawning achieved speedups for 15 out of 18 benchmarks in the experiments in the previous section and could be realized in an aggressive hardware implementation.

The thread setup and commit costs are set to values between 5 and 1000 cycles each. This again covers a relatively wide spectrum between aggressive hardware-

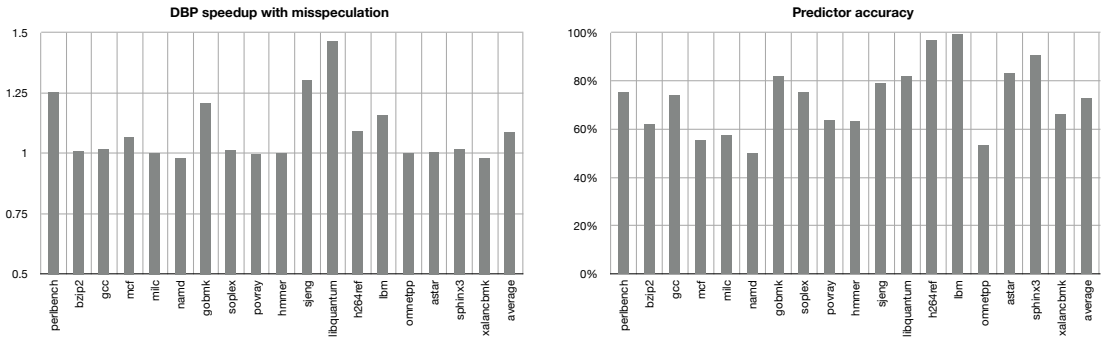


Figure 4.9: Results for realistic DBP model with mis-speculation and history-based predictor (all costs set to 20 cycles).

and slower software-based DBP implementations. We also include the result for a thread setup and commit cost of zero from the previous section. The results are shown in Figure 4.8.

With the addition of these costs, the average speedup now ranges from `1.16x` for 5 cycles setup and commit costs to `1.02x` for 1000 cycles. This is a decrease of `3.1`% to `14.8`% in comparison to the speedup achieved with a thread spawn cost of 20 cycles and no other costs.

As the costs are increased, the average speedup decreases at a lower rate than it did when the thread spawn cost was increased in the previous section. This indicates that the introduction of a thread spawn cost mainly resulted in filtering out parallel sections that are too small to lead to any benefit on a realistic system where dynamic parallel execution incurs a penalty. The remaining parallel sections exhibit enough parallelism to absorb additional costs more readily. Nevertheless, some benchmarks, such as `omnetpp` and `hmmer` which exhibited speedups in our previous experiment, do not tolerate the introduction of increased setup and commit costs very well since their parallel sections are still too small or lack a sufficient degree of parallelism.

4.4.4 Mis-speculation

We have thus far assumed the availability of a perfect predictor to decide whether a section of code should be parallelized. While such a model is useful to show the impact of individual parameters, a realistic scheme has to account for the possibility of mis-speculation and cancellation of threads.

We now introduce a history-based predictor into our model to determine whether a section of the instruction stream should be parallelized and which traces to execute speculatively. We implement a *path-based next trace* predictor [59]. Thread spawn, setup, and commit costs are set to 20 cycles each (in line with previous TLS studies such as [44]) and the cycle penalty for thread invalidation is varied between 5 and 1000 cycles.

It is important to note that, as explained in Section 4.2.3, the thread invalidation penalty represents the cost of operations that need to be carried out when mis-speculation is identified, such as rolling back memory transaction and stopping successor threads. These costs are a feature of the particular implementation of the DBP system and they are what we model by the thread invalidation cost parameter. In addition, mis-speculation incurs a different type of 'cost', namely the wasteful execution of code until mis-speculation is detected. That cost is determined by the length of the erroneously executed threads and consequently varies in each speculative execution episode. Since we simulate actual parallel execution of the code, our system inherently models this overhead.

The experiments using this configuration show that the amount of the thread invalidation penalty is relatively insignificant compared to other parameters. The change in speedup resulting from varying this cost between 5 and 1000 cycles is `<0.01x` on average. We show the speedup results for a penalty of 20 cycles in Figure 4.9.

The reason for this low impact of mis-speculation is twofold: the history-based predictor performs surprisingly well with an average accuracy of `72.7%` across benchmarks (see Figure 4.9); and the amount of cycles gained due to successful speculation is several orders of magnitude larger than those lost due to mis-speculation. Mis-speculation is hence a comparatively rare and low-impact event, the cost of which is easily absorbed by successful speculative parallel executions. This suggests that a DBP scheme can be implemented in a way that permits mis-speculation penalties to be relatively high in relation to other parameters.

The model presented in this section represents a realistic DBP scheme since it includes actual costs for all the events related to parallel execution. The assumption of a penalty of 20 cycles for each of these, in line with previous studies such as [44], implies the need for hardware support for DBP. In this light, the execution speedup results in Figure 4.9 paint a rather somber picture of the benefit of DBP under realistic assumptions. On one hand, we do achieve a speedup of `1.09x` on

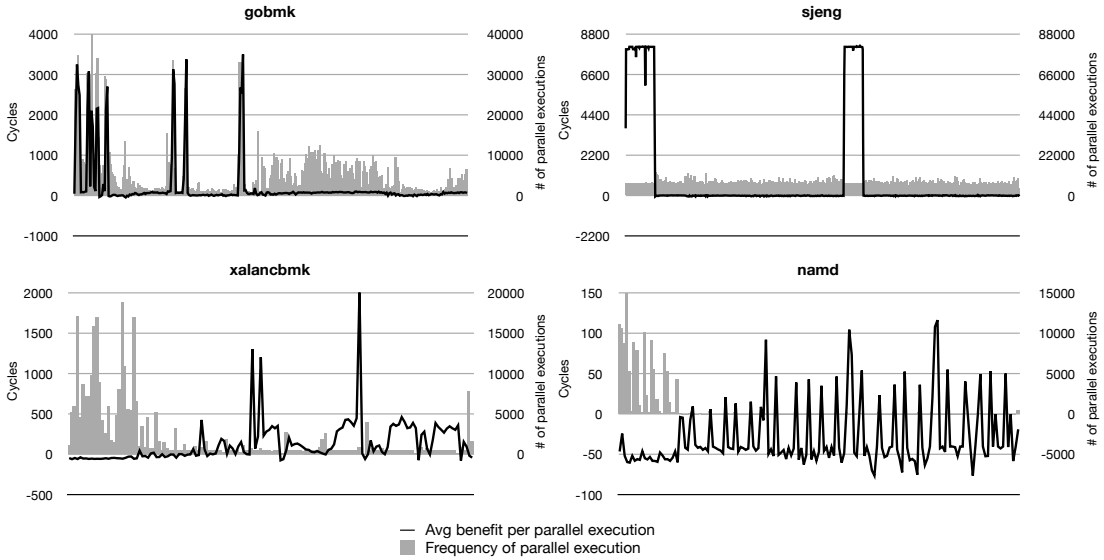


Figure 4.10: DBP behavior over time, showing both the frequency of speculative parallel execution and the average number of cycles taken off the critical path per parallel execution epoch. Time is quantized into 2.5s intervals.

Benchmark	Speedup	Reason
gobmk	1.21	Moderate, steady gains with high frequency of parallel execution
sjeng	1.30	Brief periods of very high gains
xalancbmk	0.98	High losses due to mis-speculation at beginning of execution
namd	0.98	Overall lack of opportunities for parallelization

Table 4.2: Sources of speedup or slowdown for benchmarks shown in Figure 4.10.

average. Three benchmarks exhibit speedups of above 1.25x. Only two of the benchmarks, namd and xalancbmk, suffer a minimal slowdown of 0.98x. On the other hand, the die area likely to be taken up by a hardware DBP implementation and the associated increase in power consumption limit the benefits of using this configuration.

4.4.5 Benchmark Sensitivity

Our results above show that there are large differences across benchmarks in terms of their response to dynamic binary parallelization. To gain a better understanding of this behavior, we investigate four benchmarks in more detail using the realistic DBP model from section 4.4.4.

The two key indicators for the success of DBP are a) *how often* we can speculatively execute code sections in parallel, and b) *what benefit*, or loss, in terms of cycles taken off the critical path can be derived from such parallel executions.

In Figure 4.10, we show the development of these two factors over time during a complete run of four of the benchmarks. Time is quantized into 2.5s intervals. For each interval, we show the number of parallel executions and the average amount of cycles that were offloaded to the speculative cores per parallel execution epoch. This cycle count can be negative for two reasons: firstly, in the case of mis-speculation; and secondly, when speculative execution was successful but the parallel section is not large enough to recoup the costs associated with parallelization.

The first two benchmarks shown, `gobmk` and `sjeng`, both achieved significant speedups of above `1.20x` in the previous section and were chosen as typical examples for the class of benchmarks that benefit from DBP. On the other hand, `xalancbmk` and `namd` are characteristic of the group of benchmarks that do not respond well to DBP, showing slight slowdowns of `0.98x`.

The first half of `gobmk`'s execution time is characterized by several brief episodes of high gains from parallel execution. The second half shows a long period of moderate but steady gains. The average number of cycles gained per parallel execution during this period is relatively low, but the high frequency of parallelization means that it becomes one of the main sources of speedup for this benchmark.

`Sjeng` derives most of its speedup from periods of very high parallelization gains both at the start of its execution and at the beginning of its second half, which result from hot loops that can be parallelized in their entirety. The remainder of the execution time exhibits only moderate gains.

The `xalancbmk` benchmark incurs significant losses in the first quarter of its execution due to the high frequency of mis-speculated parallel executions. There are some gains later on, but much less opportunities for parallel execution. The gains cannot fully compensate for earlier losses, so we observe an overall slowdown.

Finally, `namd` presents much less opportunity for parallel execution than other benchmarks. A period of increased attempts at parallelization at the beginning of its execution incurs losses. During the remainder of the running time, DBP oscillates between gains and losses with even fewer attempts at parallelization. This again results in an overall slowdown.

A summary of these findings is provided in Table 4.2.

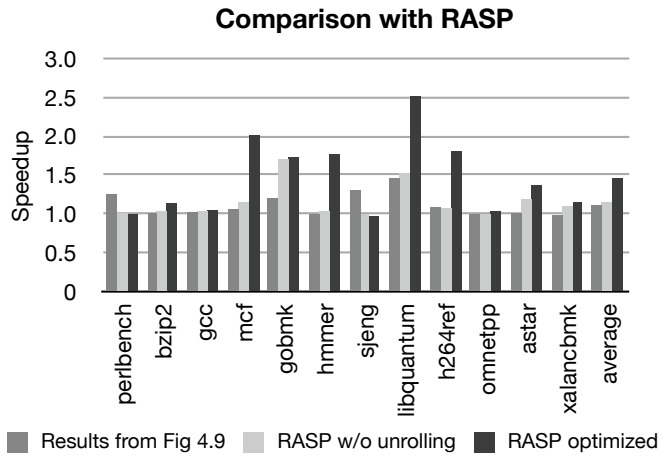


Figure 4.11: Comparison with RASP [53] showing SPECINT2006 results for realistic DBP system obtained in Section 4.4.4, RASP without dynamic loop unrolling, and RASP with all optimizations enabled.

4.4.6 Comparison with RASP

Hertzberg and Olukotun [53] recently proposed an implementation of a DBP system of the type characterized by our study. RASP is a runtime system based on a dynamic binary translator from x86 to RISC. It leverages idle cores in a CMP to analyze, optimize, and speculatively parallelize sequential programs at runtime and thus enables a collection of simpler cores to achieve sequential performance on par with a significantly more complex core without any need for recompilation or hardware support beyond transactional memory.

RASP relies on aggressive runtime optimizations, such as global value numbering and feedback-guided dynamic loop unrolling, which are not exclusive to DBP and would also benefit sequential runtime systems. If any one of these optimizations is disabled, RASP achieves very similar results to those our study predicted for a realistic system in Section 4.4.4. We measured an average speedup of 1.12x for the integer benchmarks in SPEC CPU2006, while RASP achieves 1.16x for the same benchmarks if dynamic loop unrolling is disabled. With all optimizations turned on, their speedup reaches 1.46x on average.

This indicates that the main benefit of RASP arises from dynamically applied sequential code optimizations – similar to a system like DYNAMO [11] – rather than DBP. We chose not to implement such optimizations in our model as they obscure the innate benefits and drawbacks of DBP.

A detailed comparison of these results and those predicted by our model is shown in Figure 4.11. The minor variations between our predictions and the raw RASP results are mainly due to differences in the underlying parallelization approach (loops vs. generic *traces*).

4.5 Summary and Conclusions

In this chapter we have experimentally evaluated the limits of dynamic binary parallelization. We target a CMP platform with lightweight support for speculation and employ a region trace-based just-in-time parallelization scheme to extract threads for parallel execution. Using a parameterizable cost model for speculation based transactions and cycle-accurate simulation of pipeline and memory behavior we demonstrate that for a small number of relevant benchmarks DBP shows good performance gains, whereas for other benchmarks the improvements are rather small.

This is despite a seemingly larger scope for overlapping execution threads, since our results show that the number of critical path instructions can be reduced by up to 62% using DBP for SPEC CPU2006 benchmarks. The introduction of realistic cycle penalties, however, makes it difficult to translate this into a speedup beyond 1.09x on average.

The experiments in this chapter were conducted using a model for *region-based* DBP. Generic region-based tracing has emerged as the state-of-the-art in dynamic instruction tracing [19] and subsumes other, more fine-grained, methods proposed in previous research. Nonetheless, it is conceivable that future developments in the field of instruction tracing may have an impact on the upper bound potential of DBP. We discuss this and other limitations of our approach in Chapter 7.

We draw three conclusions from the study presented in this chapter:

- **Significance of dynamic information.** While it is difficult to exploit in practice using DBP, there is a noticeable amount of parallelism in the benchmarks we investigated (up to 62% reduction in critical path instructions). Static compilation approaches have failed to uncover this parallelism. On the other hand, DBP is capable of exploiting dynamic information about application behavior and hence has greater potential for parallelism detection.

This suggests that dynamic information is key to exploit parallelism in these benchmarks.

- **Overhead of dynamic parallelization.** Parallelization requires computationally expensive analyses and transformations. In a purely dynamic system like DBP, these overheads become part of the critical path of execution. To overcome this limitation, it will be necessary to move all or part of these expensive analyses off-line while still leveraging dynamic information, e.g. in a profiling-based approach.

- **Need for higher-level techniques.** DBP by its nature is a purely dynamic technique and does not have access to the application source code. This is also its main limiting factor as it complicates analysis and restricts the parallel structures that can realistically be detected in the application.

For the remainder of this thesis, we thus focus on techniques that assume the availability of source code and exploit dynamic information in tandem with static analysis to extract parallelism.

In the next chapter, we seek to gain a deeper understanding of the variability of dynamic information. This is an important factor for any technique relying on dynamic information as it affects the validity of inferences made from such data.

Chapter 5

Variability of Data Dependences and Control Flow

In the previous chapter we have seen the benefits of exploiting dynamic information in the context of dynamic binary parallelization. This chapter investigates the variability of such information.

It develops a simple, yet powerful profiling-based analysis to capture data and control flow dependences for program executions with different input data sets. The variability of both data and control flow dependences for the whole cBench benchmark suite [49, 41] is analyzed using 100 randomly chosen input data sets from the kDataSets [25] collection. The performance implications of the dynamically collected dependence information with respect to the ability to exploit loop-level parallelism is investigated and compared against static parallelization approaches.

The chapter is structured as follows. We introduce the background and motivation for our study in Section 5.1. In Section 5.2 we present our profiling-based dependence analyses. This is followed by an evaluation of our empirical results in Section 5.3. We summarize and conclude in Section 5.4.

5.1 Introduction

Besides Dbp, the use of dynamic information to complement static analyses has been the subject of several studies, e.g. [10, 99, 100, 37, 143, 125, 65, 64, 60, 141, 130, 109, 142], as discussed in Chapter 2. The use of dynamic information allows

```
void hist(char *input, int length, int *histogram)
{
  for (int i = 0; i < length; ++i)
    histogram[input[i]]++;
}
```

Listing 5.1: Example of a histogram computation

these techniques to overcome the limitations imposed by *may*-dependences, which may or may not manifest during program execution [96].

The profitability of all of these dynamic parallelization methods is directly dependent on the frequency of dependence violations. An overall speedup can only be achieved if a large enough fraction of parallel executions is successful. However, not much is known about the (probabilistic) nature of these *may*-dependences.

In this chapter we seek to gain a better understanding of dependence patterns arising from *may*-dependences. For this we apply profile-guided data and control flow dependence analyses across the MiBench-derived cBench benchmark suite [49, 41] with 100 randomly chosen input data sets from the kDataSets [25] collection. For each benchmark we analyze and characterize how different input sets cause variation in data and control flow patterns and how these variations in turn affect loop parallelization.

5.1.1 Motivating Example

Consider the example of a trivial histogram computation in Listing 5.1. This is a common code pattern that appears in a more complex form, for instance, in graphics-related benchmarks such as `consumer_jpeg` or `tiffmedian`. While pure static analysis would determine this loop to be sequential due to a potential flow dependence between iterations, profile-guided analysis may lead to rather different conclusions. With a given collection of input data sets, we may determine that the loop is:

- fully parallel due to the nature of the input data, i.e. all elements of `input` are distinct;
- partially parallel with a limited amount of aliasing and therefore amenable to techniques such as TLS or a Map-Reduce transformation; or

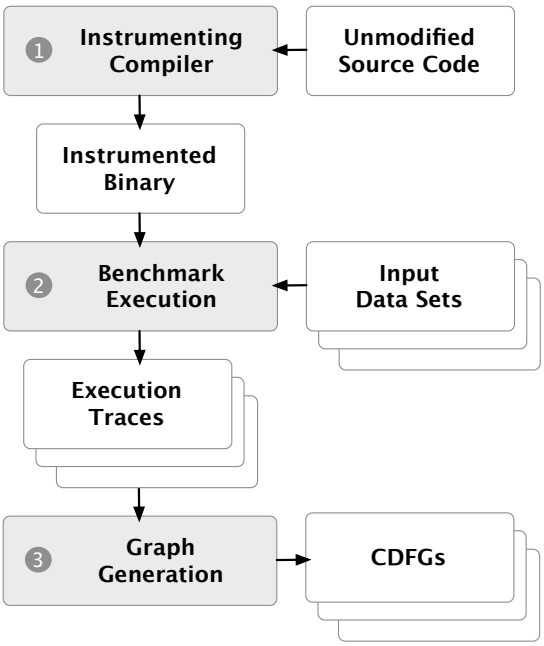


Figure 5.1: Steps to obtain CDFGs for a given benchmark and collection of input data sets.

- fully sequential, i.e. all elements of `input` are identical.

This example illustrates the significant relationship between *input data* and the *control and data flow* of an application. If we draw any empirical conclusions from profiling data, their validity is critically dependent on the quality of input data sets used during profiling. Data sets ideally need to cover all possible control and data flow in a given application. To this end, it is possible to combine profiling information gathered from multiple data sets to improve coverage. Nonetheless, the question remains: ***how variable are data dependences and control flow in typical applications?***

5.2 Dependence Analysis

Our analysis methodology consists of four distinct phases: ① *Benchmark Instrumentation*, ② *Dynamic Execution Tracing*, and ③ *Dependence Graph Construction* (see Figure 5.1); followed by *Graph Merging and Analysis* that provides the results for our study.

5.2.1 Instrumentation and Dynamic Tracing

We use the unmodified application source code of our benchmarks and compile it using CLANG/LLVM with our static instrumentation pass (see Chapter 3). The instrumentation pass inserts calls to a profiling library before every memory operation (read, write, allocation); before and after every call to an internal or external function; on branches between basic blocks; and on certain loop events (loop entry, exit, and iteration). We do not optimize the code so as not to obscure the original structure of the source code, as for instance the various loop transformations commonly used in optimizing compilers would do. This allows us to employ the concept of IR-Profiling where every instrumented event can be linked back to a specific node in the compiler IR and ultimately the program source code. In this way, information gathered statically at compile-time and profiling data can be combined easily at a later stage, for instance to discover parallelizable loops. Using unoptimized code also has disadvantages: trivial loops, which would be eliminated by the optimizer, are still considered; commonly used loop transformations, such as loop fusion or strength reduction, would remove or possibly introduce dependences and hence lead to different dependence patterns. In the interest of presenting results that are as much as possible independent of the particular compiler's optimization passes and for the advantages outlined above, the option of using unoptimized code was chosen for this study.

After compilation, the instrumented benchmark binaries are run with a large number of different input data sets. Each run produces a (lengthy) trace that records the entire control and data flow during execution of a given benchmark.

5.2.2 Construction of Dependence Graphs

For each of the program traces gathered in the previous step, we construct a dynamic, full-program, combined control and data flow graph (CDFG).

In these graphs, **nodes** represent statements with memory effects, such as reads, writes, allocations, and external function calls; as well as basic block entry and exit markers. **Edges** can either represent control flow – in the form of branches, calls, returns, or sequential execution within a basic block – or data flow as dependences between nodes.

Data dependence edges hold additional information indicating whether they are carried by a loop and the type of dependence they represent. We distinguish three separate types of data dependences:

- **Flow dependences** indicating read-after-write data flow;
- **Output dependences** indicating a write-after-write relationship; and
- **Anti dependences** indicating a write-after-read relationship between two nodes.

The loop context of a dependence is recorded in the form of a *direction vector* [135]. The elements of the vector represent levels of the surrounding loop nest. Each element indicates whether the dependence is carried by the respective loop and – if that is the case – the direction of data flow, i.e. whether the current node depends on a previous or successive iteration of the given loop.

We also record observation frequencies for nodes and edges which allows us to determine loop trip counts, likelihood of dependences, and execution frequencies of program regions.

5.2.3 Graph Merging and Analysis

Having thus obtained a CDFG for each benchmark and input data set combination, we proceed to the main object of our study, the analysis of the variability of data dependences and control flow. This process again involves a number of distinct steps.

Firstly, all graphs available for a given benchmark are merged to form a *super-*CDFG that contains all control and data flow observed across all runs. As illustrated in Figure 5.2, the merged graph does not contain duplicate edges or nodes if they were observed in multiple runs. Instead, corresponding edges and nodes are matched and only added once in the supergraph.

In the second step, we measure the *individual* control and data flow coverage of the super-CDFG by each separate graph. For each run of the benchmark, this indicates how much of the total control and data flow of the super-CDFG was observed in that particular run. We measured *Control flow coverage* by the proportion of program basic blocks observed. *Data flow coverage* is defined as the proportion of unique direction vectors observed per pair of dependent nodes. The two measures

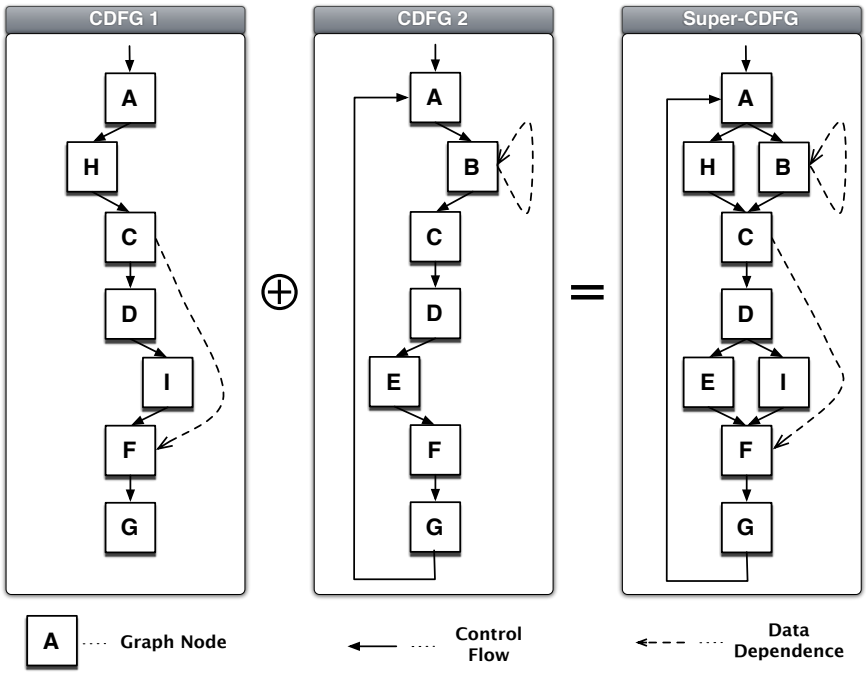


Figure 5.2: Merging of two CDFGs obtained with different input data sets to form a super-CDFG.

correlate to some degree: for instance, if a basic block is not executed in a given run, all dependences between nodes in that block and other nodes also will have not been observed.

Thirdly, we order the graphs by increasing individual data flow coverage. This simulates the worst-case scenario in which a user selects multiple low-coverage input data sets to profile their benchmark application for dynamic data flow analysis.

In the fourth step, we measure the *cumulative* control and data flow coverage of the super-CDFG by the sorted graphs. *Cumulative coverage* represents the coverage obtained by running a benchmark repeatedly with an increasing number of input data sets and combining the graphs resulting from these runs into an incrementally 'larger' graph. To illustrate: the cumulative coverage of the *third run* is the control and data flow coverage of that run and the previous two runs combined; the cumulative coverage of *all* runs is equivalent to the entirety of the super-CDFG.

5.2.4 Parallel Loop Identification

A common use case for dynamic data flow information is the detection of parallelism and specifically the extraction of parallel loops. We use the cumulative data dependence information gathered in our study to identify loops that could be parallelized.

After applying induction variable detection, reduction analysis, and privatization, we mark a loop as parallel if no loop-carried flow dependence was observed between its iterations. Each loop in a loop nest is counted individually; hence, a nest of three loops appears as three separate loops in our statistics. It is important to note that it would most likely not be sensible to parallelize all three if they were to be found parallel. However, for the sake of measuring the accuracy and variability of loop classification it is necessary to consider all relevant loops in the program.

We exclude certain loops from consideration which based on static or dynamic information cannot safely be marked as parallel despite an absence of flow dependences. This applies, for instance, to loops for which only a single iteration has been observed.

5.3 Empirical Evaluation

5.3.1 Experimental Setup

We use the MiDatasets/cBench benchmark suite [41, 42] for our evaluation. Our input data is taken from the kDatasets collection [25]. The benchmarks and input data sets were described in Chapter 3. Each benchmark is run with 100 different, randomly selected input datasets.

We compile the benchmarks using Llvm Clang 3.2 and instrument all control and data flow instructions at the Ir level. At runtime, the instrumented binaries produce traces of all memory accesses and control flow decisions. Each trace is processed to construct a whole-program dynamic control and data flow graph (Cdfg). The graphs resulting from runs with different input data sets are then merged to perform the analyses described above.

The scale of these experiments is vast. All profile data is obtained from complete runs of the respective benchmarks with unmodified input data. The instrumentation increases the running time of the benchmarks considerably and produces large

amounts of data. The size of individual program traces can reach gigabytes of compressed data. Overall, the experiments used several months of compute time and terabytes of storage.

It is important to note that there is no guarantee that the 100 input data sets randomly selected for this study cover *all possible* control and data flow in the cBench benchmarks. When measuring the accuracy of profiling information gathered from individual or cumulative runs, we do so only in relation to the cumulative information obtained from these 100 input data sets. The results should thus *not* be understood as 'absolute' figures, which would be impossible to obtain in the general case, but as probable indicators of the underlying application behaviors.

5.3.2 Variability of Data Dependences

In this first part of our study, we aim to answer the question ***how many input data sets does it take to gain an accurate picture of the data dependences of a given application?***

Figure 5.3 shows the data dependence coverage obtained with our randomly chosen input data sets and the 32 cBench benchmarks. The graphs illustrate both the data flow coverage of individual data sets as well as the cumulative coverage obtained by combining the results of an increasing number of data sets.

It becomes immediately obvious that the benchmarks fall into at least three distinct categories. Firstly, there are benchmarks such as `automotive_bitcount`, `network_patricia`, or `security_blowfish_d/e` that exhibit very little variation in response to different input data sets. The individual coverage of each data set is very close, or even equal to 100% of the total observed data flow and hence cumulative coverage reaches the optimum of full coverage already at the very beginning. Consequently, running the benchmark with additional data sets has little or no effect on data flow coverage. This is the best case scenario for any profile-guided optimization strategy since accurate conclusions can already be drawn from a very small number of profiling runs. Perhaps unsurprisingly, the benchmarks exhibiting this behavior are small algorithmic benchmarks, which tend to be more amenable to static analyses as well.

Secondly, we can see benchmarks where a small set of input data sets show very low individual data flow coverage, but the majority reach higher levels that rather quickly lead to complete cumulative coverage. This is the case for `automo-`

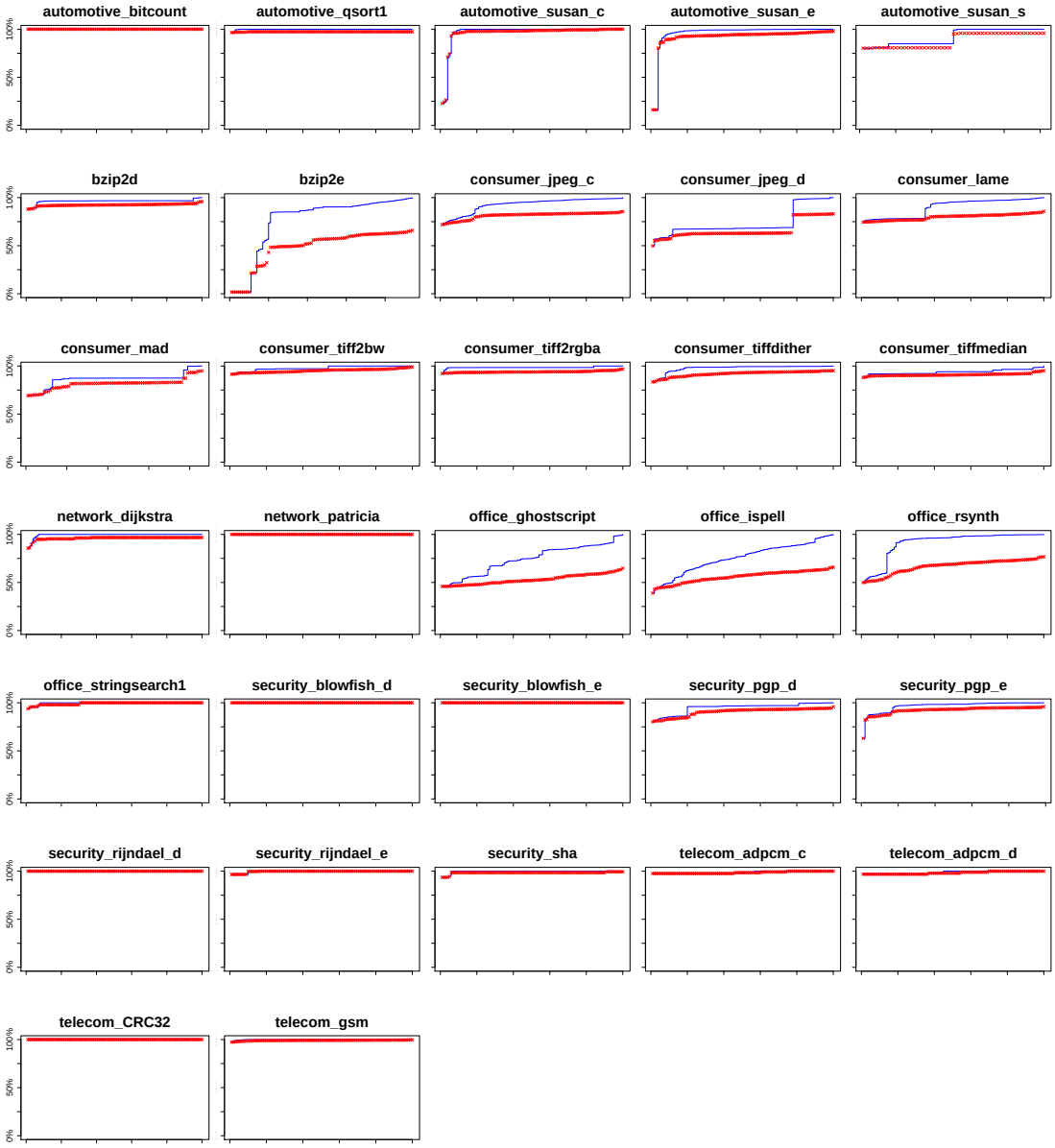


Figure 5.3: Data dependence coverage of individual (×) and cumulative (blue line) runs of cBench benchmarks with 100 different data sets. Cumulative data combines data from all previous runs up to and including a given run. Y-axis shows coverage of total observed data dependences; x-axis represents data sets ordered by increasing individual dependence coverage.

`tive_susan_c/e`, `security_pgp_e`, and – very noticeably – for the 20 worst data sets of `bzip2e`. Closer inspection of the benchmarks and data sets reveals that this is often due to very small input data set sizes or distinct behavior triggered by a certain type of input data. With `bzip2e`, for instance, the outliers stem from

input files that are already bzip2-compressed and carry a *.bz2* file ending. The benchmark detects these and quits without any of the actual compression routines being run. Once the benchmarks are presented with larger input data, coverage is much greater. However, it may not always be obvious what the 'right' size of input data is. Such benchmarks thus need to be run with a much larger number of data sets than the algorithmic benchmarks considered previously to obtain valid profile information.

Finally, the third category comprises benchmarks such as `office_ispell`, `office_ghostscript`, `office_rsynth`, and `consumer_jpeg_c/e` with two noticeable characteristics: the individual data flow coverage of data sets is relatively low (at most approximately 75%); and, the cumulative coverage increases at a very slow rate and still grows even after the results gathered with a large number of data sets have been combined. This suggests that individual data sets trigger subtly different program behavior (e.g. phase orderings) that leads to complex dependence patterns that can manifest in a multitude of different ways. Convergence is only achieved after most of our data sets have been run, which suggests that profile-guided optimizations can be meaningfully applied only if a considerable number of input data sets are considered. The benchmarks in this class are also among the largest in the cBench suite.

5.3.3 Variability of Control Flow

Our results on data flow coverage lead to the question whether there is a significant correlation between data dependence coverage and control flow coverage. In Figure 5.4, we show similarly formatted graphs for control flow as we did for data flow. The data sets are ordered by increasing individual *data flow* coverage, as before. If control and data flow correlated exactly, we would observe a monotonic increase in individual control flow coverage.

This is indeed roughly the case suggesting that limited control flow coverage is one of the main reasons for a lack of data dependence coverage. However, it is notable that for some benchmarks, e.g. `consumer_tiffdither` and `office_ispell`, the observed individual control flow coverage oscillates around a common trend line but does not necessarily grow monotonically with increasing data flow coverage (which was used to determine the ordering of data sets, as mentioned previously).

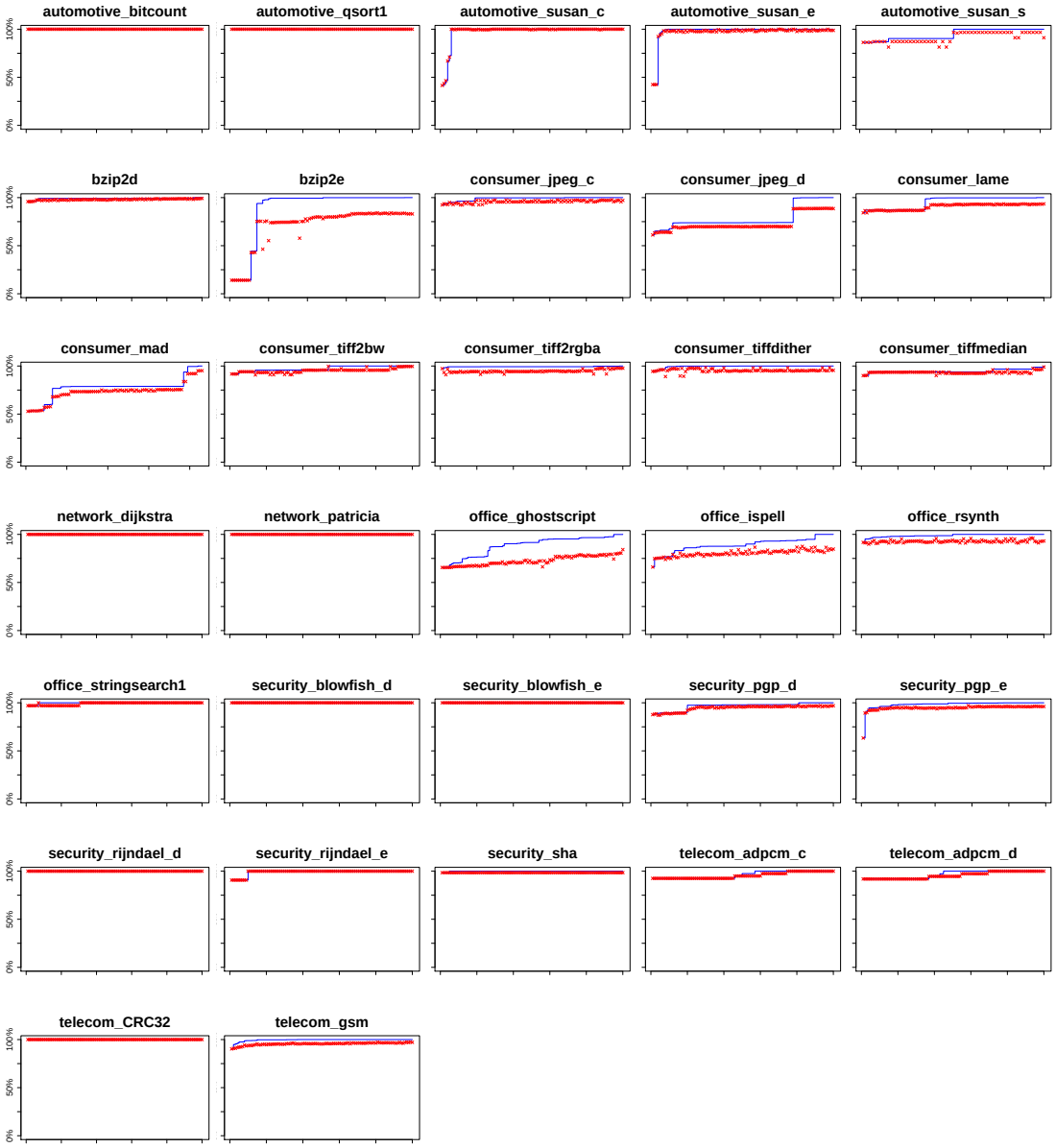


Figure 5.4: Control flow coverage of individual (×) and cumulative (blue line) runs of cBench benchmarks with 100 different data sets. Data sets are ordered by their data flow coverage. Y-axis shows coverage of total observed control flow; x-axis represents data sets.

The reason for this behavior is that the most variable data dependences are not evenly distributed across the benchmark code but rather clustered in specific code regions. Thus, a given data set may still exhibit higher individual *data* flow coverage despite lower individual *control* flow coverage as long as the most 'interesting' code regions are executed. This again occurs mostly with the more complex benchmarks.

5.3.4 Parallelizable Loops

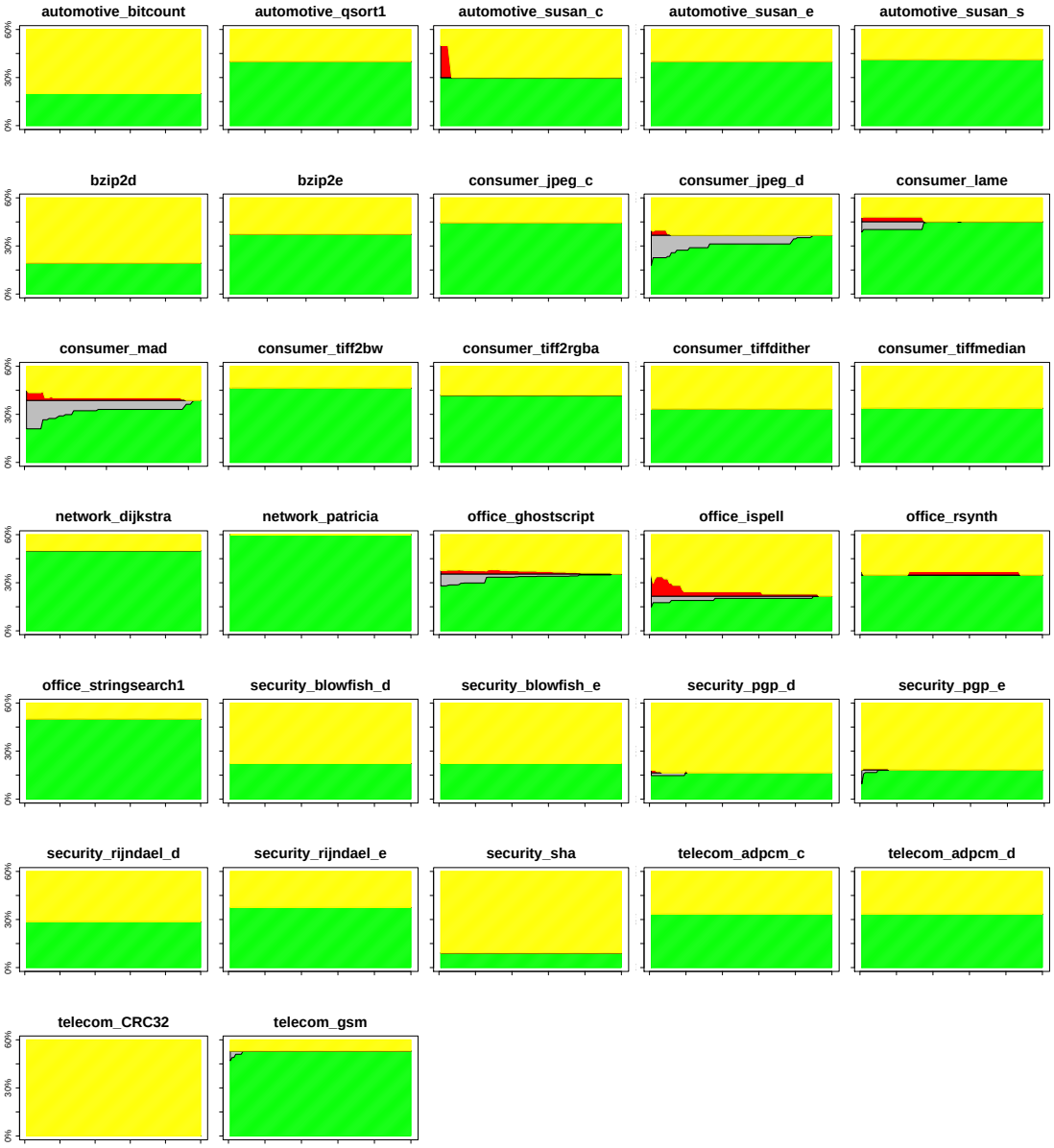


Figure 5.5: Loop classification based on cumulative data flow analysis results. X-axis represents the number of data sets combined to obtain a given classification result (from 1 to all data sets, ordered by their data flow coverage); y-axis shows the percentage of loops identified as parallel or sequential. Segments from bottom to top stand for ■ parallel loops; ■ parallel loops misclassified as sequential; ■ sequential loops misclassified as parallel; and ■ sequential loops. The range from 60-100% represents sequential loops for all benchmarks and is therefore not shown.

Having obtained individual and cumulative data dependence and control flow information for our benchmarks, we now exploit this data to identify dynamically parallel loops. These are loops that exhibit *may*-dependences and cannot be parallelized purely based on static analysis. In this context, we focus on the use of cumulative profile information obtained by combining results from a number of individual runs. As shown in previous sections, some benchmarks exhibit data dependence patterns much too complex to accurately derive from single runs.

Since we rely on dynamic data dependence information, there is a risk of mis-classifying loops. In particular, false positives – a sequential loop mis-classified as parallel – carry significant danger as they break the data flow relationships inherent in the program code. We thus aim to address the question of ***how many data sets does it take to classify the loops in our benchmarks accurately in relation to the information obtained from all 100 runs combined?***

The results for the cBench benchmarks are shown in Figure 5.5. At first glance, it is obvious that there is much less variation in the results than there was for data dependence or control flow coverage. Almost half of the benchmarks show no variation in loop classification as the number of data sets is increased, which implies that a run with even a single data set would be enough to accurately classify the loops in these benchmarks. Again, they tend to be smaller algorithmic benchmarks with more regular dependence patterns.

More complex benchmarks, such as `bzip2e`, `consumer_jpeg_d`, `consumer_mad`, and `office_ghostscript`, all show a number of parallel loops mis-classified as sequential (gray segment ■) which decreases monotonically as more data sets are added. This is entirely due to a lack of control flow coverage: if a loop has not been observed with the data sets used thus far, it is marked as sequential by default.

Finally, we see a small number of sequential loops misclassified as parallel – the worst case scenario for dynamic loop parallelization. There are two interesting observations regarding these mis-classifications. Firstly, it takes a significant amount of data sets to remedy such mistakes. Take the `consumer_mad`, `office_ghostscript`, or `office_ispell` benchmarks, for example, where these loops are only classified correctly with the cumulative information of almost all the input data sets combined. Secondly, and perhaps most surprisingly, the number of mis-classified loops may *increase* rather than decrease as the number of data sets used is increased. Why is this the case? Again, the answer lies in the control

flow coverage of data sets. A loop can only be marked as parallel once it has been observed. Thus, incorporating information from a data set which triggers the execution of a greater number of loops also increases the risk of mis-classifying these loops.

Despite the lower variability of the classification results, it appears that dynamic loop analysis still needs to rely on a large number of input data sets to provide accurate results. One potential heuristic to improve their accuracy may be to introduce a per-loop coverage threshold so as to refrain from classifying loops as parallel unless they have been observed with a majority of input data sets. As discussed in Section 5.3.1, the results are only accurate in relation to the 100 input data sets selected for the study; there may exist some other data set that triggers a dependence not previously observed, which could lead to further loops being identified as mis-classified.

5.3.5 Comparison to Static Loop Parallelization

Industry-standard compilers, such as Intel Icc, contain analysis passes that extract loop-level parallelism at compile-time. Such analyses are necessarily conservative since the absence of loop-carried *may*-dependences can often not be proven statically. In this section, we compare the performance of Intel Icc[1] to the results obtained using our dynamic analysis. To make results comparable, we disable loop distribution and inlining to retain the original program structure. We do not count loops identified in regions of the code that were not executed, since it is impossible to detect those using dynamic analysis.

The results are shown in Table 5.1. Dynamic analysis can frequently identify more than twice as many parallel loops. As expected, this is due to a combination of the practical limitations of static analyses and the nature of many dependences as being determined only at runtime by input data and program options.

5.3.6 Frequency of Sequential Iterations

We have thus far focused on a binary classification of loops as either parallel or sequential. There exist, however, loop parallelization mechanisms that can tolerate a certain degree of 'sequential' iterations that exhibit data flow dependences.

[1] Intel Icc 14.0.0.080 with options `-O2 -parallel -ipo -par-threshold0 -mP2OPT_hlo_distribution=0 -ip-no-inlining`

	Parallel Loops		Total loops
Benchmark	Icc	Dynamic	analyzed
automotive_bitcount	0	1	5
automotive_qsort1	0	4	10
automotive_susan_c	1	3	10
automotive_susan_e	2	6	15
automotive_susan_s	3	7	17
bzip2d	8	17	88
bzip2e	26	53	142
consumer_jpeg_c	21	56	126
consumer_mad	17	52	124
consumer_tiff2bw	4	13	28
consumer_tiff2rgba	3	13	31
consumer_tiffdither	5	13	39
consumer_tiffmedian	11	23	68
network_patricia	0	3	5
office_ispell	7	22	115
office_rsynth	3	17	46
office_stringsearch1	1	5	10
security_blowfish_d	0	2	9
security_blowfish_e	0	2	9
security_rijndael_d	2	2	7
security_rijndael_e	3	3	8
security_sha	1	1	11
telecom_adpcm_c	0	1	3
telecom_adpcm_d	0	1	3
telecom_CRC32	0	0	3
telecom_gsm	24	31	49

Table 5.1: Number of parallel loops detected in cBench benchmarks using a) static analysis in Intel Icc and b) dynamic analysis based on cumulative profile data gathered from 100 runs of the benchmarks.

The prime example for such a technique is thread-level speculation (Tls). Considering the cumulative data flow information gathered from our benchmark runs,

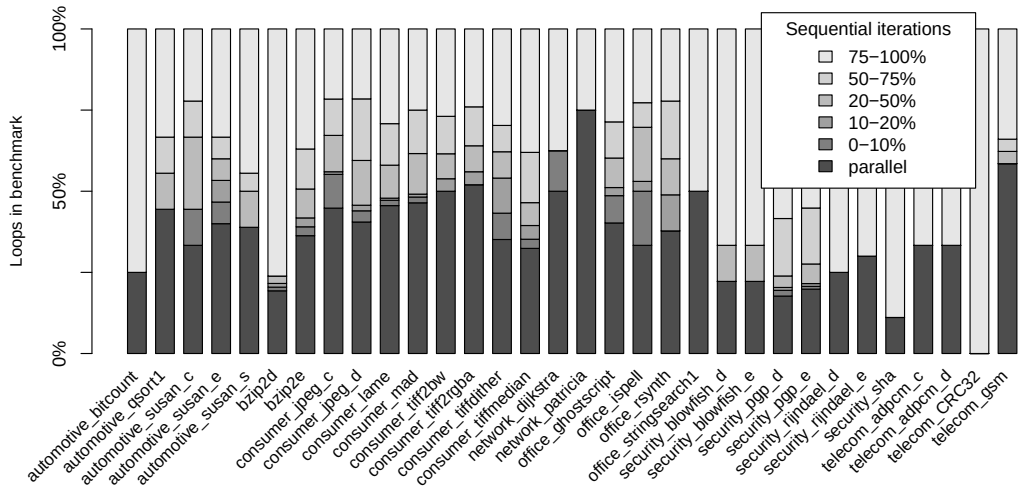


Figure 5.6: Classification of loops based on the percentage of their iterations that exhibit a loop-carried flow dependence ('sequential iterations'). Loops with a low percentage of sequential iterations are suitable for TLS and related speculative mechanisms.

we are able to answer the question of ***how many of the sequential loops in our benchmarks would be amenable to speculative techniques?***

In Figure 5.6, we show the distribution of loops across six different categories based on the percentage of their iterations that exhibit loop-carried flow dependences. Such dependences prevent parallel execution and would trigger a rollback and sequential execution of the given iterations. Limit studies on TLS [58] have shown that rollback events must be rare in order to obtain a speed-up from speculative parallel execution.

Realistically, therefore, only loops that fall into the 0-10% sequential category would seem good candidates for TLS execution. The results show that for some benchmarks, such as `automotive_susan_c/e`, `consumer_jpeg_c`, `network_dijkstra`, `office_ghostscript`, or `office_ispell`, these loops make up 5-20% of loops considered in our study. For the majority of benchmarks, however, loops with much higher dependence frequencies represent the majority of loops considered. This would suggest that these benchmarks would derive rather limited benefit from speculative parallelization.

5.4 Summary and Conclusions

We have presented an empirical analysis and characterization of the variability of both data dependences and control flow across program runs. We ran the cBench benchmark suite with 100 randomly chosen input data sets and recorded complete control and data flow traces. Based on these traces, we built a whole-program control and data flow graph (Cdfg) for each run and compared the resulting graphs to obtain a measure of the variance in the observed control and data flow. While there exist some programs where data flow patterns are almost fixed, we have shown that this is not the typical case. On average, the cumulative profile information gathered with at least 55, and up to 100, different input data sets is needed to achieve full coverage of the data flow observed across all runs. For control flow, the figure stands at 46 and 100 data sets, respectively.

This implies that profile-guided parallelization needs to be applied with care, as misclassification of sequential loops as parallel was observed even when up to 94 input data sets are used. Nonetheless, we have found that variability of data dependences is only weakly correlated with the ability to detect parallelizable loops. This is because loop parallelization is limited by the weakest link, i.e. a single loop-carried dependence is sufficient to prevent parallelization, and it appears that such dependences inhibiting parallelization are fairly stable across different program inputs. Our results also confirm that profile-guided approaches are genuinely more capable in detecting parallelism than their static counterparts.

In conclusion, our study shows that while individual dependences are rather variable in nature, the underlying parallelism in applications is less so. This suggests that purely dependence-based analysis may not in fact be the best approach to detect parallelism. In the next chapter, we introduce a novel characterization of algorithmic skeletons using commutativity and liveness *instead of* dependences. Continuing our focus on techniques involving dynamic information, we present a framework for the detection of these skeletons using a combination of static analysis and dynamic information.

Chapter 6

Automated Detection of Algorithmic Skeletons

We now conclude our vertical approach to the study of parallelization with a technique targeting a very high level of abstraction: algorithmic skeletons [26].

In this chapter, we introduce a new notion of commutativity for regions of code based on liveness and then characterize a number of popular algorithmic skeletons using this property. We combine static analysis and profiling into a practical methodology for detecting algorithmic skeletons in real-world legacy applications and apply this novel approach for parallelism detection to a number of SPEC CPU2006 benchmarks.

This chapter is structured as follows: Section 6.1 explains the motivations behind this approach. Section 6.2 provides a brief introduction to the background on algorithmic skeletons and the notions required for their characterization, which serve as the foundation of the work presented in this chapter. This is followed in Section 6.3 by a formal characterization of a set of popular algorithmic skeletons. In Section 6.4, a hybrid static/dynamic methodology for identifying these skeletons in sequential legacy applications is introduced. The results of our empirical evaluation are presented in Section 6.5. We summarize and conclude in Section 6.6.

6.1 Introduction

While algorithmic skeletons [26] have been widely adopted in the parallel programming community, for example, in the shape of Intel's *Threading Building Blocks*

(TBB) [102] or Google's MAPREDUCE [28], automatically parallelizing compilers are still largely confined to a single type of parallelism, i.e. data-parallel loops. This chapter proposes to widen the scope of automatic parallelization and enable parallelizing compilers to detect common algorithmic skeletons in sequential legacy codes. For this, we initially need to provide a formal characterization of algorithmic skeletons. We then show that for this purpose commutativity is a more suitable concept than dependence information, which is conventionally used to reason about parallelism. We define commutativity of code regions based on liveness, which is a readily available analysis in most compilers. However, as commutativity is still difficult to prove statically, profiling information is used to complement static analysis. Combining these concepts, we present a proof-of-concept skeleton detection framework implemented in the LLVM compiler and use it to demonstrate the detection of a typical skeleton.

6.1.1 Motivation

In this section, we briefly review some of the key motivations behind our approach. This is followed by a concrete example contrasting the informal definition of skeletons and their manifestation in source code.

A formal characterization of algorithmic skeletons is required. Algorithmic skeletons are an *informal* programming model, similar to design patterns. While we may gain an intuitive understanding of a skeleton using either a graphical representation, a verbal description, pseudo-code notation, or concrete code examples [47], this is not sufficient to develop a compiler analysis pass capable of detecting algorithmic skeletons in sequential code. What is needed is a *formal* characterization in terms of concepts defined in compiler theory and commonly available in compiler frameworks.

Algorithmic skeletons cannot be characterized sufficiently using dependence information. Dependence information is not suitable for characterizing complex algorithmic skeletons, which may maintain non-trivial internal data structures, e.g. a work list comprising dynamic task descriptors in a task farm. Updates to those data structures can introduce spurious dependences, which are hard to separate from those related to the actual computation carried out by a skeleton. Instead,

this chapter proposes to use liveness information and commutativity. That is, if the order of execution of two regions does not affect their live-out variables, the regions are regarded as safe to be executed concurrently, subject to synchronization.

Dynamic information is needed to compensate for the limitations of static analysis. We initially use static analysis to drive the detection of algorithmic skeletons according to their characterization developed later in this chapter. However, if this turns out to be inconclusive, additional profiling information gathered from sample executions of the program under consideration is used. Together, this provides enough information on liveness and commutativity to identify algorithmic skeleton candidates with high confidence.

Unfortunately, both the data dependence and commutativity problems have been shown to be undecidable in [103] and [23], respectively. The use of more aggressive, but possibly unsafe, dynamic analyses is the only alternative [124] to making overly conservative assumptions that would frequently prohibit parallelization altogether. As was discussed in Chapter 5, dynamic information must be used with care. However, while this ultimately implies asking the user for final approval, the framework proposed in this chapter is still able to provide useful support in automatically identifying profitable parallelization opportunities.

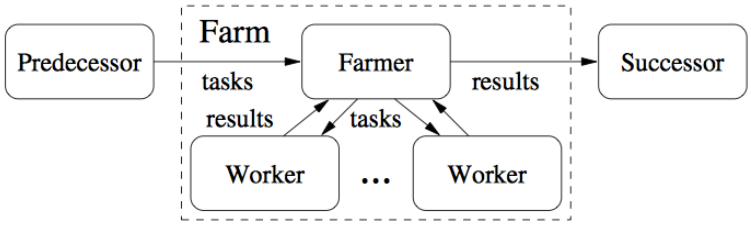
Algorithmic skeleton detection is more general than pattern matching. Detection of algorithmic skeletons differs significantly from pattern-driven automatic parallelization, e.g. [62, 112] as discussed in Chapter 2. Its goal is not the recognition of particular algorithmic idioms like matrix multiplication; instead, it captures generic algorithmic communication and coordination patterns, regardless of their specific functionality and implementation.

6.1.2 Motivating Example

Consider the two characterizations of the *task farm* skeleton in Figure 6.1. Neither the graphical representation in Figure 6.1a, nor the verbal description in Figure 6.1b are precise enough to characterize a task farm skeleton beyond a level of intuitive understanding as central concepts of a task farm, e.g. *farmer*, *workers* and *tasks*, and their interaction remain *undefined*. Now compare this to the concrete code example in Figure 6.1. This code excerpt shows a *graph traversal*

```
1  int Graph::Traverse(int s)
2  {
3      // Mark all the vertices as not visited
4      bool *visited = new bool[V];
5      for(int i = 0; i < V; i++)
6          visited[i] = false;
7
8      // Create a queue for graph traversal
9      list<int> queue;
10
11     // Mark the current node as visited and enqueue it
12     visited[s] = true;
13     queue.push_back(s);
14
15     // 'i' will be used to get all adjacent vertices
16     // of a vertex
17     list<int>::iterator i;
18
19     int result = 0;
20
21     while(!queue.empty())
22     {
23         // Dequeue a vertex from queue
24         s = queue.front();
25         queue.pop_front();
26
27         // Apply some function f to s, accumulate values
28         result += f(s);
29
30         // Get all adjacent vertices of the dequeued
31         // vertex s. If an adjacent vertex has not been
32         // visited yet, mark it visited and enqueue it.
33         for(i = adj[s].begin(); i != adj[s].end(); ++i)
34         {
35             if(!visited[*i])
36             {
37                 visited[*i] = true;
38                 queue.push_back(*i);
39             }
40         }
41     }
42
43     // Return
44     return result;
45 }
```

Listing 6.1: A *graph traversal* algorithm written in C++. Conceptually, the `while` loop in line 21 represents a *task farm*. However, many other implementations of the same skeleton are possible. Tracking dependencies introduced by the `queue` data structure is difficult, whilst the order of execution is not relevant for the calculation of `result`, the only live-out value at the end of the region.



(a) Graphical characterization of a task farm.

"Conceptually, a farm consists of a farmer and several workers. The farmer accepts a sequence of tasks from some predecessor process and propagates each task to a worker. The worker executes the task and delivers the result back to the farmer who propagates it to some successor process (which may be the same as the predecessor)."

(b) Verbal characterization of a task farm.

Figure 6.1: Both a graphical and a verbal characterization of the *task farm* skeleton (according to [93]) are not suitable for implementation in a compiler as these characterizations are informal and key concepts, e.g. *farmer*, *worker* and *task*, are not defined.

routine written in C++, which employs a worklist to iterate over all nodes of a graph, applies a function to each node and accumulates their return values. This final value is then returned to the caller. The body of the `while` loop in line 21 can be treated as a *task farm*, where the entire loop body represents a task. Obviously, there exist many dependences between iterations introduced by the (auxiliary) worklist (`queue`) and the marker array (`visited`), which would prohibit parallelization. However, if we look at the only variable **live-out** after this `while` loop, namely `result`, we will notice that this variable always holds the same value irrespective of the particular order in which the loop processes elements of the worklist. This means if we use a definition of commutativity that only enforces identity of **live-out** values, i.e values used further on in the program, we do not need to guarantee identical state for `queue` and `visited` between a sequential and a parallel implementation, thus allowing this loop to be safely executed in parallel while actually violating data dependences.

Skeleton	*Scope*	*Examples*
Data-parallel	Data structures	map, fork, scan, reduce, ...
Task-parallel	Tasks	sequential, farm, pipe, if, for, while, loop ...
Resolution	Family of problems	divide & conquer, branch & bound, dynamic programming

Table 6.1: Taxonomy of algorithmic skeletons according to [47].

6.2 Background

6.2.1 Algorithmic Skeletons

In this section we review *algorithmic skeletons* as the key concept for expressing parallel patterns beyond traditional loop parallelization. Algorithmic skeletons abstract commonly-used patterns of parallel computation, communication, and interaction. Probably the most outstanding feature of algorithmic skeletons, which differentiates them from other high-level parallel programming models, is that orchestration and synchronization of the parallel activities is implicitly defined by the skeleton patterns. Programmers do not have to specify the synchronizations between the application's computational parts.

Literature on algorithmic skeletons, e.g. [47], distinguishes between *data-parallel*, *task-parallel* and *resolution* skeletons (see Table 6.1). A representative *data-parallel* skeleton is *map*, which specifies that a function can be applied simultaneously to all the elements of a list to achieve parallelism. *Reduce* is another data-parallel skeleton, which computes prefix operations in a list by traversing the list from left to right and then applying a function to each pair of elements, typically summation. A *task farm* is a *task-parallel* skeleton, which is also known as master-slave/worker or bag of tasks. *Divide & Conquer* is an example of a *resolution* skeleton, which recursively applies a map until a condition is met.

Researchers from different communities independently use skeleton-like abstractions [69] and often refer to algorithmic skeletons using different terminology, e.g. *tao of parallelism* [91] for a class of irregular graph processing skeletons or *dwarfs* [9] for representatives for classes of algorithms from different domains.

6.2.2 Single-Entry Single-Exit Regions

Our goal is to introduce a new notion of commutativity over single-entry single-exit (SESE) code regions based on liveness of variables. The choice of SESE regions over other region formation approaches greatly facilitates both the reasoning about data flow and the re-ordering of regions for commutativity testing. As SESE regions have no side entries or exits, liveness of variables only needs to be determined at a single entry and a single exit edge. Similarly, re-ordering transformations can easily be applied and correspond to the intuitive notion of moving self-contained segments of code. We initially recall a few standard definitions directly based on [61], which we are going to use to define SESE regions and their properties.

Definition 1 *A **control flow graph** G is a directed graph with distinct nodes* `start` *and* `end` *such that every node occurs on some path from* `start` *to* `end`. `start` *has no predecessors and* `end` *has no successors.*

Definition 2 *A node x is said to **dominate** node y in a directed graph if every path from* `start` *to y includes x. A node x is said to **postdominate** a node y if every path from y to* `end` *includes x. The same definition also applies to edges.*

Equipped with this we can provide a definition of SESE regions.

Definition 3 *A **Single-Entry Single-Exit (SESE) region** in a graph G is an ordered edge pair (a, b) of distinct control flow edges a and b where:*

1. *a dominates b,*
2. *b postdominates a, and*
3. *every cycle containing a also contains b and vice versa.*

For each edge e in the graph, we want to find the smallest SESE regions, if they exist, for which e is an entry edge or an exit edge. We will call these the canonical SESE regions associated with e. We express this more formally as follows.

Definition 4 *A SESE region (a, b) is **canonical** provided*

1. *b dominates b' for any SESE region (a, b'), and*
2. *a postdominates a' for any SESE region (a', b).*

Definition 5 *A node n in a graph G is **contained** within the SESE region (a, b) if a dominates n and b postdominates n.*

The next theorem describes how canonical SESE regions in a graph are related.

Theorem 1 *If R_1 and R_2 are two canonical SESE regions of a graph, one of the following statements applies.*

1. *R_1 and R_2 are node disjoint.*
2. *R_1 is contained within R_2 or vice versa.*

In other words, canonical SESE regions cannot have any partial overlap – if two regions have any nodes in common, they are either nested or in tandem. SESE regions are either nested, sequentially composed, or disjoint. When regions are sequentially composed the exit edge of one region is also the entry edge of the following region.

It also follows from Theorem 1 that SESE regions can be organized as a tree. Each node in this tree represents a SESE region. The parent of a region is the closest containing region, and children of a region are all the regions immediately contained within it. We call this the program structure tree (PST). Chains of sequentially composed SESE regions – such as regions c, d and e in Figure 6.2b – are grouped together in the PST.

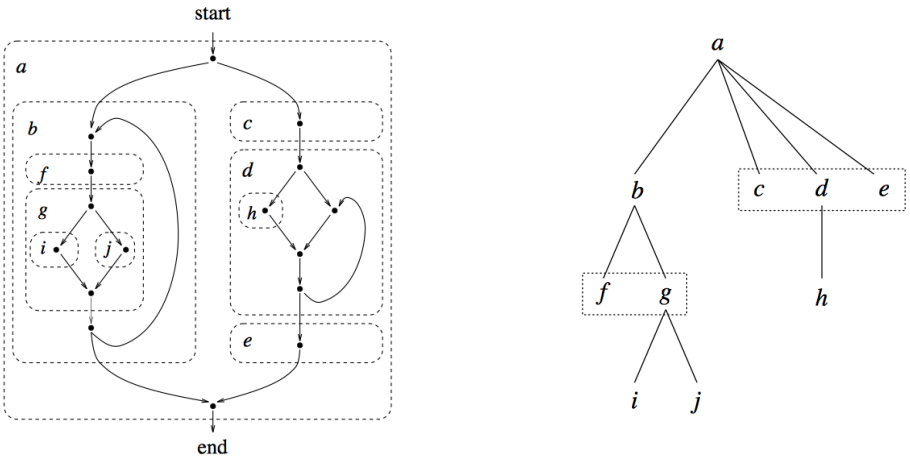
Definition 6 ***Maximal SESE regions** are non-empty chains of sequentially composed canonical SESE regions.*

6.2.3 Data-Flow Terminology

Definition 7 *A node n containing a reference that may read the value of a variable v is a **use** of the variable v. We use following notation:*

- use$[v]$ *denotes the set of CFG nodes that use variable v, and*
- use$[n]$ *denotes the set of variables that are used at node n.*

Definition 8 *A node n containing an assignment of a value to a variable v is a **definition** of the variable v. We use following notation:*



(a) Control flow graph (CFG) with hierarchically nested, canonical SESE regions.

(b) Program structure tree (PST) representing the CFG from (a) with sequentially composed canonical SESE regions grouped together. Regions f and g (and c, d, and e, respectively) can be executed concurrently if they are commutative.

Figure 6.2: A control flow graph (CFG), Figure (a), and its program structure tree (PST), Figure (b), according to [61]. Sequentially composed canonical SESE regions, grouped together in the program structure tree, are a source of task-level parallelism and can be executed concurrently if they are *commutative*, i.e. their execution order can be exchanged without changing any values *live-out* at their exit.

- def[v] *denotes the set of CFG nodes that define variable v, and*
- def[n] *denotes the set of variables that are defined at node n.*

Definition 9 *A variable v is **live** on a CFG edge e, iff*

1. *there exists a directed path from e to a use of v (node in* use[v]*), and*
2. *that path does not go through any def of v (no nodes in* def[v]*).*

Definition 10 *A variable v is **live-in** at a node n, i.e.* $v \in \text{live}_{in}[n]$*, if it is live on any of n's in-edges coming from predecessor nodes* pred[n].

Definition 11 *A variable v is **live-out** at a node n, i.e.* $v \in \text{live}_{out}[n]$*, if it is live on any of n's out-edges leading to successor nodes* succ[n].

Definition 12 *A variable v is **live-thru** at a node n, i.e. $v \in \text{live}_{thru}[n]$, if it is live-in and live-out but not defined at n. That is, if $v \in \text{live}_{in}[n] \cap \text{live}_{out}[n] \setminus \text{def}[n]$.*

Liveness information can be (iteratively) calculated from *use* and *def* as follows:

$$\begin{aligned} \text{live}_{in}[n] &= \text{use}[n] \cup (\text{live}_{out}[n] - \text{def}[n]) \\ \text{live}_{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{live}_{in}[s] \end{aligned}$$

While the above definitions refer to liveness at CFG nodes, they can be easily extended to entire SESE regions.

6.2.4 Commutativity and Commutativity Analysis

Traditionally, parallelization relies on *dependence analysis* [70]. More recently, *commutativity analysis* [104] has been proposed as an alternative analysis underpinning parallelization. This is different to dependence analysis as regions of code can still be commutative, even if there exists a dependence between them. Unfortunately, commutativity analysis is not a trivial task and, in fact, has been shown to be undecidable in general [23].

Two different notions of commutativity supporting parallelization have been suggested, namely *operations-based commutativity* [104, 105], where commutativity of code regions is based on the commutativity of the operations performed in those regions, and *effects-based commutativity* [2], where two regions are considered commutative, if their effect on the visible program state is identical for any order of execution. We have discussed these two notions and their limitations in Chapter 2.

6.3 Characterization of Algorithmic Skeletons

6.3.1 Defining Liveness-based Commutativity

Our definition combines the strengths of both versions of commutativity, while avoiding at least some of their most restrictive weaknesses. In particular, we

1. define commutativity as a *binary relation between regions of code*, which may or may not comprise function calls, rather than a property of an individual function,

2. only care about live-out and live-thru variables and their values, but do not require exact matches of memory contents for intermediate variables that are not used beyond the end of commutative regions,

3. can handle recursive commutative functions (by allowing function calls in regions) and recursive data structures (through liveness), and

4. do not require additional support for symbolic computation, but can make profitable use of it if it is available (see Section 6.4.2).

Definition 13 *Two* SESE *regions* R_1 *and* R_2 *are* ***commutative*** *iff*

1. (a) $R_1 \neq R_2$*:* R_1 *and* R_2 *are canonical* SESE *regions both contained in the same maximal* SESE *region, and* $\widehat{R}$ *is the smallest such maximal* SESE *region, and* R_1 *is not contained in* R_2 *(and vice versa), and all variables and their values in* $\text{live}_{out}[\widehat{R}]$ *are the same for any execution order of* R_1 *and* R_2*,* ***or***

   (b) $R_1 = R_2$ *("*R_1 *is commutative with respect to itself"):* R_1 *is a maximal* SESE *region contained in another* SESE *region which contains a direct control flow path from the exit of* R_1 *back to its entry, and* $\widehat{R}$ *is the smallest such containing region, and all variables and their values in* $\text{live}_{out}[\widehat{R}]$ *are the same for all execution orders for (dynamically) repeated executions of* R_1*.*

2. *The values of all variables in* $\text{live}_{thru}[R_1]$ *and* $\text{live}_{thru}[R_2]$ *are unaffected by the execution order of* R_1 *and* R_2*.*

For example, consider regions f and g in Figure 6.2a, which are contained in the same maximal SESE (f,g) in the PST in Figure 6.2b. If the live-outs and their values of (f,g) are indistinguishable for any execution order of f and g, then f and g are commutative. Similarly, if different executions orders for c, d and e result in the same live-out variables and their values of (c,d,e), then c, d and e are commutative. Note that any SESE may contain one or more function calls, and that we are not asking for commutative regions to be immediate control flow predecessor/successor pairs.

For an example of part 1(b) of Definition 13, consider the maximal SESE (f,g) in Figure 6.2a. There exists a control flow edge from the exit of (f,g) back to its entry, so that (f,g) can be executed multiple times. If any execution order of (f,g)

produces the same set of live-out variables and their values are identical once the loop terminates, (f, g) is commutative with respect to itself.

Generally, we assume that regions do not contain I/O statements or produce any other side-effects not captured by liveness (exceptions, `volatile` memory accesses, etc.). The commutativity relation is symmetric, but it is not transitive or reflexive.

6.3.2 Task-Parallel Skeletons

Primitive Task Parallelism

We start off with the simplest algorithmic skeleton, namely primitive *task parallelism*, where tasks represented as nodes (= SESE regions) appear in a sequential order in the (static) program source, but can be executed concurrently.

Definition 14 *Two SESE regions a and b are **task parallel** iff a and b are commutative.*

For example, consider Figure 6.2. Regions f and g are potential sources of task parallelism as f and g are sequentially ordered within a single SESE in the CFG, in Figure 6.2a, and, thus, are part of the same maximal SESE region in the PST, in Figure 6.2b. If f and g are additionally commutative, i.e. their execution order can be exchanged, then f and g are task parallel. Analogously, c, d, and e can be shown to be task parallel.

Functional Task Parallelism

This form of parallelism is a special case of task parallelism, where two or more task-parallel regions (according to Definition 14) each contain a function call.

Task Farm Parallelism

A *task farm* is a dynamic task-parallel algorithmic skeleton, already introduced in the motivating example.

Definition 15 *A loop L is a **task farm** iff*

1. *L is a canonical SESE region,*

2. *the maximal SESE region R representing its loop body is commutative according to Definition 13.1(b),*

3. *it consumes data* $|\text{use}[L]| = \Omega(n)$ *and produces live-out data* $|\text{live}_{out}[L] \cap \text{def}[L]| = \Omega(1)$*, where n is the number of loop iterations.*

We use Knuth-style asymptotic bounds to reason about the volume of data produced and consumed. Clause 3 can be read informally as "the loop has to consume at least n data items and has to produce at least one data item".

We allow arbitrary control flow in the loop body, any number of (non-statically determined) loop iterations, and in fact, we may generate more work items in the loop body. Our commutativity based characterization avoids complex shape analysis [92] to identify internal work list data structures. Note that we do not require tasks (=operations performed in R) to be completely independent of each other.

6.3.3 Data-Parallel Skeletons

Data parallelism is the most widely used form of parallelism and refers to scenarios in which the same operation is performed concurrently on elements of a collection (usually arrays). Data parallel operations are partitioned so that multiple threads can operate on different segments concurrently. Traditional data parallelization requires that there are no data dependences between loop iterations.

"Conventional" Data-Parallel Loops

We initially provide a commutativity characterization for ordinary data-parallel loops, also called *DO* loops [70].

Definition 16 *A simple, non-nested loop L is* ***data parallel*** *iff*

1. *L is a canonical SESE region,*

2. *the maximal SESE region R representing its loop body is commutative according to Definition 13.1(b) and does not contain any cyclic control flow,*

3. *it consumes data* $|\text{use}[L]| = \Theta(n)$ *and* $\text{use}[L] \subseteq \text{live}_{in}[L]$*, and*

4. *it produces data* $|\text{live}_{out}[L] \cap \text{def}[L]| = \Theta(n)$*, where n is the number of loop iterations.*

Note that we do not explicitly model cross-iteration dependences, but require commutativity of the loop body. We also ask for the loop to read, write and produce a live-out volume of data proportional to its iteration count. The definition can be suitably adapted for nested loops. Note that any data-parallel loop is a task farm, but the inverse is not true.

"Conventional" Parallel Reductions

Our characterization of parallel reductions is similar to that of ordinary parallel loops, but we only demand commutativity and a constant volume of live-out data to be produced irrespective of the iteration count.

Definition 17 *A simple, non-nested loop L is a* ***reduction loop*** *iff*

1. *L is a canonical Sese region,*
2. *the maximal Sese region R representing its loop body is commutative according to Definition 13.1(b) and does not contain any cyclic control flow,*
3. *it consumes data $|\text{use}[L]| = \Theta(n)$ and $\text{use}[L] \subseteq \text{live}_{in}[L]$, and*
4. *it produces data $|\text{live}_{out}[L] \cap \text{def}[L]| = \Theta(1)$, where n is the number of loop iterations.*

Map Parallelism

Map is the first actual data-parallel algorithmic skeleton we define. We use a functional view of this skeleton and demand non-destructive mapping of inputs to outputs [28].

Definition 18 *A simple, non-nested loop L is a* ***map skeleton*** *iff*

1. *L is data parallel according to Definition 16, and*
2. *it does not modify its input, i.e. $(\text{use}[L] \cap \text{def}[L]) \not\subseteq \text{live}_{out}[L]$.*

Reduce Parallelism

Similar to the previous definition of the *map* skeleton, we now provide a functionally inspired definition of the *reduce* skeleton.

Definition 19 *A simple, non-nested loop L is a **reduce skeleton** iff*

1. *L is a reduction loop according to Definition 17, and*
2. *it does not modify its input, i.e.* $(\text{use}[L] \cap \text{def}[L]) \not\subseteq \text{live}_{out}[L]$.

Map/Reduce Parallelism

Map and *Reduce* skeletons can be merged together to a combined *Map/Reduce* skeleton in following way:

Definition 20 *A sequence of a map skeleton M and a reduce skeleton R is a **map/reduce skeleton** MR iff* $\text{live}_{out}[M] = \text{live}_{in}[R]$ *and* $\text{use}[R] \subseteq \text{def}[M]$.

Fused Data-Parallel Skeletons

In practice, legacy C code will contain frequent use of fused data-parallel skeletons. This means that sequences of data-parallel loops over arrays may have been transformed into a single data-parallel loop using loop fusion. Although common in practice, the individual characterizations of such fused skeletons are outwidth the scope of this chapter, but can be derived from the definitions of the basic skeletons.

6.3.4 Resolution Skeletons

Divide & Conquer

Definition 21 *A function f implements the **divide & conquer skeleton** iff the CFG representing the body of f contains a maximal SESE region R and there exist two distinct nodes $x_f, y_f \in R$, which*

1. *each comprise a recursive call to f,*
2. *x_f dominates y_f and y_f postdominates x_f,*
3. $\text{use}[x_f] \neq \text{use}[y_f]$ *(but possibly* $\text{use}[x_f] \cap \text{use}[y_f] \neq \emptyset$*), and*
4. *x_f and y_f are commutative.*

Informally, divide & conquer is represented by a function that contains two commutative recursive calls. Clause 3 further stipulates that these calls may share some of the same arguments (e.g. in the case of MergeSort, a pointer to the array being sorted) but must differ in at least one argument (e.g. the bounds of the array segments to be sorted).

A similar definition applies for divide & conquer skeletons of degree greater than two. A dependence based characterization and parallelization approach of *divide & conquer* algorithms is provided in [106].

6.3.5 From Concurrency to Parallelism

The commutativity-based characterization of skeletons presented above enables the identification of sources of parallelism, but so far only shows that commutative tasks can be executed *concurrently*, i.e. in any (non-deterministic) order. However, this does not provide correctness guarantees for the simultaneous (=*parallel*) execution of tasks.

To accomplish this, dependences need to be brought back into play during parallelism extraction and mapping. Additional synchronization of accesses to shared variables or privatization may be required to allow parallel execution. The degree to which this needs to be done and the specific mechanisms that can be used, such as locking or speculative execution, depend on the target platform and execution model. The approach described in this chapter specifically only targets the detection of skeletons; the step from concurrency to parallelism extraction and mapping is left for a subsequent code generation and scheduling stage of parallelization.

6.4 Detection of Algorithmic Skeletons

In the following sections, we describe a hybrid static/dynamic framework to detect algorithmic skeletons in application code. As the characterization of skeletons presented in this chapter is founded on established concepts such as SESE regions, commutativity, and liveness analysis, it is possible to leverage existing analyses commonly available in many compiler frameworks and, in particular, the LLVM compiler infrastructure.

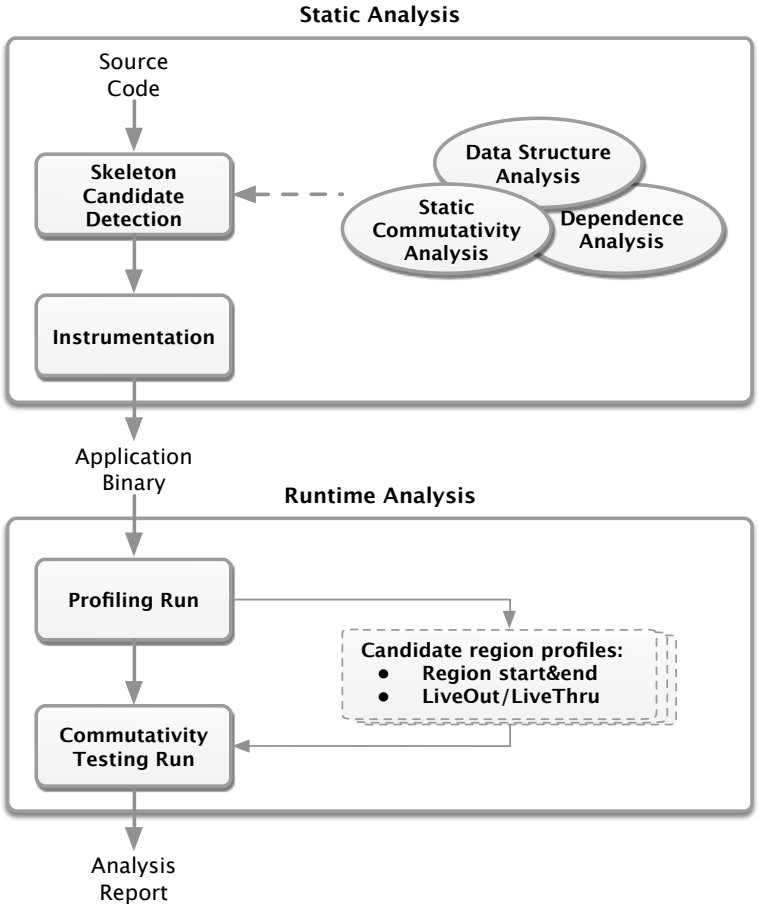


Figure 6.3: Overview of the skeleton detection framework

6.4.1 Overview

Figure 6.3 shows a high-level overview of the skeleton detection framework. It combines both static analysis, implemented in a compiler framework, and dynamic analysis using instrumentation that targets a custom runtime library.

6.4.2 Static Analysis

The static analysis step takes unmodified original source code of the application as its input. This is compiled in the usual way to obtain a whole-program intermediate representation of the application. A number of well-known compiler analyses are then applied to this intermediate representation, such as dependence analysis, shape analysis of data structures, liveness analysis of values, and the detection of SESE regions. If available, static commutativity analysis is also used.

The analysis then selects candidate skeletons. Skeletons are detected based on the characteristics described earlier, which can be expressed as a combination of

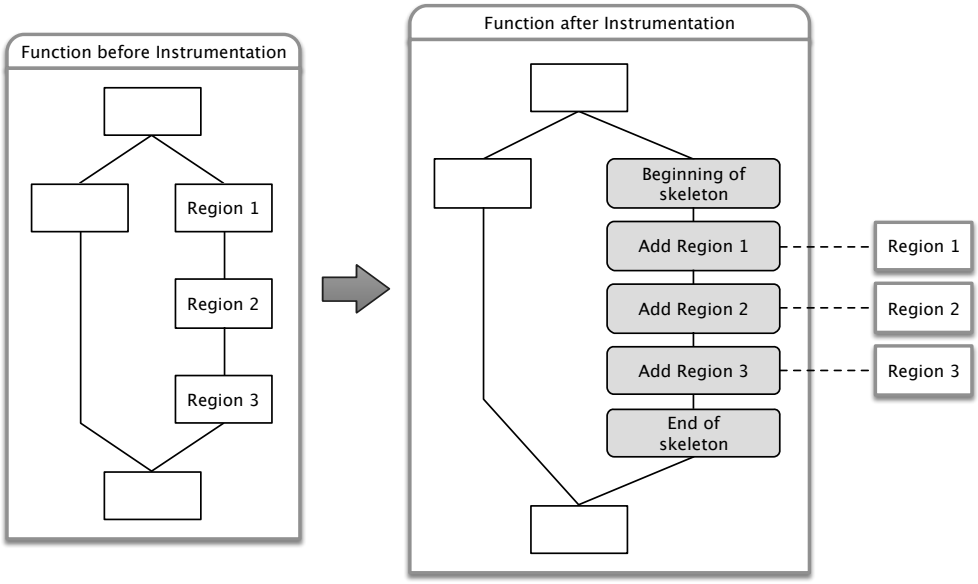


Figure 6.4: Instrumentation of skeleton for dynamic analysis. In this case, the skeleton candidate consists of three possibly commutative regions (Region 1, 2, and 3). The regions are outlined into separate functions, and all memory references and function calls contained in them are instrumented. The original code is replaced by calls into the commutativity testing runtime library (gray boxes) that indicate start and end of the candidate skeleton and add the regions for commutativity testing.

structural traits and the results of existing compiler analyses. The detectors are currently hard-coded for each skeleton. For instance, the *functional task parallelism* skeleton is detected by traversing the control flow graph and identifying all sequences of two or more calls that do not have register dependences between them. The development of a more concise domain-specific language (DSL) to describe the syntactic structure of skeletons is conceivable in future work. Lattner's data structure analysis (DSA) [73] is used to determine commutativity between calls statically. In most cases, the conservative nature of static analysis renders final determination of the commutativity of code regions and the effects on live values impossible. This makes it necessary to also leverage dynamic information about the actual application behavior and combine it with the results of static analysis.

In the instrumentation phase (Figure 6.4), we therefore instrument all memory accesses and external function calls. We mark the start and end of possible skeletons by calls into a runtime library. Potentially commutative code regions are extracted into separate functions; their original locations are replaced by calls into

the runtime library with pointers to the extracted functions. This allows the library to later re-order the regions for commutative testing. In summary, we produce an instrumented application binary that behaves identically but allows profiling of the skeleton instances and commutativity testing at runtime. The next section describes this step in detail.

6.4.3 Dynamic Liveness and Commutativity Analysis

The dynamic analysis is performed in two stages by running the instrumented application twice and applying different modes of the runtime library.

Live Memory Profiling. The first stage produces a profile of live memory locations for each instance of a candidate skeleton. We collect all memory addresses that are defined by the skeleton's code and used later in the execution (the skeleton instance's live-out locations). We furthermore record the memory addresses that were defined before the skeleton instance's execution and will be used afterwards but are not overwritten by the skeleton's code (its live-thru locations, see Section 6.2.3). Finally, we record a number of other data points such as the start and end of the skeleton instance (in terms of instruction counts) to allow us to verify that the application behavior has not changed between runs as this would potentially invalidate our profile. It is important to stress that we profile each *instance* of a candidate skeleton. This is because a skeleton candidate may appear in many different execution contexts during application runtime, only some of which may be actual manifestations of the skeleton.

Commutativity Testing. The second stage loads the profiling information for each candidate skeleton instance that was gathered in the profiling stage. For each instance, we test the commutativity of the skeleton's code regions with respect to live-out and live-thru locations using a fork-join model (see Figure 6.5). The main application process always executes the original order of the potentially commutative regions. Before the execution of this original region order, we fork out a separate testing process which in turn forks out a number of child processes to execute all possible order permutations of the regions to be tested for commutativity. We mark regions as non-commutative if a child process overwrites live-thru locations or produces a different value in a live-out location than the main process

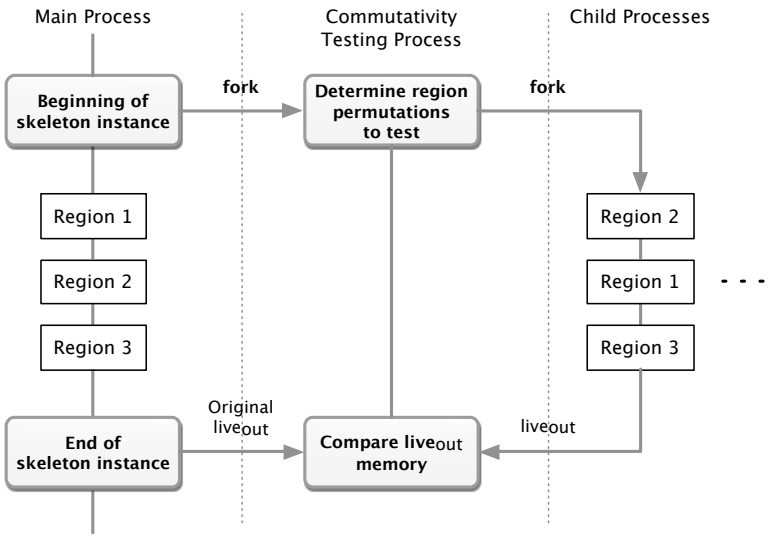


Figure 6.5: Runtime commutativity testing using a fork-join model

which ran the original execution order. Depending on the skeleton, such violations either establish weak-ordering constraints between some of the regions or discount the skeleton candidate entirely for the particular instance.

6.4.4 Importance of Context Sensitivity

We make a distinction between **candidate skeletons** and **skeleton instances**. The former refers to a region of *source code*, identified during static analysis, that *may* represent a skeleton but could not be proven as such. The latter denotes an actual execution of this code region at runtime where it has been *confirmed* as the supposed skeleton using dynamic analysis.

A candidate skeleton may appear in different program contexts at runtime. This is the case, for instance, with skeletons in library routines that are called from multiple locations within the application. The input data and the use of the output data may differ depending on the program context. Hence, the candidate skeleton may behave very differently as well.

For this reason, we see the question of whether a candidate skeleton is an actual skeleton as a *context-sensitive* property. A candidate skeleton may be an actual skeleton instance in one program context but not in another in the same run of the application. The dynamic analysis described in the previous section therefore tests each instance of a candidate skeleton separately.

6.4.5 Implementation Issues

To ensure correctness and practical feasibility of this approach, a number of implementation pitfalls have to be avoided.

Stability of memory addresses between runs. Live memory profiling and commutativity testing cannot happen in the same run of the application since the live-out locations for any given candidate skeleton instance can generally only be known once the application terminates. As described above, our approach first collects liveness information in a profiling run and then tests for commutativity in a second run of the application. Thus, we need to ensure that the application not only executes the same code between profiling and testing runs, but also accesses the exact same memory locations each time. Modern operating systems use *address space layout randomization* (ASLR) for security, so this needs to be disabled. Secondly, the runtime library needs to use its own heap and stack to ensure that its different modes of operation have no impact on the original application's address space. This is achieved using a combination of a custom memory allocator and explicit context switching. Finally, heap state and stack pointer values are compared between runs to prevent reporting incorrect results.

Nesting of candidate skeletons. At runtime, skeleton candidates may appear nested within other skeleton candidates. In the interest of detecting as many skeletons as possible it is important to expose all skeletons regardless of nesting depth. To simplify commutativity testing, our framework tests each nesting depth separately.

Risk of combinatorial explosion. As we are testing commutativity of a set of regions, all possible permutations of the set have to be executed. This has factorial growth, so a candidate skeleton instance with 10 regions already needs to be executed 3,628,800 times. While this cannot be avoided for actually commutative regions without risking correctness, there are some ways to mitigate this problem. Firstly, the runs can take place in parallel. Secondly, we employ an iterative strategy to gradually cut down the permutation space by quickly identifying pairs of regions that are not in fact commutative. At first, neighboring regions are swapped and those execution orderings tested. This will establish neighboring pairs of regions that do not commute. In each of the following iterations, we increase the

distance between the pairs of regions that are swapped while maintaining non-commutativity constraints already identified in previous iterations. At the end of this process, we know about the commutativity between each pair of regions in our given set. If there is no commutativity, we already know this after the first iteration. If only some of the regions commute, the remaining permutation space is at least reduced to some extent.

Unexpected behavior by regions. Dynamic commutativity testing is somewhat brute-force in its reordering of program regions. If a guess for a candidate skeleton is incorrect and the commutativity property does not hold, this may lead to pathological application behavior in the child processes that execute region orderings different from the original program order. Firstly, the child process may never terminate. This is prevented by measuring the runtime of each region during profiling and using this as a timeout during commutativity testing. Secondly, a child process may write into random memory locations. Since all memory accesses are instrumented, *address sanitization* can be employed. On every memory access, we verify that it is not illegal (e.g. a null pointer dereference, writing to code regions, or a write to a live-thru location). Any such illegal access is seen as a commutativity violation. Note however that we explicitly allow access to any *legal* memory location even if such an access did *not* happen in the original execution order. This is a fundamental principle of our liveness-based definition of commutativity. Such an access will either affect a memory location that has no further bearing on program execution after the end of the skeleton or will be a live-out, whose value will be checked explicitly against that resulting from the original execution order.

6.5 Empirical Evaluation

We have evaluated our proof-of-concept implementation of the skeleton detection framework on a number of benchmarks. We focus on the detection of the *functional task parallelism* skeleton (see Section 6.3.2), a fundamental pattern that has also been the target of previous commutativity studies [2]. It allows us to demonstrate the working of the hybrid static-dynamic detection framework and to evaluate key factors such as the context sensitivity of skeleton detection. The functional task parallelism skeleton includes *Divide and Conquer* (see Section 6.3.4) as a special case. Other skeletons will be added to the framework as part of future work.

6.5.1 Benchmarks

Our evaluation focuses on the SPEC CPU2006 benchmark suite, which was described in Chapter 3. We use benchmarks implemented in C and C++; they include both integer and floating point codes. The benchmarks were compiled using CLANG/LLVM 3.4 with `-O1` optimization level applied to the application code at link-time. Dynamic information is obtained from complete runs of the benchmarks. In order to reduce runtimes, we use smaller data sets provided with the SPEC benchmark suite.

Limitations. Some of the SPEC benchmarks (`dealII`, `soplex`, `povray`, `omnetpp`, and `xalancbmk`) have non-deterministic memory layouts due to the use of library functions for timing and subsequent printing of this information. Dynamic commutativity testing requires the memory layout to be stable between runs, so these benchmarks could not be run as-is and had to be excluded. A possible workaround may be to capture and emulate non-deterministic system library calls. Furthermore, the number of candidate skeleton instances in `astar` and `sphinx3` was extremely high (tens of millions) due to their location in loops with high iteration counts. This exceeded the processing capabilities of the system used for this study. This is a particular problem with the *functional task parallelism* skeleton, a skeleton which is detected very frequently due to its generic nature, and is likely not to be an issue with more complex skeletons. It may be possible to mitigate this by using additional heuristics in the static analysis stage to reduce the number of candidates. The above limitations are practical rather than conceptual and should not affect the validity of the empirical evaluation in this chapter.

6.5.2 Prevalence of Functional Task Parallelism

The skeleton selected for our evaluation, functional task parallelism, is a very generic skeleton. Any sequence of calls in a basic block that cannot statically be determined as dependent qualifies as a candidate. We therefore expect to encounter a large number of potential instances of this skeleton at runtime.

Figure 6.6 shows the time spent in potential and actual skeleton instances as a share of execution time for our benchmark applications.

The results show that many of the benchmarks spend nearly all of their execution time in potential skeleton instances. In `perlbench` and `namd`, on the other

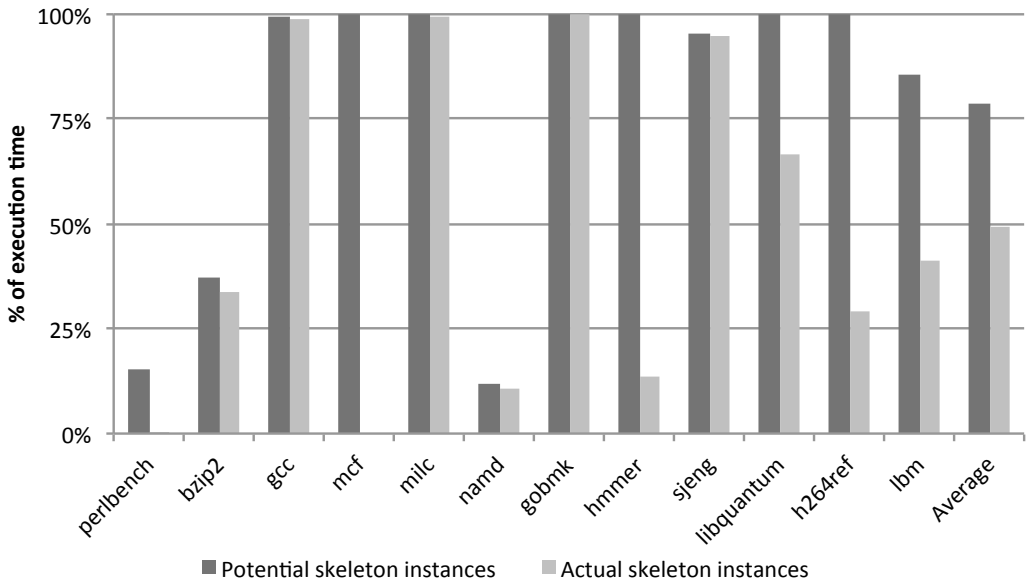


Figure 6.6: Percentage of execution time spent in *potential* and *actual* functional task parallelism skeleton instances. Actual skeleton instances are those candidates which were confirmed by dynamic analysis to be instances of the skeleton.

hand, these make up less than 20% of the running time. On average, 79% of execution time is spent in potential instances. Not all potential instances are obviously actual instances of the skeleton. During dynamic commutativity testing, differences in live-out values or the violation of the requirement to preserve live-thru values may lead to a potential instance being discounted as a skeleton instance.

This is most noticeable for `mcf`, which was found to contain no functional task parallelism at all, and for `perlbmk`, where less than 1% of execution time is spent in regions that could be confirmed as actual skeleton instances. Other benchmarks that exhibit significant drops are `hmmer`, `libquantum`, and `lbm`. The remaining benchmarks spend nearly the same amount of execution time in actual skeleton instances as they do in potential ones, suggesting that the majority of these candidates could be confirmed as containing some degree of commutativity. The average lies at just over 50%, which indicates that the SPEC benchmarks spend a significant amount of time in regions representing the functional task parallelism skeleton.

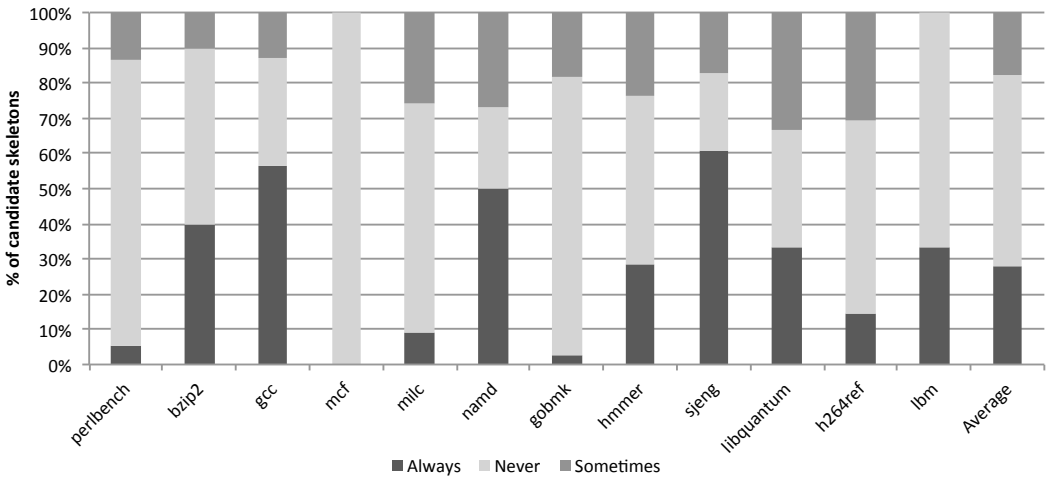


Figure 6.7: Context sensitivity of skeletons, showing the percentage of candidate skeletons in benchmark source code confirmed as actual instances in all, none, or some of their occurrences at runtime.

6.5.3 Context Sensitivity

We have previously discussed the importance of context sensitivity for skeleton detection. A candidate skeleton in the source code can manifest in several potential instances at runtime, only some of which might later be confirmed as actual skeleton instances by dynamic analysis. We can therefore divide candidate skeletons into three categories: firstly, candidates whose instances at runtime are *always* actual instances of the skeleton; secondly, those which are *never* found to be actual instances and hence were misdetected as candidates in the first place; and thirdly, candidate skeletons which are only *sometimes* found to be actual instances of the skeleton.

Figure 6.7 shows these statistics for the SPEC CPU2006 benchmarks. The results confirm the importance of a context-sensitive approach. Across the benchmark suite, an average of 18% of the candidate skeletons fall into the *'sometimes'* category. It thus depends on the execution context whether these candidates are actual skeleton instances. More than half of the candidate skeletons (54% on average) turn out never to be actual skeleton instances. This exposes the weaknesses of static analysis which is unable to make the determination at compile-time. However, this is to be expected especially for the *functional task parallelism* skeleton since disproving commutativity of two or more function calls requires complex inter-procedural

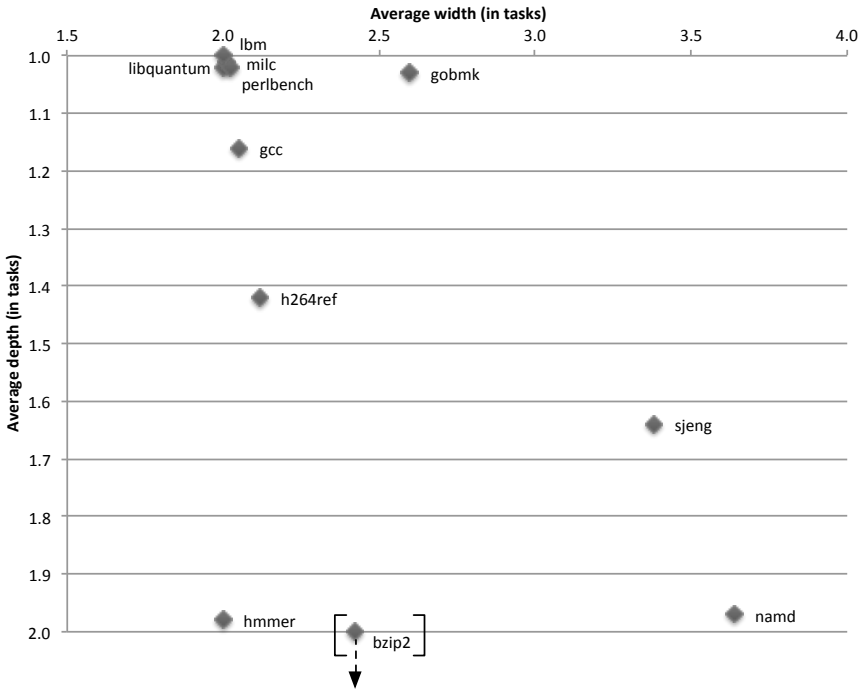


Figure 6.8: Average width and depth of skeleton instances (in number of tasks) after parallel scheduling. `bzip2` has an average depth of 38.4 tasks, which exceeds the range of the chart.

analysis that stretches the limits of even state-of-the-art techniques (discussed in Chapter 2).

Some benchmarks, such as `gcc` and `sjeng`, show a high percentage of candidates (>50%) that can always be confirmed to be actual instances of the skeleton. On average, however, only about one fifth of all candidates are always actual skeleton instances. `mcf`, in fact, has no actual skeleton instances at all.

6.5.4 Degree of Concurrency and Critical Path Length

Having determined that a particular candidate instance is an actual instance of the skeleton, the degree of concurrency of the instance becomes an important aspect. It determines the number of tasks that can be run in parallel given adequate synchronization of accesses to shared memory locations. For the functional task parallelism skeleton, the degree of concurrency is not necessarily equal to the number of function calls in the skeleton; this is because dynamic commutativity testing may reveal that some of the calls are not commutative and thus need to be run in sequential order. Such sequentialization establishes the critical path of

the skeleton, the longest sequence of tasks that needs to be run in sequential order, which is another aspect to characterize the detected skeleton instances.

Borrowing terminology from digital circuit theory [140], the maximum degree of concurrency can be described more concisely as the **width** of a skeleton instance and the critical path length as its **depth**.

We obtain the widths and depths of skeleton instances from the results of dynamic commutativity analysis by scheduling each skeleton instance using the dominant sequence clustering algorithm (DSC) of the PYRROS task scheduler [138]. This algorithm is widely used and a reference implementation is freely available from its authors. As we are interested in the maximum degree of concurrency for each instance, we set no limit on the number of available processors.

In Figure 6.8, the average width and depth of the skeleton instances across our benchmarks is shown. We can immediately see a clustering of benchmarks in the 2 to 3 width range. The average depths are also concentrated in one region, between 1 and 2 tasks. A typical function task parallelism instance can thus overlap about two tasks and may have to run a third one sequentially. However, there are some outliers: `bzip2` has a very high average depth of 38. This is mainly due to the effects of loop unrolling which creates basic blocks with large numbers of function calls only some of which are commutative. `namd` achieves the highest average width of 3.6 tasks suggesting greater potential for parallelism in that benchmark.

Overall, the results imply that a modest but noticeable amount of concurrency can be exposed using the functional task parallelism skeleton. Given the relatively primitive nature of the skeleton, this is an encouraging result especially since the data is not the result of an abstract static analysis but represents actual application behavior.

6.6 Summary and Conclusions

In this chapter we have developed a formal characterization of some popular algorithmic skeletons based on a novel notion of commutativity. This notion is based on well-understood concepts in compiler theory: liveness and SESE regions. We have shown that many algorithmic skeletons have a simple and elegant commutativity characterization. We overcome limitations of static analysis by complementing it with profiling information and have presented a hybrid static/dynamic framework for the detection of algorithmic skeletons in application code. In our empirical

evaluation, we have demonstrated how a proof-of-concept implementation of this framework can be used for the detection of a typical skeleton and investigated some of the important factors in algorithmic skeleton detection, such as context sensitivity and degree of concurrency.

The novel characterization of skeletons based on commutativity and liveness overcomes limitations of earlier dependence-based approaches and represents a promising new direction for the detection of structured parallelism in legacy applications. To widen its practical applicability, further skeletons as well as variations of the ones presented in this chapter will need to be characterized in future work. Our static/dynamic detection framework will be extended to cover a wider range of skeletons and will be integrated with a code generation/mapping stage to exploit the parallelism in practice.

Chapter 7

Conclusions

This thesis has investigated the extraction of parallelism from sequential legacy applications using two distinct techniques: region-based dynamic binary parallelization on one hand, and the detection of algorithmic skeletons on the other hand. We have presented a parameterizable model of dynamic binary parallelization and used it to establish the limits of the technique. The variability of data dependences and control flow, a critical factor in dynamic parallelization approaches, has been investigated in a large-scale study involving 100 different input data sets. Finally, we have introduced a novel characterization of algorithmic skeletons that enables their detection in legacy source code and developed a hybrid static/dynamic framework to support this process.

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 our work and discusses directions for future research.

7.1 Contributions

7.1.1 Limits of Dynamic Binary Parallelization

Chapter 4 introduced a parameterizable model of DBP capable of representing a wide range of possible implementations. Using this model, we experimentally evaluated the limits of region-based dynamic binary parallelization for a CMP platform with hardware support for speculation.

We demonstrated that there is room for a significant reduction of up to 62% in the number of instructions on the critical paths of legacy SPEC CPU2006 bench-

marks. This underlines the power of dynamic analysis, since traditional static parallelization approaches fail to extract parallelism from these benchmarks.

However, we showed that it is much harder to translate these savings into actual performance improvements, with a *realistic* hardware-supported implementation achieving a speedup of 1.09x on average. For a small number of relevant benchmarks DBP shows good performance gains, whereas for other benchmarks the improvements are rather small.

Our study confirms that for realistic speculative execution costs DBP suffers from diminishing returns. Cores in a many-core CMP cannot be made arbitrarily small without impeding single-thread performance, since DBP in its current form can only compensate for a relatively small loss of single-core performance through parallel execution.

7.1.2 Variability of Data Dependences and Control Flow

In Chapter 5, we presented an empirical analysis and characterization of the variability of both data dependences and control flow across program runs. We ran the CBENCH benchmark suite with 100 randomly chosen input data sets and recorded complete control and data flow traces. We introduced a new variability analysis for data dependence and control flow information based on an overlay control and data flow graph (CDFG) that captures the aggregated dependences and control paths from all executions.

For the CBENCH benchmarks, we found that for most applications data dependences and control flow show significant variability. On average, the cumulative profile information gathered with at least 55, and up to 100, different input data sets was needed to achieve full coverage of the data flow observed across all runs. For control flow, the figure stood at 46 and 100 data sets, respectively. This suggests that profile-guided parallelization needs to be applied with utmost care, as misclassification of sequential loops as parallel was observed even when up to 94 input data sets are used.

We also found that variability of data dependences was only weakly correlated with the ability to detect parallelizable loops. This is because loop parallelization is limited by the weakest link, i.e. a single loop-carried dependence is sufficient to prevent parallelization, and it appears that such dependences inhibiting parallelization are fairly stable across different program inputs. Our data confirms

that profile-guided approaches are genuinely more capable in detecting parallelism than their static counterparts.

We determined that TLS based parallelization techniques, which can exploit parallelism in loops where the majority of iterations, but not all, are independent, have limited potential for the benchmarks considered in our study.

7.1.3 Automated Detection of Algorithmic Skeletons

Chapter 6 argued that automatically parallelizing compilers often have a narrow focus on detecting data parallel loops, while parallelism exists in a plethora of other shapes commonly described as algorithmic skeletons. Motivated by the fact that skeletons are largely informal programming abstractions and lack a formal characterization in terms of established compiler concepts, we developed a formal characterization of some popular algorithmic skeletons based on a novel notion of commutativity, which in turn is largely based on liveness and SESE regions – well understood concepts in compiler theory and available in most compilers. We showed that many algorithmic skeletons have a simple and elegant commutativity characterization.

We described a hybrid static/dynamic analysis framework for the context-sensitive detection of skeletons in legacy code that overcomes limitations of static analysis by complementing it with profiling information. A proof-of-concept implementation in the LLVM compiler infrastructure was evaluated against SPEC CPU2006 benchmarks for the detection of the *functional task parallelism* skeleton. We showed that this skeleton is prevalent across the benchmark suite and at times context-sensitive in nature (18% of occurrences on average). We also introduced the concept of width and depth of a skeleton instance as a way of measuring a skeleton's concurrency.

7.2 Critical Analysis and Future Directions

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

7.2.1 Limitations of our DBP Study

We briefly discuss some of the limitations of the approach chosen for our study of dynamic binary parallelization in Chapter 4.

Choice of thread identification method. We use region tracing as opposed to other thread identification methods previously described in the literature (see Section 2.2). This choice of tracing was motivated by the large body of work on JIT compilation, where region-based tracing has emerged as a sweet spot between the two extremes of method- and instruction-based JIT compilation [118, 56, 19], balancing the cost of JIT compilation and scope for optimization. Such a region trace-based approach is sufficiently generic to subsume other, more fine-grained, thread identification algorithms since it can express loop, task, and data parallelism at various granularities. It is thus a step towards the detection of *structured* parallelism in the context of DBP.

Granularity of region traces. Nonetheless, region traces might be too fine-grained to capture all parallel regions. In our approach, regions are bounded by the page size and can thus reach up to 8KB. This should capture most parallel loops. In fact, our experiments have shown that larger traces provide limited benefit due to the increasing number of memory and register dependences.

Overhead of tracing. Our approach is lightweight because we only record basic block entry points (i.e. memory addresses) as nodes, and pairs of source and target entry points as edges in the CFG region traces. In our experiments, we simulate the incremental construction of region traces in trace intervals and thus the delayed availability of opportunities for parallelization (see section 4.2.2) but we do not model the specific overheads of recording the region traces in the course of execution. It would be possible to implement this lightweight trace recording efficiently in hardware. Compared to the overheads of speculative execution, the analysis of which is the goal of our study, these overheads are significantly smaller.

Level of abstraction. We operate on the level of machine instructions. No attempt was made to raise the level of abstraction, e.g. to reconstruct polyhedral loop representations [95]. This is in line with the definition of DBP in related work [53, 29, 137]. A higher level of abstraction would almost certainly create a larger scope for the discovery of parallelism in loops, but the costs would be prohibitive for an on-line method like DBP.

Future work. Our study of DBP focused on a purely dynamic implementation, where the parallelization system has no prior access to the binary before execution and every application run is treated as a new event. However, we envisage a much greater potential for the technology if prior information – such as profiling data from previous runs of the application and static analysis of the binary – is available in a hybrid system and can be exploited during dynamic parallelization. As mentioned in our review of related work, binary parallelization schemes that rely on a static analysis stage have already been proposed [68, 67, 95], but these have not yet been integrated into a full DBP system where parallelism is discovered at runtime. This further evolution of DBP would seem a promising area for future research and may be able to overcome the practical limitations we have encountered in our study.

7.2.2 Profile Sensitivity of DBP and Skeleton Detection

In Chapter 5, we investigated the variability of data dependences and control flow given different input data sets and determined that this variability can be considerable. This has implications for the evaluation of parallelization approaches that depend on profiling information, such as DBP and our hybrid static/dynamic skeleton detection approach.

In our experimental evaluation of these two techniques, we only used one input data set from the SPEC CPU2006 benchmark suite. This is common practice; in fact, the benchmark suite comes with only three input data sets, each of a different size. It was also a pragmatic choice due to the experimental overheads of full-system simulation for DBP and dynamic analysis for skeleton detection.

The reported results are certainly valid for the input data sets used and allow adequate conclusions about the techniques to be drawn. We also expect the results to have relatively low variance if the experiments were to be repeated with further data sets; this is because they rely on macro- rather than microscopic dependence patterns, and our study showed that these exhibit much lower variance.

Nonetheless, it would be an interesting direction for future research to investigate the profile sensitivity of both DBP and algorithmic skeleton detection.

A further avenue for future research is in the automated generation of input data sets, which sufficiently cover a program, while at the same time retaining

"typical" usage patterns such that profile-guided parallelization can be used with more confidence and without degenerating to conservative worst-case behaviors.

7.2.3 Expansion of Skeleton Detection Framework

In Chapter 6, we characterized a number of popular skeletons using a novel commutativity based approach, but that list is by no means exhaustive. Additional skeletons, e.g. dynamic programming, stencils, etc., can be added in future work. It remains to be seen how well these additional skeletons will lend themselves to our characterization approach and hybrid static/dynamic analysis. The development of a domain specific language (DSL) to describe the structural characteristics of skeletons would facilitate this expansion of our framework. In this way, the developers of skeleton libraries could deliver both a highly optimized *implementation* as well as a compiler-friendly *characterization* of skeletons for their detection in legacy applications.

Not all algorithmic skeletons may be characterized by commutativity, especially not those which do not change the order of execution, such as pipelines. Specialized techniques, which we discussed in Chapter 2, may still be required to address these skeletons separately.

While we describe techniques to reduce the permutation space of dynamic commutativity testing, the availability of better commutativity analysis would reduce the overheads further. It would also potentially widen the range of skeletons that can be detected. We envisage the development of novel commutativity analyses integrating static, dynamic, and probabilistic techniques.

At the moment, we require bitwise identity of live-out variables. We cannot deal with e.g. unordered container data structures, which contain the same elements, but in a different order. Awareness of data structures and the commutativity of operations on them [63] will be a crucial aspect of future work in this area.

While our work is primarily concerned with the *detection* of parallelism, it would be desirable to develop an end-to-end framework that actually exploits the detected parallelism by generating code for existing, optimized skeleton frameworks, such as TBB, MAPREDUCE, or any of the other skeleton libraries described in [47]. Previous work on auto-tuning of skeletons [46] could be leveraged in the process of mapping the extracted parallelism to the target architecture. The use of transac-

tional memory for optimistic, speculative parallelization also currently remains unexplored in this context.

It is our hope that this research will spark renewed interest in supporting techniques, such as commutativity analysis, shape analysis, and pattern recognition, since the detection of algorithmic skeletons provides a compelling and unifying use case for these hitherto disjoint techniques.

Finally, the automated detection of algorithmic skeletons may also have applications to the optimization of *parallel* legacy code. It remains to be investigated whether this approach could be used for the detection of communication patterns in parallel code and the transformation of applications from legacy programming models to more recent ones.

7.2.4 Safety of Dynamic Analysis

We advocate the use of a hybrid static/dynamic approach for algorithmic skeleton detection. The dynamic stage of the analysis relies on profiling data from actual application runs to make inferences about parallelizable code regions. Like all profiling-based approaches, this is inherently unsafe as such information is only valid for the input data sets the application was profiled with. In the general case, it is theoretically impossible to disprove that there exists input data that will lead to these inferences being incorrect.

This leaves three options: firstly, speculation could be used to detect and roll back cases where incorrect assumptions have been made from profiling data[1]; secondly, the user could be presented with the results of the analysis and asked to make the final decision using additional semantic knowledge about the application and input data; and finally, one could rely solely on the provably correct results of static analysis. Given that such analyses are strictly less powerful even for simple cases like loop parallelization (see Chapter 5), this last option cannot be considered a viable alternative.

After decades of research, we see the field of automatic parallelization and compiler optimization in general at a turning point. In the past, the compiler and its optimizer was largely treated as a black box, producing highly optimized binary code from user-supplied source code, but leaving the user in the dark on its inner workings. This leads to overly conservative analyses as only provably correct

[1] Recent work by Wang et al. [133] has shown that the costs of speculation can be significantly reduced for 'probably parallel' programs, making this a more attractive option again.

deductions can be made without involving the user's semantic domain knowledge. As software development itself is becoming an interactive process connecting the user and the compiler (cf. Apple's new 'Swift Playgrounds' in the XCode Ide [6]), we believe that a similar evolution will happen in code parallelization and optimization. Powerful, dynamic, skeleton-based analyses and transformations will detect and extract structured parallelism at multiple granularities in an interactive process. Adaptive runtime environments will then exploit this parallelism on heterogeneous target platforms, leveraging the full power of modern computer systems.

Bibliography

[1] C. Aguston, Y. Ben Asher, and G. Haber. Parallelization hints via code skeletonization. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 373–374, New York, NY, USA, 2014. ACM.

[2] F. Aleen and N. Clark. Commutativity analysis for software parallelization: Letting program transformations see the big picture. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 241–252, New York, NY, USA, 2009. ACM.

[3] F. Allen, M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. A framework for determining useful parallelism. In *Proceedings of the 2nd International Conference on Supercomputing*, ICS '88, pages 207–215, New York, NY, USA, 1988. ACM.

[4] O. Almer, I. Böhm, T. Edler von Koch, B. Franke, S. Kyle, V. Seeker, C. Thompson, and N. Topham. Scalable multi-core simulation using parallel dynamic binary translation. In *2011 International Conference on Embedded Computer Systems (SAMOS)*, pages 190–199, July 2011.

[5] E. R. Altman, D. R. Kaeli, and Y. Sheffer. Welcome to the opportunities of binary translation. *Computer*, 33(3):40–45, Mar. 2000.

[6] Apple. Swift - a new programming language for iOS and OS X. `https://developer.apple.com/swift/`, June 2014.

[7] ARC International. ARCompact instruction set architecture: Programmer's reference, April 2008.

[8] M. Arenaz, J. Touriño, and R. Doallo. Xark: An extensible framework for automatic recognition of computational kernels. *ACM Trans. Program. Lang. Syst.*, 30(6):32:1–32:56, Oct. 2008.

[9] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, Oct. 2009.

[10] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 342–351, New York, NY, USA, 1992. ACM.

[11] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 2000. ACM.

[12] P. Banerjee, J. Chandy, M. Gupta, I. Hodges, E.W., J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The paradigm compiler for distributed-memory multicomputers. *Computer*, 28(10):37–47, Oct 1995.

[13] E. Bangeman. Intel pulls the plug on 4GHz Pentium 4. Ars Technica, `http://arstechnica.com/uncategorized/2004/10/4311-2`, Oct 2004.

[14] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In R. Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 283–303. Springer Berlin Heidelberg, 2010.

[15] A. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, Oct 1966.

[16] P. Berube, A. Preuss, and J. N. Amaral. Combined profiling: Practical collection of feedback information for code optimization. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*, ICPE '11, pages 493–498, New York, NY, USA, 2011. ACM.

[17] A. Bhattacharyya. Do inputs matter?: Using data-dependence profiling to evaluate thread level speculation in BG/Q. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 401–402, Piscataway, NJ, USA, 2013. IEEE Press.

[18] G. Blake, R. G. Dreslinski, and T. Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, Oct. 2009.

[19] I. Böhm, T. J. Edler von Koch, S. C. Kyle, B. Franke, and N. Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 74–85, New York, NY, USA, 2011. ACM.

[20] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[21] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th*

Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, pages 69–84, Washington, DC, USA, 2007. IEEE Computer Society.

[22] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. MAPS: An integrated framework for MPSoC application parallelization. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 754–759, New York, NY, USA, 2008. ACM.

[23] A. Charlesworth. The undecidability of associativity and commutativity analysis. *ACM Trans. Program. Lang. Syst.*, 24(5):554–565, Sept. 2002.

[24] M. Chen and K. Olukotun. The jrpm system for dynamically parallelizing java programs. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 434–445, June 2003.

[25] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. Evaluating iterative optimization across 1000 data sets. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI'10)*, Toronto, Canada, 2010.

[26] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[27] D. Cordes, M. Engel, O. Neugebauer, and P. Marwedel. Automatic extraction of pipeline parallelism for embedded heterogeneous multi-core platforms. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, pages 4:1–4:10, Piscataway, NJ, USA, 2013. IEEE Press.

[28] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[29] M. DeVuyst, D. M. Tullsen, and S. W. Kim. Runtime parallelization of legacy code on a transactional memory system. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 127–136, New York, NY, USA, 2011. ACM.

[30] B. Di Martino and A. Bonifacio. Algorithmic concept recognition support for skeleton based parallel programming. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 132–, Washington, DC, USA, 2003. IEEE Computer Society.

[31] J. Dou and M. Cintra. A compiler cost model for speculative parallelization. *ACM Trans. Archit. Code Optim.*, 4, June 2007.

[32] C. Duckett. COBOL still not dead yet, taking on the cloud. ZDNet, `http://www.zdnet.com/cobol-still-not-dead-yet-taking-on-the-cloud-7000023462/`, Nov 2013.

[33] J. Dünnweber, S. Gorlatch, M. Griebl, and C. Lengauer. Making a Task Farm Component Parallelize Loops for the Grid. In *Integrated Research in Grid Computing CoreGRID Integration Workshop*, page 93, 2006.

[34] T. J. Edler von Koch and B. Franke. Limits of region-based dynamic binary parallelization. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 13–22, New York, NY, USA, 2013. ACM.

[35] T. J. Edler von Koch and B. Franke. Variability of data dependences and control flow. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS'2014, March 2014.

[36] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5(1):1–33, 2003.

[37] K.-F. Faxén, K. Popov, L. Albertsson, and S. Janson. Embla - data dependence profiling for parallel programming. In *International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS 2008, pages 780–785, 2008.

[38] M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[39] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 85–95, New York, NY, USA, 1992. ACM.

[40] B. Freisleben and T. Kielmann. Automated transformation of sequential divide-and-conquer algorithms into parallel programs. *Computers and Artificial Intelligence*, 14:579–596, 1995.

[41] G. Fursin and O. Temam. Collective optimization: A practical collaborative approach. *ACM Transactions on Architecture and Code Optimization*, 7:20:1–20:29, December 2010.

[42] G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam. MiDataSets: Creating the conditions for a more realistic evaluation of iterative optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2007)*, January 2007.

[43] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented design*. Addison-Wesley, Reading, MA, 1995.

[44] L. Gao, L. Li, J. Xue, and T.-F. Ngai. Loop recreation for thread-level speculation. In *International Conference on Parallel and Distributed Systems*, 2007.

[45] M. Gillespie. Preparing for the second stage of multi-core hardware: Asymmetric (heterogeneous) cores. Technical report, Intel, 2009.

[46] L. Góes, N. Ioannou, P. Xekalakis, M. Cole, and M. Cintra. Autotuning skeleton-driven optimizations for transactional worklist applications. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2205–2218, Dec 2012.

[47] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, Nov. 2010.

[48] M. Gupta, S. Mikhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 139–148, Oct 1999.

[49] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 2001 IEEE International Workshop on Workload Characterization*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

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

[51] T. Harris, J. Larus, and R. Rajwar. *Transactional memory*, volume 5 of *Synthesis Lectures on Computer Architecture*. Morgan & Claypool Publishers, 2nd edition, 2010.

[52] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[53] B. Hertzberg and K. Olukotun. Runtime automatic speculative parallelization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 64–73, Washington, DC, USA, 2011. IEEE Computer Society.

[54] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41:33–38, July 2008.

[55] H. C. Hunter and W.-M. W. Hwu. Code coverage and input variability: effects on architecture and compiler research. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 79–87, New York, NY, USA, 2002. ACM.

[56] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A trace-based java jit compiler retrofitted from a method-based compiler. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 246–256, Washington, DC, USA, 2011. IEEE Computer Society.

[57] Intel. Single-chip cloud computer: Project. `http://www.intel.co.uk/content/www/us/en/research/intel-labs-single-chip-cloud-computer.html`, 2012.

[58] N. Ioannou, J. Singer, S. Khan, P. Xekalakis, P. Yiapanis, A. Pocock, G. Brown, M. Lujn, I. Watson, and M. Cintra. Toward a more accurate understanding of the limits of the TLS execution paradigm. In *2010 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, Dec 2010.

[59] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 14–23, Washington, DC, USA, 1997. IEEE Computer Society.

[60] D. Jeon, S. Garcia, C. Louie, S. Kota Venkata, and M. B. Taylor. Kremlin: like gprof, but for parallelization. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 293–294, New York, NY, USA, 2011. ACM.

[61] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 171–185, New York, NY, USA, 1994. ACM.

[62] C. W. Kessler. Pattern-driven automatic parallelization. *Scientific Programming*, 5(3):251–274, Aug 1996.

[63] D. Kim and M. C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 528–541, New York, NY, USA, 2011. ACM.

[64] M. Kim, H. Kim, and C. Luk. Prospector: A dynamic data-dependence profiler to help parallel programming. In *HotPar'10: Proceedings of the USENIX workshop on Hot Topics in parallelism*, 2010.

[65] M. Kim, H. Kim, and C.-K. Luk. SD3: A scalable approach to dynamic data-dependence profiling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 535–546, Washington, DC, USA, 2010. IEEE Computer Society.

[66] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3): 155–163, Oct. 1988.

[67] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua. Automatic parallelization in a binary rewriter. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 547–557, Washington, DC, USA, 2010. IEEE Computer Society.

[68] V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *Proceedings of the 12th International Conference on Supercomputing*, ICS'98, pages 85–92, New York, NY, USA, 1998. ACM.

[69] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.

[70] L. Lamport. The parallel execution of DO loops. *Commun. ACM*, 17(2): 83–93, Feb. 1974.

[71] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, Dec. 1992.

[72] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[73] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 129–142, New York, NY, USA, 2005. ACM.

[74] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam. SUIF Explorer: An interactive and interprocedural parallelizer. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 37–48, New York, NY, USA, 1999. ACM.

[75] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Large-scale variability-aware type checking and dataflow analysis. Technical Report MIP-1212, Fakultät für Informatik und Mathematik, Universität Passau, Nov 2012.

[76] 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 *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[77] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 365–372, New York, NY, USA, 1999. ACM.

[78] P. Marcuello and A. González. Thread-spawning schemes for speculative multithreading. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 55–, Washington, DC, USA, 2002. IEEE Computer Society.

[79] B. D. Martino and G. Iannello. Pap recognizer: A tool for automatic recognition of parallelizable patterns. In *Proceedings of the 4th International Workshop on Program Comprehension*, WPC '96, pages 164–173, Washington, DC, USA, 1996. IEEE Computer Society.

[80] B. D. Martino and C. W. Keßler. Program comprehension engines for automatic parallelization: A comparative study. In *Proceedings of the First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 146–157, London, UK, UK, 1996. Chapman & Hall, Ltd.

[81] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[82] K. S. McKinley. Evaluating automatic parallelization for efficient execution on shared-memory multiprocessors. In *Proceedings of the 8th International Conference on Supercomputing*, ICS '94, pages 54–63, New York, NY, USA, 1994. ACM.

[83] C. Moore. Data processing in exascale-class computer systems. In *The Salishan Conference on High Speed Computing*, April 2011.

[84] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[85] C. Nugteren and H. Corporaal. Introducing 'Bones': A parallelizing source-to-source compiler based on algorithmic skeletons. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 1–10, New York, NY, USA, 2012. ACM.

[86] C. E. Oancea and L. Rauchwerger. Logical inference techniques for loop parallelization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 509–520, New York, NY, USA, 2012. ACM.

[87] T. Olzak. Virtualization: Dealing with legacy apps. *TechNet Magazine*, Apr 2012.

[88] J. Ortega-Arjona. *Patterns for parallel software design*. Wiley, 2010.

[89] V. Packirisamy, A. Zhai, W.-C. Hsu, P.-C. Yew, and T.-F. Ngai. Exploring speculative parallelism in SPEC2006. *IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[90] K. Pingali. Why compilers have failed and what we can do about it. Keynote Languages and Compilers for Parallel Computing, 2010.

[91] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[92] J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 37–56. Springer Berlin Heidelberg, 1994.

[93] M. Poldner and H. Kuchen. On implementing the farm skeleton. In *Proceedings of the 3rd International Workshop HLPP 2005*, 2005.

[94] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 1–11, New York, NY, USA, 2011. ACM.

[95] B. Pradelle, A. Ketterlin, and P. Clauss. Polyhedral parallelization of binary code. *ACM Trans. Archit. Code Optim.*, 8(4):39:1–39:21, Jan. 2012.

[96] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.

[97] W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM Trans. Program. Lang. Syst.*, 20(3):635–678, May 1998.

[98] S. Rapps and E. J. Weyuker. Data flow analysis techniques for test data selection. In *Proceedings of the 6th International Conference on Software Engineering*, ICSE '82, pages 272–278, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[99] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 218–232, New York, NY, USA, 1995. ACM.

[100] L. Rauchwerger, N. M. Amato, and D. A. Padua. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th international conference on Supercomputing*, ICS '95, pages 137–146, 1995.

[101] M. Reilly. When multicore isn't enough: Trends and the future for multi-multicore systems. In *Proceedings of the Workshop on High Performance Embedded Computing*, 2008.

[102] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[103] T. Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, Jan. 2000.

[104] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 54–67, New York, NY, USA, 1996. ACM.

[105] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 19 (6):942–991, Nov. 1997.

[106] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 72–83, New York, NY, USA, 1999. ACM.

[107] J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[108] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 35–50, New York, NY, USA, 2014. ACM.

[109] Y. Sato, Y. Inoguchi, and T. Nakamura. Whole program data dependence profiling to unveil parallel regions in the dynamic execution. In *IEEE International Symposium on Workload Characterization*, IISWC, pages 69–80, 2012.

[110] N. Scaife, S. Horiguchi, G. Michaelson, and P. Bristow. A parallel SML compiler based on algorithmic skeletons. *J. Funct. Program.*, 15(4):615–650, July 2005.

[111] R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6): 52–59, Jun 1997.

[112] A. Shafiee Sarvestani, E. Hansson, and C. Kessler. Extensible recognition of algorithmic patterns in DSP programs for automatic parallelization. *Int. J. Parallel Program.*, 41(6):806–824, Dec. 2013.

[113] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 414–425, New York, NY, USA, 1995. ACM.

[114] M. Sottile, A. Dakshinamurthy, G. Hendry, and D. Dechev. Semi-automatic extraction of software skeletons for benchmarking large-scale parallel applications. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles*

of Advanced Discrete Simulation, SIGSIM-PADS '13, pages 1–10, New York, NY, USA, 2013. ACM.

[115] J. Stokes. Into the Core: Intel's next-generation microarchitecture. Ars Technica, `http://arstechnica.com/gadgets/2006/04/core/`, Apr 2006.

[116] K. Streit, C. Hammacher, A. Zeller, and S. Hack. Sambamba: A runtime system for online adaptive parallelization. In *Compiler Construction*, pages 240–243. Springer, 2012.

[117] K. Streit, C. Hammacher, A. Zeller, and S. Hack. Sambamba: Runtime adaptive parallel execution. In *Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems*, ADAPT '13, pages 7:1–7:6, New York, NY, USA, 2013. ACM.

[118] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a java just-in-time compiler. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 312–323, New York, NY, USA, 2003. ACM.

[119] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[120] S. Tallam, R. Gupta, and X. Zhang. Extended whole program paths. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 17–26, 2005.

[121] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[122] N. Topham. EnCore: A low-power extensible embedded processor. `http://groups.inf.ed.ac.uk/pasta/pub/hipeac-wroclaw.pdf`, October 2009.

[123] N. Topham and D. Jones. High speed CPU simulation using JIT binary translation. In *Proceedings of the Annual Workshop on Modelling, Benchmarking and Simulation*, MOBS '07, 2007.

[124] G. Tournavitis and B. Franke. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 377–388, New York, NY, USA, 2010. ACM.

[125] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009*

ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, pages 177–187, New York, NY, USA, 2009. ACM.

[126] A. Udupa, K. Rajan, and W. Thies. ALTER: Exploiting breakable dependences for parallelization. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 480–491, New York, NY, USA, 2011. ACM.

[127] N. Vachharajani, M. Iyer, C. Ashok, M. Vachharajani, D. I. August, and D. Connors. Chip multi-processor scalability for single-threaded applications. *SIGARCH Comput. Archit. News*, 33:44–53, Nov 2005.

[128] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[129] H. Vandierendonck, S. Rul, and K. De Bosschere. The paralax infrastructure: automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 389–400, New York, NY, USA, 2010. ACM.

[130] R. Vanka and J. Tuck. Efficient and accurate data dependence profiling using software signatures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 186–195, New York, NY, USA, 2012. ACM.

[131] D. W. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 59–70, New York, NY, USA, 1991. ACM.

[132] C. Wang, Y. Wu, E. Borin, S. Hu, W. Liu, D. Sager, T.-f. Ngai, and J. Fang. Dynamic parallelization of single-threaded binary programs using speculative slicing. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 158–168, New York, NY, USA, 2009. ACM.

[133] Z. Wang, D. Powell, B. Franke, and M. O'Boyle. Exploitation of GPUs for the parallelisation of probably parallel legacy code. In *Compiler Construction*, pages 154–173. Springer, 2014.

[134] D. Wentzlaff and A. Agarwal. Constructing virtual architectures on a tiled processor. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 173–184, Washington, DC, USA, 2006. IEEE Computer Society.

[135] M. Wolfe. *Optimizing supercompilers for supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, 1982.

[136] M. Wong. C++ benchmarks in SPEC CPU2006. *ACM SIGARCH Computer Architecture News*, 35(1):77–83, 2007.

[137] J. Yang, K. Skadron, M. L. Soffa, and K. Whitehouse. Feasibility of dynamic binary parallelization. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, 2011.

[138] T. Yang and A. Gerasoulis. Pyrros: Static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th International Conference on Supercomputing*, ICS '92, pages 428–437, New York, NY, USA, 1992. ACM.

[139] E. Yardımcı and M. Franz. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 127–138, New York, NY, USA, 2006. ACM.

[140] H. Yasuura. Width and depth of combinational logic circuits. *Information Processing Letters*, 13(4):191–194, 1981.

[141] H. Yu and Z. Li. Fast loop-level data dependence profiling. In *Proceedings of the 26th ACM International conference on Supercomputing*, ICS '12, pages 37–46, New York, NY, USA, 2012. ACM.

[142] H. Yu and Z. Li. Multi-slicing: a compiler-supported parallel approach to data dependence profiling. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 23–33, New York, NY, USA, 2012. ACM.

[143] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 47–58, Washington, DC, USA, 2009. IEEE Computer Society.

[144] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 85–96, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.