



THE UNIVERSITY *of* EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Structured Parallelism Discovery with Hybrid Static-Dynamic Analysis and Evaluation Techniques

Christos Vasiladiotis



Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2022

Abstract

Parallel computer architectures have dominated the computing landscape for the past two decades; a trend that is only expected to continue and intensify, with increasing specialization and heterogeneity. This creates huge pressure across the software stack to produce programming languages, libraries, frameworks and tools which will efficiently exploit the capabilities of parallel computers, not only for new software, but also revitalizing existing sequential code. Automatic parallelization, despite decades of research, has had limited success in transforming sequential software to take advantage of efficient parallel execution. This thesis investigates three approaches that use commutativity analysis as the enabler for parallelization. This has the potential to overcome limitations of traditional techniques.

We introduce the concept of *liveness-based commutativity* for sequential loops. We examine the use of a practical analysis utilizing *liveness-based commutativity* in a symbolic execution framework. Symbolic execution represents input values as groups of constraints, consequently deriving the output as a function of the input and enabling the identification of further program properties. We employ this feature to develop an analysis and discern commutativity properties between loop iterations. We study the application of this approach on loops taken from real-world programs in the OLDEN and NAS Parallel Benchmark (NPB) suites, and identify its limitations and related overheads.

Informed by these findings, we develop Dynamic Commutativity Analysis (DCA), a new technique that leverages profiling information from program execution with specific input sets. Using profiling information, we track liveness information and detect loop commutativity by examining the code’s live-out values. We evaluate DCA against almost 1400 loops of the NPB suite, discovering 86% of them as parallelizable. Comparing our results against dependence-based methods, we match the detection efficacy of two dynamic and outperform three static approaches, respectively. Additionally, DCA is able to automatically detect parallelism in loops which iterate over Pointer-Linked Data Structures (PLDSs), taken from wide range of benchmarks used in the literature, where all other techniques we considered failed. Parallelizing the discovered loops, our methodology achieves an average speedup of $3.6\times$ across NPB (and up to $55\times$) and up to $36.9\times$ for the PLDS-based loops on a 72-core host. We also demonstrate that our methodology, despite relying on specific input values for profiling each program, is able to correctly identify parallelism that is valid for all potential input sets.

Lastly, we develop a methodology to utilize *liveness-based commutativity*, as implemented in DCA, to detect latent loop parallelism in the shape of patterns. Our approach applies a series of transformations which subsequently enable multiple applications of DCA over the generated multi-loop code section and match its loop commutativity outcomes against the expected criteria for each pattern. Applying our methodology on sets of sequential loops, we are able to identify well-known parallel patterns (i.e., maps, reduction and scans). This extends the scope of parallelism detection to loops, such as those performing scan operations, which cannot be determined as parallelizable by simply evaluating *liveness-based commutativity* conditions on their original form.

Lay Summary

The use of computers and software, the instructions that tell a computer what calculations to perform over an input set, is found everywhere nowadays. Computers are powered by billions of tiny components called transistors contained in their processing units (microprocessors). For decades, the leaps in execution speed enjoyed by software were largely due to the advances in the miniaturization transistors. However, this growth trend is ending due to physical limitations in the manufacturing process. In response, manufacturers have been creating units with multiple processors (multicore), able to perform computations independently of each other and in parallel.

However, for software to take full advantage of the available parallelism, extensive changes to its instructions are typically needed. These modification steps, termed parallelization, are challenging and error-prone when performed manually. Hence, considering also the multitude of software programs and different computer types, it is important to use tools to automate and assist in these steps. One such tool is a compiler, a program that translates software from a set of instructions written and understood by humans, to one that can be executed by a computer. This thesis focuses on developing and examining compiler techniques to assist in the parallelization of software.

Compilers analyze software based on specific models to deduce which parts of a program can be executed in parallel. In this thesis, we examine and develop techniques based on a novel commutativity model that aims to handle certain types of programs that traditionally used analyses are unable to. In our commutativity model, we rearrange the order of operations in a program section and if the outcome is the same as in the original order, then we declare it as potentially parallelizable. Initially, we implement and study a new technique based on our model that represents program inputs as sets of inputs and executes the candidate programs with them to deduce aggregated program properties. We note the challenges and limitations associated with this approach and take the lessons learned forward to develop another novel technique. This technique relies on actual program inputs for execution and analysis, thus allowing us to make very precise claims about program properties for parallelization, despite being specific to these inputs. Nonetheless, our experiments show that these results are not only profitable for parallelization, but also safe to generalize for all legal, potential program inputs. Lastly, we examine the use of our commutativity model towards identifying not only parallelism, but frequently occurring, higher-level patterns of it.

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.

(Christos Vasiladiotis)

Acknowledgements

First, I would like to thank my supervisor, Björn Franke, for his tireless support and invaluable guidance through the duration of my studies. I am especially grateful to Björn for his collaborative spirit, practical optimism and sincere advice, which, with the benefit of hindsight, I continue to appreciate more and more over the years. I also want to express my gratitude to the rest of my supervisors; Murray Cole, for his energetic involvement and diligent feedback; Roberto Lozano Castañeda, for his attentive mentorship and steady encouragement; and Michael O’Boyle, for his crucial advice and pragmatic foresight. I am indebted to Tobias J.K.E. von Koch, Stanislav Manilov and Georgios Tournavitis who laid the foundations for my research. I am also grateful to Kuba Kaszyk, for being that rare, serendipitous combination of a great colleague, collaborator and friend.

I wish to thank all my colleagues who have made the Institute for Computing Systems Architecture, the Compilers and Architecture Design group, and the Pervasive Parallelism CDT program, great places to conduct research in. I want to thank all the people whom I have shared an office with in the Informatics Forum, particularly Rui Li, for making it an interesting and fun space to study. Special thanks to all the people who have provided feedback during the writing of this thesis, especially Murray Cole, for enduring the anguish of reading multiple drafts.

I could not have done this without the support of all my friends who have tolerated my social elusiveness and unreliability, allowing me to appear and, more often, disappear as I please. Special thanks to my long-time friends from Samos and previous work lives, Yiannis, Vasilis and Tasos, for being there whenever I felt like picking up the phone; the friends I made in Edinburgh, Michael, Kate, Andreas, Andrea, Dave and Thomas, for their support and providing the necessary diversion with wee Scottish adventures; and my late friend, Roussis, for his continuous urges to embrace change.

I thank my family for their unwavering love and support.

Last but not least, I am grateful to my wife, Ellyse, for her patience, support and love through this journey.

Στον πατέρα μου.

To my father.

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. E. von Koch, Stanislav Manilov, **Christos Vasiladiotis**, Murray Cole, and Björn Franke

“Towards a Compiler Analysis for Parallel Algorithmic Skeletons”

in Proceedings of the 27th International Conference on Compiler Construction (CC 2018), Vienna, Austria, 2018

— This publication forms the basis for parts of Chapter 4.

- Stanislav Manilov, **Christos Vasiladiotis**, and Björn Franke

“Generalized Profile-Guided Iterator Recognition”

in Proceedings of the 27th International Conference on Compiler Construction (CC 2018), Vienna, Austria, 2018

— This publication forms the basis for Section 3.2.8.

- Aleksandr Maramzin, **Christos Vasiladiotis**, Roberto Castañeda Lozano, Murray Cole, and Björn Franke

““It Looks Like You’re Writing a Parallel Loop”: A Machine Learning Based Parallelization Assistant”

in Proceedings of the 6th ACM SIGPLAN International Workshop on AI-Inspired and Empirical Methods for Software Engineering on Parallel Computing Systems (AI-SEPS 2019), Athens, Greece, 2019

- Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, **Christos Vasiladiotis**, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O’Boyle, Scott Mahlke, Trevor Mudge, Ronald Dreslinski

“Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design”

in Proceedings of the 27th IEEE International Symposium on High-Performance Computer Architecture, (HPCA 2021), Virtual, Republic of Korea, 2021

- **Christos Vasiladiotis**, Roberto Castañeda Lozano, Murray Cole, and Björn Franke

“Loop Parallelization using Dynamic Commutativity Analysis”

in Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2021), Virtual, Republic of Korea, 2021

— This publication forms the basis for Chapter 5.

Table of Contents

1	Introduction	1
1.1	The Problem: Porting Software to Parallel Hardware	3
1.2	The Ideal Solution: Automatic Parallelization	5
1.3	The Challenges of Automatic Parallelization	8
1.4	Goals and Contributions	10
1.5	Thesis Overview	11
1.6	Summary	12
2	Related Work	13
2.1	Introduction	13
2.2	Dependence Analysis	13
2.2.1	Static Dependence	16
2.2.2	Dynamic Dependence	16
2.2.3	Limit Studies on Dependence	17
2.3	Commutativity Analysis	18
2.3.1	Separability-Based Commutativity	18
2.3.2	Output-Based Commutativity	21
2.4	Automatic Parallelization	23
2.4.1	Overview of Approaches	23
2.4.2	Parallelization in the Presence of Dependences	24
2.4.3	Speculative Parallelization	26
2.4.4	Analyzing Pointer-Linked Data Structures	28
2.4.5	Parallelization with User Interaction	29
2.4.6	Parallelization Frameworks	30
2.5	Symbolic Execution	33

2.6	Algorithmic Skeletons	35
2.7	Conclusion	37
3	Infrastructure and Related Tools	39
3.1	Introduction	39
3.2	Software Tools	39
3.2.1	LLVM Compiler Infrastructure	39
3.2.2	INTEL ICC Compiler	41
3.2.3	LLVM/POLLY Extension	42
3.2.4	IDIOMS Technique	42
3.2.5	DISCOPOP Technique	43
3.2.6	DEPENDENCE PROFILING Technique	44
3.2.7	Parallelization with OPENMP	44
3.2.8	Generalized Iterator Recognition Analysis	46
3.2.9	KLEE Symbolic Execution Tool	48
3.3	Benchmarks	49
3.3.1	NAS Parallel Benchmark (NPB) Suite	49
3.3.2	Applications using Pointer-Linked Data Structures	51
3.4	Hardware Setup Configurations	53
3.5	Summary	54
4	Liveness-based Commutativity Analysis using Symbolic Execution	55
4.1	Introduction	56
4.2	Liveness-Based Commutativity	57
4.3	Liveness-Based Loop Commutativity	59
4.4	Motivating Examples	61
4.5	Operation of Symbolic Execution	63
4.6	Symbolic Crosschecking of Commutativity	66
4.7	Symbolic Crosschecking Technique for Liveness-based Commutativity	67
4.7.1	Approach	67
4.7.2	Dealing with Limited Code Coverage	69
4.7.3	Symbolic Sizes for Memory Allocations	71
4.7.4	Handling Floating-point Operations	72
4.8	Case Studies	73
4.8.1	Pointer-based Loops	73
4.8.2	Array-based Loops	79

4.9	Summary and Conclusions	82
5	Loop Parallelization using Dynamic Commutativity Analysis	83
5.1	Introduction	83
5.2	Motivating Examples	85
5.3	Revisiting Liveness-based Loop Commutativity Analysis	87
5.3.1	Applicability of Parallelization	88
5.4	Dynamic Commutativity Analysis	88
5.4.1	Static Stage	90
5.4.2	Dynamic Stage	93
5.4.3	Parallelization	94
5.4.4	Safety	94
5.4.5	Challenges and Limitations	95
5.5	Empirical Evaluation	95
5.5.1	Experimental Setup	96
5.5.2	Performance against Dynamic Techniques	98
5.5.3	Performance against Static Techniques	101
5.5.4	Aspects of Detection Profitability and Precision	104
5.5.5	Scope of Parallelization Beyond Loops	105
5.6	Summary and Conclusions	106
6	Detecting Data-Parallel Patterns with Liveness-based Commutativity	109
6.1	Motivation	110
6.2	Methodology	113
6.2.1	Overview	113
6.2.2	Syntactic Transformations	118
6.3	Case Studies	121
6.3.1	Scan Pattern	121
6.3.2	Reduction Pattern	123
6.3.3	Map Pattern	126
6.3.4	Discriminating between Patterns	127
6.4	Summary and Conclusions	130
7	Conclusions	133
7.1	Introduction	133
7.2	Contributions	134

7.3	Critical Review and Future Directions	136
7.3.1	Liveness-Based Commutativity Analyses	136
7.3.2	Pattern Detection with Liveness-based Commutativity	139
7.3.3	User Interface Improvements	141
	Bibliography	143

Introduction

Today, the majority of computers are parallel and heterogeneous. Computers comprise multiple Central Processing Units (CPUs), Graphics Processing Units (GPUs) and other specialized processing components. The trend towards increasing use of parallelism is not new, but computer architects were, for a time, able to hide it from the software stack [187]. This allowed software to readily reap the benefits of new generations of processors while retaining, for the most part, the same computation model with minimal changes [12]. However, since the mid-2000s this hardware and software contract, that largely insulated one from changes in the other, has broken down [240]. Architects resorted to replacing single-core processors with many processors (i.e., multicore) in a chip in order to sustain the ever-increasing performance, thus declaring the future for the industry was in parallel computing. Therefore, putting enormous pressure on software systems, existing and new, since programs need to be explicitly modified (i.e., parallelized) to fully take advantage of this newly exposed parallelism.

For several decades, from the mid-1960s till the early 2000s, two main factors in hardware design allowed processor manufacturers to achieve exponential performance growth [187]. The first reason is concerned with miniaturization, popularly termed as “Moore’s Law” [174] (Figure 1.1) and is the roughly biannual trend of doubling the number of transistors in Integrated Circuits (ICs) of the same area [165]. This has been coupled with advancements in circuit design (e.g., increased clock rates) and microarchitecture to provide even higher performance than that predicted by Moore’s Law. These microarchitectural techniques aim at discovering and exploiting Instruction-Level Parallelism (ILP) (e.g., pipelining, prefetching, Out-of-Order (OoO) execution, etc.) unbeknownst to the software running on them and its developers. This extraction of latent parallelism from the instruction stream of a program allowed software to

remain practically unchanged while gaining in performance at similar rates.

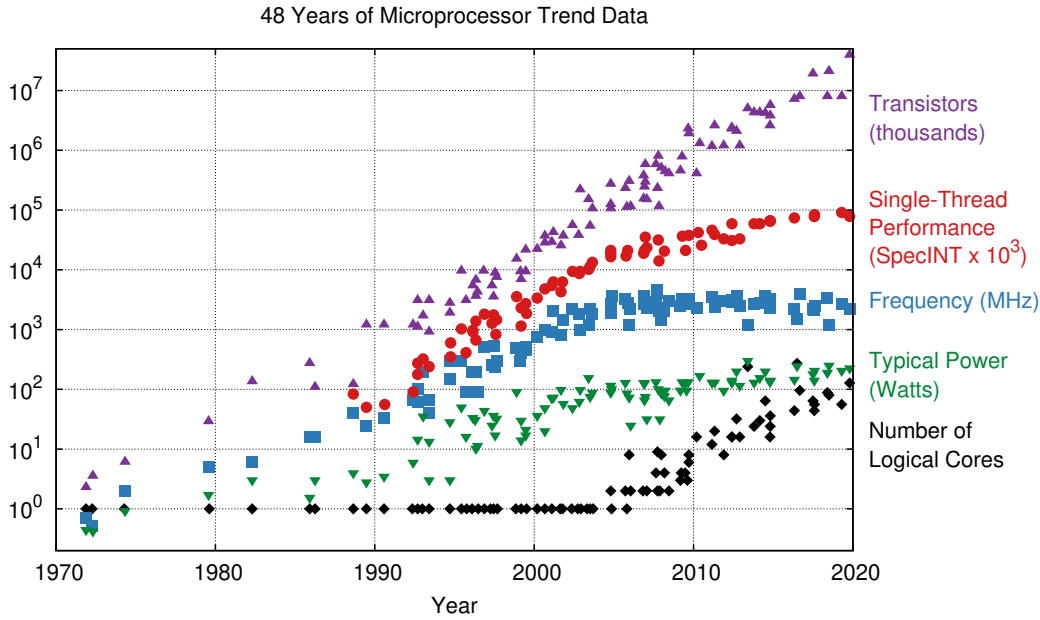


Figure 1.1: 48 years of microprocessor trend data. Transistor counts grow exponentially following Moore’s Law, while single-thread performance, frequency and power consumption largely flatten after 2005. The number of cores continue to increase since it is one of the main ways of delivering more transistors. The slight increase in single-thread performance is due to gains from advancements in compilers. (Illustration source data [215]).

However, chip manufacturing is rapidly reaching the limits of miniaturization and exhausting the effect of these microarchitectural advancements. The single biggest factor has been the inability of chip designers to deal with power delivery and heat dissipation, as each new chip generation required exponentially more power to support the aforementioned features. This “power wall” constraint has led the industry to seek new means to deliver regular performance improvements and initially paved the way for the current wide adoption of Chip Multi-Processors (CMPs) and heterogeneous computer systems [1, 35, 113, 64, 187, 88]. The ensuing technology switch has had multiple ramifications on software and its development, which are still percolating through the software stack, and are expected to continue and intensify [226, 225]. The most disruptive of these consequences for software are [225]:

- i) Devising new programming models to facilitate and advance the use of the new and ever-evolving parallel architecture, using abstractions that hide their low-level implementation details.

- ii) Developing new algorithmic approaches in a co-design process, taking into account the potential performance and impact of various choices.
- iii) Lessen the burden of transitioning, refactoring and redesigning software for the new architectures.

Thus, due to the lack of an immediate solution that would enable the necessary year-on-year technological scaling that we have grown accustomed to, the path forward leads to architectural specialization in specific application domains and extreme heterogeneity [226, 243, 186, 230]. This is already a reality in a wide range of hardware, from specialized Machine Learning (ML) accelerators deployed in data centers, like Google's Tensor Processing Unit (TPU) [132], to mobile devices like Apple's 11th generation iPhone System-on-Chip (SoC) which contains over 40 discrete accelerators [228]. The unifying trend of all these diverse processing units is that they are all predominantly parallel systems.

1.1 The Problem: Porting Software to Parallel Hardware

The tectonic shift in processor design has caused an equally dramatic reactionary jolt in the software ecosystem, after a relatively long period where software architectures remained largely unchanged and oblivious to the changes of the underlying hardware. One of the primary goals of software and associated tools is to abstract and ease the use of the ongoing feature changes of parallel computing units, that are only expected to increase, by developers. In response, academia and industry have feverishly focused on exploring and producing techniques, tools and environments that will alleviate the task of developing software on new parallel hardware [171, 63, 173, 103, 99, 225, 243, 62, 9]. These efforts have already produced a plethora of flavors in parallel algorithms, programming models, languages, compilers and runtime systems, and have provided a few promising, albeit very narrow, glimpses of a way forward. Thus far, a panacea for all problems and challenges of the evolving parallel landscape has not been found.

A major hurdle is the amount of legacy software that is still operating today in a wide range of fields [213, 284, 245, 170]. For example, a Reuters report states that 220 billion lines of COBOL code are active today which power 80% of all in-person transactions with banking systems [55]. For instance, in the spring of 2020, the US

state of New Jersey made a public appeal for volunteer programmers who know COBOL to help in the maintenance of their 40-year-old systems, a situation that is also shared among other US states [67]. The definition of what constitutes legacy software varies widely, from reasons relative to its development process (i.e., lack of unit tests), to the phase of its life cycle (i.e., “once delivered, it’s legacy”). In the broadest sense, legacy software describes a system that is *still* in use, but some aspects of which are now outdated. This “bit rot” [29] can occur due to several factors, such as: i) lack of maintenance, ii) insufficient testing or hard to test codebases, iii) antiquated tools and processes, iv) lack of resources (human or otherwise), v) lack of communication and documentation. It is noteworthy that legacy software does not strictly have to do with the underlying codebase alone, hence the associated high costs with the upkeep and migration of it.

In relation to the continuously emerging parallel hardware technologies, legacy software poses serious migration challenges to them in order to keep gaining in performance. First, the majority of legacy software has been written with no consideration or notion of parallel execution. Most programming languages, with various degrees of abstraction, have closely followed the traditional von Neumann computational model. In this model, a computer consists of main processing unit, executing instructions sequentially while moving data from and to a main memory as needed. For a programmer, this perspective has been maintained for decades, and combined with the speedups gained by moving to each successive processor generation, it has amplified and sustained the resistance to modernize and adopt newer paradigms.

The strong ties with a sequential execution model exacerbate the challenges of revamping existing software in order to accommodate the trend of Domain-Specific Architectures (DSAs). The use of specialized processors is not new and it was more common place in the early days of computing, such as in supercomputers like the Cray architecture [216]. However, it was phased out and became progressively financially nonviable as a result of the advances of general purpose processors. Since advances in general single-core processors have nowadays slowed, this trend has started to reverse itself, initially by re-purposing GPUs and continuing with more specialized accelerators. Higher efficiency is achieved by tailoring these architectures to the characteristics of the domain, utilizing specialized operations, parallelism, tailored memory hierarchies and targeted overhead reductions [99, 62].

In terms of software, there is a price to be paid when adopting such an approach since most existing applications necessitate modifications to attain high speedups on DSAs.

The need to rewrite and refactor programs puts extra pressure on programmers and tools in order to revitalize and keep them up to date with new parallel architectures [69]. This applies not only to legacy sequential code, but also legacy *parallel* code, which exhibits tight coupling as a result of development by a combination of hand-tuning, exploiting specific hardware features and dated parallel models (e.g., Message Passing Interface (MPI)) [164]. Thus, simply porting already parallel code might result in poor performance and modifications might be required to properly exploit new parallel systems, even as extensive as revisiting the underlying algorithm used by the application. In addition to the low-level interaction with parallel hardware, parallel programming is hard because of:

- i) the required work partitioning,
- ii) resource access synchronization, and
- iii) resource partitioning and replication.

In summary, the move of computer architecture towards processors that utilize parallelism as the main means to increase performance creates disruptions and increasingly higher tensions on the whole software stack, up to the user applications level. The amount of legacy software in combination with the difficulties and challenges of porting it on new architectures inhibit the full utilization of the parallel features on these devices.

1.2 The Ideal Solution: Automatic Parallelization

The field of compilers, since the 1950s, has been concerned with the translation of software to machine code and systematizing an increasingly wide range of tasks in support of this [103]. Since then a vast body of knowledge has been produced in more focused areas such as program analysis and transformation, code generation and runtime systems, interconnecting with other disciplines like computer architecture, software engineering and security. Hence, the task of converting sequential software to its equivalent parallel version falls naturally under the domain of compilers and it is typically termed as automatic parallelization [173]. However, in practice, very few compilation systems have managed to employ a fully automatic parallelization procedure, with most systems using feedback to complement various stages of their operation [69, 193, 9]. Those systems, that have enjoyed some, success are often

limited to a narrow application domain [7]. We discuss the challenges of automatic parallelization in the next section. This section provides a brief summary of this research area describing its various characteristics at a high level. We present a more detailed view of related work in Chapter 2.

The aim of automatic parallelization is to rid the programmer of the burden of understanding and extracting parallelism from an application. Similarly to a manual, human-led analysis, when implemented in a compiler system, automatic parallelization occurs in two stages:

- i) a detection phase, which is responsible for identifying any available parallelism, and
- ii) an extraction and mapping phase of this parallelism on the target architecture.

During the detection phase, code is analyzed in order to identify candidate regions that could be executed in parallel. The code can be denoted as binary machine instructions, a high-level programming language or anything in between. During this phase, the behavior and properties of the code are analyzed and represented using a model (e.g., unimodular analysis [272, 22]), which is used to discover parallelization opportunities. The size of the targeted code sections can range from a single instruction, a loop nest, a series of basic blocks or a function, and it determines the granularity of the extracted parallel tasks in the next phase. Typically, task granularity is indicative and proportional to the amount of work which is performed. However, there are other factors that affect task behavior during execution such as control and data flow, access to common resources and communication overheads.

During the extraction and mapping phase, the identified code regions are subject to possible transformations as described by the used representation model (e.g., unimodular [272, 22] or polyhedral [82]). The primary goal of this phase is to generate parallel code that is semantically equivalent to the original sequential form. The safety and legality of the transformations can be [69, 280]:

- i) guaranteed by the model itself at varying degrees,
- ii) verified using a supplementary method (e.g., revert to prior state or version if a violation is detected),
- iii) guaranteed by the user (e.g., with source-level annotations), or
- iv) a combination of all the previous.

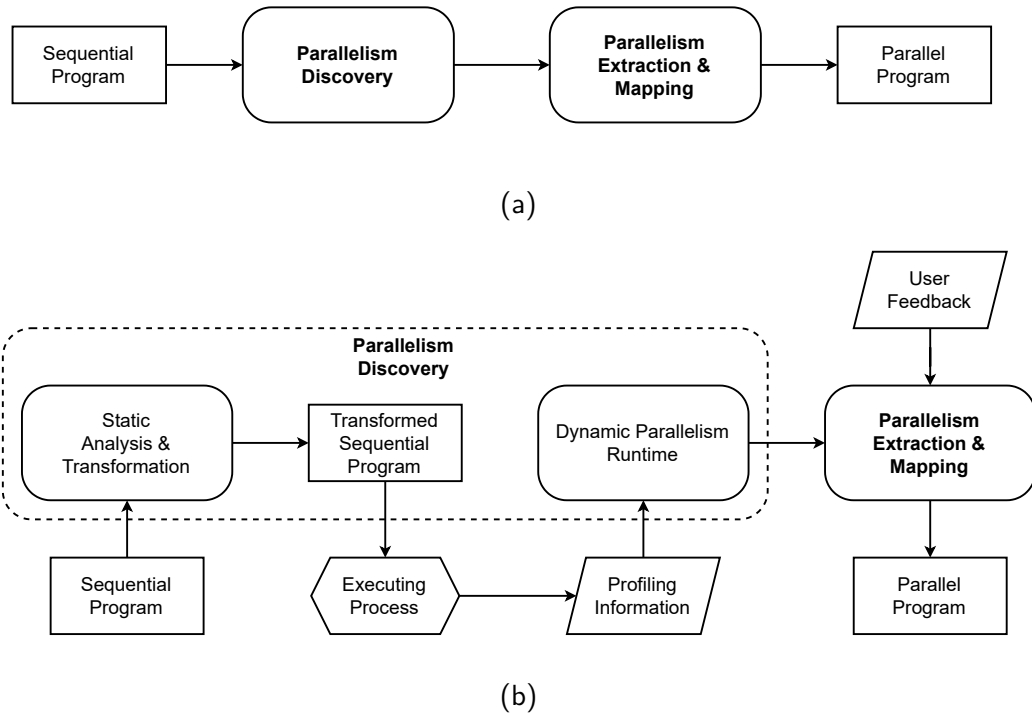


Figure 1.2: A generic bird's eye view of the stages of a parallelizing compilation system (top) and a high-level architectural outline of an actual system described in Chapter 5 (bottom).

Moreover, the mapping part of this phase has to ensure that the underlying parallel architecture features are exploited profitably. This relies on many factors, including the amount of available parallelism and the way it is exploited, other compiler optimizations, runtime support, data layout, workload balancing and so on. The mapping process has both a spatial and temporal component, which combined with the aforementioned factors make it a challenging task to solve. For example, the problem of deriving an optimal schedule for parallel jobs on multicore computer has been shown to be NP-hard [265, 126]. For a long time, research in this area focused on finding analytical cost models that incorporate knowledge from both hardware (e.g., the size of L2 cache) and software (e.g., analyzing loop array accesses) [7]. These methods have also evolved to incorporate heuristics and information derived during the parallel execution of a program, and more recently, utilize profiling information and machine learning techniques to select a profitable mapping [248].

Automatic parallelization techniques are not necessarily limited to the compiler, and they can be implemented at various stages of the software lifetime and layers of the hardware/software stack. It can occur statically, prior to a program's execution, using its

source code (or any textual representation) [236], or dynamically during execution [280, 205, 248]. This distinction is not absolute and precise, since parallelization methods, in order to bridge the semantic information gap [103], can use information from previous program application runs, such as in a profile-guided approach or the training phase of a model based on machine learning [248]. User knowledge can also be exploited by providing source-level annotations prior to any parallelization attempt or at certain stages of the process by confirming or rejecting the proposed modifications. Lastly, in cases where knowledge relies on the input and the code exhibits a highly dynamic-dependent behavior, speculative execution can push the decision of parallelization to the moment when it might be needed during execution, while preserving correct behavior through checks and the ability to revert to a prior correct program state. An example of a system that combines both profiling information and user input is the one presented in [248] and our study in Chapter 5. A typical workflow of such a system is depicted in Figure 1.2; the top figure shows the high-level view of the main stages consisting of i) finding latent parallelism in a sequential program, and ii) exploiting it by determining how it should be executed. In Figure 1.2(b), the workflow of an actual parallelizing compiler used in Chapter 5 is presented, where that aforementioned stages are shown with more detail as they relate to the specific technique. Parallelizing techniques are generally classified based on their time of operation during a program's lifetime. For example, a completely static approach would use no information from prior program executions, whereas a fully dynamic one would perform all stages during execution.

1.3 The Challenges of Automatic Parallelization

As we briefly mentioned in the previous section, it might seem that automatic parallelization has been able to successfully tackle the difficult task of porting sequential software to parallel hardware. However, compilers for automatic parallelization have largely failed to deliver on their promise to seamlessly transition sequential legacy software into the multicore era [193, 9]. Despite intensive research, the problem of discovering and exploiting parallelism hidden in sequential code is far from solved [10], except for limited success in a few narrow domains (e.g., regular array-based computations) [7].

The success of a compiler technique relies on meeting several criteria [9], such as:

- i) Effectiveness: select and exploit the best parallel version out of a number of possible choices, automating decisions and minimizing reliance on the user for

these.

- ii) **Stability:** resilience to small program changes and perceiving the compiler as an opaque component.
- iii) **Portability:** operate on a range of software inputs and hardware, avoiding embedded assumptions and enduring the appearance of new parallel architectures with small changes.
- iv) **Scalability:** the parallel code should perform well on machines with a wide range of parallel capabilities.
- v) **Simplicity:** simple techniques benefit compilation performance, ease of development and maintenance of the compiler itself.

With these criteria in mind, we can summarize the main sources for the lack of widespread success of automatic parallelization to the following reasons:

- i) *Semantic information gap:* compilers have had success in lowering abstractions from higher level languages, whereas they have been mainly ineffective moving in the other direction. A lot of the previous work in automatic parallelization relies on static program information extracted at compile time [173, 7]. The analyses and transformations built on this information are undecidable [153], which in turn leads to models that have to use conservative approximations, thus missing parallelization opportunities. Using information from program execution runs can shorten the gap, increase their effectiveness and simplify the implementation of such models.
- ii) *Parallelism structure and granularity:* typically, the majority of parallelization efforts has been confined to regular loop nests which exhibit limited control flow and simple iteration patterns over dense arrays, with the mapping phase mostly focusing on discovering profitable memory hierarchy access configurations.
- iii) *Disjoint compiler and developer practices for parallelization:* the ‘modus operandi’ of programmers trying to discern parallelizable sections in a sequential application, depends upon understanding of the algorithms, the data structures and how their use can be partitioned and shared across processing elements. On the other hand, most program approximations and models used in parallelizing compilers aim at preserving a specific aspect of the code (e.g., the order of the read and

write operations that occur on the same memory locations) and get entangled in implementation details that are either inessential parts of the algorithm or do not affect the final result of the program.

1.4 Goals and Contributions

The objective of this thesis is to overcome the aforementioned challenges and address the reasons for some of the limitations identified in Section 1.3. We make a series of closely related contributions by examining the use and efficacy of an analysis model that is based on *commutativity* of program operations for automatic parallelization.

Intuitively, two code regions are commutative when permuting their order of execution does not change their outcome. Various commutativity analyses aim to discern the conditions under which the commutativity of code regions can be identified and utilized for parallelization. We present our novel concept of *liveness-based commutativity* in Chapter 4. We discuss our previous notions of commutativity in Chapter 2.

The goal of our study is to examine the potency of this model, combining static and dynamic program information at varying degrees. We begin with a brief description of our model and investigate its use via approximate program behavior using symbolic values. This informs our subsequent study and development of a novel analysis for discovering parallelism that incorporates profiling information from program executions using concrete input values. We apply this to fully fledged applications from the High Performance Computing (HPC) and scientific domains which exhibit complex control flow, aliasing and implement irregular algorithms over complex dynamic data structures. Finally, we extend the application of our model to encompass the detection high-level parallel algorithmic patterns.

The main original contributions of this thesis are as follows:

- i) We investigate the use of a novel *liveness-based* commutativity analysis when implemented in a symbolic execution context and provide an identification of the overheads and limitations associated with it (Chapter 4).
- ii) We introduce DCA, a technique utilizing liveness, for testing the commutativity and hence potential parallelizability of arbitrarily complex loops, the program regions with the highest parallelization potential, including traversals of Pointer-Linked Data Structures (Chapter 5).

- iii) We evaluate the efficacy of DCA in detecting parallelizable loops from a wide range of benchmarks, outperforming a combination of state-of-the-art dependence-based approaches, demonstrating that it is able to discover loops with significant parallelization profitability and high precision (Chapter 5).
- iv) We study the application of our commutativity analysis in the identification of parallel patterns in sequential code, explore its limitations and suggest potential solutions to overcome where applicable (Chapter 6).

1.5 Thesis Overview

This thesis is organized as follows:

Chapter 2 presents related work. We discuss prior work in the field of automatic parallelization, spanning static and dynamic techniques, along with brief descriptions of the main enabling models, dependence analysis and the polyhedral framework. We also provide a brief overview of prior commutativity analyses and symbolic execution techniques. The chapter ends with a presentation of parallel algorithmic skeletons and patterns as used in parallelization discovery.

Chapter 3 gives a brief account of the infrastructure used in this work. We describe the frameworks (e.g., LLVM, POLLY, etc.), toolchains and analyses utilized, along with various benchmarks, inputs and setup configurations.

Chapter 4 investigates the use of a novel *commutativity* analysis for parallelization which uses the *liveness* results of a code section when performed in a symbolic execution context. We develop an approach for applying this analysis using symbolic execution and use it to evaluate and uncover the challenges and limitations of such a system on a set of real-world codes. Parts of this chapter are based on work published in [266].

Chapter 5 examines the use of profiling information along with our notion of *liveness-based commutativity*. We develop a novel hybrid analysis, DCA, that combines both static and dynamic information for the discovery of profitable parallelism in sequential legacy code. DCA focuses on loops as the program regions that typically capture most of the program execution time. It has been evaluated on over 1200 loops across 10 HPC benchmarks and a wide range of irregular pointer-based loops. DCA handles, in *uniform* manner, both loops that are in the scope of traditional dependence approaches (such as affine loops) and loops that are beyond the capabilities of traditional methods, such as PLDS traversals, which are usually dealt with *ad hoc* or ignored altogether.

We also explore aspects that impact profile-driven parallelizing techniques such as the profitability and safety of parallelization. This chapter is based on work published in [263].

Chapter 6 revisits our *liveness-based commutativity* analysis and investigates its application to the challenge of finding instances of potential parallel patterns. We demonstrate how this analysis could be performed to identify various parallel skeletons, classifying loops that have already been identified by DCA, establish its limitations and explore ways to overcome them by applying a series of transformations.

Chapter 7 summarizes the work presented and its main contributions, providing a critical analysis and suggesting directions for potential future work.

1.6 Summary

This chapter has provided an overview of the thesis. First, we gave a short introduction to the challenges in the modern multicore era of computing that place automatic parallelization at the forefront of our attempts to efficiently take advantage of the growing number of parallel architectures. Then, we moved on to highlight the main challenges and reasons that have led automatic parallelization to fall short of the initial expectations. Finally, we enumerated the goals and contributions of this work, aiming to explore the effectiveness of identifying commutativity of operations in various execution contexts and over different parallel code patterns. The following chapter presents prior research in areas relevant to the work presented in this thesis.

Related Work

2.1 Introduction

This chapter presents a critical review of prior research which is most relevant to the work in this thesis. Our aim is to compare, juxtapose and motivate the approaches we present in this thesis with existing work in automatic parallelization. Research in the field dates back to the 1960s [27], hence there is an extensive body of literature. Accordingly, we restrict our discussion by presenting the main enabling technologies and highlighting the main trends in the area.

We start in Sections 2.2 and 2.3 by discussing the main lines of research in dependence and commutativity analysis, which most frequently constitute the basis for the detection and exploitation of parallelism. Then, in Section 2.4 we review automatic parallelization with an emphasis on hybrid static and dynamic systems. These relate more closely to our work presented in Chapters 4 and 5. Section 2.5 offers a brief discussion on symbolic execution that is relevant to our hybrid approach in Chapter 4. We conclude with a discussion on parallel algorithmic skeletons related to the study in Chapter 6.

2.2 Dependence Analysis

Traditionally, the vast majority of parallelizing techniques and compilers use dependence analysis in order to build a model for the relationships between code sections in a program [173]. In the most general terms, dependence analysis is concerned with the exploration and description of the program in terms of its memory accesses (i.e., locations that it reads from or writes to) during execution. This broad definition can be

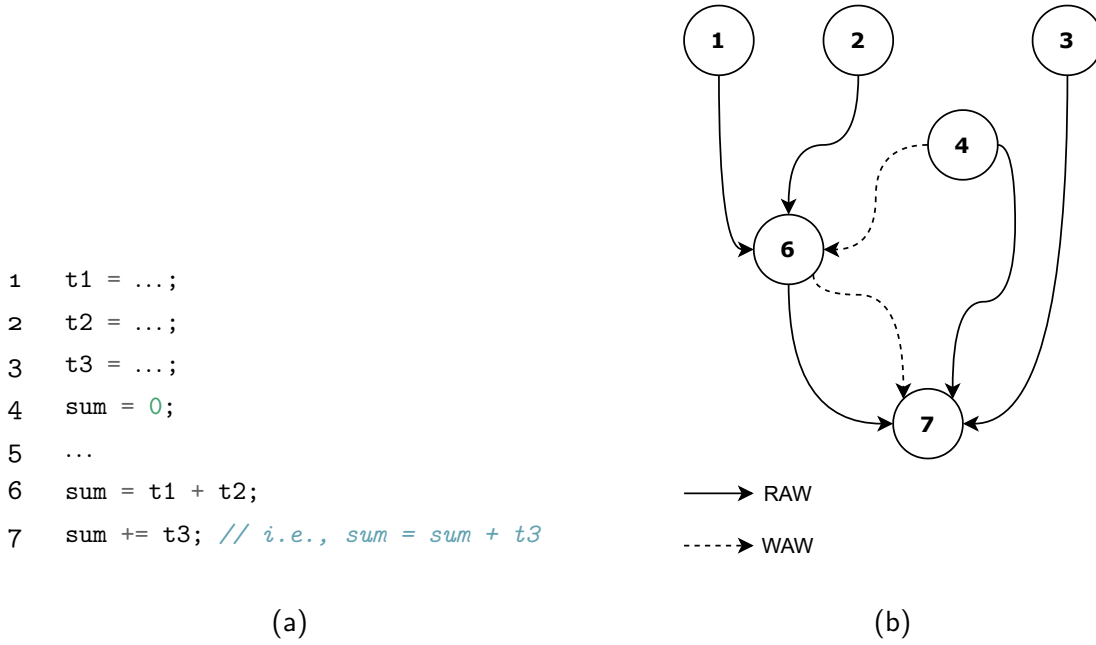


Figure 2.1: A simple, straight-line (i.e., no branching) program (left) and its corresponding Data Dependence Graph (DDG) (right). Each node's number corresponds to the statement at that source line number.

easily conflated with alias analysis, which is often an enabling technique for dependence analysis. Alias analysis attempts to determine which program constructs reference the same memory objects during execution. Dependence analysis goes beyond this goal, trying to discern execution-order constraints between them using more information, including, but not limited to, loop nesting levels, relations between loop iterators, relations between indexed array accesses, direction of data flow and awareness of control flow. Regarding parallelization, the goal is to identify the independence of statements in order to be able to reorder them and execute them in parallel (see Section 2.4.6.1).

The basic definition of dependence states that [7]:

Definition 2.2.1 (Data Dependence). There is a data dependence from statement S_1 to statement S_2 (i.e., statement S_2 *depends on* statement S_1) iff:

1. both statements access the same memory location and at least one of the stores into it, and
2. there is a feasible runtime execution path from S_1 to S_2 .

Definition 2.2.1 allows classification of dependences based on various criteria,

assignment of properties to them and enables their graph-based representation where each node is a program statement and an edge between them denotes a dependence.

The load-store classification takes into account the memory operations of the dependence statements [7]:

- i) *True, Flow or Read-After-Write (RAW)* dependence: The first statement stores into a memory location that is later read by the second statement.
- ii) *Anti- or Write-After-Read (WAR)* dependence: The first statement reads from a memory location into which the second statement later stores. Essentially, this type of dependence is the reverse of a *RAW* dependence and prevents a program transformation to introduce a new *RAW* dependence when such a relation did not exist in the original program.
- iii) *Output or Write-After-Write (WAW)* dependence: Both statements write into the same memory location.

Dependence is meant to be used as a tool for determining the safety of program transformations. Safety, in this context, means that the original and transformed programs are equivalent or, in other words, the two versions of the program do the same thing. Thus, for dependence analysis, any reordering transformation that preserves every dependence relation in a program, then preserves the meaning of that program [7].

Consider the small program in Figure 2.1(a) and its corresponding dependence graph in Figure 2.1(b). According to dependence, executing the first four assignment statements can occur in any order prior to line 6. Moreover, the assignment of `t3` can be executed anywhere prior to line 7. Note that we can remove *WAW* dependences by variable renaming [108]. On the other hand, the *RAW* dependence between lines 6 and 7 persists and prevents any reordering between these two statements. However, the view of this program is still more restrictive than is necessary when the actual contents of the `sum` variable are considered. Even if we reorder the execution of the last two lines of this program, the result (i.e., the value of `sum`) will be the same.

Therefore, while dependence analysis presents a more relaxed partial order of a program's statements, enabling specific statement rearrangements, there are cases where this view is overly limiting. We present such cases in the motivating examples of Chapters 4 and 5, arguing that dependence analysis is inherently unable to capture certain possible reorderings in the context of loop constructs (e.g., reduction operations).

2.2.1 Static Dependence

Similarly to automatic parallelization techniques described later in this chapter, dependence analyses schemes can also be categorized as either static or dynamic. Early research from the 1970s on dependence analysis focused on the detection of the absence of data dependences between program statements and the formalization of these tests (e.g., the GCD test [21]) solely at compile-time. Into the 1990s, researchers have continued to evolve and refine these techniques, incorporating ideas from other domains such as integer programming (e.g., the OMEGA test [198]), aiming at increasing detection precision and speed. Moreover, during the same period, work has been conducted to expand the use of dependence in optimizations [249, 148, 4, 276], to organize the dependence information in graphs and combine it with control flow [6, 86]. This has enabled further transformations with regard to the memory hierarchy, especially for loop iteration space manipulation, such as tiling and loop interchange [7]. The wider adoption of programming languages such as C and its derivatives, has introduced aliasing challenges in disambiguating memory accesses using pointers and dynamic memory allocation, which have percolated to dependence analysis [112], resulting in recent work on improving alias analysis [101, 238].

2.2.2 Dynamic Dependence

Later studies make use of knowledge extracted from a program's behavior during execution to determine exactly which instructions are dependent on each other for specific input sets in dynamic dependence analyses [155, 267, 14, 81, 285, 160, 248, 144, 143, 127, 262, 219, 282, 283, 277, 141]. This is because statically proving the absence of dependences is generally undecidable [209], similarly to other challenges in static analyses [153, 118, 51, 163]. In the rest of this section we discuss work that is most pertinent to dynamic dependence analysis and its limits. Section 2.4 reviews the main research directions in automatic parallelization that are enabled by dynamic data dependence.

One of the first techniques to use runtime information in supplementing dependence was [155], which tracked the frequency and location of loop-carried data dependences. From early on, it was obvious that the amount of data produced and the execution overheads of profiling were huge. [155] attempted to alleviate both these issues by tracking a subset of program events and using this partial profile when needed to generate a full profile, by executing again only the portion of the program associated

with the subset trace [154].

ALCHEMIST [285] is a profiler able to analyze binaries and determine the distances of loop intra-iteration, inter-iteration and inter-invocation dependences. SHADOW PROFILING [175] is a technique that invokes a shadow process periodically in combination with Copy-on-Write (CoW) to create a detailed sampled profile of the target application. SUPERPIN [268] uses a similar approach based on forking separate non-overlapping program regions used as instrumentation targets during program execution. SD3 [144] implements a scalable approach that compresses the trace representation memory accesses that exhibit stride patterns and calculates dependence relations directly in this compressed format. More recently, [158] uses signatures, a concept borrowed from transactional memory systems, to record memory accesses and treats dependence similarly to conflict detection in such systems. With the advent of multicores, some of these techniques also apply parallelization to the profiling task itself to speed up the whole process [175, 268, 144, 158].

2.2.3 Limit Studies on Dependence

Studies that focus and explore the effects and efficacy of dependence analysis are presented in [76, 179, 180]. These are useful since they allow us to understand the limitations of dependence-based approaches, and further motivate and support our work in Chapters 4 and 5.

[76] develops a profiling-based analysis that captures data and control dependences using program executions from 100 randomly chosen input data sets. The goal of the study is to discover the effect that dynamically collected dependence information has when compared against static approaches with respect to the ability to exploit loop-level parallelism. In essence, it seeks to understand the patterns and impact of *may*-dependences (i.e., dependences that may or may not manifest themselves during execution) in parallelization. One of the main findings of this variability analysis, most relevant to this thesis, is that although a single loop-carried dependence is sufficient to prevent parallelization, it appears that these parallelization-inhibiting dependences are *stable* across executions with different inputs. Moreover, it confirmed that profile-based approaches are truly more effective than static alternatives in parallelism detection, although safety concerns must be considered.

A further interesting result regarding the limits of dependence analysis is presented in [179, 180]. Here, starting with a state-of-the-art dependence analysis, the generated

DDG is manipulated to generate an oracle DDG. More specifically, using knowledge from profile-guided runs of the dependence analysis, *may*-dependences that never manifest during execution, are removed, thus producing the oracle DDG. Subsequently, the oracle DDG is used for parallelization, thus determining an upper-bound on the potential speedups for purely static parallelization. When compared with the parallelization using only compile-time information, it was demonstrated that no additional speedups were obtained, despite the removal of dependence edges. Thus, dependence analysis, even when informed with perfect profiling information, is inherently unable to identify any further latent parallelism.

2.3 Commutativity Analysis

In recent years, research has demonstrated that commutativity is a key enabler of efficient concurrent execution in diverse contexts, ranging from transactional memory to software scalability [54]. In this work, we are interested in the use of commutativity as the basis of compiler analyses for program parallelization. A number of different notions of commutativity for parallelization can be found in the literature [211, 3, 266]. [27] was one of the earliest works that examined the effect of the commutativity of operations for parallelization.

2.3.1 Separability-Based Commutativity

A pioneering notion of commutativity and an associated analysis have been first introduced in [211]. The underlying idea of separability-based commutativity is simple: it views computation as composed of separable operations on objects, where each operation has a receiver object and parameters that are passed by value to the operation. If all operations required to perform a given computation commute, the compiler can generate parallel code. Operations commute when they generate the same final result regardless of the order in which they execute.

Commutativity analysis focuses on separable operations, where operations can be decomposed into an object section and an invocation section. The object section performs all accesses to the receiver. The invocation section invokes other operations and does not access the receiver. The motivation for separability is that the analysis requires that each operation’s accesses to the receiver execute atomically with respect to the operations that it invokes. Separability ensures that the actual computation obeys

1	<code>class node {</code>	1	<code>class node {</code>
2	<code>bool marked;</code>	2	<code>bool marked;</code>
3	<code>int value, sum;</code>	3	<code>int value, sum;</code>
4	<code>node *left, *right;</code>	4	<code>node *left, *right;</code>
5		5	
6		6	<code>struct statistics {</code>
7		7	<code>int visited;</code>
8		8	<code>} stats;</code>
9		9	
10	<code>void visit(int p) {</code>	10	<code>int visit(int p) {</code>
11	<code>// object section</code>	11	<code>// interleaved sections</code>
12	<code>sum += p;</code>	12	
13		13	
14		14	
15	<code>if(!marked) {</code>	15	<code>if(!marked) {</code>
16	<code>// invocation section</code>	16	<code>stats.visited++;</code>
17	<code>if(left) left->visit(value);</code>	17	<code>if(left) left->visit(value);</code>
18	<code>if(right) right->visit(value);</code>	18	<code>if(right) right->visit(value);</code>
19	<code>}</code>	19	<code>}</code>
20		20	<code>sum += p;</code>
21		21	<code>return sum;</code>
22	<code>}</code>	22	<code>}</code>
23	<code>};</code>	23	<code>};</code>

(a)
(b)

Figure 2.2: Separability-based commutativity [211] examples for a class definition representing a binary tree data structure that conforms to its required computation model (left) and one that does not (right). The lines that violate the requirements of this model are highlighted (right).

this constraint.

To verify that method invocations commute, the compiler represents and reasons about the new values of the receiver’s instance variables and the multiset of operations directly invoked when the two methods execute. The compiler represents the new values and multisets of invoked methods using symbolic expressions.

Unfortunately, this approach quickly reaches its *limits* for real-world applications. In short, applications are required to be coded as “clean” object-based programs. Yet, sev-

```

1  node *n;
2  int p1, p2;
3  ...
4  // commutativity under different execution orders
5  // order 1                // order 2
6  n->visit(p1);              n->visit(p2);
7  n->visit(p2);              n->visit(p1);

```

Figure 2.3: The invocations of the `visit` method for the code in Figure 2.2(a) are commutative, according to separability-based commutativity [211], since different execution orders produce the same state.

eral further restrictions apply which exclude: i) virtual methods, ii) operator or method overloading, iii) multiple inheritance and templates, iv) typedef, union, struct and enum types, v) global variables other than class types, vi) pointers to members or static members, vii) casts between base types such as `int`, `float` and `double` that are used to represent numbers, viii) default arguments or methods with variable numbers of arguments, and ix) accesses to instance variables of nested objects. It is not entirely clear which of these restrictions are fundamental to the approach and which stem from the prototype implementation. Separability-based commutativity has been evaluated on highly sanitized, sequential implementations derived from parallel benchmarks.

For instance, Figure 2.2(a) defines a simple class representing the nodes in a binary tree data structure which conforms to the model of computation required by separability-based commutativity. One of the strictest requirements is to maintain the data accesses and other method invocations in each method separate. This corresponds to the *object section* and *invocation section* starting at lines 11 and 16, respectively. Moreover, method calls in the *invocation section* should not alter the state of the current object (i.e., calling a setter method). The order of the `visit` method invocations does not affect the resulting internal state as shown in Figure 2.3. However, introducing changes, such as those highlighted in Figure 2.2(b), commonly found in real-world codebases, results in the inability of this approach to analyze and discern commutativity conditions. First, the interleaving of modifying accesses (i.e., write/store operations) to its internal state are interleaved with method invocations (lines 16 to 20). Next, the `visit` method returns a value (line 21) which is explicitly not tolerated by this model and requires use of reference variables (e.g., pointers) to pass information to callers. Lastly, the access to the inner object is also not allowed (line 16). Converting this functionality to be

compliant with the computation model of separability-based commutativity analysis, would require to: i) refactor the `struct statistics` to a standalone class, ii) add access methods (i.e., setter and getter methods) to it, and iii) either pass it by reference to each invocation of `visit` or make it a data member of the node class.

Later work has focused on verifying commutativity conditions for linked data structures [142]. For a number of data structure implementations, including `ListSet`, `HashSet`, `AssociationList`, `HashTable`, and `ArrayList`, it can verify commuting operations that produce semantically equivalent (but not necessarily identical) data structure states in different execution orders.

More recently, [23, 146] have moved towards the automatic generation of conditions which verify commutativity properties.

SERVOIS [23] presents an automatic iterative algorithm that successively refines provided pre-conditions and post-conditions to generate commutativity conditions, assuming that these descriptions describe precisely all effects of the pair of operations under examination.

[146] describes techniques that aim to synthesize commutativity conditions automatically from data structure implementations. This is achieved by providing an abstraction that captures the effects of two methods depending on their execution order and an algorithm that reduces the burden of proving commutativity to a reachability problem, thus allowing the use of off-the-shelf program analyses.

2.3.2 Output-Based Commutativity

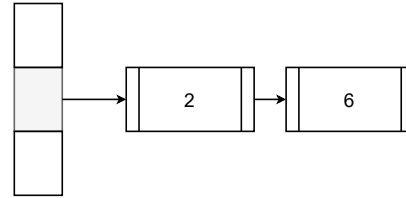
In [3] an alternative notion of commutativity (and analysis) is proposed for individual functions, which considers the *output* of a function at its point of use. In order to automatically detect commutative functions, a candidate function is symbolically executed in two different orders to create an abstract representation of the result of the two execution orders. This symbolic result is then used as input to all functions that could potentially read the results, and those functions are symbolically executed. If the outputs of these reader functions are identical, then the initial function is commutative.

Here, commutativity is a property of a *single function* w.r.t. multiple invocations of the same function. It is not the memory state created by the execution of this function, but in fact the state generated by all the potential consumers of the values generated by the function that determines the commutativity property. For example, consider Figure 2.4, the *insert* function might place numerous items in a hash set.

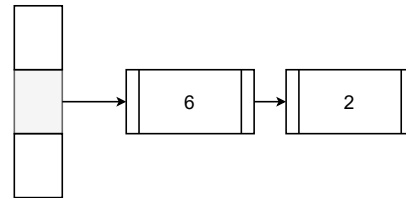
```

1  // definition
2  class hash_set {
3      ...
4  public:
5      void insert(int x) {...}
6      bool is_member(int x) {...}
7      void remove(int x) {...}
8  };
9
10 // declaration
11 hash_set hs;
12
13 if(comm_test) {
14     // input set 1
15     hs.insert(2);
16     hs.insert(6);
17 } else {
18     // input set 2
19     hs.insert(6);
20     hs.insert(2);
21 }
22
23 hs.is_member(2) // true

```



(b)



(c)

Figure 2.4: Output-based commutativity [3] example (based on [3]) where a hash set data structure contains a commutative insert method (left). Invoking insert with two diff input sets at lines 15 and 19 (left) creates two different memory layouts (top right and bottom right, respectively). However, the function is still commutative since subsequent invocations of `is_member` (e.g., at line 23) or `remove` will produce the same results for both memory states.

Reversing the order of multiple invocations to this function creates a list under the same hash bucket containing the same items, but in a different order (i.e., different memory layout as depicted in Figures 2.4(b) and 2.4(c)). If a subsequent consumer of this list produces the same result, irrespective of the order in which items are stored in the list, then the *insert* function is considered commutative. For the code in Figure 2.4, the membership call at line 23 will be true for either insertion order (i.e., input sets at lines 15 and 19).

While this notion of commutativity has strength in handling e.g., unordered container data structures and their use in the same spirit as [142], it is limited to the repeated, possibly commutative invocation of a single function.

2.4 Automatic Parallelization

2.4.1 Overview of Approaches

The objective of automatic parallelization is to automate the task of transforming a sequential application into an equivalent parallel version. Code analysis supporting parallelization has a long history [152, 173, 197].

Initial efforts in the field were concentrated on discovering and exploiting DOALL parallelism and some forms of DOACROSS [117]. These culminated in the development of influential compiler frameworks such as PFC/PARAFRASE [5], POLARIS [32, 195] and SUIF [236]. Other outcomes in compilation technology were the formalization of dependences (discussed earlier in this chapter) and frameworks that reason about a loop's iteration space (e.g., polyhedral model Section 2.4.6.2).

PFC/PARAFRASE was an early Fortran source-to-source compiler which pioneered the exploration and development of dependence-based parallelization (see Section 2.4.6.1).

Later in the 1990s, the POLARIS compiler pushed these techniques further to target actual real-world programs and not only isolated loop nests. Apart from data dependence tests, it implemented privatization, induction variable elimination and symbolic analysis for loop bounds. It also implemented data placement and parallel execution scheduling to map the discovered parallelism to the target machine.

An important contemporary to POLARIS was SUIF, another dependence-based parallelizing compiler that contained a larger set of interprocedural analyses in its arsenal and employed them more systematically in order to tackle the parallelization of larger and complicated loop nests across program scope.

More recent work in the static analysis field geared towards cyclic and pipelined multithreading, such as DSWP and HELIX is presented in the next section and Chapter 3. Another static analysis based on idiom recognition via constraints [96, 95], which we compare to our evaluation of work in this thesis, is also discussed in Chapter 3.

Moving hand in hand with advancements in dependence analysis, research in parallelization has also progressed to the use of execution-time information. Tracing and

dynamic profiling have been used to capture application properties for subsequent exploitation in benchmark characterization [74], collaborative runtime verification [33] and dynamic program optimization [26]. We discussed in Section 2.2 how unsafe dependence profiling has been used to overcome the limits of static dependence analysis [248, 160, 81]. Such approaches, including [14, 81, 285, 248, 144, 143, 127, 262, 219, 282, 283, 277], combine dependence profiling information with static dependence analyses to gain additional information on *may*-dependences.

SAMBAMBA [237] combines a number of techniques in a dynamic framework for automatic parallelization. The main component is a runtime system that performs profiling, parallelism detection/extraction and Just in Time (JIT) recompilation. It also applies multiversioning, containing several versions of the candidate parallel sections, specialized to the current execution context. DISCOPOP [159, 160] is another profile-driven dependence-based approach aiming at code regions of varying granularity. DEPENDENCE PROFILING [248] uses profiling information to create a dynamic Program Dependence Graph (PDG), combining control and data dependences, and uses this representation to discover loop independence (or lack of it). More akin to our approach presented in Chapter 5 is the DEPENDENCE PROFILING [248]. In our approach, we extend this analysis for PLDS-based loops using a uniform iterator recognition analysis (Section 3.2.8) and apply commutativity for determining parallelizability. We present DEPENDENCE PROFILING and DISCOPOP in Chapter 3.

2.4.2 Parallelization in the Presence of Dependences

In this section we present research that tackles parallelization despite the presence of loop-carried or other parallelization-inhibiting dependences. The underlying analysis of these techniques can be dependence-based, commutativity-based or use a mix of both.

Pipeline-like parallelism from outer program loops has been exploited in PARALAX [261]. However, this approach relies on manually inserted code annotations to mark up absence of dependencies or directly influence the result of alias analysis. The parallel code generation is based on the DSWP technique [189].

ALTER [255] is a dynamic technique which exploits *breakable dependences* for parallelization, extracting loop parallelism by reordering iterations or allowing stale reads. ALTER relies on the user to dictate the parallelism policy (out of order or stale read) and, partially, on a set of tests to infer potential annotations, and it has been evaluated on hand-picked loops, some of which have been manually parallelized.

DSWP [189, 201] and HELIX [46] are both designed to derive parallel execution schedules for loops with data dependences. Some variants rely on additional hardware support for the fast communication of values and synchronization between threads, typically outperforming their software-only counterparts.

DSWP is loop partitioning technique which schedules the computations that exhibit loop-carried dependencies on the same core, thus avoiding the communication costs of forwarding the dependent values from core to core as is the case for DOACROSS parallelism. The candidate loops are presented to the compiler after being selected by the user.

HELIX executes each candidate loop’s iterations in parallel. It utilizes a range of analyses to identify potential loop-carried dependences and ensures that these parts of the loop are executed sequentially, while minimizing the number and optimizing the placement of synchronization points. Synchronization is achieved via the memory system during runtime, thus HELIX uses helper threads that hide a lot of overhead associated with memory.

[142] presents a technique to verify commutativity conditions, which are logical formulae that characterize when operations on a linked data structure commute. Code annotations for commutative functions are also proposed in parallelization frameworks by [38], as well as in GALOIS [150] and PARALAX [261].

[196] presents a commutativity-based programming model called COMMSET (Commutative Set) and its associated compiler technology. 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.

FRACTAL SYMBOLIC ANALYSIS uses a commute operation to verify correctness of restructuring transforms, which have a user-provided, rule-based description in their implementation [172]. The analysis applies these simplification rules recursively and aims to prove that the generated simpler program is equivalent with the original program before proceeding further.

DCA, our commutativity-based approach presented in Chapter 5, shares this notion of iteration reordering and breakable dependences with ALTER, but instead of programmer annotations we derive this information automatically. ALTER relies on a weaker test-driven framework to infer likely annotations, and it has been evaluated on hand-picked loops, some of which have been manually transformed for parallelization. In contrast, we evaluate our technique across standard benchmarks and demonstrate

parallelization using existing techniques. Lastly, while DSWP and HELIX focus on the exploitation of parallelism, DCA complements them by identifying sources of profitable parallelism. Our comparison with DSWP is mainly centered on PLDS-based loops.

2.4.3 Speculative Parallelization

We have already described how the inability to utilize and discern a program's behavior during execution has led researchers seek insight in dependences analysis and parallelization using data from profiling runs. In the same vein, researchers have turned to methods that attempt to predict the properties of a program's execution and transform it in anticipation of these assumptions, i.e., execute the program with speculation. Speculative parallelization techniques optimistically execute potentially independent regions of code in parallel [80, 280, 178]. As such, they circumvent the issues associated with the precision of static dependence analysis and other input-dependent factors that adversely affect static analysis. Moreover, having knowledge of the iteration space, they can facilitate partial parallelization.

Briefly, speculative parallelization operates in two stages [280]:

- i) *compile-time*: parallelization of the target program, assuming its sequential semantics are preserved, and instrumentation with safety tests and recovery/propagation code.
- ii) *execution-time*: parallel execution of transformed code segments (Item i)), execution of rollback/propagation based on the results of the safety tests.

In the first stage, a speculative compiler must keep track of the memory access patterns of the candidate code section for parallelization, generate precise test conditions capable of detecting violations, and place them appropriately in the code text. A violation of a safety check means that the assumed program semantics have not been maintained during parallel execution, at which point any local changes kept by the offending thread are discarded and a rollback operation takes place to ensure that execution resumes from a valid state. Otherwise, if there are no conflicts, the thread-local modifications are propagated and become visible to the other parallel-executing threads (i.e., the changes are committed). The compile-time stage can also be enhanced with profiling information from previous program executions [131, 180].

An early precursor of this line of work on parallelization was known as the inspector/executor model. In this model, the candidate loop was converted to two versions:

- i) the *inspector* loop is a stripped down version of the original aiming to examine if any accesses to shared data materialize dependences that would prevent parallelization.
- ii) the *executor* loop will execute a parallelized version of the original, assuming that no violations were detected.

The LRPD test [205] is one of the most influential examples of work in this area of speculative parallelization, targeting DOALL loop parallelism. Each inspector loop receives a chunk of the iteration space, keeping track of each memory access in auxiliary data structures. At the end of this stage, inspector threads check each other's access trace to determine if a violation occurred. Earlier, in [287] implemented a multiphase inspector/executor model for each chunk of iteration space, scheduling their parallel using inter-chunk dependences, thus tackling partially parallelizable loops. The main disadvantage of this precursor technique is that sufficient stripping of the inspector loop might not be feasible, resulting in an overall slowdown when both stages are accounted for.

Since then, research has moved on to explore hardware support, compiler extensions, software behavior characterization and overheads reduction [80]. In terms of the software elements that work on speculation has targeted, they can be classified as [137]: i) control speculation for conditional statements, ii) data-dependence speculation, with goals similar to [205] described earlier, and iii) data values (i.e., value prediction) for predicting the result of instructions to avoid recomputation and reduce overheads associated with misprediction.

The main overheads of speculative parallelization stem from the processing that takes place during its execution. Therefore, a lot of work has focused on investigating ways to reduce the costs of safety tests and state handling (i.e., discarding, committing and reverting state) [73]. More recent studies [121] have further identified the limitations: i) limited coverage, ii) load imbalances, iii) under-utilization of synchronization and value prediction, and at a lesser degree, iv) data dependences.

[180] proposes three approaches (coarse-grained, fine-grained and judicious) of taking advantage of loops when there are a few transient loop-carried dependences, augment the HELIX compiler [46] with speculation.

[281] introduces a new data structure to reduce memory overheads by hashing memory accesses. PRIVATEER [131] performs a LRPD-style parallelization, using profiling information to selectively privatize data structures. This is achieved by observing accesses and organizing them in heaps which are speculated to be independent, instead of reasoning about them individually. Subsequent work on PERSPECTIVE [10] improves

upon the overheads of PRIVATEER and also introduces a planning phase to select the most profitable set of parallelization-enabling transformations. SPEC-DSWP [257] adds speculation to the DSWP described earlier in this chapter, with the objective to increase the stages in the loop pipeline by the assumed absence of dependences.

For PLDS-based programs, [244] extends an earlier approach to software-based parallelization that separates the program states to speculative and non-speculative by adding a mapping table to keep track of accesses between these sections.

[70] analyzes program behavior to assist in manual parallelization, by examining manually annotated code regions as potential parallelization candidates, and running both the sequential and speculatively parallelized version of the code. If a violation is detected in the parallel version, the system uses the results of the sequential execution.

APOLLO [129, 41, 40] is polyhedral compiler/runtime system that uses speculation to guide the parallelization of loop nests that are determined to be affine at runtime. This line of work is also discussed in Section 2.4.6.2.

SAMBAMBA [237] (also presented in Section 2.4.1) uses speculative execution to supplement its profiling-based analysis.

Our hybrid technique, described in Chapter 5, avoids the usual costs associated with speculation during runtime, and even though user interaction might be required during analysis, in practice, our technique offers high precision. We consider speculative techniques orthogonal to the detection capabilities of our approach, which can be used on top of it for further exploitation.

2.4.4 Analyzing Pointer-Linked Data Structures

In this section we give a brief overview of research in automatic parallelization for programs that use extensively Pointer-Linked Data Structures (PLDSs).

[102] presents an approach that is based on programming language annotations to exploit parallel execution of PLDSs.

Pioneering work by Laurie Hendren in the early 1990s has focused on parallelizing programs with recursive data structures [107], analyzing pointer-linked data structures using description of their properties [106], annotation of pointer based programs [115], and programming language design supporting the analysis of PLDSs [116]. The same research group also investigated the effect of improving the precision of shape analysis [93] and alias analysis [94] as enablers for the parallelization of PLDSs. [114] presents a novel technique to improve dependence testing on PLDSs whose nodes

exhibit some regularity using theorem proving. It attempts to prove properties that should hold for all nodes of the data structures, from simple tree-like to complex structures which contain cyclic dependences. It is worthwhile to note the direction of research away from pointer analysis and towards a programming model where programmers describe their intentions [116].

The next wave of research in the second half of the 1990s and early 2000s has focused on shape analysis [212, 25], which seeks to discover and verify (shape) properties of linked, dynamically allocated data structures by means of static analysis.

More recently, [260] studied the performance of programs containing PLDSs on an INTEL architecture and found that the observed degradation in execution times is due to the irregularity of control flow that is more pronounced in such programs. However, the study is limited to a single architecture and admits that there are other factors that can have adverse effects on performance when dealing with PLDSs. Informed by these findings, subsequent work [259] has investigated ways of optimizing the data layout of PLDSs.

2.4.5 Parallelization with User Interaction

Parallelizing systems relying on user-provided feedback may require that input at various stages [161]; from annotating code [261, 18, 41] to validating the proposed transformations [288, 248]. In all cases the user is responsible for ensuring correctness or providing such guarantees. Involving the programmer in different steps of the parallelization process allows supplementing analyses and transformations with information which is difficult to derive automatically. The potential for user involvement is vast, hence we limit the discussion to outline research that has introduced ways of taking advantage of user input and is more closely related to this thesis.

In the last few decades, a trend for systems focused on user-assisted parallelization has been towards easing user reasoning and understanding with information associated to source locations [53, 104], interactivity and step-wise refactoring [104, 123, 161, 69, 286], and various visualization approaches [271, 288].

PAT [235] operates on Fortran source and performs a preliminary dependence analysis of candidate loops for parallelization. Then, it presents to the user a classification proposal for variables references in the loop (e.g., private, shared, etc.) and dependence relations which can be accepted or rejected, modifying the dependence graph appropriately.

PIPS [122] is a source-to-source compiler that also targets Fortran loop nests using interprocedural analysis. The user is able to specify and order analyses or transformations, such as the selection of specific induction variable analysis.

The PARASCOPE EDITOR [139, 104] works in similar vein to the aforementioned tools, building upon the PFC Fortran parallelizing compiler. It lets the programmer change both the source and the underlying supporting data structures (e.g., dependence graph). This two-way update capability results in a shorter “inspect-edit-update” cycle in the manual parallelization process. The system also allows execution of the code with part or all of the input set to provide another source of information to the user.

HPFIT [37] is a set of integrated tools that allows the user not only to annotate source code, but also direct the mapping of parallelism and supply cost model values (e.g., cost of communication) for the optimization of the parallel code generation.

SUIF EXPLORER[161] is also an interactive parallelization tool that adds more sophisticated support from execution-time with extensive profiling, in combination with program-slicing analysis in order to focus the programmer’s attention to long-running loops and generate a dynamic dependence graph.

Our system in Chapter 5 adopts an approach that uses profiling information similar to [248, 167], extending the potential parallelization candidates to PLDS-based loops, for: i) identifying the hottest loops, ii) ignoring dependences that do not materialize, and iii) comparing computation results between sequential and transformed executions.

2.4.6 Parallelization Frameworks

2.4.6.1 Dependence-based Parallelization

As mentioned in Section 2.2, dependence has been the underpinning analysis of parallelizing compilers for decades, hence a lot of the developments in each field go hand in hand (e.g., dependence direction and distance vectors). Moreover, the vast volume of literature focuses on the parallelization of loops, since these account for the majority of a program’s execution time. The main objective of parallelizing compilers is to identify independence among loop iterations to facilitate reordering or be aware of dependences to allow for locality and synchronization optimizations [274].

The main parallelization enabling transformations which either remove dependences or arrange them in a manner that does not prohibit parallelization are privatization and induction variable elimination [7]. Privatization of a scalar or an array aims to identify and move these potentially shared variables to the private local storage of each parallel

running thread of execution, thus providing a distinct and separate instance of that variable to each processing element [253, 205]. Induction variable elimination is able to detect the variables that control the loop iterations from their read (e.g., in the loop condition statement) and write (e.g., at the increment step) access patterns, thus recognizing that this loop-carried dependence is not disallowing parallelization [2]. The concept of iterator variable recognition has been recently expanded and generalized in [166], presented in more detail in Section 3.2.8.

Based on dependence and the previously described fundamental transformations, further and more extensive loop restructuring aims to expose and exploit fine-grained and coarse-grained parallelism with a focus on its legality and profitability. In the most general sense, the legality of each transformation, when all essential dependences have been captured, is guaranteed when it preserves all dependence relations. Some of the most important loop restructuring transformations [273, 274, 7, 275, 17] are: i) statement reordering, ii) loop unswitching, iii) loop peeling, iv) loop unrolling, v) loop fusion, vi) loop fission, vii) loop interchange, viii) loop reversal, ix) loop skewing and x) loop tiling/blocking.

A lot of these transformations do not only work in isolation, but many operate and are enabled in conjunction with others. However, since each loop restructuring has its own set of rules and checks for legality and transformation, it is difficult to reason or predict the result of the composition of transformations. A partial resolution to this problem was the development of unimodular loop transformations which aims to provide a unified framework able to describe a set of transformations and their compositions in a single representation [272, 22]. In short, the effects of all unimodular transformations can be represented as a product of a square unimodular matrix (i.e., a square matrix with only integral components and a determinant of 1 or -1) and a dependence distance vector. Unimodular transformations have two limitations: i) they can only be applied to perfectly nested loops, and ii) all statements in a loop nest are transformed in the same manner (i.e., it manipulates the entire iteration space). Hence, transformations such as statement reordering, loop fusion, distribution and tiling are not unimodular. Further work to overcome these limitations has been conducted in [138, 202], which although it has managed to increase the framework's expressiveness, it has also complicated its operation (e.g., more complex checks for legality).

Unless otherwise specified in the following sections of this chapter, dependence is used as the main enabler for parallelization. In Chapter 3 we describe the INTEL ICC which is a state-of-the-art compiler implementing several of the dependence-based loop

restructuring transformations referenced in this section.

2.4.6.2 Polyhedral-based Parallelization

The polyhedral (or polytope) model is a mathematical framework for analyzing and optimizing loop nests, tracing its origin to the study of systems of uniform recurrence equations [136]. In this model, a loop's iteration space and its dependences are represented as convex polyhedral containing lattice points that correspond to each loop iteration (i.e., the dynamic instances of a loop's body statements). This allows for a succinct representation of the iteration space (or domain) as a set of linear inequalities. It also requires that the loops bounds and all access in the loop nest can be conveyed as affine functions of the loop integer induction variables and all conditionals in the loop can be computed statically. The loop nest, its bounds and affine access functions are collectively known as a Static Control Part (SCoP) in polyhedral compilation [82].

The polyhedral model builds on top of the concept of dependence, hence, a lot of the early work in it, described in Section 2.2.1, is also part of the origins of this model (e.g., [198]). The identification of the presence (or absence) of dependences among scalars and array access is critical for the polyhedral model and the precise description of the loop iteration space as a polyhedron. The application of transformations over this polyhedron aims to reorder each corresponding statement by assigning a different execution order or *schedule* [83]. Intuitively, the schedule assigns a logical date that dictates the order of execution of each iteration instance. The polyhedral model enables the expression of various transformations and allows reasoning about their composition under a unified framework [56]. Further research has spurred the implementation of various polyhedral compilers able to parallelize loops in certain application domains [251, 100, 34, 222].

More recent work has targeted the limitations that arise when certain knowledge about a loop cannot be determined statically (e.g., loop bounds or non-affine accesses). More specifically, [234] studied the detection of Static Control Parts (SCoPs) in real-world programs from various domains and showed that: i) aliasing issues are limiting, ii) high sensitive to the ordering of other compiler optimizations, and iii) performing the analysis at runtime instead of at compile time increases its potential. [24] extends the polyhedral model in the presence of more complicated conditionals in the loop body (e.g., an *if*-based break statement) using control predication of statements, an approach similar to *if*-conversion. [231] combines polyhedral and syntactic transformations, using them in an optimization flow with a more precise cost model that generating code

that exhibits improved data locality and reuse. In [71], the polyhedral framework is augmented with the generation of assumptions that describe and verify the necessary preconditions in order for polyhedral compilation to take place at runtime if these are met.

The presence of loop-carried dependences still constitute a parallelization-inhibiting factor in the original polyhedral model. Early research [208] focused on pattern-based detection of constructs such as reductions and scans, by applying transformations such as array expansion and renaming to break loop-carried dependences. [72] enables the reduction-aware polyhedral optimizations by detecting the commutativity and associativity of reduction operations using constraints on the access functions and the flow of dependences. [207] allows the user to annotate reductions at the source-level and uses a template-based code generation scheme for their exploitation on GPUs.

Other work has examined the application of speculative parallelization in conjunction with the polyhedral model. [129] uses the polyhedral model in a Thread-Level Speculation (TLS) system for parallelizing loops by re-scheduling its iterations at runtime. APOLLO [41, 40] extends [129] with a runtime code generation that reduces time overheads and handles loop nests with pointers and indirections as long as they conform to polyhedral behavior during execution. The framework uses interpolating functions to assist and predict the representation of loop indices as linear functions. The parallel execution step uses profiling information to choose the polyhedral transformations that would yield better speedups.

We compare our proposed commutativity-based approach for loop parallelization against the APOLLO and POLLY polyhedral compilers in Chapter 5.

2.5 Symbolic Execution

Symbolic execution is a program analysis technique introduced in the 1970s to test whether certain program properties and assumptions hold [145, 45]. In concrete execution a program is run using a single input set, leading to under-approximations of the required properties by the corresponding analyses. Symbolic execution, however, uses symbolic values which are *classes* of inputs represented in a succinct manner (e.g., value ranges within a domain), to execute the program on a symbolic execution engine [20]. The key idea is to derive useful and strong guarantees on program properties for which there is no automated method to determine their status, by reasoning about more than one input at once.

For these reasons, symbolic execution has been a fertile ground for research in automated software testing and security via automatic test case generation, input filter generation, vulnerability detection and malware analysis [44, 221, 168]. On the other hand, the ability to represent multiple potential inputs at once poses several challenges for symbolic execution [221]. The main challenges are: i) path selection strategy, ii) interaction with the environment (e.g., system calls), iii) representation of memory, and iv) representation of floating point operations. In response, researchers have started using concolic (i.e., both *concrete* and *symbolic*) inputs to overcome some of these limitations [223, 199, 92]. Other techniques sidestep the hurdles of path selection and path explosion by focusing execution on regions of interest [184, 250], using a minimal representation of that program that exhibits the same properties [59] or selecting program transformations that maximize its efficiency [43]. For the memory representation issues, some tools like [44] use a memory model which concretizes pointers on multiple dereferences and clones the maintained program state when a pointer can refer to multiple objects. MEMSIGHT [61] reduces the need for explicit concretization of pointers by using a compact symbolic address representation which increases the precision of pointer reasoning. Symbolic floating point operations are either approximated or not supported, however, recent work has studied the use of constraint solvers that allow floating point reasoning in symbolic execution engines [214, 162].

While symbolic execution is predominantly popular in the domains of security and software testing, its capacity to systematically explore all the feasible paths along with the generation and solution of constraints has found some application in parallelization [59, 206].

KLEE-FP [59] employs symbolic crosschecking to verify that implementations of programs that use Single Instruction Multiple Data (SIMD) instructions on processor architectures with such support are equivalent to their original scalar versions. It is argued that although automatic vectorization is an active area of research, a lot of code aimed at running on Single Instruction Multiple Data (SIMD) processors to exploit data level parallelism has been rewritten manually. KLEE-FP avoids representing floating point numbers by using a series of canonicalization transformations and comparing the expressions in each program version syntactically.

SYMPLE [206] presents a system that allows the parallelization of user-defined aggregations on MapReduce-style *group-by* queries. SYMPLE allows the parallel execution of these complex aggregations when iterating on chunks of the input despite

the existence of loop-carried dependences. Its key insight is to treat the values related to these dependences as “unknown” symbolic values (essentially breaking them) and execute the aggregation symbolically, thus returning as output a symbolic *summary*. These summaries are combined and computed at a final reduction step to produce the actual concrete output. Programmers are required to code their aggregations using SYMPLE’s programming model which facilitates the added requirement of this system for both soundness and precision, unlike typical applications of symbolic execution in testing and security.

We use KLEE to perform our symbolic execution experiments in Chapter 4. KLEE’s design characteristics and operation are described in more detail in Chapter 3.

2.6 Algorithmic Skeletons

The concept of algorithmic skeletons was introduced by [57] to abstract recurrent patterns of computation, communication and synchronization. They were intended to address challenges in parallel programming [63], in the spirit of structured (sequential) programming and design patterns [90] for software engineering, by:

- i) the developer’s choice of suitable compositions of parallel skeletons to model the parallel behavior, and
- ii) the developer’s responsibility to provide a suitable and efficient skeleton in support of the required parallel behavior.

Since their initial inception, they have been widely adopted by industry and research community, resulting in a plethora of programming models, Domain-Specific Languages (DSLs), libraries and frameworks, such INTEL’s TBB, GOOGLE’s MAPREDUCE, SKEPU and FASTFLOW [58, 98, 171].

Based on their features, such as scope, structure, granularity and internal interaction of their subcomponents, skeletons can be classified as [98]:

- i) *Data-parallel*: operating of bulk data, with examples such as *map*, *reduce* and *scan*.
- ii) *Task-parallel*: operating of tasks and their interactions, with examples such as *farm*, *for*, *sequential* and *pipe*.

- iii) *Resolution*: take on a family of problems, with examples such as *divide-and-conquer* and *branch-and-bound*.

A focal point in research, relevant to this thesis, has been the identification of these patterns in legacy sequential code. The key idea is that having the computation expression in algorithmic skeletons provides a pliable form of the program, which is more amenable to parallelization, not tied down and obscured by low-level implementation decisions explicitly expressed in the code. We give an overview of representative work in this section.

PARAMAT [140] uses a static, hierarchical, pattern-matching approach over an Intermediate Representation (IR) to detect common constructs (e.g., matrix-matrix multiplication) in scientific codes. This restricts the approach to detecting fairly idiomatic operations that match exact specifications, thus it is not able to deal with more generic patterns.

PAP RECOGNIZER [68] is another tool for parallelizable pattern discovery that shares similarities with PARAMAT, such as the hierarchical and IR-based operation. However, PAP RECOGNIZER uses Prolog to describe inference-based rules for matching over a hierarchical PDG. All found matches are presented to the user to make the final choice.

XARK [11] uses an extension of the popular Static Single Assignment (SSA) form for IRs to analyze use-def chains in Strongly Connected Components (SCCs), aiming at reduction and recurrence computations. The idiom recognition is a two-step process that, first, examines intra-SCC and, then, inter-SCC dependences to reveal more complicated computation kernels.

[247] presents a profiling-based pipeline detection technique which extracts pipeline stages at multiple loop nest levels. It is also capable of replicating pipeline stages that might be a performance bottleneck, if it proves that they operate on independent data elements, thus creating opportunities for OoO execution.

STNG [134] uses program synthesis to identify stencils in Fortran source code and translate them to a high-level predicate language for use with HALIDE's DSL [200]. In this way it leverages the specialized optimization offered by HALIDE and enjoys easy migration to GPUs.

[52] is technique that offers a path towards progressive raising of multi-level IRs, complementary to the typical progressive lowering performed by multi-level IR compilers. It uses a declarative description language to specify the computational pattern to which a rule might apply along with a set of replacement expressions. It currently

targets affine and Basic Linear Algebra Subprograms (BLAS) constructs, thus enabling domain-specific optimizations at these abstraction levels.

[164] presents a profiling-based technique that applies constraint matching in the produced traces to discover specific patterns (i.e., maps, reductions, map-reduction fusions and other variants of these). The key idea is that each pattern can be identified by the topology of the dynamic data flow it creates. This technique is able to operate on both sequential and already parallelized code, aiming to help the rejuvenation of legacy parallel code.

In Chapter 6 we present a brief description of a unified framework for the description of algorithmic skeletons built on liveness-based commutativity, and its adaptation for discovering loop-level parallelism.

2.7 Conclusion

We have presented a concise overview and comparison of relevant prior research most relevant to this thesis. Here we summarize the main differences of our work and highlight its placement in the presented spectrum of literature in the field of automatic parallelization.

Automatic parallelization and supporting approaches have increasingly used dynamic information to fill in the knowledge gap. Moreover, recent studies have shown that dependence analysis, which has been the prevailing enabler for parallelization discovery, has reached its limits even when augmented with dynamic information. Thus, in Chapters 4 and 5 we examine the use of a novel commutativity analysis for the parallelization of loops, first, in a symbolic execution context and, then, with an execution-time profiling-based approach. The foundations of this novel analysis are presented in Chapter 4 and adapted in the context of loop execution. We overcome shortcomings related to previous approaches, especially when dealing with irregular and PLDS-based loops. Chapter 5 also shows a study of the precision of the predictions made by our profiling-based approach.

Finally, a lot of earlier work on algorithmic skeletons was directed towards their description and programming libraries for the development of parallel code. More recently, researchers have focused on detecting these patterns in legacy code. In Chapter 6 we revisit the liveness-based loop commutativity definitions of parallel skeletons and examine the criteria for their detection taking hints from their corresponding parallel versions.

In the next chapter, we briefly present the infrastructure and tools used in this thesis, before moving on to present our studies and key contributions.

Infrastructure and Related Tools

This chapter presents details about the infrastructure used in this thesis.

3.1 Introduction

First, we present the software tools used in our empirical evaluation such as compilers, runtimes, frameworks and specific analyses. These include software components that were used to both assist, support and supplement the techniques developed in this thesis and to compare to in our studies. We continue with a presentation of the application benchmarks and the hardware platform.

3.2 Software Tools

3.2.1 LLVM Compiler Infrastructure

We rely on the LLVM compiler infrastructure (version 7.0.1) for the implementations of the various analyses and transformations used in this thesis. LLVM is also an enabling technology for other tools that we make use of, such as KLEE (Section 3.2.9). There is no inherent dependency of our techniques and tools with the specific LLVM version, but it is rather an artifact of our software development process and cycles.

LLVM started in academia in the early 2000s, but since then it has evolved to a mature, industrial-strength open-source compiler framework [156]. While initially its adoption was slow and limited to academic research, the design decisions taken from its inception, allowed its uptake by the industry, with active backing from industry. Companies like Apple and Intel are involved in the development of LLVM, building on

top of it programming languages (e.g., Apple's Swift), runtimes and toolchains (e.g., INTEL's ONEAPI DPC++ compiler). LLVM's modular and composable architecture has greatly changed the landscape in compiler research and related fields by allowing quick experimentation and the adoption of a wide variety of components at various stages in the compilation process.

Hence, the plethora of LLVM uses has created a whole ecosystem of tools and a federation of related projects:

- i) compiler frontends/backends and optimizers,
- ii) linkers and assemblers,
- iii) libraries and Application Programming Interfaces (APIs),
- iv) JIT engines and runtimes, and
- v) code representation formats.

While several subprojects are officially under the umbrella of the project, the vast majority are independent and only tie in with specific parts of the LLVM project based on their functionality.

The architecture of LLVM [39] follows a classic three-phase design which consists of: i) frontend, ii) optimizer, and iii) backend. This allows the decoupling and composition of different parts that implement each of these phases. Moreover, separation of concerns is facilitated by the use of a single, core, target-independent IR. LLVM IR is a strongly-typed, well-specified, low-level code representation that can be augmented with metadata information. Optimization passes use mainly units of LLVM IR as input and output for their operation.

More recently, the LLVM project introduced support for Multi-Level Intermediate Representations (MLIRs). Multi-Level Intermediate Representation (MLIR) is a new representation format and associated utilities that allow the description, implementation and processing of IR variants that closely model a specific domain. This further increases flexibility, interoperability and composability among compiler-based tools and frameworks.

Hence, implementing our techniques in LLVM IR enables to isolate ourselves from source code idioms and peculiarities, providing a uniform layout on which to operate on. In addition, this allows us to base our work upon a multitude of well-established and mature compiler techniques. Lastly, directly comparing with other techniques

implemented on the same foundational framework is more straightforward, since a common experimental base is typically obtained via small configuration adjustments. Nevertheless, the approaches presented in this thesis are not conceptually tied to a specific technology or tool.

3.2.2 Intel Icc Compiler

ICC is a collection of C/C++ compilers and tools from INTEL, available on most popular operating systems and targeting a wide range of INTEL architectures. This compiler is highly tuned for INTEL processors with support for the latest language standards and is routinely ranked among the top compilers in producing fast code, especially for computational kernels [210]. Although details on its internal operation are undisclosed since it is a commercial, closed-source application, it is generally agreed that this demonstrates its ability to better exploit the target machine architecture (i.e., floating-point operations, registers, etc.) [210]. ICC’s optimization pipeline is organized in the following categories [120]: i) floating-point, ii) processor-specific, iii) interprocedural, iv) profile-guided, v) high-level language optimizations, vi) parallel programming, and vii) optimization support features.

The main optimization support feature is the state-of-the-art data dependence analysis. More specifically, for array references, ICC’s dependence analyzer is organized as a series of tests, performed in a dimension-by-dimension manner, which progressively increase in power, with compilation time and memory also increasing proportionally.

In our experiments, we use INTEL ICC “Classic” version 19.0.5.28120190815 and also utilize the emitted reports (i.e., via the `-qopt-report*` option group) to parse information about the parallelization of each specific loop. Moreover, we use ICC’s profiling capabilities to measure loop execution coverage and INTEL’s OPENMP implementation for parallelization (Section 3.2.7).

It is worth noting that INTEL is moving its compiler infrastructure to the LLVM framework with the release (late in 2020) of the oneAPI DPC++ compiler [65]. Since then, the previous toolchain that is not LLVM-based has been marketed as “Classic”. The new compiler features proprietary optimizations which are implemented as a separate backend.

3.2.3 LLVM/Polly Extension

POLLY is a high-level loop and data-locality optimization infrastructure built on top of LLVM [100]. POLLY uses the polyhedral model to perform loop optimizations by combining several tools:

- i) Integer Set Library (ISL): a portable C library for handling polyhedral and computing dependences.
- ii) PLUTO: a library providing a data locality optimizer and parallelizer for selecting valid transformations.
- iii) CLOOG: a library for code generation via a generic Abstract Syntax Tree (AST) to a final target (e.g., OPENMP).

The central novel feature of POLLY is that it enables polyhedral compilation by directly manipulating LLVM IR.

The main advantages of POLLY are:

- i) enabling easy use via integration in an existing popular compiler framework,
- ii) leveraging existing and enabling new optimizations by offloading the polyhedral model details to specialist libraries,
- iii) decoupling polyhedral optimization from high-level languages, and
- iv) overcoming analysis limitations by accepting optimization descriptions from external tools or the user via an import/export facility (i.e., SCoP description files).

In our empirical evaluation, we parse the exported reports to collect data on which loops are considered parallelizable by POLLY. POLLY’s versioning system follows that of LLVM (version 7.0.1 in our evaluation).

3.2.4 Idioms Technique

Constraint-based idiom recognition (which will be referred as “IDIOMS” in this thesis) is a static compiler technique which automatically detects a wide class of reduction operations [96, 95]. This allows us to extend the breadth of our comparison in relation to the static tools used in our empirical evaluation.

The IDIOMS technique uses a constraint-based approach to describe what constitutes a reduction. To this end, it combines the results of other classic compiler data flow analyses such as the Control Flow Graph (CFG), dominance analysis and SSA form to formulate the conditions that need to be fulfilled for detection. The technique proceeds to evaluate loop code piecewise and allows backtracking if only partial solutions are found while there is still code that has not been considered. Following an intuitive approach in the application of constraints, it avoids the exponential complexity associated with a naive evaluation order.

The technique is implemented as a compiler pass in the LLVM infrastructure (version 7.0.1), operating at the IR level. The constraints are specified using a custom DSL embedded in C++, placed directly in the source of the pass. The parallel code generation is using a pthreads implementation which we were not able to use. Instead, we opted to parallelize the reported loops using OPENMP.

3.2.5 DiscoPoP Technique

DISCOPOP [159] is a dynamic technique that aims to discover pipelines, task and loop parallelism operating in two phases. During the first phase, it places instrumentation calls for profiling memory accesses and control flow during execution. In the second phase, it builds a dynamic PDG based on the collected profiling information. Then, this graph is transformed to another condensed graph consisting of nodes that do not contain parallelization-inhibiting *read-after-write* dependences internally. These condensed nodes are termed Computation Units (CUs) in DISCOPOP's internal representation and, in essence, correspond to *read-compute-write* patterns.

DISCOPOP uses the resulting CU graph for parallelism detection at various granularity levels. At loop level, it tries to determine the absence of loop-carried dependences and, thus, identify DOALL parallelism. For the discovery of parallel tasks, the CU graph is further condensed to SCCs, chains of SCCs (i.e., nodes connected one after the other, without branch or join points). Additional graph partitioning techniques (e.g., a min-cut algorithm) are applied to discern groups of parallelizable tasks. Lastly, a template-matching approach is used to detect pipelines.

DISCOPOP is implemented in LLVM (version 3.3) and it was evaluated in the same version of the NPB suite (version 3.3) that we use for our empirical evaluation. In our comparison we used DISCOPOP's results from [159] in combination with profiling traces from its execution over the NPB suite, which we obtained after private

communication with the authors.

3.2.6 Dependence Profiling Technique

DEPENDENCE PROFILING [248] is the technique closest to the profiling approaches in the techniques used in this thesis and it is extensively described in [246]. DEPENDENCE PROFILING has been used in combination with an ML-based parallelization mapping approach and in the extraction of hierarchical pipeline parallelism from sequential applications [247].

In our experiments, we used the results presented in [248], since we did not have access to a working implementation. The initial prototype was developed in the commercial COSY [8, 242] compiler toolchain.

3.2.7 Parallelization with OpenMP

OPENMP is a shared-memory Application Programming Interface (API) which aims to facilitate shared-memory parallel programming with an emphasis on efficiency, portability and user-friendliness [50]. It is published as a specification and implemented as a language extension, allowing the user to insert annotations in the source to direct parallel compilation and operation with an OPENMP-supporting compiler [188]. OPENMP's success in industry and academia can be attributed to these factors [50]: i) emphasis on structured parallel programming, ii) comparative easy of use, iii) wide adoption, and iv) following community-driven design and development, allowing timely addition of features and implementation releases on broad variety of multicore architectures.

We use OPENMP for parallel code generation due to the following reasons:

- i) Some of our benchmarks (Section 3.3.1) have already manually parallelized versions in OPENMP, giving an upper bound of the potential parallel performance that can be achieved.
- ii) All the compiler toolchains used in our experiments offer a mature OPENMP implementation.
- iii) It was relatively uncomplicated to convert the parallelization suggestions of other tools, which emit parallel code using other models, to OPENMP for a uniform comparison (e.g., IDIOMS used PTHREADS).

```
1  #pragma omp for reduction(+:sum)
2  for (i = 0; i < N; ++i) {
3      sum += array[i];
4  }
```

Figure 3.1: A simple reduction loop parallelized using an OPENMP pragma directive.

Under the hood OPENMP consists of: i) compiler support of directives, ii) a library realizing the specification API via function calls which control execution-time behavior of threads, and iii) a set of environment variables that can configure and influence its execution-time operation. An OPENMP-enabled compiler must recognize, validate, and translate OPENMP directives according to their semantics in order to generate a multithreaded program. The flexibility of the compiler support allows easy integration with source languages (e.g., pragma preprocessor statements in C/C++ and metadata in LLVM IR). The directives map to work-sharing constructs that are implemented by the runtime library, and allow users to:

- i) create teams of threads for parallel execution,
- ii) specify how work is shared among them,
- iii) declare both shared and private variables, and
- iv) synchronize threads.

The environment variables can be used to control execution features such as, for example, the number of available threads or their scheduling policy [191]. OPENMP adheres to the *fork-join* model of execution. The fork-join model is a versatile execution pattern which subdivides a problem to subparts recursively and can be used for both regular (e.g., a map pattern) and irregular parallelization (e.g., a divide-and-conquer strategy). New serial execution control flows (i.e., threads) are created at specified fork points by splitting up an existing thread into two or more threads. Conversely, two or more execution threads are merged together at a join point. It is important to note that the execution model describes the desired parallel behavior and omits any implementation details on how a library may achieve this on a specific Operating System (OS) or architecture [124].

Figure 3.1 presents an example use of OPENMP for the parallelization of simple loop. This directive denotes a for loop and specifies a reduction variable (*sum*) that

is built using the addition operator (+). The programmer does not need to specify how to assign the execution of the iterations to threads or to manage the reduction (i.e., maintain the partial sums and their final accumulation and assignment to sum). Typically, this is internally implemented by privatizing the variables (Chapter 2) that are indicated as the accumulators of the reduction operation by the pragma statement. Moreover, all data are shared by default in the OPENMP memory model, thus giving all threads access to the elements of the array variable. Lastly, at the end of the loop (after line 4) there is an implicit synchronization barrier for this parallel region, meaning that no thread can progress until all other threads have reached that program point. All these details are left to be carried on by the OPENMP compiler and runtime.

Our empirical evaluation aims at the parallelization of loops, hence we make extensive use of OPENMP's `for` work-sharing construct, following similar code generation methodology as described in [248]. We use OPENMP version 4.5 and INTEL's OPENMP runtime library for all executions in our experiments. This allows to intentionally avoid any effects in performance due to differences in implementation details among libraries [125].

3.2.8 Generalized Iterator Recognition Analysis

In the course of the work described in this thesis, we have developed *generalized iterator recognition*, a compiler-based, supporting analysis for the recognition of loop iterators over a wide class of loops [166]. *Generalized iterator recognition* allows us to identify the instructions in a loop that contribute to advancing the iteration (i.e., the “iterator” code); in other words, the instructions that contribute to and determine if the loop exit conditions are satisfied or not per iteration. All the remaining instructions in the loop are characterized as the loop's “payload”. This intuitive description of loop iterators is the core of our *generalized iterator recognition* analysis.

In compilers, iterators have been typically associated with structured `for` loops where the iteration proceeds over a set of integer values by a fixed amount (i.e., it is an induction variable). This allows easy identification (e.g., pattern matching), normalization of lower/upper bounds and step count [7], further exploitation in other models such as dependence and the polyhedral analysis (Chapter 2), and enabling other optimizations (e.g., induction variable elimination) [177]. However, there is a wider range of iterator idioms as a result of the abundance of programming languages, styles and APIs.

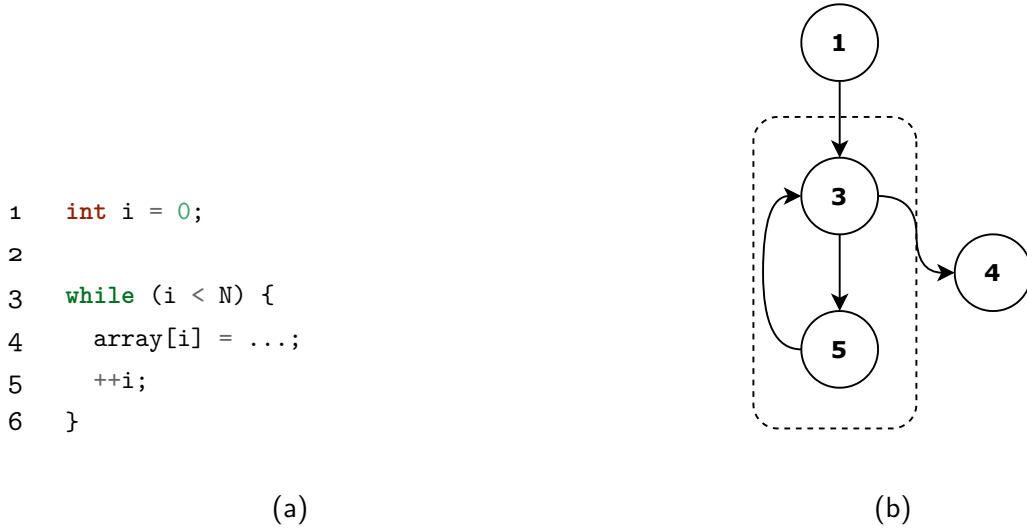


Figure 3.2: A simple `while` loop (left) and the corresponding SCC condensation of its DDG (a PDG without control dependences for simplicity) (right). Each node's number corresponds to the statement at that source line number. *Generalized iterator recognition* analysis will mark the nodes inside the dotted node, which is an SCC node that contains more than one PDG node, as “iterator” code.

The analysis uses dependences to generate a PDG of a loop and, then, condense the nodes to SCC nodes. The SCC node with no incoming edges and a loop-carried dependence of distance 1 is marked as the “iterator” code. Figure 3.2 presents a minimal example of a loop and its corresponding condensed PDG. The analysis can also incorporate profiling information from a program’s executions in order to build a dynamic PDG with may-dependences disambiguated.

We have implemented our *generalized iterator recognition* as an external (i.e., out-of-source) compiler pass based on the LLVM compiler infrastructure (version 7.0.1) [264]. Our current prototype implementation is able to mark the “iterator code” of a loop as: i) in-memory LLVM IR instructions, ii) embedded metadata in LLVM IR, and iii) exported as text in JavaScript Object Notation (JSON) format. Apart from the work in this thesis, our *generalized iterator recognition* has been utilized in other research projects such as rejuvenating legacy parallel code to patterns [164] and extracting code features in a ML-based parallelization assistant. It has also been considered for future inclusion in research-oriented compilation frameworks [169].

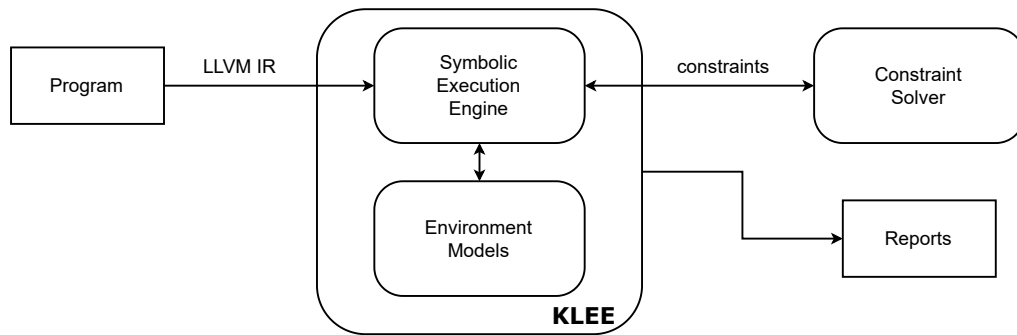


Figure 3.3: Overview of KLEE's architecture.

3.2.9 KLEE Symbolic Execution Tool

KLEE [44] is a symbolic execution tool built on the LLVM compilation framework that automatically generates test case for programs that can heavily interact with their environment. The main goals of KLEE are:

- i) achieve high code coverage (i.e., execute every instruction in the program), and
- ii) detect potential dangerous operations (i.e., bugs such as out-of-bounds pointer dereference) if there are *any* input values that would trigger them.

KLEE, at its core, consists of:

- i) A symbolic engine which:
 - parses and executes symbolically LLVM IR,
 - generates constraints that describe exactly the set of values at a given execution state/path, and
 - offloads constraints for resolution to an external constraint solver.
- ii) Environment models which describe the semantics of interactions with executing programs environment (e.g., command-line arguments, environment variables, file I/O, system calls, etc.).

In other words, KLEE operates as hybrid combination of an OS for symbolic processes and an interpreter. The modelling of an application's environment and its interactions with the OS (i.e., system call API) is implemented via a modified version of the UCLIBC library [254], a minimalistic implementation of the C standard library. Internally, KLEE

strives to compact state representation, optimization of constraint query representation and path selection strategies. A high-level overview of KLEE’s architecture is presented in Figure 3.3. We describe how symbolic execution is performed with an example in Chapter 4.

In our experiments we use KLEE version 1.4.0.0 (built on top of LLVM version 7.0.1) and its default constraint solver (STP [91]).

3.3 Benchmarks

3.3.1 NAS Parallel Benchmark (NPB) Suite

The NPB benchmark suite is named after the NASA Advanced Supercomputing Division where they have been developed and maintained since 1991 for the evaluation of highly parallel supercomputers [19]. Through the years the suite has been regularly updated with new applications, kernels and data sets to reflect the evaluation needs of contemporary parallel architectures. We are using version 3.3 in this thesis, which consists of 10 applications which implement representative algorithms from the field of Computational Fluid Dynamics (CFD). The NPB suite is used in industry and academia in a wide variety of studies, such as comparing parallel programming models [227] or describing specific implementation and their performance [130]. Moreover, NPB provides a diverse set of different implementations in OPENMP, MPI, Fortran and Java. We use the version of the suite that is written in C for our empirical evaluation and is based on [224] (version SNU 1.0.3).

The most desirable attribute of the suite for our studies is that it includes both a sequential and manually parallelized versions of each program. In our experiments we use the parallel version based on an OPENMP shared memory implementation to derive: i) an upper bound of what is possible to detect and exploit as parallelizable, and ii) an upper bound of the optimal speedup that can be achieved by parallelization, according to expert developers with deep knowledge of the algorithms and the program implementations. The aim of the experts has been to exploit coarse-grain parallelism that might encompass multiple loops, in large code sections. Since our evaluation in this thesis is considering loops, where a large part of an application’s execution time is typically spent, we have adapted the parallelism to the enclosed loops. This allows us to compare our techniques against the upper bound of both loop-level parallelism and beyond as presented in Chapter 5.

Table 3.1: Source code details of the applications in the sequential version of the NPB benchmark suite.

Benchmark	Lines of Code (LOC)	Loops	Loop Nests
	(#)	(#)	(#)
BT	2676	182	52
CG	548	47	26
DC	2715	105	37
EP	166	9	7
FT	606	42	16
IS	464	16	12
LU	2428	186	63
MG	879	81	38
SP	2127	250	53
UA	5768	479	157
Total	18377	1397	461

We present a breakdown of the LOC and loops contained in each sequential application in Table 3.1. A brief description of each application follows:

- i) *BT* (Block Triagonal): an application that solves a synthetic system of non-linear Partial Differential Equations (PDEs) using a block (of size 5×5) triangular solver for CFD codes.
- ii) *CG* (Conjugate Gradient): a kernel using the conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse matrix employing unstructured matrix multiplication.
- iii) *DC* (Data Cube): a data-intensive application that implements the data cube operator [36] over an arithmetic data set. Data cube is a common operation in data mining and database systems where it is used to compute view of multidimensional data sets. It primarily aims to help in the evaluation of the memory hierarchy of systems and data movement across it [89].
- iv) *EP* (Embarrassingly Parallel): a kernel which evaluates an integral using a Monte Carlo method, generating pseudorandom numbers. It requires almost no inter-processor communication, and thus exemplifies the potentially attainable upper bound of parallel performance on an architecture.

- v) *FT* (Fourier Transform): a 3D PDE solver using Fast Fourier Transforms (FFTs), common in spectral methods.
- vi) *IS* (Integer Sort): a kernel that performs a bucket sorting operation.
- vii) *LU* (Lower-Upper): an application like *BT* which uses a lower-upper symmetric Gauss-Seidel solver.
- viii) *MG* (MultiGrid): a kernel that uses the multigrid method to solve a 3D discrete Poisson equation.
- ix) *SP* (Scalar Pentagonal): an application like *BT* which uses a scalar pentagonal solver.
- x) *UA* (Unstructured Adaptive): an application which solves a heat equation with convection and diffusion from moving ball. It features an unstructured adaptive mesh, displaying dynamic and irregular memory access.

We applied the flags presented in Table 3.2 for the compilation of the benchmarks. The main objective of the two option groups is to preserve as many loops as possible throughout the compilation pipeline and in the final compiled output, without hindering the power of analysis and the efficacy of parallelism mapping. Thus, we enable the majority of optimization passes (optimization level O2), while we disable vectorization and loop unrolling. In addition, we enable ICC’s more aggressive parallelization discovery heuristics that operate using information from interprocedural optimization (the `-parallel` and `-ipo` flags, respectively) and processor-specific mappings (`-xcorei7` and `-axcorei7`). We obtained a single LLVM IR archive for each application that had to further go through a LLVM-based compilation pipeline in order to reduce any differences that occur due to phase ordering [13]. Any further additions, modifications or exceptions to the compilation options are discussed on a per-case basis in the remainder of the thesis.

3.3.2 Applications using Pointer-Linked Data Structures

We use a wide variety of benchmarks that employ PLDS-based loops for the empirical evaluation of our techniques in this thesis. These range from industry standard suites, such as SPEC CPU2000 [109] and SPEC CPU2006 [110], to standalone open source scientific applications. We provide a brief description of each application in this section.

Table 3.2: Options per toolchain that were applied during the compilation of applications in the sequential version of the NPB benchmark suite.

Compiler	Options
CLANG/LLVM-based	-O2
	-fno-vectorize
	-fno-slp-vectorize
	-fno-unroll-loops
INTEL ICC	-O2
	-no-vec -unroll=0
	-parallel -ipo -mcmmodel=medium
	-xcorei7 -axcorei7
	-g -rdynamic

- i) *429.mcf* (source: *SPEC CPU2006*): an integer benchmark application that solves the problem of vehicle scheduling in mass transportation using a network simplex algorithm (combinatorial optimization). The task is formulated as a large-scale minimum-cost flow problem employing a custom binary tree data structure. Each tree node contains links (i.e., pointers) to the predecessor node and the two children nodes, and the left/right sibling nodes (if any).
- ii) *300.twolf* (source: *SPEC CPU2000*): an integer benchmark application which is a simulator for deriving an optimal area layout (placement and routing) of chip components. Components are modelled as doubly-linked list nodes.
- iii) *em3d* (source: *Olden* [47, 48]): an application that simulates the propagation of electro-magnetic waves in an object in 3 dimensions. It uses an irregular bipartite graph to represent electric and magnetic field values, each a separate node.
- iv) *mst* (source: *Olden*): an application that computes the minimum spanning tree of an n-node graph.
- v) *bh* (source: *Olden*): an application that solves the N-body problem, represented as an n-node tree, using hierarchical methods (Barnes-Hut).
- vi) *perimeter* (source: *Olden*): an application that computes the perimeter of a set of quad-tree encoded raster images.

- vii) *treeadd* (source: *Olden*): an application that adds the values in a balanced, binary tree data structure.
- viii) *BFS* (source: *Lonestar* [149]): an application implementing a breadth-first search traversal over a graph.
- ix) *ks* (source: *PtrDist* [15]): an application that performs graph partitioning via the Kernighan-Lin, modified Kernighan-Lin, or Kernighan-Schweikert algorithms.
- x) *hash* (source: *Shootout* [241]): an application that performs hash operations.
- xi) *spmatmat* (source: *SPARK00* [258]): an application that performs a sparse matrix-matrix multiplication operation. The matrices are implemented using adjacency linked lists.
- xii) *water-spatial* (source: *SPLASH3* [217]): an application that evaluates forces and potentials that occur over time in a system of water molecules. The water molecules are arranged in a uniform three-dimensional grid of cells. Each particle is represented as a node which can access (via pointers) N other nodes that constitute its neighborhood using a singly-linked list. These neighborhoods also form nodes in a larger singly-linked list that encapsulates the whole problem domain.
- xiii) *otter* (source: *FOSS*): a theorem prover for first-order logic making use of singly-linked lists.
- xiv) *ising* (source: *community* [269]): a kernel implementing a condensed matter physics program using the ISING model. Particles are modeled as nodes in a singly-linked list.

3.4 Hardware Setup Configurations

All our execution experiments were conducted on an INTEL Xeon Gold 6154 (Skylake) CPU with 72 cores, running at 3 GHz with 540 GB RAM on Ubuntu 18.04.4 LTS (Linux kernel 4.15.0–91).

3.5 Summary

In this chapter we described the range of tools, benchmarks and hardware setup configurations used in the remainder of this thesis for the empirical evaluation studies. In the next chapter, we begin our examination of liveness-based commutativity using symbolic execution.

Liveness-based Commutativity Analysis using Symbolic Execution

We now start our study of the use of commutativity instead of dependence analysis for the parallelization of legacy sequential code. In this chapter, we present and motivate our concept of commutativity which is based on the liveness of variables in a code region of interest. Our *liveness-based commutativity* overcomes a lot of the limitations of previous descriptions of commutativity targeting parallelization (Chapter 2), while preserving and getting inspiration from their best features.

We investigate the potential of implementing a practical analysis based on *liveness-based commutativity* in a hybrid static/dynamic context. To this end, we apply an analysis based on symbolic execution, which allows us to describe and assess the required conditions for determining the commutativity of code regions.

We begin the chapter presenting the background of our investigation in Section 4.1 and introduce our concept of commutativity in Sections 4.2 and 4.3. Then, we continue by motivating our approach in Section 4.4 and describing its combination with symbolic execution in Sections 4.5 and 4.6. Section 4.7 outlines a technique based on symbolic execution and Section 4.8 presents our case studies. We conclude in Section 4.9, summarizing the obstacles that symbolic execution poses to the creation of a practical analysis for the parallelization of real-world applications. These realizations inform the approach presented in Chapter 5.

Part of this chapter is based on the work in *generalized iterator recognition* [166] for which I partly contributed to its inception, implementation design and conducting experiments.

4.1 Introduction

As we have discussed in Chapter 1, extracting and exploiting parallelism is essential to maximizing the performance of legacy sequential code on modern multicore systems. While an attractive proposition, compiler-based parallelization techniques have largely failed to deliver the much-needed boost in application performance outside a narrow domain of scientific codes [7].

Over the last few decades, dependence analysis has grown to be an essential enabling component of the analyses and transformations offered by parallelizing compilers. Recent work has identified inherent limitations in the expressiveness of dependences to model program operations [180]. This is mainly due to the inherent inability of dependence analysis to prove that violating certain relations between memory locations does not always affect program results [255, 172, 209]. Or, to put it another way, some dependences are safe to violate (e.g., in the case of a reduction operation). Therefore, this leads to over-approximating (i.e., *sufficient* but not *necessary*) assumptions regarding program functionality that inhibit detection of further available parallelization opportunities.

A promising approach to overcome the limitations of dependence analyses is the use of *commutativity analysis* [211, 3, 266]. In a nutshell, commutativity analysis detects code regions whose order of execution can be exchanged without altering the program outcome. Prior uses of commutativity have either been applied in a limited manner [38] or expressed within restricted models of computation [211, 3], thus failing to expose the full potential of such an analysis.

In this chapter, we present a new notion of commutativity based on the liveness of operations, which relaxes the strict data dependence constraints imposed by traditional parallelization approaches. Our *key insight* is that some of the detected dependences are not, in fact, fatal to parallelization, and therefore cause some loops to be needlessly discarded. Instead, *liveness-based commutativity* analysis focuses on the more crucial issue of whether any such dependences have a detectable effect on the eventual result. We have also introduced an adaptation of this concept for loop constructs, since often these are code regions where applications spend most of their execution time.

Our objective is to explore the capacity of a practical analysis using *liveness-based commutativity* in extending the scope of automatic parallelization, while avoiding the usual limitations of a purely static system (Chapter 2). As such, an important aspect of our approach is the potential of using a combination of static and dynamic

information to model program inputs and outputs. For this, we employ *symbolic execution* which is a technique that models program values as classes of data, i.e., symbolic values, and represents the result of the computations over them as symbolic expressions and functions of their input [145, 45]. This enables reasoning about higher-level program properties (e.g., complex assertions) and has found extensive use in software testing. Consequently, we develop an approach based on symbolic execution to verify commutativity by testing equivalence on program outputs (i.e., live-outs), and ultimately parallelizability of loops. However, symbolic execution is associated with certain innate scalability challenges. We examine the limitations and impact that these characteristics have in the applicability of our *liveness-based commutativity* analysis with our case studies on loops taken from real-world applications. These results, subsequently, inform our approach in Chapter 5.

4.2 Liveness-Based Commutativity

We have already discussed a number of different notions of commutativity for parallelization [211, 3] in Chapter 2. In this section we give a synopsis of the main previous concepts of commutativity from the literature and continue to briefly describe the basic concepts and building blocks of our *liveness-based commutativity* [266].

Separability-based commutativity considers computations as separable operations on objects, where their semantics are clearly defined by the programmer. Thus, having a well-defined description of operations, the compiler is able to determine the effects of operations and reason about commutativity in this restricted programming model.

Output-based commutativity determines if invocations of a *single* candidate function commute, using a technique supported by abstract interpretation to model its inputs and follow its result over the use-def chain (i.e., recursively following readers of readers).

The *key idea* behind our concept of commutativity based on liveness [266] is to restrict commutativity requirements to only those variables which are live-out at a region of interest. This allows for the separation of transient variables and associated computations, whose values are not used anywhere later in the code to be relaxed.

The objective of our definition of commutativity is to combine the best aspects from existing commutativity concepts. From separability-based commutativity we incorporate the expressive power to reason about any two (or more) regions of code, while avoiding its practical restrictions. We adopt the idea to only consider values “live-out” at the end of candidate regions from output-based commutativity, but not limit

our analysis to invocations of a single function at a time. We define commutativity as a binary relation over regions of code and only require equality of observable state leaving the candidate code regions (i.e., live-out and live-through sets and values). Hence, *liveness-based commutativity* does not require exact matches for the contents of intermediate, dead variables. Or, in other words, temporary results from intermediate computations are not required to contain equal values with respect to sequential execution.

This definition of commutativity allows us to build upon well-established compiler techniques such as liveness analysis (live-in, live-through and live-out variables) for capturing the active elements of computations, and Single Entry Single Exit (SESE) regions, for selecting candidate regions [177, 60, 2]. Moreover, by utilizing basic compiler concepts, it is less tied to specific implementation details, while still allowing the use of other analyses and information, if profitable, and adjusting the granularity of the candidate SESE regions.

Definition 4.2.1 (Liveness-based Commutativity of Regions). Two SESE regions R_1 and R_2 are **commutative** iff:

1. (a) $R_1 \neq R_2$ are canonical SESE regions both contained in the same maximal SESE region, and $\hat{R}$ 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 the set of live-outs of $\hat{R}$ are the same for any execution order of R_1 and R_2 ; **or**
 (b) $R_1 = R_2$ (i.e., “ 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 $\hat{R}$ is the smallest such containing region, and all variables and their values in the live-out set of $\hat{R}$ are the same for all executions of R_1 .
2. The values in the live-through set of R_1 and live-through set of R_2 are unaffected by the execution order of R_1 and R_2 .

Note that the definition does not make any claims about control flow within a region, thus any SESE region may contain one or more function calls, and we do not require for commutative regions to be immediate control flow predecessor/successor pairs. In fact, the definition is general enough that it does not necessitate any other relations among statements in a region (e.g., read-write constraints or affine accesses), therefore giving us flexibility in implementation, as mentioned earlier in this section. If a function call

has side effects that are not commutative, then this will impact its set of live values, thus being detected with respect to the above definition. In this thesis, we ignore functions with I/O side effects (e.g., `printf`) as this has been the focus of prior work [84], where I/O related dependences are broken and extensive use of I/O buffering in special helper threads is used.

Loops are the focal point of our empirical evaluation in this thesis, since programs usually spend the majority of their execution time in such code constructs. In the following section, we refine the concept of commutativity presented here for loops and related to clause 1b from Definition 4.2.1. Intuitively, this means that we can treat the body of loop as a SESE region and test its intra-iteration invocations for commutativity.

Moreover, we seek to examine possible implementations of a practical *liveness-based commutativity* for loop parallelization, thus we examine potential prototype implementations by composing static and dynamic analyses and information. Lastly, commutativity based on liveness was previously considered as part of a formal characterization of parallel algorithmic skeletons [266]. We return to this issue in Chapter 6, after having concluded our evaluation of potential implementations in this and the next chapter, and propose a new methodology for the challenge of finding instances of potential parallel algorithmic skeletons.

4.3 Liveness-Based Loop Commutativity

The notion of commutativity has a natural interpretation for loops: a loop is commutative if rearranging its iterations preserves the semantics of the original program. In other words, in the spirit of the Item 1b, we can treat the body of a loop as a SESE region. Figure 4.1 gives an example of how a simple loop would be considered for commutativity based on different execution orders of its body.

This section introduces and formalizes the notion of *loop commutativity* both for the specific case (single executions) and for the universal case (all possible executions), and refines it using liveness information as described in the previous section.

We denote a given loop L with an original iteration space I (a sequence of iterations) as L_I , and the memory state resulting from an execution of that loop over a specific input x as $L_I(x)$. Loop L_I is *commutative* w.r.t. a given input x and a permutation π of I iff executing $L_{\pi(I)}$ on x produces the same output:

$$L_I(x) = L_{\pi(I)}(x), \quad (4.1)$$

```

1  for (i = 0;
2      i < N;
3      ++i) {
4      sum += array[i];
5  }

```

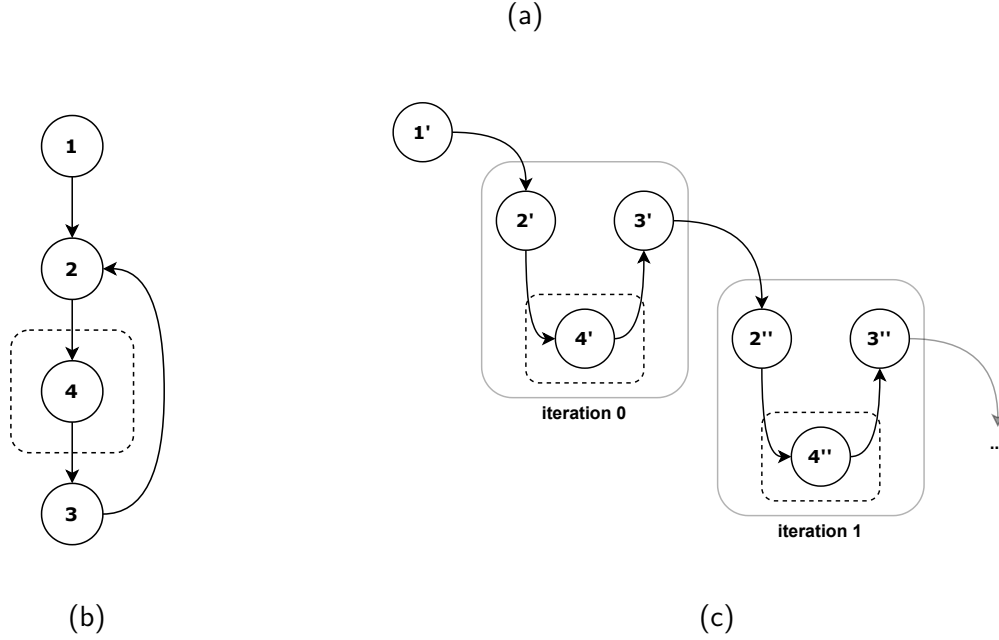


Figure 4.1: A small loop (top) and its corresponding body as a SESE region (dashed rectangles) when considered statically (bottom left) and during execution (bottom right). The loop initialization, continuation test and update step components are separated to multiple lines in the source code to help clearly identify the flow of control.

and *universally commutative* iff it is *commutative* for all possible inputs of L and permutations of I :

$$L_I(x) = L_{\pi(I)}(x) \quad \forall x \forall \pi. \quad (4.2)$$

As discussed in the previous section, requiring the memory state resulting from the execution of the original and the permuted loop to be identical is overly conservative, and can be relaxed by limiting the comparison to the variables that are live-out of the loop. This observation gives rise to the notion of *liveness-based loop commutativity*.

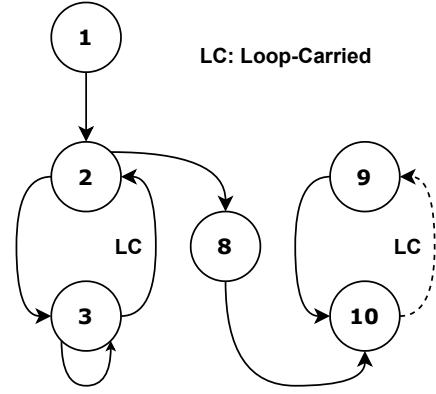
We denote the live-out set of variables of a loop L as $\text{live-out}(L)$, and the comparison of a subset V of the variables in two loop executions $L(x)$, $L(y)$ as $L(x) \stackrel{\text{live-out}(L)}{=} L(y)$. A loop L_I is *live-commutative* w.r.t. a given input x and a permutation π of I iff it is commutative w.r.t. $\stackrel{\text{live-out}(L)}{=}$. *Universal live-commutativity* is similarly defined. Unless

```

1  for (i = 0;
2      i < N;
3      ++i)
4  {
5      // these next 3 instructions
6      // essentially perform:
7      // sum += array[i];
8      tmp1 = array[i];
9      tmp2 = sum;
10     sum = tmp1 + tmp2;
11 }

```

(a)



(b)

Figure 4.2: A simple loop performing a reduction operation over an array (left) and its corresponding DDG (right). Each graph node’s number corresponds to the statement at that source line number. The loop reduction is separated to multiple intermediate instructions in the source code to help clearly identify the memory operations. The dashed edge represents the cross-iteration *parallelization-inhibiting* dependence.

otherwise stated, the rest of the thesis refers to *live-commutativity* as just *commutativity*.

Different loop commutativity analyses with varying scalability-precision trade-offs can be naturally derived from the above definitions. Universal commutativity of a loop L can be examined statically by making the input and permutation domains symbolic and proving that no combination of input and permutation leads to different values for the live-out variables of L .

4.4 Motivating Examples

Figure 4.2(a) presents a simple scalar reduction over the elements of an array written in C. Briefly, a reduction operation combines every element in a collection into a single element using an associative *combiner* function. The combiner function in this code example is the addition operation. Reductions have been long studied in the literature [133, 87] and it is known that the associativity of the reduction operator enables different execution orderings. The parallelization of a reduction requires the provision of buffers to store the intermediate results from the constituent threads that perform the partial computations over subsets of the input. This is typically handled with a privatization transformation (Section 2.4.6) which makes sure that isolated temporary

```

1  for (i = 0; i < N; ++i) {
2      // conditional branch creates
3      // control dependence
4      // on reduction variable
5      if(max < array[i])
6          max = array[i];
7  }

```

Figure 4.3: A simple loop finding the maximum value in an array.

storage is supplied to each thread.

However, detecting reductions is a challenging task for parallelizing compilers. The main issue lies with the dependence analysis that compilers use to model program statement relations. In our example in Figure 4.2(a) there is a loop-carried dependence due to the *read* occurring after the *write* (i.e., RAW dependence) at the reduction variable *sum*. This is highlighted by the dashed dependence arrow of the DDG of the same loop in Figure 4.2(b).

Compilers such as INTEL ICC or the LLVM polyhedral compilation framework POLLY fail to detect this simple code as parallelizable. In most cases, compiler writers are handling reductions by providing a tailored custom approach (e.g., pattern matching [205]) or extending their representation model with a narrow definition (e.g., “reduction-like” computations [72]). [72] extends POLLY by defining reductions based on computation properties (e.g., operator associativity) and chains of dependences relations among instructions in a candidate region. This permits POLLY to describe the example of Figure 4.2(a) as an affine relation and parallelize it.

Despite such extensions and additions, detection of reductions remains fragile and newer tools opt to skip it [176]. Consider the example in Figure 4.3 that computes the maximum of a list of integers in an array. The *max* operation (often provided as a built-in or primitive operation) is associative and can, obviously, also be parallelized. However, when written imperatively, it challenges parallelization based on dependence analysis since this form adds a control dependence to the reduction variable due to its participation in the *if* condition.

On the other hand, using commutativity based on liveness enables us to discover parallelism that escapes traditional dependence analysis. Based on the definitions we presented in the previous section, we model and capture the live-in and live-out values of a candidate loop and determine if the contents of the latter (i.e., the live-outs) remain

the same after we permute the loop's iteration order. The key idea is that if the live-outs remain unaffected, in other words, the loop is found to be commutative, then we can execute it in parallel. Regions are commutative and can be executed in parallel, subject to synchronization of accesses to shared variables.

For the examples in Figures 4.2 and 4.3, this means that we can direct our attention to the variables `sum` and `max` that hold the useful results from the computations of these loops. We avoid the complications that arise from the overly restrictive program view of dependence and avoid the issue of explicitly expecting or modelling program properties at the instruction-level (i.e., associativity).

4.5 Operation of Symbolic Execution

In this section, we provide an example of the steps followed during the symbolic execution of a program. We use KLEE [44] as our symbolic execution engine. KLEE's architecture and design goals were presented in Chapter 3. We also gave a synopsis of the symbolic execution literature related to this thesis in Chapter 2. Briefly, symbolic execution allows the execution of multiple paths using symbolic input (i.e., a class of values) while maintaining a symbolic state. The symbolic state of a program consists of symbolic values bound to variables and constraints on them. Symbolic program execution starts with no constraints and incrementally generates them upon branches when their decisions are dependent on symbolic input.

Consider again the *max* reduction motivating example from Figure 4.3 and assume that it operates on a small, statically known (i.e., concrete) array of values. Initially, we represent the contents of variable `max` with the symbolic value `x`. Then, executing the next program statement, such as the `if` condition `if (max < 6)` for state *A*, will create two new states depending on whether the condition is true or not (state *B* and *E* respectively in Figure 4.4). This will also update the maintained path constraints *C*, program store *S* and next program statement *P* to be executed by the symbolic engine. Whenever a branch is reached, the engine checks if the outcome on either side is feasible before proceeding to explore it.

Note that each execution path has its own path constraint expression at the end (i.e., upon loop exit). Symbolic execution is able to succinctly represent input classes and, together with the initial input, it can provide a transfer function that determines the current state. Another benefit of this approach is that it can handle a mix of both concrete and symbolic values and move from one to the other depending on the path

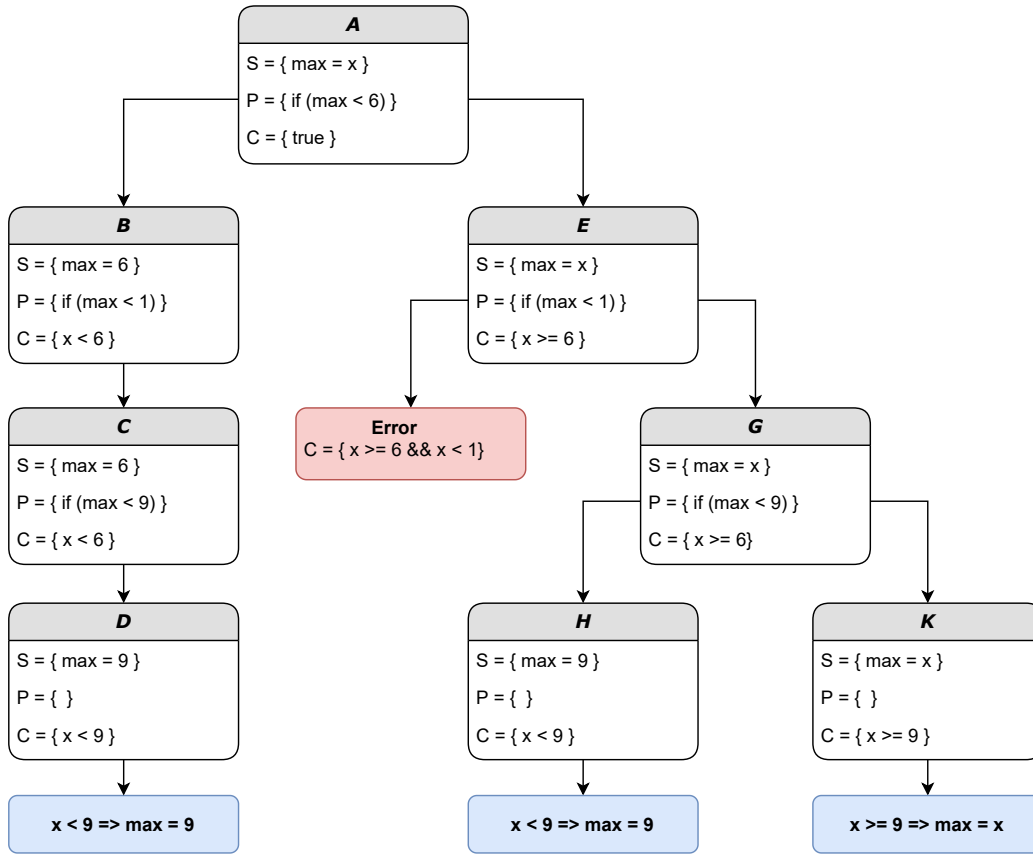


Figure 4.4: Symbolic execution tree of loop from Figure 4.3. Each rectangle represents an execution state consisting of the current value store S , the next program statement to execute P and the path constraints C . Each tree level (except the top) corresponds to a loop iteration invocation for the input array $[6, 1, 9]$.

exploration and available values. For example, in Figure 4.4, the value of variable `max` is set to a concrete value for paths A to D and A to H , while it remains symbolic at the end of the path from A to K . Moreover, additional constraints can be inserted in the program which can influence the construction of path constraints (e.g., the programmer might know that a certain variable can never acquire a certain value down a specific path). Lastly, a symbolic execution engine allows the evaluation of assertions, essentially discovering classes of inputs that might cause an assertion to fail at a program point, thereby avoiding the pitfalls of testing with random or hand-picked values.

KLEE is able to evaluate assertions by interpreting the standard C library call to the `assert` function. It also offers intrinsics (special functions) to offer functionality that is useful in the context of symbolic execution. We make use of the `klee_make_symbolic`

function, which declares a variable as symbolic, and the `klee_assume`, which takes a condition as an argument and is used to constrain the values of symbolic variables down the path where it is inserted.

```

1  int reduce_max(int reverse_iteration_order, int *array, int size) {
2      unsigned i = 0;
3      int max = INT_MIN;
4
5      if (!reverse_iteration_order) {
6          /* original iteration order */
7          for (i = 0; i < size; ++i) {
8              if (max < array[i])
9                  max = array[i];
10         }
11     } else {
12         /* reverse iteration order */
13         for (i = size - 1; i >= 0; --i) {
14             if (max < array[i])
15                 max = array[i];
16         }
17     }
18
19     return max;
20 }
21
22 int main() {
23     int array[32];
24     int max1 = INT_MIN, max2 = INT_MIN;
25
26     klee_make_symbolic(array, sizeof(array), "array");
27
28     max1 = reduce_max(0, array, sizeof(array));
29     max2 = reduce_max(1, array, sizeof(array));
30
31     /* crosschecking */
32     assert(max1 == max2);
33
34     return 0;
35 }

```

Figure 4.5: Symbolic crosschecking example of the reduction loop from Figure 4.3.

4.6 Symbolic Crosschecking of Commutativity

In Section 4.3 we described our notion of *liveness-based commutativity* for loops. We consider a loop commutative if, after reordering its iterations, the values of its live-out variables remain the same. In this section, we briefly describe how we utilize symbolic execution to allow us to combine a mix of symbolic and, where available, concrete values, and exploit its systematic exploration of execution paths for arbitrary sections of code to test *liveness-based commutativity* conditions.

Our approach, inspired by [59], aims to execute a candidate loop using various iteration orders and verify that the resulting contents of the live-out variables from these reorderings are equivalent, i.e., crosscheck their results. Using our motivational *max* reduction example from Figure 4.3, a possible implementation of this crosschecking technique is presented in Figure 4.5.

The reduction operation has been wrapped in the function named `reduce_max` and duplicated under each branch of the `if/else` conditional. This function returns an integer value (i.e., the result of the reduction) and accepts three arguments: i) the input integer array over which reduction is performed, ii) its size iii) `reverse_iterator_order` that controls the order of iteration.

The use of the `main` function is similar to that of a setup routine, often found in test harnesses. The `main` starts with the declaration the input (array) and output variables (`max1`, `max2`). The input is declared as *symbolic* (line 26) using KLEE’s intrinsic call. Then, we call `reduce_max` twice in order to execute with the original iteration order (line 28) and its reverse (line 29). Lastly, we compare the outcome of the two execution orders (i.e., crosscheck) with an assertion (line 32) expecting their equality.

While this simple example captures the essence of our approach, it does not cover concerns relating to aspects such as:

- i) detecting and bounding the input values (i.e., the array variable),
- ii) exploring paths, program coverage and scalability issues, and
- iii) complex program features (e.g., global variables, pointers and aliasing, etc.).

Lastly, regarding our choice of crosschecking commutativity conditions against the reverse iteration order of a loop, this is in response to the scalability concerns discussed in Section 4.3 and it also relates with item ii) above. We discuss and explore these concerns in the remainder of this chapter.

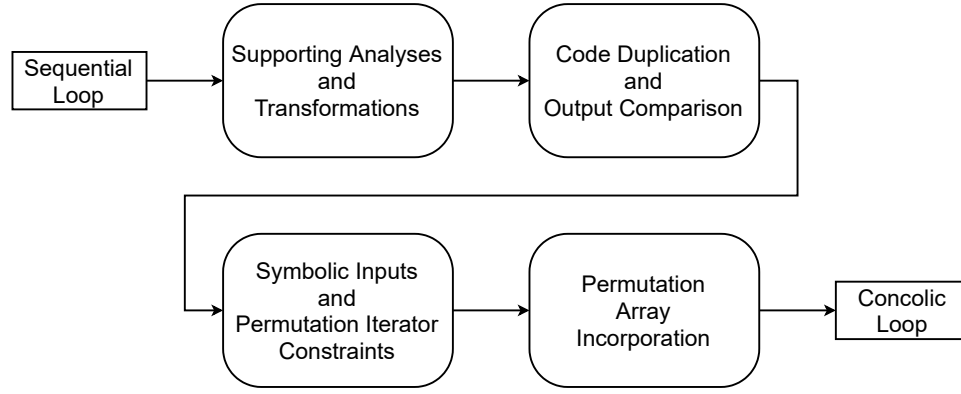


Figure 4.6: Steps of our *liveness-based commutativity* technique using symbolic execution.

4.7 Symbolic Crosschecking Technique for Liveness-based Commutativity

The outline of our technique consists of the steps depicted in Figure 4.6. We also provide source code examples of the transformations happening in each step to ease comprehension in Figures 4.7 and 4.8. We conclude the section by describing and addressing issues and limitations that directly affect our technique.

4.7.1 Approach

Our approach begins by applying liveness analysis on our candidate loop as presented in Figure 4.7(a). During this step, we also perform a *generalized iterator recognition* analysis in order to identify the loop statements that constitute its iterator [166]. Once the live-out values are determined, the loop is duplicated and placed after the original loop (line 10 in Figure 4.7(b)). The objective is to perform all the required transformations over this duplicated loop which should implement the iteration reordering steps for commutativity. Since we have the liveness information for both loops, we make the necessary transformations to save the state of input and output variables for each loop variable. Then, we add the assertions about the corresponding live-out variables of each loop which should be satisfied after symbolic execution by KLEE.

Figure 4.7(a) depicts the original, unmodified candidate loop. This loop is extracted into a separate function in order to sidestep the issue of limited code coverage associated

<pre> 1 2 3 for (i = 0; i < N; ++i) 4 sum += array[i]; 5 6 7 8 9 10 11 12 13 14 15 16 </pre>	<pre> 1 +int sum_shadow = sum; 2 3 for (size_t i = 0; i < N; ++i) 4 sum += array[i]; 5 6 +int sum_ref = sum; 7 +sum = sum_shadow; 8 9 10 +for (size_t i = 0; i < N; ++i) 11 + sum += array[i]; 12 13 14 +int sum_perm = sum; 15 16 +assert(sum_ref == sum_perm); </pre>
---	---

(a) Original loop from Figure 4.2(a). (b) Loop duplication and loop output comparison assertion.

Figure 4.7: Code output of the first two steps of our approach. The **+** sign indicates a line added when compared to the code from the previous step.

with symbolic execution tools. However, for the description of our approach in this section, we omit this step and assume the transformation has already taken place. We discuss the decision to extract candidate loops in separate functions and concerns about code coverage in Section 4.7.2.

Figure 4.7(b) presents the final code at the end of the first transformation step. The duplicated loop is at line 10. The variables `sum_shadow`, `sum_ref` and `sum_perm` and the associated assignments represent state saving before and after the execution of each loop. They are the source-level approximation of the steps needed in the SSA form of the LLVM IR. The variables `sum_ref` and `sum_perm` represent the result of the loop from the programmer-intended and the permuted iteration order, respectively, which are compared at the final assertion statement at line 16. The variable `sum_shadow` stores the initial value of `sum` to restore it for the duplicated loop, so that both loops have the same initial input values.

The last two steps of our approach are presented in Figure 4.8. We proceed by introducing the code required to perform the permutation of iterations for the duplicated loop (line 26). We create a permutation array (line 13) which is able to hold the

iteration space of the loop under test. For this we utilize the information provided by the *generalized iterator recognition* analysis performed at the first step of our approach. We then transform the iteration control statements of the duplicated loop (marked with the ★ sign) to use the contents of the permutation array to index each iteration.

The output of the last transformation step is presented in Figure 4.8(b). First, the loop input and loop iterator variables, identified in the first step of our approach, are made symbolic in lines 3 and 15, respectively. The array sizes are set to an arbitrary upper bound since KLEE is unable to use symbolic sizes for memory allocations. We further discuss this limitation in Section 4.7.3. Then, we apply constraints to the permutation array between lines 19 and 24. The first `klee_assume` statement (line 20) restricts each value in the permutation array within the iteration domain by requiring it to be less than the upper bound. The second `klee_assume` statement (line 22) adds the constraint which enforces the ordering of the array contents to be a permutation (i.e., pairwise different).

4.7.2 Dealing with Limited Code Coverage

Path selection and the strategy employed by a technique or a tool is a core operation in symbolic execution (Chapter 2). These strategies prioritize the most promising paths to be explored first, typically based on various heuristics and criteria that are tailored to the purposes of a particular symbolic execution engine.

KLEE aims at maximizing code coverage (i.e., the number of LOC executed) using distance metrics and assigning probabilities to branches based on their arity, path length, times visited, etc. Missed paths result in possible erroneous conclusions, since a symbolic execution engine cannot reason about paths which it does not process at all [20].

In our approach, we take inspiration from the analysis used in [184] (although it is aimed at detecting security vulnerabilities), which takes a compositional approach examining first at a lower granularity (function level). It also resembles the dual implementations (scalar and SIMD versions) subject to crosschecking used in the technique described in [59]. We avoid starting symbolic execution from a program's default entry point (i.e., main function) and extract the loop nest under test into a separate function, while still embedding it into the original application. This allows us to:

- i) use all the global context used in the loop and any other potential function calls,

<pre> 1 int sum_shadow = sum; 2 3 4 5 6 7 for (size_t i = 0; i < N; ++i) 8 sum += array[i]; 9 10 int sum_ref = sum; 11 sum = sum_shadow; 12 13 +int p[N]; 14 15 16 17 18 19 20 21 22 23 24 25 26 ★for (int k = 0; k < N; ++k) { 27 + int i = p[k]; 28 sum += array[i]; 29 } 30 31 int sum_perm = sum; 32 33 assert(sum_ref == sum_perm); </pre>	<pre> 1 int sum_shadow = sum; 2 3 +klee_make_symbolic(&array, 4 sizeof(array), 5 "a"); 6 7 for (size_t i = 0; i < N; ++i) 8 sum += array[i]; 9 10 int sum_ref = sum; 11 sum = sum_shadow; 12 13 int p[N]; 14 15 +klee_make_symbolic(&p, 16 sizeof(p), 17 "p"); 18 19 +for (int i = 0; i < N; i++) { 20 + klee_assume(p[i] < N); 21 + for (int j = i + 1; j < N; j++) { 22 + klee_assume(p[i] != p[j]); 23 + } 24 +} 25 26 for (int k = 0; k < N; ++k) { 27 int i = p[k]; 28 sum += array[i]; 29 } 30 31 int sum_perm = sum; 32 33 assert(sum_ref == sum_perm); </pre>
--	--

(a) Insertion of permutation array.

(b) Marking of symbolic inputs and application permutation iterator constraints.

Figure 4.8: Code output of the last two steps of our approach. The **+** and **★** signs indicate, respectively, a line added or modified when compared to the code from the previous step.

```
1  int N;  
2  klee_make_symbolic(&N, sizeof(N), "N");  
3  int array[N];
```

Figure 4.9: Allocation of an array using a symbolic value for its size.

and

- ii) specifically select this extracted function as our program entry point (via KLEE's option `-entry-point=FUNC_NAME`).

4.7.3 Symbolic Sizes for Memory Allocations

Figure 4.9 presents a small code fragment in which the array's size `N` is declared as a symbolic value. As mentioned previously in Section 4.7.1, KLEE is not able to handle memory allocations (i.e., both dynamic via `malloc` or static) that use a symbolic value for their size specification. In these cases KLEE resorts to concretization of the symbolic variable representing the allocation size, asking the constraint solver to provide an example initial value, which it subsequently attempts to iteratively reduce by issuing additional constraint queries. Hence, further symbolic execution uses the concrete value for any following path constraints. In code coverage terms, this avoids reporting errors that would not normally occur (i.e., false positives), but is liable to potentially skip certain paths.

In our case, we explicitly select a size for any arrays that we need to represent, utilizing information, where available, from LLVM's implementation of recurrence chains for integer induction variables [16] (i.e., scalar evolution). This compromise might cause issues with the detection of loop parallelization when there is a data dependence relation between a pair of memory locations and the dependence distance is greater than the selected array allocation size. We have not been able to identify such codes in the NPB suite, but a possible solution would be to perform the symbolic execution runs multiple times with varying allocation sizes. We discuss challenges posed by PLDS in Section 4.8.

There has been work on designing a symbolic-size memory allocation model and implementing it in KLEE [233], however, the results have shown marginal improvements due to path explosion problems resulting from these allocations. Other recent work focuses on enhancing the modelling of dynamic memory allocations and reasoning about them [135, 181] or improving constraint solving for array operations that are

indexed by complex symbolic expressions [192].

4.7.4 Handling Floating-point Operations

The representation of floating-point operations in computers are finite approximations of real numbers. Typically, these follow scientific format notations and attempt to provide trade-offs between precision, range and storage space. Moreover, they handle other aspects, such the representation of special number classes (e.g., zero, infinity, Not a Number (NaN), etc.), rounding modes and exceptions occurring upon invalid computations (e.g., division by zero). The details of various formats are usually incorporated into standards, such as the IEEE-754 standard [119]. These standards are implemented by language implementers and processor manufacturers alike in order to facilitate interoperability, conformance and reproducibility of results.

Hence, handling code that makes heavy use of floating-point arithmetic is strenuous and error-prone. This is because common assumptions about real numbers do not hold for floating-point (e.g., the associativity property), leading to subtle bugs that may appear only under specific input values. These challenges extend to constraint solvers, which symbolic execution engines rely upon, since it is extremely difficult to model accurately these semantics in them. As a result, there are currently no such stable constraint solvers available [59], with research in such a direction being only preliminary [214, 162]. KLEE-FLOAT [162] is prototype, implemented as KLEE extension, that provides initial support from floating-point arithmetic. However, as of late 2020, it has not yet been incorporated into KLEE’s main development branch, because of concerns on its performance impact, due to its extensive modifications to the internal expression representation [42].

Nevertheless, we are interested in examining real-world programs coming from various application domains (e.g., HPC) which make extensive use of floating-point arithmetic. We have already discussed the approach taken by [59] in Chapter 2. This does not attempt to solve the issue, but makes specific assumptions on the order of operations. In addition, it requires extensions to the symbolic execution engine and involves the programmer.

For our purposes, we sidestep the issue by transforming the selected code regions, converting floating-point types to integers. This allows us to still utilize the ability of constraint solvers to reason about ranges of values, while avoiding overly limiting strategies that typical symbolic execution engines employ, such as selecting a single

random concrete value to carry on path exploration with. Given that this process risks introducing various floating-point errors (e.g., division by zero exceptions), we involve the programmer to make the required adjustments wherever possible. Note that in code presented in the rest of the chapter, taken from benchmark suites, we show the original types for simplicity.

4.8 Case Studies

In this section, we discuss the challenges of the crosschecking approach of our methodology using symbolic execution. We examine characteristic loops taken from real-world programs of the NPB and OLDEN suites. These loops have been chosen due to their known potential to accelerate the execution the programs which contain them when parallelized.

4.8.1 Pointer-based Loops

4.8.1.1 Loop Conditions and Branching

Consider the loop presented in Figure 4.10 taken from the *em3d* program of the OLDEN benchmark suite. We have described the high-level purpose of each benchmark in Chapter 3. This loop iterates over a singly linked list of nodes and updates their electric and magnetic field values (i.e., `localnode->value`, `value`, `cur_value` and `other_value` variables in Figure 4.10). The loop moves to the next node (line 35) and its condition (line 4) terminates the iteration when the pointer to the next node is NULL.

There are several difficulties with this code which center around the operation of the loop and the representation of this dynamic (i.e., heap-allocated) data structure. Symbolically representing the loop exit conditions is one of the most common causes which trigger path explosion in symbolic execution tools. The objective of symbolic execution is to maximize coverage and explore as many program paths as possible. Hence, for a loop condition that involves symbolic values, a symbolic execution engine creates two additional paths (i.e., one for when the condition is true and one when it is false). This process continues for each subsequent loop iteration, thus the execution engine faces an exponential blow up of path constraints and states (i.e., execution time and memory).

Even if we set the number of iterations of the loop in Figure 4.10 to some specific concrete value, this code fragment still faces path explosion challenges. This is because

```

1  int i;
2  node_t *nodelist;
3
4  for (;nodelist;) {
5      int from_count;
6      double *other_value, cur_value,coeff, value;
7
8      cur_value = *nodelist->value;
9      from_count = nodelist->from_count - 1;
10
11     for (i = 0; i < from_count; i += 2) {
12         other_value = nodelist->from_values[i];
13         coeff = nodelist->coeffs[i];
14
15         if (other_value)
16             value = *other_value;
17         else
18             value = 0;
19
20         cur_value -= coeff * value;
21         other_value = nodelist->from_values[i + 1];
22         coeff = nodelist->coeffs[i];
23         ...
24         cur_value -= coeff * value;
25     } // inner for loop end
26
27     if (i == from_count) {
28         other_value = nodelist->from_values[i];
29         coeff = nodelist->coeffs[i];
30         ...
31         cur_value -= coeff * value;
32     }
33
34     *nodelist->value = cur_value;
35     nodelist = nodelist->next;
36 } // outer for loop end

```

Figure 4.10: Simplified code of the hottest loop in the *em3d* program from the OLDEN benchmark suite [47]. The loop iterates over a singly linked list of objects in 3D space and updates their electric and magnetic field values.

it also contains an inner loop (line 11) whose value we also want to represent symbolically, since it is also an input value (i.e., `from_count` specifies the number of nodes that affect the electromagnetic forces of the current node). For the same reasons, the situation is also worsened by the use of branch constructs such as `if/else` statements. The loop of Figure 4.10 contains four `if/else` statements dependent on symbolic values (two of them have been obviated for brevity). More importantly, all these issues are further exacerbated by our methodology's requirement to permute iterations for symbolically crosschecking commutativity (Section 4.7).

4.8.1.2 Memory Allocation and Initialization

Apart from the challenges due to control flow constructs, symbolic execution suffers from another source of path explosion which deals with the representation of a program's memory. This is because representing a memory allocation symbolically (i.e., symbolic pointers), will prompt the symbolic execution engine to explore all potential paths: (i) either by forking upon any possible symbolic memory address a pointer might be assigned, leading to path explosion, or (ii) by utilizing `if/else`-style expressions to describe the path constraints, stressing the constraint solver with very large and complicated formulas. As discussed in Section 4.7.3, a common simple alternative, trading completeness for performance, which is adopted by KLEE, is to concretize the pointer to a value that satisfies the current path constraint based on a strategy (e.g., the maximum heap value the pointer may acquire). While this might suffice for specific programs (i.e., minimal control flow), it is problematic for programs that make heavy use of PLDSs. Considering Figure 4.10 again, the output of the program is dependent on loading values via pointers (e.g., lines 12 and 16), since this program propagates electromagnetic force values among neighbouring nodes.

An alternative approach to the allocation and initialization of PLDSs would be to involve the user and resolve them manually. This could alleviate the aforementioned problems with the use of pointers and heap allocations with programs that make basic use of these data structures (e.g., *ising* [269], *tree* from OLDEN [47]). For instance, Figure 4.11 shows the simple allocation pattern employed by *ising* (line 9), a small (~200 LOC) kernel which also performs particle physics simulation (Chapter 3). In this code, it is straightforward to manually rewrite the single dynamic allocation using a stack-based alternative. However, in practice, the allocation and initialization patterns of PLDSs can be complex and require domain knowledge, since these data structures provide a flexible representation and lend themselves naturally to particular domains.

```

1  struct node {
2      uint32_t spins[ny];
3      struct node *next;
4  };
5  ...
6  int main(int argc, char *argv[]) {
7      ...
8      // allocate nodes
9      nodes = (struct node *)malloc(nodes_num * sizeof(struct node));
10     if (!nodes) { return -1; }
11     ...
12     // set up the initial linked list
13     for (i = 0; i < nodes_num - 1; i++) {
14         for (j = 0; j < ny; j++)
15             nodes[i].spins[j] = (rand() < (theta * RAND_MAX)) ? 1 : 0;
16         nodes[i].next = &nodes[i + 1];
17     }
18     ...

```

Figure 4.11: A simplified excerpt from the *ising* [269] kernel benchmark showing it dynamic allocation (line 9) and subsequent initialization (lines 13 to 17).

Figure 4.12 depicts a limited view of the irregular bipartite graph data structure allocated and configured by *em3d*. We notice that, although the traversal was iterating over a linked list, the actual data structure and relationships between nodes are more intricate. This can also be the case for initialization, as seen between lines Section 4.8.1.2 and Section 4.8.1.2 of the *ising* kernel in Figure 4.11, where a specific algorithm, as required by the simulation model, is used to set the initial values of the nodes. Lastly, as discussed in Section 4.7.3, selecting specific concrete values for input size and inputs can potentially misrepresent the data flow in a candidate loop, leading to false positives. In general, the extent of the required user changes can range broadly from a local function scope (e.g., a specific code fragment implementing an array access pattern) to the whole program.

4.8.1.3 Pointer Analysis

Another challenge associated with the allocation of the required storage for our cross-checking functions, is the efficacy of the pointer analysis which aims to identify and disambiguate pointer accesses and heap storage locations. More importantly for our

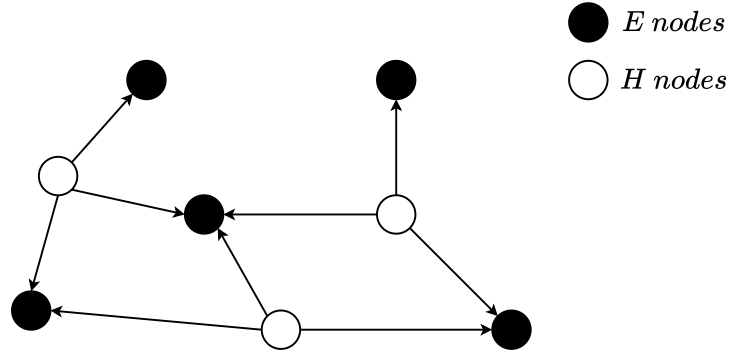


Figure 4.12: A small instance of the main graph data structure in the *em3d* program from the OLDEN benchmark suite [47] displaying the complex relationship between nodes (based on [48]). *E* and *H* nodes represent electric and magnetic field values, respectively. For simplicity, only edges from *H* nodes are shown.

implementation of *liveness-based commutativity*, it reduces the ability of our analysis to discover and distinguish live-in and live-out values for the candidate loops and, consequently, automatically select the correct symbolic inputs that could affect their outcome. As discussed in Chapter 2, performing alias analysis in a static context is difficult and limiting, which is compounded by:

- i) the use of global variables and extensive pointer usage in PLDS-based applications, and
- ii) the loss of contextual information that our technique suffers by extracting the candidate loops into separate functions to avoid issues with code coverage (Section 4.7.2).

For the former point, recent work on symbolic execution [135] has utilized state-of-the-art alias analysis [239], but required extensive integration in the symbolic execution engine of KLEE. Regarding our loop extraction, we could perform the alias analysis on the original loop and propagate that information to its extracted version for crosschecking. Nevertheless, it is not clear how to compensate for the potential differences (i.e., varying path constraints and states) that could arise from different call path explorations orders when symbolically executing the original loop.

4.8.1.4 Results on PLDS-based loops

Table 4.1: Results from running KLEE on PLDS-based loops with two problem sizes (N) and two different solver timeouts (in minutes). The ✓ and ✗ signs denote successful and failed (i.e., execution exceeded timeout value) completion of each test case, respectively.

Loop	Program	Problem Size			
		N = 300		N = 150	
		Timeout		Timeout	
		30mins	60mins	30mins	60mins
compute_nodes	emd3d	✗	✗	✗	✗
BlueRule	mst	✗	✗	✗	✗
walksub	bh	✗	✗	✗	✗
perimeter	perimeter	✗	✗	✗	✗
treeadd	treeadd	✗	✗	✗	✗
ising	main	✗	✗	✗	✗
spmatmat	main	✗	✗	✗	✗
INTERF	water-spatial	✗	✗	✗	✗

We proceed to manually amend the issues with memory allocations, selecting symbolic inputs and discerning liveness information, as identified in Section 4.8.1.2, for the extracted candidate loops from the OLDEN benchmark suite and other sources, presented in Table 4.1. For each of the candidate loops we present four results, two different problem sizes and two different KLEE timeout values. The problem size (denoted as **N** in Table 4.1) represents the iteration trip count for each loop. The choice of **N** was based on a compromise between the actual input sizes for each benchmark and various timeouts used in the evaluation of symbolic execution engines from recent literature [279]. Loops can incur the exploration of numerous paths and the decision of which loop body variables to introduce as symbolic can affect path exploration strategies. Hence, following the practice in previous work [279, 220], we select problem size and timeout values that aim to: i) allow path exploration to detect potential cross-iteration dependences of within certain iteration distances, and ii) provide execution time for symbolic execution engines to explore the opened paths. The timeout value sets an upper time limit for KLEE and its constraint solver to finish execution, otherwise symbolic execution is aborted. The selected timeout values are chosen based on suggestions in [279] where the performance of several execution engines (including KLEE) is

```

1  ...
2  for (i = 1; i <= grid_points[0]-2; i++) {
3      for (j = 0; j <= grid_points[1]-1; j++) {
4          ru1 = c3c4*rho_i[k][j][i];
5          cv[j] = vs[k][j][i];
6          rhoq[j] = max(max(dy3+con43*ru1, dy5+c1c5*ru1), max(dymax+ru1, dy1));
7      }
8
9      for (j = 1; j <= grid_points[1]-2; j++) {
10         lhs[j][i][0] = 0.0;
11         lhs[j][i][1] = -dtt2 * cv[j-1] - dtt1 * rhoq[j-1];
12         lhs[j][i][2] = 1.0 + c2dtt1 * rhoq[j];
13         lhs[j][i][3] = dtt2 * cv[j+1] - dtt1 * rhoq[j+1];
14         lhs[j][i][4] = 0.0;
15     }
16 }
17 ...

```

Figure 4.13: A reduced excerpt from the *SP* program of the NPB suite showing the *y_solve.c:58* (i.e., line 58 in source file *y_solve.c*) loop. This loop performs part of the operations required for matrix factorization. The upper bounds of each (outer and inner) loop are highlighted.

examined.

We note that *none* of the selected loops in Table 4.1 is able to successfully terminate in any combination of the selected execution configurations. These findings corroborate with results from other recent work on the limitations of symbolic execution [279, 252] and lead us to consider that it is currently not able to attain practical results in the aforementioned experimental settings. However, our study has allowed us to identify fundamental issues that need to be addressed before tackling the parallelization of PLDS-based loop constructs. In practice, even if the symbolic execution reaches a determination regarding the commutativity of the examined loops, we would still need to involve the user, as it is uncertain if results could be generalized over all inputs or even the full specific input set provided by the benchmarks (see Chapter 2).

4.8.2 Array-based Loops

Figure 4.13 presents a loop from the *SP* program of NPB suite. This is an inner loop

and contained in a function that performs the matrix factorization computation of a PDE solver. We notice that all the loops are following a regular `for`-style iteration idiom, commonly found in HPC and other scientific codes which deal with systems of linear algebra and PDE equations [7]. These programs make extensive use of multidimensional arrays and employ deeply nested loops to iterate over them. The loops use an integer induction variable as their iterator, and are typically *normalized* [7] (i.e., running from an index of 0 or 1 up to some upper bound, using a step increment of 1).

Despite using a simpler data structure (i.e., arrays) and loop iteration idiom, this loop also suffers from path explosion limitations, which is due to the same reasons as in the case of PLDS-based loops detailed in Section 4.8.1. In Figure 4.13, the upper bounds for all three loops in the loop nest are directly dependent on the program’s input, stored in different locations of the `grid_points` array (lines 2, 3 and 9). Lastly, notice the calls to the `max` function from the `LIBM` math library (line 6). External function calls are another source of path explosion [279], however, in this case KLEE’s system call API model implements such standard library functions since it includes a modified `UCLIBC` version [254, 44]. Nevertheless, function calls remain problematic since they potentially include further loop constructs or branching themselves which can further exacerbate the overhead of symbolic execution.

As we have already discussed when we examined PLDS-based loops, dynamic allocation is another program feature that can potentially lead to path explosion. In the case of the NPB, the arrays for all programs in the suite (except *DC*) are allocated statically as global variables. While this obviates the need to compensate with dynamic allocation, it still skips any application-specific initialization, typically carried out in the default program entry function (i.e., `main`). We have discussed the implications of avoiding the programmer-intended initialization in Section 4.8.1.

In terms of pointer analysis, the required disambiguation is less complicated than that necessary to analyze PLDS-based loops. This is also due to the extensive use of global variables for the declaration of each program’s data structures and their subsequent direct reference in all the functions where they are accessed. Thus, the use of pointers, which compounds aliasing effects, is fairly limited in the NPB programs (with the notable exception of *DC* which employs a tree-like data structure). However, considering that the suite was originally written in `Fortran`, this idiomatic style seems to be a residual trait from using that programming language and it might not necessarily represent other programs which make extensive use of array-based loops.

Table 4.2: Results from running KLEE on array-based loops from the NPB suite with two problem sizes (N) and two different solver timeouts (in minutes). The ✓ and ✗ signs denote successful and failed (i.e., execution exceeded timeout value) completion of each test case, respectively. The loop names follow a *filename:line_number* format convention.

Loop	Program	Problem Size			
		N = 300		N = 150	
		Timeout		Timeout	
		30mins	60mins	30mins	60mins
y_solve.c:397	BT	✗	✗	✗	✗
z_solve.c:151	BT	✗	✗	✗	✗
cg.c:460	CG	✗	✗	✗	✗
mg.c:550	MG	✗	✗	✗	✗
rhs.c:179	SP	✗	✗	✗	✗
y_solve.c:58	SP	✗	✗	✗	✗

4.8.2.1 Results on Array-based loops

Similarly to the PLDS-based loop in Section 4.8.1, we manually amend the issues identified, for the extracted candidate loops from the NPB suite, presented in Table 4.2. For each of the candidate loops we present four results, two different problem allocation sizes and two different KLEE timeout values. Again, we observe that *none* of the selected loops in Table 4.2 is able to successfully terminate in any combination of the selected execution configurations. Despite using simpler programming constructs than programs containing PLDSs, the loops in the NPB suite lead to the same challenges for symbolic execution.

Considering the results from both loop iteration idioms (i.e., over arrays and PLDSs), we notice that program properties emerge from different uses of programming constructs which can have adverse effects and generate varying tensions on symbolic execution and associated analyses. The use of profiling, which can sample specific program properties and information during (concrete) execution, is a potential avenue that could alleviate the issues identified in this section and provide a more widely applicable approach.

4.9 Summary and Conclusions

In this chapter we have presented our notion of commutativity based on liveness information for parallelization. This definition of commutativity is in turn built upon well-understood concepts from compiler theory: liveness and SESE regions. We have also introduced an adaptation of this concept for loop constructs, since often these are code regions where applications spend most of their execution time. We then sought to explore expressing our *liveness-based commutativity* in a practical analysis. Consequently, we apply a method based on symbolic execution to determine commutativity conditions. Symbolic execution allows expressing inputs in a condensed manner, while also being able to combine concrete values, in order to generate and test conditions on program properties. We presented a procedure that integrates *liveness-based commutativity* on top of a symbolic execution engine and explored its efficacy on loops taken from the NPB suite, the OLDEN benchmark suite and other sources. We have identified several non-trivial challenges and obstacles that limit the practical use of such an approach on real-world applications. We discuss possible ways to mitigate some of these shortcomings in Chapter 7.

Our findings imply that despite the recent progress of symbolic execution techniques, such an analysis is still unlikely to handle realistic problems due to a combinatorial explosion of possible permutations and inputs. Taking these lessons forward, we seek to alleviate some of these shortcomings by leveraging information extracted from actual (i.e., with fully concrete inputs) program execution. One such potential route is the use of profiling where a candidate program is instrumented, executed and sampled using a specific data set as input.

In the next chapter, we develop a novel analysis founded on our notion of *liveness-based commutativity* for the parallelization of loops. To this end, we utilize profiling information to test commutativity and explore the precision of the results when applying our proposed approach on complex, real-world benchmarks along with their accompanied data sets (albeit not exhaustive inputs). We also investigate the potency of the parallelism detection and mapping that it offers when compared against applications manually parallelized by domain experts.

Loop Parallelization using Dynamic Commutativity Analysis

In the previous chapter we presented the concept of *liveness-based commutativity* and explored its application in a static and limited hybrid context by abstracting the input sets, exploring the associated challenges with an underlying technique like symbolic execution. In this chapter, we investigate the application of *liveness-based commutativity* in a dynamic context during the execution of programs with user provided input sets.

We develop a novel Dynamic Commutativity Analysis (DCA) for identifying parallelizable loops by combining static analysis and profiling. We have used a wide collection of benchmarks, representing HPC applications which utilize regular array-based accesses from the NAS Parallel Benchmark suite [19] along with a wide collection irregular pointer-based codes. The results of the detection performance and multi-threaded exploitation of the discovered parallel loops is investigated and compared against both dynamic and static parallelization approaches.

This chapter starts with Section 5.1 which introduces the background for our study, continues with our motivation in Section 5.2 and reassessment of the commutativity analysis definitions from the previous chapter in Section 5.3. Section 5.4 describes DCA and Section 5.5 evaluates our approach with an empirical study. Section 5.6 summarizes and concludes.

5.1 Introduction

As discussed in the previous chapter and Chapter 2, static analyses are restricted by certain limitations when a compiler is trying to deduce if various properties of a code

section hold. The same applies when using hybrid contexts, as symbolic execution does, where features of the input are modelled as classes of symbols and thus represented in an abstract and comprehensive manner. This has proven useful in determining properties for questions relating to an application's robustness against vulnerabilities or its testing coverage.

However, for program analyses which aim to exploit latent performance, such as the automatic parallelization of legacy sequential codes, researchers have moved to the use of runtime information to supplement static analyses [14, 81, 285, 248, 160, 144, 143, 127, 262, 219, 282, 283, 237, 74, 33, 26, 205, 277, 163], presented in more detail in Chapter 2.

Using a subset of all potential inputs, these analyses are not guaranteed to be correct for every possible workload. In practice, however, it has been shown that they can be applied without sacrificing program safety [248, 160, 76]. The most common tactics to address any safety concerns are to: i) either, involve the user at various stages in order to confirm the results and provide auxiliary feedback [161, 41, 288, 248], or ii) generate and embed in the program special preconditions that can be checked during execution, where a safe execution path is chosen when these assumptions are not met [71].

In addition to the above, automatic parallelization methods have been heavily relying on dependence analysis for decades. Thus, a notable amount of effort has been channeled to improve the precision of dependence analyses techniques. However, recent work has shown that dependence analysis, even when informed with perfect profiling information, is inherently unable to identify any further latent parallelism [180].

In this chapter, our objective is to explore the use of profiling information along with the notion of commutativity that we presented in the previous chapter. More specifically we are interested in explore the following properties:

- i) the capability of *liveness-based commutativity* to discover hidden parallelism in loops when used with profiling information from specific input sets,
- ii) the efficacy of parallelism when parallelization is performed, and
- iii) the safety of the proposed transformations.

To this end, in this chapter, we develop a novel, hybrid (static and dynamic) Dynamic Commutativity Analysis (DCA) for the discovery of profitable parallelism in sequential legacy code. We apply DCA to the sequential version of the NPB suite (almost 1400 loops in non-trivial scientific applications) and a wide range of PLDS-based loops. For

<pre> 1 for (i = 0; i < N; ++i) { 2 array[i]++; 3 4 }</pre>	<pre> 1 while (ptr) { 2 ptr->val++; 3 ptr = ptr->next; 4 }</pre>
--	---

(a) Array-based loop.

(b) PLDS-based loop.

Figure 5.1: Simple loops that perform the same *map* operation. The right-hand side version defeats dependence analysis.

each loop nest, we collate the reported parallelizability and single-threaded execution time coverage with five other techniques (three static and two dynamic). We then proceed to parallelize the discovered loops using a common parallel code generation scheme (OPENMP) and compare the obtained speedups. We also assess the accuracy of the predictions produced by DCA. Lastly, since the NPB suite provides an already manually parallelized version, we examine how DCA’s parallelization matches up against a human expert for each program.

5.2 Motivating Examples

Consider the simple loop traversal over an array in Figure 5.1(a), performing a *map* operation written in C. Such patterns can readily be detected as parallel using data dependences to reason about the independence of array accesses across the iteration space of this loop. However, using a different data structure to write what is in essence the same trivial code, shown in Figure 5.1(b), defeats dependence analysis.

Dependence analysis tries to establish the *read* and *write* operations that occur at a program’s memory locations and associate these locations with various relations (e.g., *RAW*, *WAR* and *WAW* dependences). In Figure 5.1(b), the *ptr* pointer is being read in order to update the *ptr->val* data element of each node in the linked list. More crucially, *ptr* itself is being updated (i.e., read and written) to point to the next node for processing by the next loop iteration. This creates a cross-iteration *RAW* dependence on *ptr* which dependence analysis finds present even when profiling information is used. Hence, in this case dependence analysis is inherently incapable of determining the independence of iterations and cannot further propose this loop as a valid candidate for parallelization.

While there have been various attempts to deal with such codes (e.g., pattern

```

1  Graph *g;
2  int *dist;
3  WorkList *frontier, *next_frontier;
4  ...
5  push(frontier, g->adjacent[source]);
6  /* while there is still work in the frontier */
7  while (frontier->size) {
8      /* top down step */
9      while (frontier->size) {
10         /* remove node from worklist for processing */
11         current = pop(frontier);
12         /* go over its adjacent nodes */
13         Node *n = current->next;
14         while (n) {
15             if (dist[n->vertex] > dist[current->vertex]) {
16                 dist[n->vertex] = dist[current->vertex] + 1;
17                 /* this node's distance was updated,
18                 * so in turn we need to recheck its neighbors */
19                 push(next_frontier, n);
20             }
21             n = n->next;
22         } /* end of adjacent node loop */
23     } /* end of top down step */
24     swap(frontier, next_frontier);
25 } /* end of worklist loop */

```

Figure 5.2: A BFS implementation from Lonestar [149] employing complex and irregular PLDS-based loop traversals.

matching), they have been proven inflexible and very limited against real programs. Consider Figure 5.2 that shows a Breadth-First Search (BFS) graph traversal from the Lonestar benchmark suite [149]. BFS employs a worklist to iterate over all the nodes of a graph, traverses the adjacent nodes for each of them and conditionally updates their distance from the selected source node. Its result is the array `dist` containing the distance of each node from source.

The issues with parallelization of the innermost loop in Figure 5.2 (lines 14–20) are similar to those discussed above. In addition, dependences between loop iterations induced by the dynamically updated `frontier` and `next_frontier` worklists via the `push` and `pop` operations prevent conventional parallelization of the top-down step.

This motivates the development of a new method that extends the automatic detection of parallelizable code in real-world programs, addressing some of the aforementioned problems. To this end, we propose Dynamic Commutativity Analysis (DCA); an analysis which utilizes liveness-based commutativity, instead of dependences, to detect potentially parallel loops. By using the live-out variables of a loop (i.e., those consumed later in the program), DCA focuses on the parts of the computation that have an impact on its outcome, irrespective of its traversal idiom. Moreover, using liveness, DCA observes if the outcome remains unaffected when permuting the iterations of a loop, thus strongly suggesting that this loop's iterations can be executed in any order, or in fact in parallel with appropriate synchronization. In other words, we say that DCA detects the loop as *commutative*.

For example in Figure 5.2, the significant result of the outermost loop is the variable `dist`, which is indeed found as live-out after line 25. In this case, DCA determines that this update loop (lines 9–23) is *commutative* and can subsequently be executed in parallel. This is because processing of the current nodes produces the same `dist` values, regardless of the order it occurs.

Overall, DCA detects more potential parallelism in complex and diverse loops in a *uniform* manner, overcoming obstacles which thwart dependence analysis.

5.3 Revisiting Liveness-based Loop Commutativity Analysis

We have already presented the basic concept of our *liveness-based commutativity* and our progressive formalization for loops which we termed *loop commutativity* and *liveness-based loop commutativity*. We then proceeded to qualify this definition depending on the number of permuted input sets used when comparing the corresponding live-out sets, inspiring the definitions of *live-commutativity*, when limiting the comparison to an input and a permutation of it, and *universal live-commutativity*, when comparing all inputs and all possible permutations. We continue in this chapter to refer to *live-commutativity* as just *commutativity*, unless other explicitly stated.

Different loop commutativity analyses with varying scalability-precision trade-offs can be naturally derived from the above definitions. Universal commutativity of a loop can be examined statically by making the input and permutation domains symbolic and proving that no combination of input and permutation leads to a different values for the

live-out variables of the loop.

As we concluded in the previous chapter, despite the recent progress of symbolic execution techniques [20], such an analysis is still unlikely to handle realistic problems, due to a combinatorial explosion of possible permutations and inputs.

This chapter takes a more pragmatic approach and studies the problem of detecting commutative loops dynamically. Interestingly, and despite the fundamental lack of guarantees offered by dynamic analysis, the results reported in this chapter show that in practice there is a close correspondence between commutative and universally commutative loops.

5.3.1 Applicability of Parallelization

Identifying a code region (i.e., function, loop, etc.) as commutative does not immediately guarantee its correct parallel execution. Being able to execute two regions in any order requires the introduction of access synchronization to shared state. For example, in the case the motivating example in Figure 5.1(b) or the one from Figure 4.2(a) (in the previous chapter), the integrity of the accesses to the shared variables `ptr` and `sum`, respectively, has to be maintained. This can be accomplished with access protection mechanisms (e.g., barriers or locks) or other code transformations (e.g., privatization).

The separability-based commutativity model in [211] already fulfills a lot of the requirements for parallelization as a natural consequence of its strict computation model and also employs a series of conservative checks to guarantee the safe parallelization of a method. For output-based commutativity [3], the step to parallelism is not clearly described and the evaluation takes place in a simulation environment with infinite-issue capability.

At this stage in our approach, we combine DCA's profile-guided nature with reemployment of data dependences to extract parallelism and guide synchronization, similarly to [248]. Wherever safe parallelization cannot be conclusively determined, we turn to the user for final approval (Section 5.4.4).

5.4 Dynamic Commutativity Analysis

Our analysis consists of a static and a dynamic stage. Figure 5.3 depicts the components of our approach. We have also implemented a parallelization stage to evaluate the effectiveness of our scheme. To easily follow DCA's operation, we provide examples in

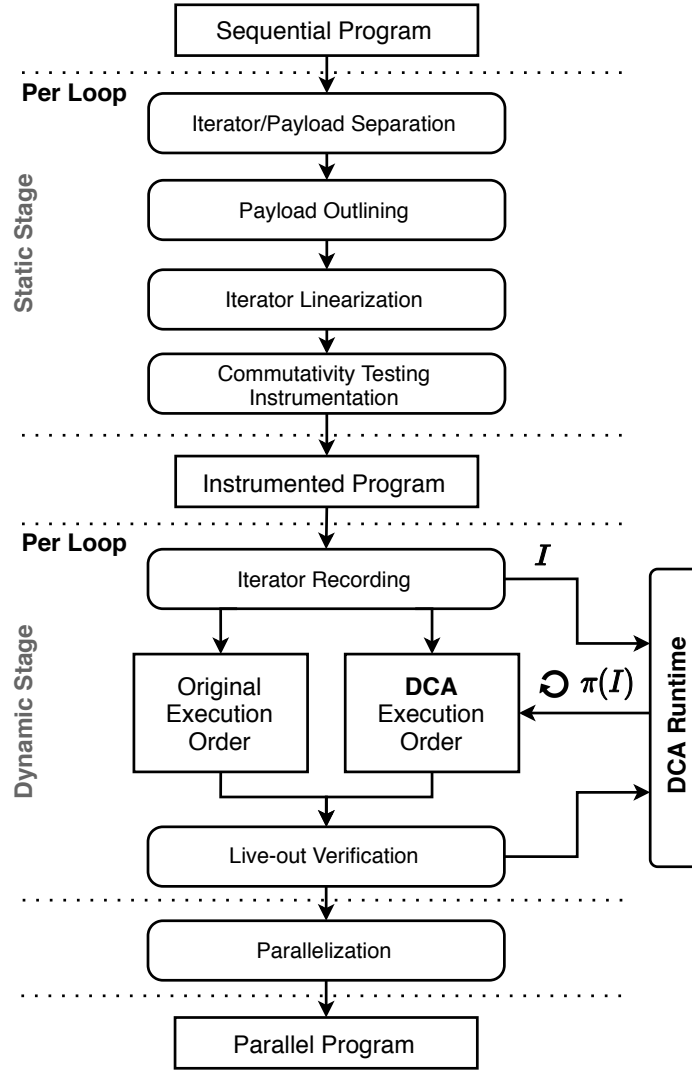


Figure 5.3: Overview of the static and dynamic stages of Dynamic Commutativity Analysis. The interactions with the runtime marked as I and $\pi(I)$ represent an iteration set and a permutation of it respectively.

<pre> 1 for (i = 0; i < N; ++i) { 2 array[i]++; 3 4 }</pre>	<pre> 1 while (ptr) { 2 ptr->val++; 3 ptr = ptr->next; 4 }</pre>
---	--

(a) Iterator/Payload Separation.

<pre> 1 for (i = 0; i < N; ++i) { 2 payload1(i, array); 3 4 } 5 6 void 7 payload1(int it, int *v1) { 8 v1[it]++; 9 }</pre>	<pre> 1 while (ptr) { 2 payload2(ptr); 3 ptr = ptr->next; 4 } 5 6 void 7 payload2(struct node *it) { 8 it->val++; 9 }</pre>
--	---

(b) Payload Outlining.

Figure 5.4: Code output for the first 2 intermediate steps (top to bottom) of DCA's static stage for an array-based (left) and a PLDS-based loop (right) respectively.

C of the intermediate code generated by each step in Figures 5.4 and 5.5, for the simple loops of Figure 5.1.

5.4.1 Static Stage

Our analysis goes over the source of an input program and selects loops for further processing. For every loop nest, each loop is considered separately by a series of compiler passes operating on intermediate representation code.

5.4.1.1 Iterator/Payload Separation

One of the crucial aspects of this stage is our ability to detect which parts of the loop form the *iterator* code and which the actual computation, i.e., the *payload*, using a *generalized iterator recognition* analysis [166]. Intuitively, this analysis identifies the set of variables that are updated on each iteration *and* determine if execution continues in the loop body or exits out of it.

This allows the static stage of DCA to tackle a range of non-affine loop iterators,

while avoiding the use of a limited *ad hoc* scheme. We have already discussed the challenges of dealing with a wider range of iterator styles in Section 5.2. Figure 5.4(a) highlights which lines would be identified and *separated* as iterator from the code of Figure 5.1.

5.4.1.2 Payload Outlining

Next, we outline the *payload* part of the loop in a separate function. This allows us easier handling of the code section in subsequent stages and in particular during execution when coupled with our instrumentation. We identify the live-in, live-out and live-through variables of the outlined region and provide them to the outlined function as arguments. By construction the iterator values are consumed as live-in variables. Figure 5.4(b) shows the resulting code after outlining.

5.4.1.3 Iterator Linearization

We proceed by instrumenting the identified iterator code in order to extract its values during profiling. This process *linearizes* our subsequent accesses to values of the iterator and is similar in spirit to the linearization described in [258]. Our variant is able to tackle a wider range of loop idioms and data structure traversals as it is powered by a *generalized iterator recognition*.

Figure 5.5(a) shows the code from Figure 5.4(b) after inserting calls to the DCA runtime library. While this is depicted in a separate standalone loop (lines 1–4) for simplicity, in practice it can be accomplished by proper placement of these calls in the loop header when operating with low-level intermediate representation code.

5.4.1.4 Commutativity Testing Instrumentation

This stage concludes by placing additional calls to our runtime, enabling commutativity testing of the loop (Figure 5.5(b)). These calls serve two basic purposes: (i) permute the order of the loop iterations, and (ii) verify the commutativity property of the tested loop. Their operation during execution is discussed in detail in the next section. The final output is an instrumented program along with auxiliary reports on the loops that were transformed.

<pre> 1 for (i = 0; i < N; ++i) { 2 rt_iterator_linearize(i); 3 4 } 5 6 for (i = 0; i < N; ++i) { 7 payload1(i, array); 8 9 } 10 11 void 12 payload1(int it, int *v1) { 13 v1[it]++; 14 } </pre>	<pre> 1 while (ptr) { 2 rt_iterator_linearize(ptr); 3 ptr = ptr->next; 4 } 5 6 while (ptr) { 7 payload2(ptr); 8 ptr = ptr->next; 9 } 10 11 void 12 payload2(struct node *it) { 13 it->val++; 14 } </pre>
--	---

(a) Iterator Linearization.

<pre> 1 for (i = 0; i < N; ++i) { 2 rt_iterator_linearize(i); 3 4 } 5 6 rt_iterator_permute(); 7 8 while(rt_iterator_next()) { 9 payload1(10 rt_iterator_get(), 11 array); 12 } 13 ... 14 rt_verify(); 15 void 16 payload1(int it, int *v1) { 17 v1[it]++; 18 } </pre>	<pre> 1 while (ptr) { 2 rt_iterator_linearize(ptr); 3 ptr = ptr->next; 4 } 5 6 rt_iterator_permute(); 7 8 while(rt_iterator_next()) { 9 payload2(10 rt_iterator_get() 11); 12 } 13 ... 14 rt_verify(); 15 void 16 payload2(struct node *it) { 17 it->val++; 18 } </pre>
--	--

(b) Commutativity Testing Instrumentation.

Figure 5.5: Code output for the last 2 intermediate steps (top to bottom) of DCA's static stage for an array-based (left) and a PLDS-based loop (right) respectively.

5.4.2 Dynamic Stage

During the dynamic stage of our analysis, the instrumented program produced by the static stage is executed multiple times by our runtime library. The goal of this stage, in the spirit of Section 4.3, is to determine commutativity for a loop by executing its iterations in different orders and comparing the resulting outcome with the one obtained by the original, programmer-intended execution order (i.e., a “golden” reference).

5.4.2.1 Iterator Recording

Once a DCA-transformed loop is reached during execution, the linearization step records the iterator values, using a random-access sequence container. Next, a set of permutation schedules is selected which control the exact reordering of the loop iterations. The originally prescribed order is executed by default for every loop under test since the output is required as a reference for comparing against the subsequent permuted executions.

5.4.2.2 DCA Execution

Exhaustively executing, using input data from the benchmark suite, all possible permutations for a set of iterator values is exponentially expensive for loops with a large trip count. Therefore, further addressing the scalability issues discussed in Section 4.3, we provide reduced permutation presets (e.g., reverse or a configurable number of random shuffles). This means accepting a chance of missing a commutativity violating permutation, i.e., that our dynamic analysis is generally not safe. In our experiments in this chapter, the configurable number of random shuffles was set to 100, adjusted for the input class of the benchmarks. For example, in the case of the *EP* microbenchmark from the NPB suite, the main top-level loop performs 2^{16} (i.e., 65536) iterations using input class *C*. Fully testing commutativity of this single outer loop, with the specific input set, would require checking all $2^{16}!$ iterator order permutations. There is no clear indication of the required number of random shuffles to provide sufficient accuracy in our profiling, since a single violation would classify a loop as not commutative. However, as evidenced and discussed in Section 5.5.4, this trade-off is still surprisingly powerful in practice.

5.4.2.3 Live-out Verification

The final step of the analysis tests the candidate loop for commutativity. After the execution of loops in the preselected iteration permutation schedules is complete, the runtime compares the produced output computation (i.e., live-outs). If the output of at least one permuted execution differs from the original output (“golden” reference) we mark the loop as non-commutative.

5.4.3 Parallelization

While the focus of this work is mainly on the detection of parallelizable loops, we have also implemented a simple parallel code generation scheme to allow evaluation. We achieve significant speedups for PLDS-based loops (Section 5.5.2.2).

Our strategy is limited at loop-level parallelism and employs the same techniques as described in [248] concerning identification of variables for privatization and reduction operations. Profiling information is used to detect privatizable variables per loop-level by following the reader and writer statements for each memory location. The exploitation of reduction operations uses the approach outlined in [195]. The produced parallel code uses the OPENMP framework which lends itself naturally and easily to this scope of parallelism, since it does not involve any other high-level code restructuring.

For PLDS loops, we adapt these techniques by traversing the elements of the data structure ahead-of-time and place the visited nodes in a dynamically allocated array. Then, we proceed to execute the generated array in parallel using similar OPENMP pragmas as with array-based loops described above. Despite incurring additional time and space overheads from the double traversal and the use of a helper array, our approach still obtains speedups as discussed in Section 5.5.2.3 and Fig. 5.7. Adopting a less naive parallelization mapping scheme, such as the one presented in [85], could contribute in further alleviating the associated costs.

5.4.4 Safety

As already mentioned in Chapter 4 and Section 5.1, profile-guided parallelization cannot inherently guarantee correctness for *every* potential concrete input. However, it overcomes the overly conservative nature of static analysis, unlocking more potential parallelism. In our system, we let the user approve the cases where correct parallelization is not conclusively guaranteed.

In this work, we have also studied the rate of loop misclassification (false positives) and found that DCA correctly identifies all the reported loops as parallelizable (Section 5.5.4) for the specific inputs. This agrees with prior research which shows that the occurrence or absence of *potentially* parallelization-inhibiting dependences are fairly stable across different program inputs [76]. It is unclear how these results could be generalized for other inputs (e.g., for classes or ranges of input values) since for dynamically determined dependences, only a single counterexample suffices to inhibit parallelization transformations. As noted above, we have used inputs provided with the benchmark suites.

5.4.5 Challenges and Limitations

Candidate loops can be deeply nested, thus we explore commutative loops hierarchically in a top-down fashion, using one loop per test invocation. We mitigate this by executing several test instances at the same time.

A candidate loop can appear in different execution contexts (e.g., different call sites of containing function) during application runtime. Loop candidates can exhibit commutativity in some execution contexts, but not in others. Currently, our analysis is not context-sensitive. We leave this for future work.

Execution of regions in permuted order can lead to unpredictable behavior if those loops are not commutative. We reliably detect these situations.

Generally, we assume that candidate loops do not contain I/O statements or produce any other side effects not captured by liveness (volatile memory accesses, etc.). Any such loops are excluded during the selection step of the static stage.

5.5 Empirical Evaluation

We evaluate DCA’s efficacy and the performance obtained by simple parallelization of the detected commutative loops against dynamic (Section 5.5.2) and static (Section 5.5.3) approaches. We also study aspects of its profitability and precision (Section 5.5.4), along with its potency at the loop scope and beyond against expert parallelization (Section 5.5.5).

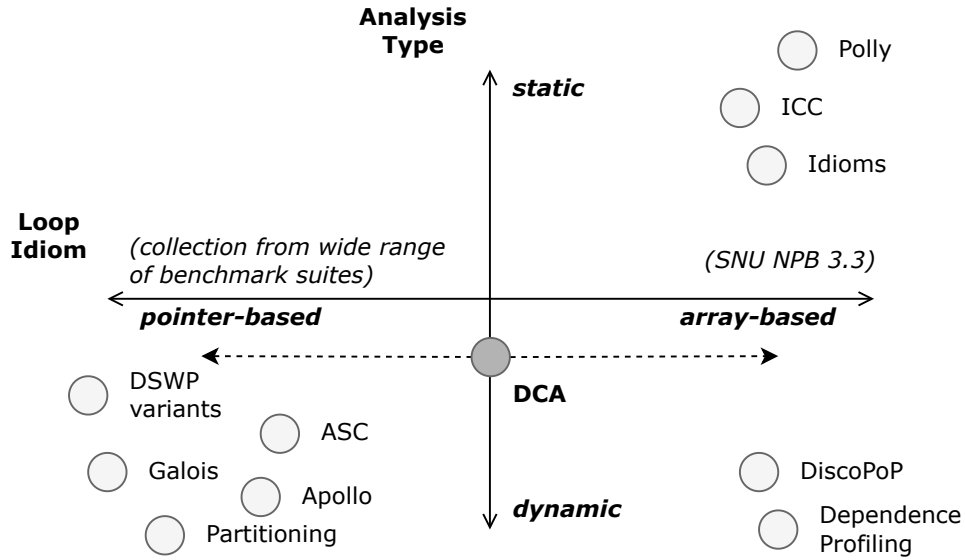


Figure 5.6: Overview of the scope of the baseline techniques used to compare with DCA. We classify them based on their use of dynamic information (y-axis) and the loop idioms which are able to operate on (x-axis). DCA is able to handle both array and PLDS-based loops using the same analysis, employing static and dynamic compiler/runtime stages.

5.5.1 Experimental Setup

We provide a summary of benchmarks, compilers and other configuration elements of our setup for the evaluation described in the remainder of this chapter. We give a more detailed overview of the below compilers and benchmarks in Chapter 3.

Benchmarks. We use the NAS Parallel Benchmark (NPB) suite [19] (NPB 3.3, SNU 1.0.3) to evaluate array-based loops. The suite contains ten programs with a total of 1397 loops (listed in Tables 5.1 and 5.3). The NPB programs implement numerical analysis kernels, in both sequential and OPENMP versions, written in C (originally derived from FORTRAN). We also use a diverse selection of programs that employ PLDS-based loops drawn from several benchmark suites, listed in detail in Table 5.2. Olden [47] benchmarks employing recursion were rewritten in imperative form as in [203].

Inputs. We use input workload class *B* for NPB programs, except for *MG* and *IS* which use class *C*. This is due to the very short execution time of *MG* and *IS* when using input of class *B* on our hardware platform. For the selection of programs with PLDS loops, we use the biggest workload, when available, otherwise we provide a custom workload to also enable a long-running execution for the sequential version.

Compilers. DCA was prototyped on the LLVM compiler infrastructure [156]. We also use five state-of-the-art dependence-based tools as a baseline to compare with our approach. These are:

- **DEPENDENCE PROFILING** [248]: a profile-driven dependence-based parallelism detection approach targeting loops.
- **DISCOPOP** [160]: another profile-driven dependence-based approach aiming at code regions of varying granularity.
- **IDIOMS** [96]: a constraint-based analysis focusing on the detection of complex reduction and histogram operations.
- **POLLY** [100]: a polyhedral transformation framework.
- **INTEL ICC** [120]: a mature industrial compiler that uses data dependence analysis and supports auto-parallelization.

IDIOMS and POLLY are also implemented on LLVM, which allows a loop-by-loop comparison. ICC was also used to obtain loop profiling information across the benchmark programs that we assessed and the INTEL OPENMP runtime library for all parallelized execution runs. The DISCOPOP results are taken from the literature since the tool was not available at the time of writing [160]. Although DISCOPOP’s authors, after private communication, kindly provided us with raw data from their experiments, the discrepancies in the results are due to differences in the produced LLVM IR (e.g., similar loops duplicated at different call sites of the containing function due to inlining).

Figure 5.6 shows the scope of each baseline technique when broken down to the following dimensions: i) their operation during execution-time or use of dynamic information, and ii) the loop idioms they operate on from the benchmarks used in our evaluation.

Configuration. To meet our evaluation goals we configured the tools with the following criteria in mind: (i) during detection, maximize the identification capability for each tool regardless of profitability, (ii) while during code generation, maximize the profitable parallelization exploitation for each tool. Hence, we disable ICC’s parallelization profitability heuristic (i.e., `par-threshold` option) for detecting parallelizable loops, and maximize it for the code generation phase. POLLY’s profitability heuristic is disabled during detection (via the flag `-polly-process-unprofitable`) and it is only applied during code generation. All the static tools compile using optimization level 02 with loop unrolling and vectorization disabled.

Other. The Coefficient of Variation (CV) was 5% or less for all execution time measurements, except for *CG* generated by POLLY where the CV was 40%. We verified

Table 5.1: NPB loops reported as parallelizable by the baseline dynamic approaches, DEPENDENCE PROFILING [248] and DISCOPOP [160], and as commutative by DCA. The discrepancies with DISCOPOP's results are due to the results being taken from the literature (see also Section 5.5.1).

Benchmark	Loops	DEPENDENCE PROFILING	DISCOPOP	DCA
	(#)	(#)	(#)	(#)
BT	182	168	176	168
CG	47	33	21	33
DC	105	—	—	41
EP	9	6	8	6
FT	42	36	34	36
IS	16	12	20	12
LU	186	160	164	160
MG	81	48	66	48
SP	250	233	231	233
UA	479	—	—	466
Total	1397	696	720	1203

DCA's output on NPB using the internal suite verification routines. For the PLDS programs, we used a combination of their reference output and custom profiling.

ICC failed to compile a parallelized version of *UA*, so we use its sequential execution time in all further speedup measurements. Both DISCOPOP and DEPENDENCE PROFILING did not report results on *DC* and *UA*.

5.5.2 Performance against Dynamic Techniques

5.5.2.1 Detection of Array-Based Loops

Our experimental results on almost 1400 loops of the NPB suite show that DCA is effective at detecting commutative array-based loops. This is significant, since further experiments (Section 5.5.4) provide evidence that all the detected loops are indeed comprehensively commutative and hence parallelizable. Given this close correspondence, the rest of this section also refers to commutative loops as *potentially parallelizable*.

Table 5.2: PLDS-based loops which DCA detects as commutative automatically, while existing parallelization techniques fail to identify any. These loops contain profitable parallelism that was exploited manually by previous work. These previous results have either reported the speedup over the sequential execution time of that loop only (subcolumn titled “Loop”) or over the whole program execution time (subcolumn titled “Overall”).

Benchmark	Origin	Loop-Containing Function	Profitability			Detection Technique	
			Sequential Coverage (%)	Potential Speedup (×)			Expert Manual
				Loop	Overall		
429.mcf	SPEC CPU2006 [110]	refresh_potential	30	2.2	—	DSWP variant 1 [204, 203]	
300.twolf	SPEC CPU2000 [109]	new_dbox_a	30	1.5	—	DSWP variant 2 [201]	
ks	PtrDist [15]	FindMaxGpAndSwap	99	1.5	—	DSWP variant 1	
otter	FOSS	find_lightest_geo_child	15	2.5	—	DSWP variant 2	
em3d	Olden [47]	compute_nodes	100	~ 2	—	DSWP variant 1	
mst	Olden	BlueRule	100	1.5	—	DSWP variant 1	
bh	Olden	walksub	100	2.75	—	DSWP variant 1	
perimeter	Olden	perimeter	100	2.25	—	DSWP variant 1	
treeadd	Olden	TreeAdd	100	—	~ 7	Partitioning [85]	
hash	Shootout	ht_find	50	—	~ 4	Partitioning	
BFS	Lonestar [149]	BFS	99	—	21	Galois [194]	
ising	community	main	95	—	~ 6	ASC [147]	
spmatmat	SPARK00 [258]	main	89	—	~ 4	APOLLO [41]	
water-spatial	SPLASH3 [94, 217]	INTERF	63	—	2	OPENMP	

For each benchmark, Table 5.1 reports the total number of loops and the number loops that each baseline dynamic approach reports as parallelizable, and compares these loops with the loops detected as commutative by DCA. As seen in the table, DCA uncovers potential parallelism that closely matches each of the dynamic techniques. The detection effectiveness of DCA is attributed to its ability to capture the effects of code that spans large regions with complex control flow, function calls and non-linear array accesses.

5.5.2.2 Detection of PLDS-Based Loops

While DCA is effective at detecting commutative array-based loops, its true potential as a unified analysis for parallelism discovery is best seen on PLDS-based loops. DCA is able to detect *automatically* as commutative a broad collection of popular PLDS-based

loops from earlier compiler studies, whereas the baseline tools, both dynamic and static, hit their limits and *fail to detect any of them as parallel*. The focus of these earlier studies has been on profitably exploiting these loops on modern parallel architectures rather than detecting them as parallel. Unlike DCA, these studies rely on *ad hoc* manual methods to identify and utilize PLDS-based loops. The PLDS-based loops detected as commutative are summarized in Table 5.2. The table also reports, for each loop, its origin, sequential coverage and potential speedup found by *ad hoc* manual methods as described in the literature [204, 203, 201, 85, 194, 147, 41]. Sequential coverage is defined as the percentage of total execution time spent in a specific program scope, which in this context is one or more loops [137]. Remarkably, DCA detects commutative loops even in complex programs, such as *300.twolf* which contains doubly-nested linked list traversals, similar in nature to Figure 5.1(b). Other loops, such as *treeadd*, employ a worklist traversal idiom akin to BFS presented in Figure 5.2.

An interesting case is *429.mcf*, the only loop in Table 5.2 known *not* to be statically commutative. *429.mcf* performs a complicated tree traversal, accessing sibling and predecessor nodes, and contains a cross-iteration dependence. The dependence is not exercised by the test or the reference workloads, hence DCA reports it as commutative. Speculative parallelization approaches in the literature rely on the assumption that this dependence is infrequent to parallelize the loop profitably.

In summary, DCA’s uniform approach is able to discover potential parallelism in loops such as PLDS traversals that are well beyond the limits of dynamic dependence-based analysis approaches. This opens a potential avenue for leveraging techniques aiming at efficient parallelism exploitation of complex loops, such as speculative parallelization.

5.5.2.3 Parallelization of PLDS-Based Loops

Our results demonstrate that DCA’s simple parallelization scheme can yield speedups for PLDS-based loops (up to $36\times$), although it is not as widely effective as for array-based loops. Figure 5.7 reports the parallelization speedup achieved by DCA for a selection of the PLDS-based loops found as commutative in Table 5.2. The selection combines simple kernels (such as *spmatmat*, a sparse matrix-matrix multiplication) and loops from larger, more complex programs (such as *water-spatial*, an n -body wave simulation).

For the remaining loops in Table 5.2, DCA’s simple parallelization scheme does not yield significant speedups and requires more specialized parallel code generation

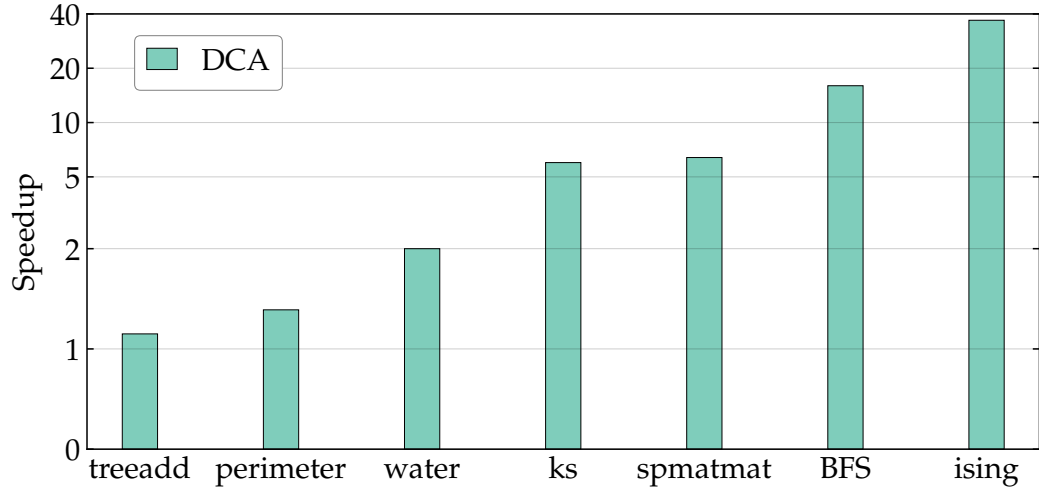


Figure 5.7: Overall speedup over sequential code achieved by DCA parallelization for PLDS loops. The parallel code generation techniques from Table 5.2 fail to detect parallelizable loops automatically.

techniques (last column of that table) for profitable exploitation. In most of those cases, while the target loops account for a significant fraction of execution time, any benefits from our parallelization scheme are negated due to the high number of relatively short-lived calls of their containing functions. However, in the cases where DCA’s simple parallelization is not effective, its analysis results might still be exploited by more sophisticated parallel code generation techniques such as those reported in Table 5.2. Exploring the interaction between DCA’s parallelism discovery and these parallelization techniques is part of future work.

5.5.3 Performance against Static Techniques

5.5.3.1 Detection of Array-Based Loops

Similarly to Table 5.1, Table 5.3 reports the number of loops that each static baseline approach reports as parallelizable. It also presents a comparison of the *combined* results of all three static approaches (*Combined Static*) and the number of loops detected as commutative by DCA. DCA uncovers nearly twice as much potential parallelism (86% of all loops) as the combined static baseline (49%).

For five out of the ten benchmarks in the suite, DCA finds over 80% of the loops as potentially parallelizable, substantially more than the combined static baseline, which achieves less than 50% for the same benchmarks. ICC is more robust in detecting

Table 5.3: NPB loops reported as parallelizable by the baseline static approaches and as commutative by DCA. “Combined Static” indicates the results of the static techniques (IDIONS [96], POLLY [100] and ICC [120]) combined.

Benchmark	Loops (#)	IDIONS		POLLY		ICC		Combined Static		DCA	
		(#)	(%)	(#)	(%)	(#)	(%)	(#)	(%)	(#)	(%)
BT	182	5	3	34	19	50	27	80	44	168	92
CG	47	9	19	8	17	23	49	25	53	33	70
DC	105	14	13	11	10	23	22	39	37	41	39
EP	9	2	22	2	22	3	33	4	44	6	67
FT	42	1	2	6	14	1	2	8	19	36	86
IS	16	7	44	3	19	3	19	11	69	12	75
LU	186	3	2	19	10	81	44	90	48	160	86
MG	81	8	10	5	6	21	26	32	40	48	59
SP	250	2	1	38	15	93	37	113	45	233	93
UA	479	23	5	43	9	180	38	209	44	466	97
Total	1397	74	5	169	12	478	34	611	44	1203	86

parallelizable loops than the other two baseline approaches. In some cases this is due to more aggressive inlining of pure (i.e., no side effects) functions. On the other hand, ICC is unable to detect complex reduction and histogram operations discovered by IDIONS. DCA correctly identifies all the above loops which are executed as commutative.

In the rest of the NPB suite, DCA also achieves high detection scores, outperforming the combined baseline. *CG* contains a higher number of loops exhibiting cross-iteration dependences, which neither DCA nor the rest of the approaches detect. *MG* displays a somewhat unusual coding style with respect to the rest of suite, using I/O in several nested loops and contains a number of loops that the input workloads do not exercise. We have asserted that DCA can detect these loops as potentially parallelizable given the appropriate input conditions, but excluded them from our final results for consistency. Unsurprisingly, DCA detects the least number of potentially parallelizable loops for *DC*, which performs numerous I/O operations.

5.5.3.2 Parallelization of Array-Based Loops

Our results show that the simple parallelization scheme proposed in Section 5.4.3 is generally effective for array-based loops. Overall, by parallelizing the profitable loops among those discovered as commutative in Section 5.5.3.1, DCA achieves an average speedup of $3.6\times$ (and up to $55.2\times$) over the sequential version of each NPB benchmark. Figure 5.8 presents the speedup results for each benchmark.

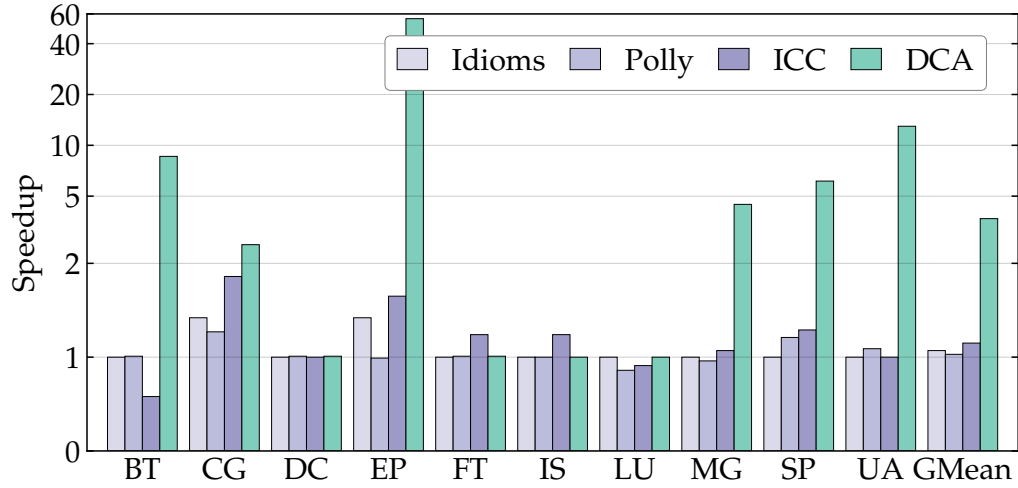


Figure 5.8: Overall speedup over sequential code achieved by IDIOMS, POLLY, ICC and DCA parallelization for NPB.

Since profitability analysis is out of DCA’s current scope, only the commutative loops deemed as profitable in the expert NPB implementation (and the hottest ones for the case where this information is not available) are selected for parallelization. Remarkably, DCA detects as commutative all data-parallel loops deemed profitable in the expert parallelization.

EP is a small kernel with a hot two-level loop nest performing an integral evaluation via pseudo-random trials. Parallelizing the outer loop which contains the complex reduction loop yields a speedup of $55.2\times$. DCA also achieves significant speedups for *BT*, *CG*, *MG*, *SP* and *UA* ($8.6\times$, $2.6\times$, $4.5\times$, $6.1\times$ and $13\times$ respectively). This is attributed to DCA’s ability to detect and exploit loops that extend across many lines of code, containing function calls and complex control flow.

For *DC*, *FT*, *IS* and *LU*, the relatively high number of detected commutative loops does not translate into profitable parallelism. For example, *DC* is an I/O intensive benchmark manipulating data at volumes much larger than a modern system’s memory capacity, while *LU* contains dependences across hot function calls. Thus, these programs require higher level of synchronization or extended code restructuring for their efficient exploitation.

DCA (together with expert profitability analysis) consistently outperforms the baseline parallelization by IDIOMS, POLLY, and ICC, reported in Figure 5.8. The same profitability analysis is applied for IDIOMS as for DCA, whereas for ICC and POLLY their optimal profitability analysis is used (Section 5.5.1). ICC and IDIOMS are able

Table 5.4: DCA covers a significant fraction of execution time, providing high precision detection results.

Benchmark Loops	DCA				Combined	
					Static	
	Found	False Positive	False Negative	Sequential Coverage	Sequential Coverage	
	(#)	(#)	(#)	(#)	(%)	(%)
BT	182	168	0	0	100	36
CG	47	33	0	0	91	7
DC	105	41	0	0	0	0
EP	9	6	0	0	100	37
FT	41	36	0	0	91	42
IS	16	12	0	0	60	56
LU	186	160	0	0	84	56
MG	81	48	0	0	87	56
SP	250	233	0	0	94	77
UA	479	466	0	0	86	57

to extract some latent loop parallelism in *CG* and *EP*, but are still outperformed by DCA. IDIOMS is able to exploit *EP*, but the effectiveness of parallelization is limited to the inner hot loop. The difference between ICC and DCA for *FT* and *IS* is due to the unavoidable compiler optimization differences: manually implementing the parallelization suggestions from ICC’s reports for *IS* and compiling it in our framework results in the same performance that ICC obtains.

5.5.4 Aspects of Detection Profitability and Precision

In order to assess the accuracy of DCA’s predictions, all loops were further analyzed semi-manually, employing expert algorithmic knowledge and a combination of targeted profiling and testing of dependences and computations.

The results (Table 5.4) show that all loops determined as commutative by DCA are indeed parallelizable (i.e., no false positives), following the spirit of the discussion in Section 4.3. This result strengthens the significance of Tables 5.1 and 5.3, as it confirms that DCA can indeed uncover many valid opportunities for parallelization on top of the baseline approaches.

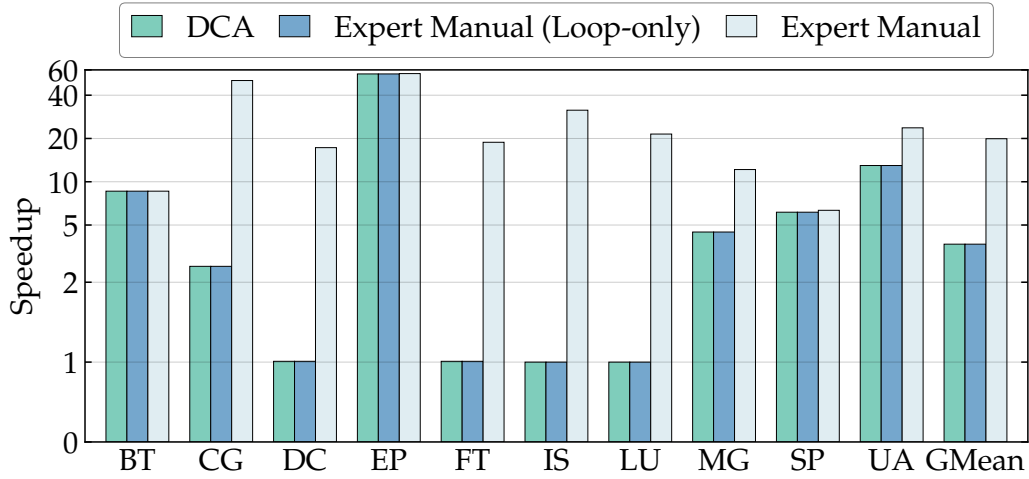


Figure 5.9: Overall speedup over sequential code achieved by DCA and Expert Manual (loop-only and whole program) parallelization for NPB. DCA matches the performance of manual parallelization by experts at the loop level.

Future work could improve DCA’s effectiveness by applying combined tests for multiple inputs and exploring inputs leading to execution paths that might affect commutativity.

The loops found by DCA are significant in that they cover a considerable fraction of the total execution time for most benchmarks in NPB, with above 80% for eight out of the ten benchmarks in the suite and almost 100% for *BT* and *EP*. As shown by the two rightmost columns of Table 5.4, DCA consistently outperforms the combined static approaches. This agrees with the high detection rate shown in Table 5.3 and the parallelism that is profitably exploited as reported in Figure 5.8.

Overall, DCA accurately detects a large amount of potentially relevant and profitable parallelism, beyond the capabilities of static analysis approaches.

5.5.5 Scope of Parallelization Beyond Loops

The evaluation has so far focused on the detection and parallelization of loops, where DCA succeeds in uncovering more hidden loop parallelism than several state-of-the-art approaches. There exists, however, parallelism that is not strictly confined within loops.

Figure 5.9 compares the speedups of DCA on the NPB suite with those obtained by the data-parallel loops as parallelized by experts (*Expert Manual (Loop-only)*) and a full expert parallelization beyond single-loop data parallelism (*Expert Manual*). The

results show that DCA succeeds in identifying and exploiting data-parallel loops and that opportunity remains for further parallelization beyond their scope.

DCA is successful in extracting all the available parallelism from *BT*, *EP* and *SP*. It is also fairly effective on *MG* and *UA*, where the performance discrepancy with the full expert parallelization is due to the fact that the latter exploits whole parallel sections, spanning across loops, thus improving locality and minimizing synchronization costs.

The rest of the benchmarks contain parallelism that is outside the loop-level focus of DCA. The full expert parallelization of these benchmarks uses additional algorithmic knowledge. *DC* and *FT* are largely restructured to take advantage of independent work-sharing. *LU* uses a pipeline pattern to overcome the bottleneck in its hottest loop nests. *CG* falls under the same category, however, DCA is able to exploit profitably some of its remaining loop-level parallelism.

The gap between loop-level parallelism and what can be achieved by taking a wider, structural view of program parallelism motivates further research on this topic. We believe that commutativity analysis can also play a key role in discovering and exploiting structured parallelism. We explore this potential in Chapter 6.

5.6 Summary and Conclusions

In this chapter, we have presented DCA, a novel, hybrid analysis for identifying parallelizable loops in sequential legacy code, which relies on liveness-based commutativity.

DCA proceeds in two phases. First, it statically identifies and instruments the potential live outputs and useful computation of a loop to allow the permutation of its iterations during execution time. The static phase relies on state-of-the-art analysis techniques such as *liveness-based commutativity* [266] and *generalized iterator recognition* [166]. Then, it determines dynamically whether the loop is commutative by comparing the live output of the permuted executions with that of the original iteration order.

The result of the analysis can be used as a *strong indication* of the loop's parallelizability. In fact, as confirmed by our extensive experiments, all loops reported as commutative by DCA in our evaluation are indeed parallelizable. We do not expect DCA to be used in a fully automated setting, rather as part of an interactive or semi-automatic parallelism advisor, where the user has the final word over any code transformations. Potential avenues to fully automate this technique along with overcoming other limitations and suggesting future directions are discussed in Chapter 7.

Applying DCA to the sequential version of the NPB suite (almost 1400 loops in non-trivial scientific applications), we found numerous parallel loops, matching two profile-driven dependence techniques, DEPENDENCE PROFILING and DISCOPOP, and outperforming the *combination* of three static dependence methods, INTEL ICC, LLVM POLLY and a constraint-based idiom recognition approach, finding nearly twice as many. DCA was also able to uncover parallelism in irregular loops dominated by PLDS structures, in the same *uniform* manner as for array-based loops, *where all other analyses tested failed*.

After parallelization, our experiments showed that DCA resulted in an average speedup of $3.6\times$ across NPB (and up to $55\times$) on a 72-core host, and up to $36.9\times$ for the PLDS-based loops.

In conclusion, our evaluation showed that DCA, a *liveness-based commutativity* loop parallelization technique that combines static and dynamic information, is capable of discovering latent parallelism beyond dependence-based methods, over a wide range of benchmarks and with high precision.

In the next chapter, we revisit the loop *liveness-based commutativity* concepts presented in Chapter 4 and incorporated in DCA, moving towards the ability to identify not just potential parallelism, but the pattern or skeleton form which that parallelism takes.

Detecting Data-Parallel Patterns with Liveness-based Commutativity

In the previous chapter we introduced DCA, a technique based on *liveness-based commutativity* and utilizing profiling information from program execution using specific input sets. Evaluating DCA on a broad range of loops, following both array-based and pointer-based traversal idioms, we were able to identify useful parallelism in sequential code that exceeded the capabilities of other competing techniques which rely on dependence analysis.

Looking into the form of the discovered parallelism, we find DOALL-style loop parallelism which is straightforward to extract. This is due to the inherent independence among the iterations of a DOALL loop. However, other loop structures exhibit data flow which requires synchronization, since they combine the outcomes of various iterations. For instance, one such case is the reduction operation that we have discussed in the previous two chapters, where each iteration is reusing the result of the one immediately preceding it, under the order specified in the execution of its sequential implementation.

Typically, the naive approach is to control and limit access around the shared state required by each thread in the parallel version of such a loop form. However, this, as we have seen in more complex cases in Chapter 5 (e.g., certain PLDS-based loops), while it can ensure the correctness of the parallelization, it can also cripple its efficiency and negate any potential benefits by the simultaneous execution of iterations. Nevertheless, several of these loop forms have been found to have an efficient parallel implementation which usually is achieved by employing a different algorithm in a parallel context.

In this chapter, we propose and apply a methodology which leverages our DCA technique in a pattern-oriented detection manner. This enables us to broaden the scope

```

1  out[0] = array[0];
2
3  for (i = 1; i < N; ++i) {
4      out[i] = out[i - 1] + array[i];
5  }

```

Figure 6.1: A loop performing a prefix sum operation over an array.

of detection towards the identification of latent *parallel patterns* in sequential loop code. Specifically, we demonstrate that our approach can identify parallelism not only in commutative loops, but also in certain *non-commutative* loops when pursued from a pattern-oriented perspective, thus expanding the potential parallelization opportunities.

This chapter starts with Section 6.1 which motivates our study. We continue with Section 6.2, which describes our pattern-oriented methodology and Section 6.3, which evaluates its application with case studies over widely used data-parallel patterns found in sequential code. We conclude and summarize in Section 6.4.

6.1 Motivation

Consider the loop in Figure 6.1 which performs a simple prefix sum operation over a set of values in an array. Essentially, every output location in array `out` contains the result of the reduction of all the input values from array up that index position.

Such operations pose a challenge to dependence-based detection methods for parallelization because they exhibit loop-carried dependences. For our example in Figure 6.1, the RAW dependence in the `out` array between the element at index `i` and the one at `(i - 1)` does not allow any iteration reordering. This is the same reason prohibiting parallelization as the one for reductions which was discussed in the motivating example for commutativity in Chapter 4.

Moreover, *liveness-based commutativity* as implemented in DCA, described in the previous chapter, is not able to detect the loop in Figure 6.1 as commutative. Consider the execution of the loop from Figure 6.1 for the first three iterations (i.e., assuming $N \geq 4$ and integer arrays) presented in Figure 6.2(a) and when permuted, following a reverse execution order, in Figure 6.2(b). The resulting contents of the `out` array variable at line 18 differ between the two execution orders, thus DCA would correctly not detect this loop as commutative and, by extension, as potentially parallelizable because it is not, in fact, commutative. Examining the results of Figure 6.2(b), we

1	<i>// initialization</i>	1	<i>// initialization</i>
2	array[] = { 5, 7, 9, 1, ... };	2	array[] = { 5, 7, 9, 1, ... };
3	out[] = { 0, 0, 0, 0, ... };	3	out[] = { 0, 0, 0, 0, ... };
4	...	4	...
5	out[0] = array[0];	5	out[0] = array[0];
6	<i>// out[] = { 5, 0, 0, 0 };</i>	6	<i>// out[] = { 5, 0, 0, 0 };</i>
7		7	
8	<i>// first iteration</i>	8	<i>// first iteration</i>
9	out[1] = out[0] + array[1];	9	out[3] = out[2] + array[3];
10	<i>// out[] = { 5, 12, 0, 0 };</i>	10	<i>// out[] = { 5, 0, 0, 1 };</i>
11		11	
12	<i>// second iteration</i>	12	<i>// second iteration</i>
13	out[2] = out[1] + array[2];	13	out[2] = out[1] + array[2];
14	<i>// out[] = { 5, 12, 21, 0 };</i>	14	<i>// out[] = { 5, 0, 9, 1 };</i>
15		15	
16	<i>// third iteration</i>	16	<i>// third iteration</i>
17	out[3] = out[2] + array[3];	17	out[1] = out[0] + array[1];
18	<i>// out[] = { 5, 12, 21, 22 };</i>	18	<i>// out[] = { 5, 12, 9, 1 };</i>
	(a)		(b)

Figure 6.2: Execution of the first three loop iterations in the original, programmer-intended order (left) and in the reverse order (right). The contents of the out array are depicted as comments right after each statement is executed.

see the contents of out are partly correct, specifically for position at index 1. This is because the value of that position used the correct preceding partial reduction (i.e., position at index 0). In the other hand, the second and third iterations, when executed out of order, used the wrong value since that had not been computed yet (i.e., the value 0 from the initialization of out).

Nevertheless, prefix sum is well-studied operation that can be parallelized [30]. In short, when the operation applied over the input set is associative, we can perform different reorderings of the intermediate computations, allowing for shorter span (i.e., the length of the longest series of computations which have to be performed sequentially) and more parallelism. Figure 6.3 highlights graphically the differences between a serial and parallel implementations of prefix sum over a small input array (i.e., size of 4 elements). First, in Figure 6.3(a), the order of computations for the intermediate results is such that only one at a time can be computed because of the dependences between them (i.e., t_2 requires t_1). However, taking advantage of the associativity of addition,

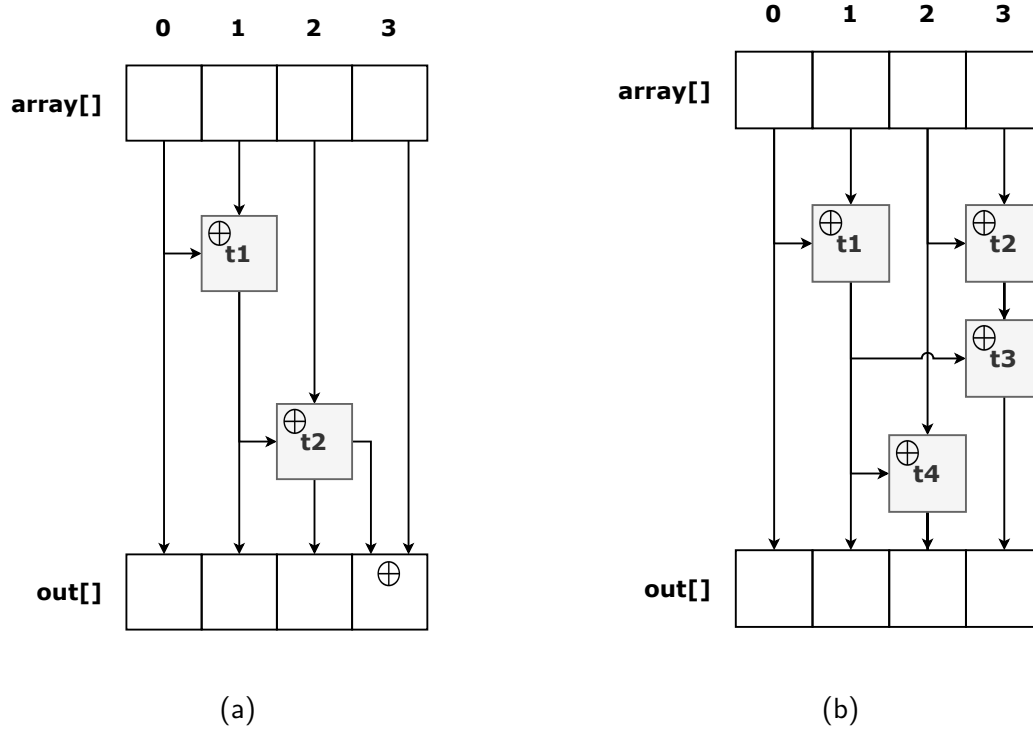


Figure 6.3: Representation of the dataflow in prefix sum when implemented sequentially (left) and in parallel (right). The required intermediate results are represented with the grey squares between the input and output arrays. The $\oplus$ sign denotes a location where a sum operation is performed to compute its contents using the incoming values.

we can calculate the intermediate results in parallel following a hierarchical approach. This means `t1` and `t2` can be computed in parallel, but prior to `t3` and `t4`, which in turn can be executed in parallel.

Note that this parallel implementation performs more sum operations (4 versus 3) and requires more intermediate space locations (4 versus 2). Despite this, it reduces both the span of execution and increases the amount of parallelism. Different parallel implementations provide different trade-offs between the work-span and the available parallelism [111, 30, 31, 171].

The prefix sum is one instance of the *scan* parallel pattern (i.e., algorithmic skeleton) for any associative operator [31]. Their wide usefulness is recognized by their inclusions as building blocks for various parallel programming languages, libraries and frameworks [185] (Chapter 2). Figure 6.4 shows a use of the built-in prefix sum function from the SKEPU library [79]. These library implementations enhance portability, allowing the control and abstraction of tuning and selection of various implementation

```

1  #include <skepu>
2
3  int plus(int a, int b) { return a + b; } // operator expressed as function
4  ...
5  const int N = ...;                      // size definition
6  skepu::Vector<int> array(N), out(N);     // input/output definition
7  ...
8  auto prefix_sum = skepu::Scan(plus);     // skeleton instantiation
9  ...
10 prefix_sum(out, array);                  // skeleton execution

```

Figure 6.4: The prefix sum operation from Figure 6.1 implemented using the scan pattern from the SKEPU parallel programming framework.

aspects that affect a pattern’s performance, e.g., such as the work span or the amount of intermediate work, as discussed earlier in this section. We took advantage of this added benefit in the case of reductions when we used OPENMP for the parallel code generation in DCA described in Chapter 5.

Therefore, a crucial challenge for *liveness-based commutativity* and, thus, DCA is to extend their applicability of detecting data parallel patterns in sequential code. More specifically, it would be important to achieve this by DCA in the detection of patterns whose sequential loop-based form is not commutative, even though these loops are parallelizable. In the rest of this chapter, we examine the potential of an analysis that can discern among such patterns, incorporating DCA’s ability to determine about the commutativity or non-commutativity of their constituent loops.

6.2 Methodology

6.2.1 Overview

In this section we present an approach which extends the applicability of DCA for the detection of data-parallel patterns in sequential loops. We describe the main insight behind it, provide a high-level overview of each stage and abstractly present its constituent transformations performed using a specific data-parallel pattern. We provide a more detailed look at the individual transformations and criteria required for each pattern in their corresponding case study in Section 6.3.

Figure 6.5 presents a high-level overview of our methodology for the identification

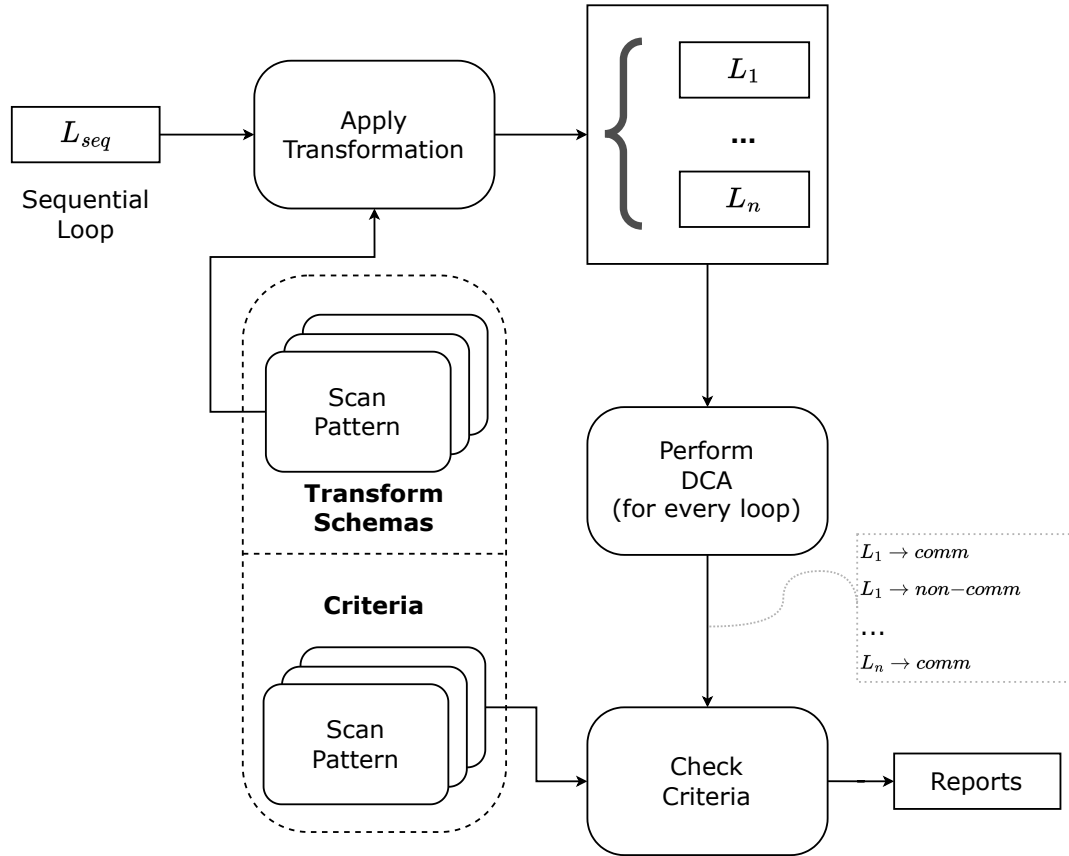


Figure 6.5: Overview of the steps performed by our data-parallel pattern identification technique leveraging DCA. Note that DCA is applied to all loops that resulted from the application of a transformation schema and the collated results are passed to the next stage. The whole process is repeated for each available pattern description (transform schema and criteria).

of data-parallel patterns. Our technique proceeds by examining a sequential program a single loop at a time. The first step is to apply the syntactic transformation, encapsulated in a schema description, to the loop. This produces a code fragment, typically consisting of a sequence of loop nests. The transformation turns the original loop into a sequence of loops which though explicitly still sequential, captures the phases of the corresponding parallel library implementation of the pattern. For example, for the initial code fragment of Figure 6.1, the corresponding transformed code is shown in Figure 6.6. This can be seen as a collection of loops L_0 to L_n inside a code section in the top right corner of Figure 6.5.

As shown in Section 6.1, decomposing a candidate pattern loop to its expected

```

1  int ssize = N / P;
2  int array[N], tmp[N], out[N];
3  int tid;
4
5  array[] = { 5, 7, 9, 1, ... };
6  out[]   = { 0, 0, 0, 0, ... };
7
8  // phase 1
9  for (tid = 0; tid < P; tid++) {
10     tmp[tid * ssize] = array[tid * ssize];
11 }
12
13 // phase 2
14 for (tid = 0; tid < P; tid++) {
15     for (i = tid * ssize + 1; i < (tid + 1) * ssize; i++) {
16         tmp[i] = tmp[i - 1] + array[i];
17     }
18 }
19
20 // phase 3
21 for (tid = 0; tid < P; tid++) {
22     for (i = tid * ssize; i < (tid + 1) * ssize; i++) {
23         out[i] = tmp[i];
24     }
25 }
26
27 // phase 4
28 for (tid = 0; tid < P; tid++) {
29     for (i = 0; i < tid; i++) {
30         for (j = tid * ssize; j < (tid + 1) * ssize; j++) {
31             out[j] += tmp[(i + 1) * ssize - 1];
32         }
33     }
34 }

```

Figure 6.6: The sequential multi-loop code which results after applying our proposed transformation for the loop in Figure 6.1.

constituent loops from its parallelized form allows us to test their commutativity. For this purpose, we utilize DCA, which as we have seen in Chapter 5, is able to detect a range of different loop idioms using profiling information. However, the results of

DCA are not used in isolation for each loop, but grouped together and compared against the expected commutativity for each component loop of the parallel pattern. The lack of commutativity is also useful here, in contrast to its treatment as a parallelization inhibitor in Chapter 5. Hence, the role of DCA is broadened, composed to a detection enabler for parallel patterns in sequential code.

Subsequently, we apply DCA over every loop in the derived transformed code section by executing the program using the provided input sets. We record the result of DCA (i.e., commutative or non-commutative) for each loop and pass the collated results to the last stage. In this stage, we apply a set of criteria associated with the selected pattern that need to be satisfied to confirm the presence of this specific pattern. These steps, from the transformation to the matching, are performed for every available pattern whose transformation schema and identification criteria has been provided by the user.

Recall from Figure 6.3(a) the corresponding graphical representation of the sequential implementation of the prefix scan where each computation per loop iteration can only occur after the previous has been completed. In Figure 6.7 we present, as an example to assist in the comprehension of our methodology, a similar graphical example of the memory state and the operations which are performed on it by the transformed scan code of Figure 6.1. These states and computations are organized in phases and are executed from top to bottom and one after the other (i.e., sequentially). We present the corresponding transformed code generated by our approach in Section 6.3 where we investigate the identification of the scan pattern.

First, we notice that the memory locations of the input array, the intermediate `tmp` and the output `out` are divided into segments (or blocks). In the fully parallel version, which we got inspiration for our approach from, these correspond to areas of the shared arrays over which each thread would operate on. This is a consequence of the labor division, allowing each thread to read or write in these segments without requiring access control (i.e., synchronization). Going over these segments in the sequential version corresponds to using a loop for each phase in Figure 6.7. In the same vein, processing each element per segment utilizes an inner loop for each (outer) segment processing loop. This occurs in all phases where either the source, the destination element or both are not drawn from a constant index (i.e., the locations marked with thicker borders), and can be seen most clearly in the third phase which involves a copy of the `tmp` contents to `out`.

The transformed code is now passed to the criteria checking phase. Our key insight

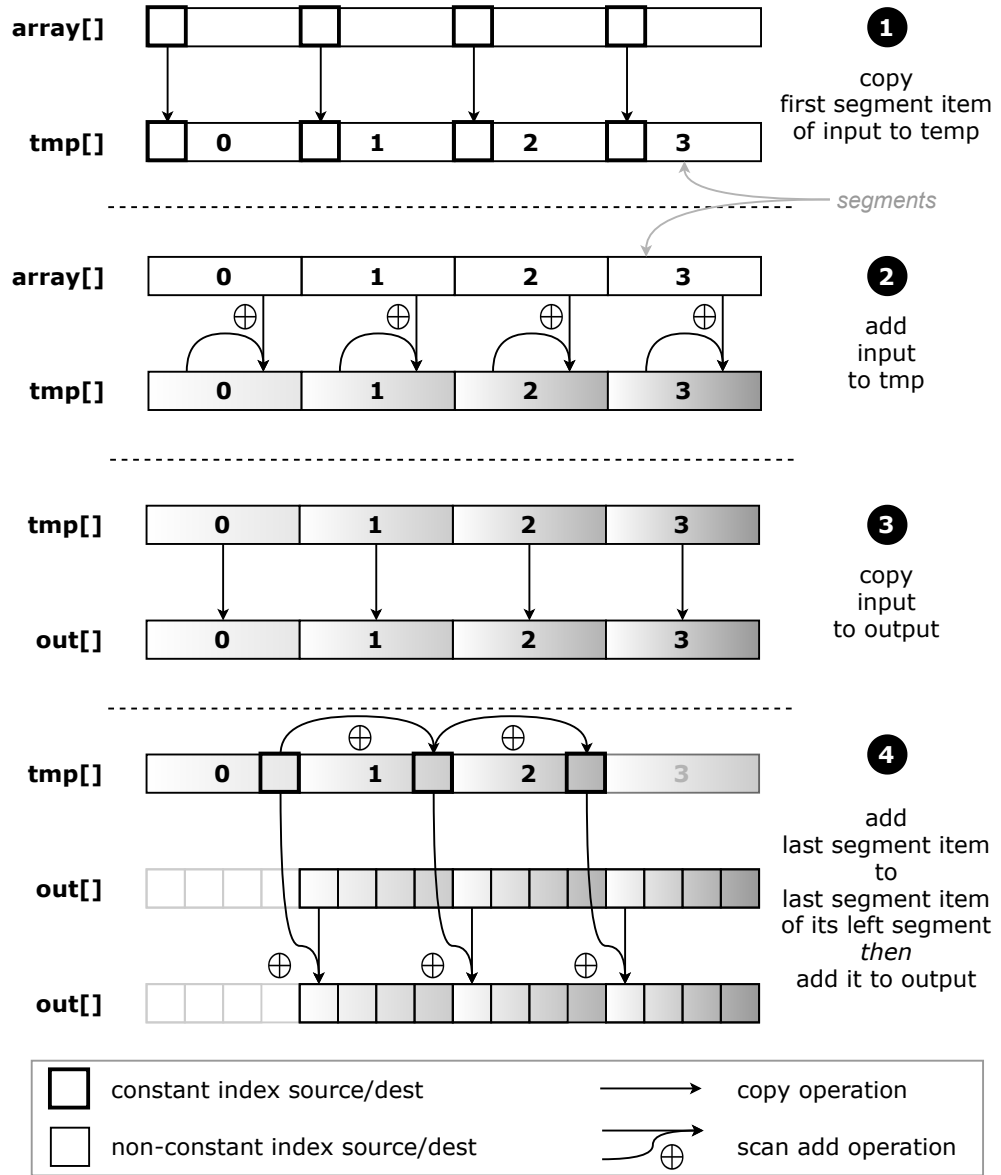


Figure 6.7: Example result by applying our methodology over the loop from Figure 6.1 depicting the phases of the generated sequential scan pattern (top to bottom). It graphically depicts the memory state and the operations on it as performed by the transformed code. Unannotated and $\oplus$ -annotated arrows represent a copy or a scan add operation, respectively.

is that, for the presence of the given pattern to be declared, the loops in the transformed code should have the same commutativity properties as those common to all correct uses of the pattern. These loops will be tested individually by DCA. It is easy to recognize from the pictorial representation of the dataflow in Figure 6.7 that the outermost loops (i.e., iterating over segments) of first and third phases are commutative. Additionally,

although less straightforward to determine, the segment loop of phase four is also commutative, since the operands of the scan within the segment are either: i) constant, with `tmp` as their source, or ii) originate from locations within the same segment of `out`. As a last example, we turn our attention to the element processing within a segment in phase 2 of Figure 6.7. We notice that the scan operation uses an element of `tmp` from a previous iteration, thus denoting that such a reuse makes the loop non-commutative for reasons already presented in Section 6.1.

6.2.2 Syntactic Transformations

In its most basic form a syntactic transformation converts a candidate loop to a series of loops, using the inputs and outputs of the original loop (as detected by liveness analysis), iterator code and basic operator (e.g., addition in Figure 6.6), substituting them in placeholder locations in the generated target code. Currently, these are implemented as a combination of compiler passes and code templates provided by the user, who also ensures their correctness after their application to a code fragment. This is on top of using user assistance to determine the result of DCA (Chapter 5).

Figure 6.8 presents succinctly the transformation steps of our methodology for the scan pattern in pseudocode. The first function (line 1) creates the generated code region that contains all the resulting code from the application of each phase (lines 9 to 12). This functions proceeds by generating: i) the placeholder code region, ii) the required storage, iii) calling the constituent phase generation functions, and iv) collating the phases in the region. For brevity, we present only the pseudocode for the generation of the loop at the second phase (line 18). The `GenerateScanLoopPhase2` generates a two-level loop nest and calls the scan operation for each input segment. The input and output values, as found by liveness analysis, are connected with the initial loop input and output variables once the transformation has taken place in the location of the original loop.

As noted in the previous section, our *key insight* is to utilize the known parallel implementations of data-parallel patterns in order to facilitate the detection of that pattern in sequential code. We have already discussed (Chapter 2) that parallel patterns help encapsulate various aspects of the commonly found computations inside a particular algorithmic pattern. Briefly, patterns encapsulate facets such as the organization of computations, the arrangement and movement of data, offering the necessary abstractions to separate semantics from implementation details. This separation does not aim

```

1  def GenerateScanTransform(seqLoop, scanOp, p):
2      region = Compiler.CreateRegion(...)
3
4      ssize = len(seqLoop.tripCount) / p
5      tmp = Compiler.CreateStore(ssize)
6      out = Compiler.CreateStore(ssize)
7      region.add([tmp, out])
8
9      loop1 = GenerateScanLoopPhase1(scanOp, region, p, ssize)
10     loop2 = GenerateScanLoopPhase2(scanOp, region, p, ssize)
11     loop3 = GenerateScanLoopPhase3(scanOp, region, p, ssize)
12     loop4 = GenerateScanLoopPhase4(scanOp, region, p, ssize)
13
14     region.add([loop1, loop2, loop3, loop4])
15
16     return region
17
18 def GenerateScanLoopPhase2(scanOp, region, p, ssize):
19     ...
20     outer = Compiler.CreateLoop(
21         0,          # lower bound
22         p,          # upper bound
23         step)
24     inner = Compiler.CreateLoop(
25         inner.iterator * ssize + 1,    # lower bound
26         (outer.iterator + 1) * ssize, # upper bound
27         step)
28     stmt = Compiler.CreateCall(
29         scanOp,          # function
30         scanOp.inputs,   # inputs
31         [region.tmp]);   # outputs
32
33     inner.add(stmt)
34     outer.add(stmt)
35
36     return outer

```

Figure 6.8: Pseudocode of our methodology's code transformation steps for the scan pattern. For brevity, only the overall orchestration (line 1) and second phase processes (line 18) are presented.

to obscure, but enable and ease the use of these patterns as building blocks where the user retains the ability to tune and adapt to their individual needs [58, 171]. In a sense, we exploit this separation between semantics and implementation in the patterns, by moving from the latter to the former. Following that, we apply a set of checks on criteria that will allow us to discern the different semantics exhibited by the transformed code and, thus, identify the corresponding data-parallel pattern. Additionally, we also compare the outputs of the transformed code and original candidate loop and we reject the pattern if they differ. This verification check expands on the notion of SESE regions and the comparison of outputs in the spirit of *liveness-based commutativity* techniques, such as DCA, described in the previous two chapters. In essence, we consider the transformed code as a single SESE region and compare its live-out values with those of the original loop.

As discussed earlier in this section, a further consequence of the transformation following a specific data-parallel pattern is that the generated code will be potentially decomposed to a series of loops or multi-level loops (i.e., loop nests). Intuitively, this occurs due to the fact that parallel patterns provide a division of work and communication for the underlying computation which is aimed at parallel execution. Therefore, it requires execution to occur over the produced segments and their contents which, in a sequential execution context, means groups of loop nests. This also enables the application of our DCA technique (Chapter 5) over each resulting loop and brings into consideration not only its determinations for commutativity, but also non-commutativity. Since DCA focuses on the parallelization of loops, by not satisfying the required commutativity conditions, a loop is not further processed. However, since our objective is to expand the scope of parallelism detection which consists of potentially multiple loop nests, both results offered by DCA are necessary and critical in describing and restricting the specific underlying pattern.

For the decomposed loops in Figure 6.6, this means we are able to apply DCA to each of them in order to determine their commutativity. Note that although almost all of the resulting loops are parallel (except the loop of phase 2), we are not solely interested in discovering their commutativity in order to proceed with parallelization. We take into account both the commutativity or its absence to match the transformed loops to the expected commutativity of the loop implementation of a data-parallel pattern, and thus identify it.

Moreover, we exploit the fact that implementations or parallel patterns are generic with respect to the number of threads involved. Taking this to the extreme of a sequential

implementation, which simply captures the intended sequence of phases as loops, provides two advantages:

- i) it permits us to use DCA, which is geared to operate with sequential programs, without any additional changes, and
- ii) it lets us omit any required synchronization aspects from our transformation templates, which would be required in multithreaded execution, simplifying our transformation templates.

Thus, although we are using a transformation derived from the parallel implementation of each pattern, our detection steps are performed in a sequential execution context. Our objective is to assemble and identify all the loop commutativity criteria which will allow us to correctly identify the underlying pattern present in sequential loop code.

6.3 Case Studies

In this section, we will apply the steps our transformation technique presented in Section 6.2 and examine the requirements that need to be fulfilled to identify specific data-parallel patterns. For clarity, we reiterate here that each pattern which we will try to identify has its own distinct transformation schema and commutativity checking criteria.

6.3.1 Scan Pattern

6.3.1.1 Background

The *scan* pattern computes all partial reductions of a collection of elements. In other words, the scan takes [30, 31]:

- i) a binary operator $\oplus$, with
- ii) an order set $[a_0, a_1, a_2, \dots, a_{n-1}]$, and
- iii) (*output*) returns the ordered set $[a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus \dots \oplus a_{n-1})]$.

For example, a popular instance of this pattern is the *prefix sum* algorithm, presented in Figure 6.1, which calculates, for every output location (i.e., the out array variable), a summation of the input elements up to that point. Addition is the binary operator

```

1  int N = 8;
2  int P = 2
3  int ssize = N / P; // 4
4
5  //          segment 1      segment 2
6  array[] = { 5, 7, 9, 1,  3, 4, 1, 2 };
7  tmp[]   = { 0, 0, 0, 0,  0, 0, 0, 0 };
8  out[]   = { 0, 0, 0, 0,  0, 0, 0, 0 };
9
10 // end of phase 1
11 tmp[] = { 5,  0,  0,  0,  3, 0, 0,  0 };
12 out[] = { 0,  0,  0,  0,  0, 0, 0,  0 };
13
14 // end of phase 2
15 tmp[] = { 5, 12, 21, 22,  3, 7, 8, 10 };
16 out[] = { 0,  0,  0,  0,  0, 0, 0,  0 };
17
18 // end of phase 3
19 tmp[] = { 5, 12, 21, 22,  3, 7, 8, 10 };
20 out[] = { 5, 12, 21, 22,  3, 7, 8, 10 };
21
22 // end of phase 4
23 tmp[] = { 5, 12, 21, 22,  3,  7,  8, 10 };
24 out[] = { 5, 12, 21, 22, 25, 29, 30, 32 };

```

Figure 6.9: Example execution and output of the transformed code in Figure 6.6. The state of intermediate and final results is depicted right after the end of each phase.

(i.e., $\oplus$) taking place in this implementation. The associativity of the binary operator is the property that determines if a scan operation can be parallelized, by reordering operations and some additional work on each step. Due to associativity, any adjacent elements can be scanned (i.e., $a_{k-1} \oplus a_k$) without having to wait for the partially reduced value prior to those elements (i.e., $a_0 \oplus \dots \oplus a_{k-2}$). This computation can happen when that value is ready.

Scans have been identified as parallel building blocks in a wide range of applications, such as [30, 151]: i) the implementation of sorting algorithms (e.g., quicksort, radix sort), ii) polynomial evaluation, iii) solving linear systems (e.g., tridiagonal), iv) searching, v) and parsing.

6.3.1.2 Detection Steps

Applying our transformation steps as described in Section 6.2 in the prefix sum loop of Figure 6.1, we derive the sequential multi-loop code in Figure 6.6. Initially, we divide the input set to (N / P) segments, assuming we had P parallel processing units or threads, at line 1. The first phase copies the leftmost element for each segment from the input array to the intermediate output variable `tmp`, allowing the per segment scanning in the next phase to start with the correct initial values. Then, during the second phase, we compute for each segment of the input computes its partial prefix sum (line 9), using the intermediate `tmp`. Note that the operation at line 16 is essentially the same as the one in Figure 6.1, using a different output variable being the only difference. The third phase, at line 21, we copy the intermediate results from the `tmp` array to the final output variable `out`. Lastly, at the fourth phase at line 28, for every segment, the rightmost value from each intermediate result in `tmp` needs to be added to every value of the output segment `out`. In other words, this phase propagates and scans the partial reduction result of each segment computation to the values of the segment after it.

Figure 6.9 presents an example execution of the transformed code when applied to an example input set. It is worth noting that the first segment in the `tmp` array already has the correct final contents after the second phase is completed. It is this segment's rightmost value (i.e., 22) that needs to be scanned with all the values in the second segment to produce the correct final result (line 24).

Next, moving to the commutativity criteria checking phase, applying DCA on every loop of the three top-level loop nests of the transformed code in Figure 6.9, we obtain that all loops are commutative, except the inner loop of the second phase at line 15. This is expected since this inner loop still exhibits a cross-iteration dependence where the output value depends on the one at the previous index position in the `tmp` array. Hence, the matching criteria for the scan pattern in the transformed code are: i) all loops in the loop nests of all four phases of the transformed code to be commutative when DCA is performed on them, *except* ii) the inner loop of the loop nest in the second phase which should be non-commutative, and iii) the original and transformed code should produce the same output.

6.3.2 Reduction Pattern

6.3.2.1 Background

```

1  out = 0;
2
3  for (i = 0; i < N; ++i) {
4      out += array[i];
5  }

```

(a)

```

1  int ssize = N / P;
2  int array[N], tmp[P], out;
3  int tid;
4
5  array[] = { 5, 7, 9, 1, ... };
6  out[]    = { 0, 0, 0, 0, ... };
7
8  // phase 1
9  for (tid = 0; tid < P; tid++) {
10     tmp[tid] = array[tid * ssize];
11 }
12
13 // phase 2
14 for(int tid = 0; tid < P; tid++) {
15     for(int i = tid * ssize + 1; i < (tid + 1) * ssize; i++) {
16         tmp[tid] += array[i];
17     }
18 }
19
20 // phase 3
21 for(int tid = 0; tid < P; tid++) {
22     out += tmp[tid];
23 }

```

(b)

Figure 6.10: A loop performing a reduction operation (accumulation) over an array (reproduced from Section 4.4) (top). The sequential multi-loop code which results after applying our proposed transformation for the loop in (a) (bottom).

The *reduction* pattern computes a single value by applying an operator over a collection of elements. This combiner operator (or function) is applied in pairwise fashion [87, 171], thus we can define a reduction in terms of:

- i) a binary operator $\oplus$, with

- ii) an order set $[a_0, a_1, a_2, \dots, a_{n-1}]$, and
- iii) (*output*) returns a single value $(a_0 \oplus \dots \oplus a_{n-1})$.

Reductions are closely related to scans since the latter can be expressed in terms of the former, by applying the combiner operator successively over all ordered subsets of the input (i.e., over one element, over two elements, and so on) and keeping the intermediate results. For the same reasons as for scan operations, the associativity of the binary operator is necessary to parallelize reductions. We have already discussed the reasons for which detecting reductions poses a challenge for parallelization techniques based on dependence analysis in the previous two chapters. Typically, such approaches resort to specific and customized extensions within their models [87, 207]. On the other hand, DCA is able to innately identify reduction loops as parallelizable within its *liveness-based commutativity* model. We reuse the reduction loop from our motivation in Chapter 4, presented in Figure 6.10, to exemplify our approach in this section.

Reductions have a plethora of applications in machine learning and scientific domains (i.e., physics, statistics, etc.) [207, 96, 128]. A notable use of this pattern is in the MAPREDUCE parallel programming framework and infrastructure where the reduce operation is performed once for each unique key over records of “Big data” sets [66, 206].

6.3.2.2 Detection Steps

The detection method for reductions proceeds similarly to the previous steps for scans, with the resulting transformed code in Figure 6.10(b). As in the case of scans, we notice the division of the input (i.e., segments of `ssize` each) and the intermediate `tmp` store arrays. This division allows each thread to work independently in an actually parallel implementation. In our version it enables us to apply DCA over each generated loop and check if this independence really exists without affecting the output.

The first phase (line 9) transfers the leftmost element from each segment of the input to each corresponding segment of the intermediate `tmp`. This ensures using the correct initial value for the subsequent computations and, more importantly, avoids requiring to match an operator with its identity value (e.g., 0 for addition). Phase two performs the reduction operation on the input for each segment, storing the result in `tmp`. Each partial reduction reads and writes in a separate index position of `tmp` (line 16). Finally, the third phase performs a final reduction of the partial results from the previous phase to `out` (line 22).

```

1  for (i = 0; i < N; ++i) {
2      out[i] = c * array1[i] + array2[i];
3  }

```

(a)

```

1  int ssize = N / P;
2  float array1[N], array2[N], out[N];
3  int tid;
4
5  array1[] = { ... };
6  array2[] = { ... };
7  out[]    = { ... };
8
9  // phase 1
10 for(int tid = 0; tid < P; tid++) {
11     for(int i = tid * ssize; i < (tid + 1) * ssize; i++) {
12         out[i] = c * array1[i] + array2[i];
13     }
14 }

```

(b)

Figure 6.11: A loop performing a map operation (SAXPY) over an array (top). The sequential multi-loop code which results after applying our proposed transformation for the loop in (a) (bottom).

The matching criteria for the reduction pattern expressed in the transformed code are: i) both loops in the loop nests of all three phases of the transformed code to be commutative when DCA is performed on them, and ii) original and transformed code should produce the same output.

6.3.3 Map Pattern

6.3.3.1 Background

The *map* pattern applies an operation or a function to all the elements in a collection. The map pattern represents computations that have no side effects and, thus, can be performed without requiring any synchronization or communication (embarrassingly parallel). It is often utilized in combination with other patterns such as reductions and

scans (fused patterns) [171]. Figure 6.11(a) presents a simple loop which performs a Single-Precision $A * X + Y$ (SAXPY) map operation, combining vector multiplication and addition, commonly used in various application domains and packaged in BLAS libraries [171].

The map pattern lends itself naturally to the processing of modern parallel multicore architectures. Due to the map’s conceptual simplicity and generality in expressing the effect of independent operations, its use has become ubiquitous in parallel programming frameworks (e.g., MAPREDUCE [66], OPENMP [50], etc.) and, even, standard libraries of languages not necessarily associated with parallelism (e.g., Python’s map function for processing iterable objects). The discovery and exploitation of map parallelism in loops has a long history, from the study of DOALL loops [152] to being the goal of compiler detection tests that utilize dependence [7].

6.3.3.2 Detection Steps

Figure 6.11(b) presents the transformed loop after applying our technique’s transformation steps on the loop in Figure 6.11(a). The input arrays (line 1) are divided into segments and the map operation is applied in turn to each of them (line 12). The simplicity of the map pattern does not require any data transfer between the partial segment calculations, thus the transformed code consists of a single phase.

The matching criteria phase for this pattern expects: i) all loops in the single loop nest of the transformed code to be commutative when DCA is performed on it, and ii) original and transformed code should produce the same output. These conditions are easily comprehensible considering the expected independence of a map’s operation.

6.3.4 Discriminating between Patterns

Table 6.1 summarizes the matching criteria of each phase for each parallel pattern used in our methodology. For every phase of each pattern, every loop in that phase’s loop nest is required to be commutative or non-commutative. In addition to these, we also require and verify that the final outcome of each transformed candidate pattern is the same as that of running the original unmodified sequential loop. We now discuss the ways in which this approach promotes true positives and discourages false positives, in pattern identification.

Table 6.1: Summary of the pattern matching criteria of each phase for our methodology. The ✓ and ✗ signs denote the requirement for a *commutative* and a *non-commutative* loop, respectively. The notation Ln_{ij} corresponds to a loop of phase n placed in i -th position at loop depth 1 and j -th position at loop depth 2.

Patterns	Phase Criteria							
	Phase 1		Phase 2		Phase 3		Phase 4	
Scan	$L1_1$	✓	$L2_1$	✓	$L3_1$	✓	$L4_1$	✓
			$\hookrightarrow L2_{11}$	✗	$\hookrightarrow L3_{11}$	✓	$\hookrightarrow L4_{11}$	✓
							$\hookrightarrow \hookrightarrow L4_{111}$	✓
Reduction	$L1_1$	✓	$L2_1$	✓	$L3_1$	✓	—	
			$\hookrightarrow L2_{11}$	✓				
Map	$L1_1$	✓	—		—		—	
	$\hookrightarrow L1_{11}$	✓						

6.3.4.1 Encouraging True Positives

Using our methodology and transforming a loop based on a schema that has its roots in the parallel forms of common data-parallel patterns, we are able to derive different criteria for discovering these patterns in sequential code, leveraging DCA. Moreover, we utilize not only the existence of loop commutativity for parallelization as in Chapter 5, but also its absence. Consider the criteria for matching the scan pattern, comprising eight loops in total across its four phases in Table 6.1. Under our criteria, loop $L2_{11}$ is expected to be *non-commutative* for correct detection of this pattern since, as described in Section 6.3.1, it performs the local scan operation on each segment of the input. DCA is unable to detect as parallelizable sequential loops which perform scan operations, when examined in isolation (i.e., at the loop level). However, this is a vital feature in our pattern-oriented detection methodology for discovering hidden parallelism. Our methodology’s transformation step is the decisive enabler for: i) multiple applications of DCA, and ii) expansion of the liveness information over a region of code (beyond a single loop nest).

```

1  int saxpy(int in1, int in2) { return 2 * in1 + in2; }
2
3  //          segment 1      segment 2
4  int array1[] = { 5, 7,      9, 1 };
5  int array2[] = { 3, 4,      2, 6 };
6
7  // phase 1
8  ...
9  // end of phase 1
10 tmp[] = { 5, 0, 9, 0 };
11
12 // phase 2
13 for(int tid = 0; tid < 2; tid++) {
14     for(int i = tid * 2 + 1; i < (tid + 1) * 2; i++) {
15         tmp[tid] = saxpy(tmp[tid], array1[i]);
16     }
17 }
18
19 // end of phase 2
20 tmp[0] = saxpy(tmp[0], array[0]) = 2 * 5 + 5 = 15;
21 tmp[0] = saxpy(tmp[0], array[1]) = 2 * 15 + 7 = 37;
22 tmp[1] = saxpy(tmp[1], array[3]) = 2 * 9 + 9 = 27;
23 tmp[1] = saxpy(tmp[1], array[4]) = 2 * 27 + 1 = 55;
24
25 // phase 3
26 ...
27 // end of phase 3
28 out = tmp[0] + tmp[1] = 37 + 55 = 92;
29 // expected correct output from map operation
30 out[] = { 13, 18, 20, 7 };

```

Figure 6.12: Example of applying our reduction transformation steps to a loop employing a map pattern. Our output verification check will disallow matching this loop to the reduction pattern since the output result between the original loop and the transformed code differ (lines 28 and 30). The code from the first and third phases is omitted for clarity.

6.3.4.2 Avoiding False Positives

It is also interesting to consider the ways in which our methodology behaves correctly when a pattern test is applied to a piece of code which actually matches a different

pattern (i.e., how we avoid false positives).

In order to review the robustness of our methodology, we now consider an example of its outcome when we test a candidate loop against a pattern which does not match. For instance, if we apply the reduction schema to a candidate loop that employs the map pattern, then it should produce a transformed program, which once tested, does not meet the criteria for the reduction pattern.

Figure 6.12 presents an example of the transformation result if we suspend the application of the schema criteria on live- and live-outs and assume that a sequential loop that performs a map operation gets transformed with the reduction transform schema. Figure 6.12 combines parts of the transformed code and its output, omitting the first and third phase for clarity and uses an instance of the SAXPY map presented earlier in this section (line 1). We notice that treating the extracted saxpy operation in the second phase (line 15) as a reduction operator yields incorrect results that propagate to the final out variable (lines 28 and 30). Even if we ignore the difference in output size (i.e., scalar versus array), the final results differ and will fail our output verification.

6.4 Summary and Conclusions

In this chapter, we have presented a methodology which utilizes *liveness-based commutativity*, as implemented in DCA, and facilitates the detection of parallelism in the form of patterns in sequential loop code.

Our methodology proceeds by applying a series of transformation to a candidate loop to obtain a code section consisting of multiple loops. The transformation steps convert the loop based on a schema that is derived from the parallel implementation of each pattern we aim to discover, but repurposed for sequential execution which avoids the need for any synchronization. This allows us to leverage DCA by applying it separately on each resulting loop of the transformed code. Then, we compare DCA's determinations for each loop, retaining results for both commutativity and *non-commutativity*, with an expected set of criteria in order to match the candidate loop to pattern.

Using this approach, we are able to discover data-parallel patterns such as maps, reductions and scans. The detection of loops performing a scan operation is a step towards extending the efficacy of *liveness-based commutativity* for parallelization, since, when examined using simple DCA, they are not indicated as parallelizable. Despite its simplicity, our transformation avoids probable misclassifications by utilizing an output

verification step that rejects a pattern if results differ from the original candidate loop. Future work will aim to automate and enhance the transformation schema for each pattern.

Conclusions

This chapter concludes the presentation of our work in this thesis.

We start with a synopsis of our investigations in Section 7.1 and continue with a summary of our contributions in Section 7.2. We conclude in Section 7.3 with a critical analysis of our work and explore potential future directions.

7.1 Introduction

In this thesis, we have examined the detection and extraction of parallelism from sequential source code.

To this end, we use *liveness-based commutativity* as the main enabler for the techniques developed and examined in our work. Commutativity is the property of a code region whose order of execution we can permute without affecting the final program outcome. *Liveness-based commutativity* utilizes information from the classic compiler liveness analysis in order to determine whether a program’s outcome is altered or not by reordering the execution of code regions.

Our study has been aimed at loops, where applications usually spend most of their execution time. The candidate loops that we are interested in exhibit array-based access idioms, found mostly in the HPC and scientific domains, and pointer-based loop chasing idioms, a hallmark of applications that utilize PLDS. Consequently, we have explored the potential of a technique built on top of symbolic execution, combining symbolic and concrete input values. Identifying the limitations of this approach, we then developed a new technique that relies on profiling information to detect latent loop-level parallelism. Finally, we investigate the use of *liveness-based commutativity* in identifying higher-level parallel algorithmic skeletons in sequential code and thereby extending its scope

of detection.

7.2 Contributions

In Chapter 4, we introduced our concept of *liveness-based commutativity* and adapted it for loops in order to express and, consequently, test commutativity conditions between their iterations. We developed a method to examine loop commutativity conditions based on liveness leveraging symbolic execution, which allows a concise representation of inputs as value ranges (i.e., symbolic values) expressed using sets of constraints. We structured the loop commutativity testing in such way that it can be expressed as a path constraint that can be checked for equivalence using a symbolic execution engine’s constraint solver. For our case studies we selected representative loops from NPB suite and other sources (e.g., OLDEN benchmarks) which follow common iteration idioms over arrays and PLDS.

Despite the fact that symbolic execution presents an attractive and elegant process of expressing outputs in terms of constraints of inputs, thus enabling the extraction and derivation of various program properties, in practice it is limited due to scalability issues. Despite having taken actions to remedy some of these issues (e.g., limited coverage, etc.), our case studies demonstrate its applicability is inhibited by features existing in real-world programs. Informed by these findings, we then seek to alleviate these shortcomings using a program’s execution with actual input sets as an additional information source.

In Chapter 5, we introduced DCA, a hybrid static and dynamic technique based on *liveness-based commutativity* which utilizes profiling information, derived from execution runs using provided input sets, to detect and exploit loop parallelism. For our evaluation, we used almost 1400 loops across 10 applications in the NPB suite which perform array-based computations for various scientific applications. We also targeted PLDS-based loops, which have been studied previously in the literature, from a wide variety of benchmarks, such as the SPEC CPU and OLDEN suites among others. We compared DCA to five state-of-the-art dependence-based approaches, matching or outperforming their detection capabilities using *liveness-based commutativity* as our single analysis model.

For the NPB programs, we found that DCA is able to match the detection capabilities of two other dynamic approaches based on dependence analysis, discovering over 1200 loops (86% of the total) as parallelizable. When compared to three static compiler

techniques over the same suite, DCA found twice as many parallelizable loops as all of them *combined*.

For PLDS-based loops, we have drawn upon the literature to collect a wide range of loops in programs that use pointer-based data structures. The approaches described in those works were focused on the exploitation of PLDS-based loops on parallel architectures and relied on *ad hoc* methods to select them. On the other hand, DCA is able to detect automatically all these loops as potentially parallelizable, going beyond the capacity of all dependence-based tools used as our baseline. DCA is able to handle *uniformly* loops using regular, array-based induction variables and pointer-chasing idioms.

For the exploitation of the discovered parallelism, we employed a simple parallelization approach, based on [248], that utilizes the profiling information collected during detection to also select the hottest (i.e., the highest sequential execution coverage) loops. We were able to obtain an average speedup of $3.6\times$ across NPB and up to $55\times$ on our test platform. For PLDS-based loops, we achieved speedups that reached up to $36\times$. DCA can be used orthogonally with more sophisticated parallel code generation schemes as discussed in Section 7.3.

Moreover, since DCA requires user input to approve results, a common aspect of profile-guided approaches, we have assessed its accuracy. We found no false positives in our technique’s predictions, hence, all discovered loops are indeed parallelizable for all valid inputs. Lastly, the NPB suite gives us the opportunity to compare our parallelization against that performed manually by experts. At the loop-level, DCA was able to extract all available parallelism, with all the remaining parallelism uncovered by manual transformations being outside our scope (i.e., extending beyond and across loops).

In Chapter 6, we developed a methodology, based on *liveness-based commutativity* and leveraging DCA, to further extend to scope of parallelism discovery in sequential code. Our methodology transforms a candidate loop, based on a schema, into a code section comprising multiple loops, applies DCA on them, and compares with a set of criteria corresponding to the commutativity properties of various data-parallel patterns. Applying these steps over sequential loops, we are able to identify specific forms of pattern-based parallelism in sequential code. This allows us to extend the scope of detection with the identification of patterns, such as scans, that DCA (and dependence analysis) is not able to find as parallelizable. This methodology not only utilizes the existence of commutativity, but also its absence in the specification of the matching

criteria for each pattern.

7.3 Critical Review and Future Directions

7.3.1 Liveness-Based Commutativity Analyses

In Chapters 4 and 5, we presented and evaluated two techniques which leveraged *liveness-based commutativity* for parallelization leveraging information derived from symbolic and dynamic execution contexts.

Currently, the overheads of *liveness-based commutativity* () using symbolic execution are very limiting for any useful determination regarding the examined loops due to the inherent path explosion associated with iteration exploration. While the characterization of inputs with symbolic classes is attractive, there is no clear approach on how this can be further lifted from low-level ranges, based on the underlying variable types, tailored without domain and problem knowledge from the user. Further, the user is required to be heavily involved during the whole process of commutativity checking, from input set manipulation to code fragment extraction and verification of results. Lastly, fundamental modelling limitations in symbolic execution (i.e., floating-point numbers) continue to pose an unresolved challenge that impacts the application of any such approach to codebases and workloads that employ them, such as HPC and numerical analysis.

One future research path would be to investigate the use of program slicing techniques in order to alleviate the shortcomings of symbolic execution when used with *liveness-based commutativity*. Program slicing [270] is a technique to determine the set of statements in a program (i.e., the program slice), which may affect the values at some program region of interest (i.e., slicing criterion). The usual application of these techniques is in debugging, but their uses have been expanded, producing multiple variants since its inception [278]. Slicing could be used to reduce and simplify a candidate program region to be tested for commutativity conditions.

Fuzzing, or fuzz testing, is a technique for the fast, automatic generation of inputs to test programs for security vulnerabilities [157, 97]. Recently published research has started investigating the combination of symbolic execution in tandem with fuzzing in order to overcome each other's limitations (whitebox fuzzing) [105, 190]. Briefly, the main challenge in fuzzing is generating such inputs that thoroughly exercise the intended program sections under test. In the context of implementing *liveness-based*

commutativity analyses, fuzzing could help selectively concretize values that could lead to a fast fail of the commutativity conditions tests. Another potential use of fuzzing, that could widen the applicability of symbolic execution, would be to direct its input generation towards parallelization-specific goals (e.g., discovering what values could cause fast failing of commutativity tests). Recent research has also examined the use of a more targeted application of symbolic execution by allowing the user to specify regions of code that can be excluded, thus focusing on the exploration of important paths [250].

Symbolic execution could also be used in an indirect manner, as a mechanism of specification inference mechanism for program parallelization, treating parallelization-inhibiting dependences as program defects. While this also suffers from scalability issues, there have been adaptations which have worked towards mitigating such challenges in other application fields, such as program repair [229], and could make future work on this aspect more tractable.

Several opportunities for future research lie in the exploration of combining DCA with other dynamic analyses.

In terms of parallelization discovery, DCA is sensitive to the potential of combinatorial explosion, due to long iteration counts, potentially deep loop nest structures and different execution contexts (e.g., different call sites of the containing function). Currently, there is no consideration in mitigating the associated overheads apart from performing independent loop invocations. There is also no attempt to reuse or compress profiling information across loops in a loop nest or execution contexts. The parallelization safety of the discovered loops is not guaranteed since it is based on profiling information using a finite input set. Hence, the user involvement in the decision to proceed with parallelization requires expert knowledge. Lastly, this information is presented in a low-level format, exposing the data dependences between variables, making this task onerous.

Probabilistic techniques can help reduce the overhead of the dynamic analysis created by the requirement to permute the iteration space of a loop. One such potential technique is loop perforation [232] which could be utilized to reduce the amount of loop iterations executed and hence the overall computational workload. While the perforated result differs from the original, its use in computations that can tolerate such changes (e.g., ML-based algorithms) and regulate trade-offs between efficiency and accuracy, can be acceptable and beneficial. This difference in results from algorithmic changes or computation reordering (e.g., parallelization) is not new in computing. For

example, developers of HPC applications already deal with the heavy use of floating-point arithmetic, where experts have included error threshold checks (e.g., the NPB suite provides a verification step for each program). However, perforation algorithms actively search the space of the generated variants for optimality (i.e., maximize performance for a specific acceptable accuracy loss).

Another set of related techniques that could alleviate the overhead of iteration reordering are those that enable program execution replay. These techniques record enough information from the execution of a program about its behavior, so that a second execution can loyally replay the behavior, producing the same result [183, 256, 182]. Typically, the objective is to enable debugging (e.g., reverse-execution debugging) or resolve difficult to reproduce faults. However, there have been techniques that aim to reproduce specific program properties for performance such as [49], where code regions are extracted and executed in isolation using an identified representative working state (i.e., memory and cache). This could allow an extension of DCA that could accurately test commutativity conditions for individual loop nests.

In Chapter 2, we briefly presented speculative parallelization approaches more closely related to our work. We consider such approaches orthogonal to DCA’s detection capabilities, thus speculation can be leveraged for the actual parallel code generation stage of the analysis. DCA is an Ahead-of-Time (AoT) analysis with zero overheads during the program’s parallel execution. However, if the two approaches are used in conjunction, then speculative execution can benefit from a reduction of the associated runtime costs (Chapter 2) and DCA can completely avoid user involvement (i.e., fully automatic). The manner in which such a combination could prove profitable is a further avenue for future research.

Moreover, although we did not use any specific hardware features, there are ways that architectural support can assist in our proposed techniques, such as Transactional Memory (TM) or even *Value Prediction*, which has received renewed interest from the community following the advent of processors with a variety of general-purpose cores [78]. For instance, if the path of speculative parallelization is investigated, there is a body of research that has studied ways in which TM can help in the efficient maintenance of speculative state and rollback implementation [218].

As we presented in Chapter 5, DCA relies on concrete inputs to execute a target application and retrieve the required values through the generated profiles. While the results are valid for those specific input sets, we have also explored the precision of the predictions made by our analysis and found that they are adequate and hold for all

potential valid inputs.

An interesting future research direction would be to explore the generation of representative program input sets. Another way would be an approach similar to [28] used in the context of Feedback-Directed Optimization (FDO), where profiles from several program runs are combined in a statistically sound method, supporting both offline and online information collection. Similarly, in [77], the effect of different input sets is measured using statistical analysis and the results are clustered together based on the program behavior that they induce. Techniques like DCA, which rely on *liveness-based commutativity* could potentially present good candidates for such statistical approaches. This is because their results are based on a candidate region's outcome, unlike dependence-based methods, where a single violating read/write operation pair between two memory locations is enough to stop parallelization.

In our implementation, the main focus of profiling was the extraction of adequate information in support of DCA's requirements. Despite that, future revisions can incorporate techniques for the efficient storage [179] (e.g., using compression) or collection of traces [144]. For example, SD3 [144] parallelizes the gathering of profiling information itself and could be applied orthogonally.

Lastly, even though parallelization detection using DCA can be applied to each individual loop in a loop nest, our current parallelization scheme exploits only the outermost loops. This strategy typically avoids the costs associated with setup and synchronization of parallelism which typically negate any speedup gains from parallelization when performed on inner loops [50]. Exploring the use of commutativity with other parallel code generation techniques, more suitable for inner-loop parallelism such as vectorization [7], could further expand DCA's applicability and advance its integration along other compiler transformations and optimizations. Deciding on the granularity of parallelism, its form and the order which these should be optimally combined is a complicated problem which could also benefit from the use of ML-based techniques for parallel code generation [248].

7.3.2 Pattern Detection with Liveness-based Commutativity

In Chapter 6, we presented a methodology to identify data-parallel patterns in sequential code leveraging DCA, thus extending the scope of our discovery capacity.

The current implementation is limited to data-parallel patterns which are sought using an inflexible detection scheme, which uses a combination of compiler transfor-

mations and code templates. Furthermore, the user is required to design, implement and verify the results of the commutativity for each decomposed loop in a pattern under detection and also check the correctness of the match with the expected criteria for it. While DCA is able to handle PLDS-based loops, pattern detection with has not been expanded to attempt discovery in such codes. It is unclear how our technique could be expanded to address combination of data-parallel patterns (fused or composable variations).

Future work can examine if our methodology can be expanded to include the identification of other skeletons (e.g., stencils, recurrences, etc.) and provide a unified way to describe them. This could allow the description of patterns by the users and facilitate a holistic integration, from detection to code generation, by taking advantage of parallel programming frameworks (e.g., SKEPU [79]). Using skeleton frameworks can alleviate the task of mapping and tuning the extracted parallelism to the target parallel architectures.

Enhancing the results of commutativity with shape analysis [93] could further amplify parallelism and pattern detection beyond the scope of loops. Additionally, incorporating methods which automatically discover and verify commutativity conditions for data structures [146] could limit the reliance of profiling information and reduce or eliminate user involvement in the process of parallelization.

Another facet of our approach is the application and search for every available pattern description at our disposal. At the moment, the interaction of each pattern transformation and matching occurs in isolation from each other. Implementing a specific order in the application of each pattern could obviate duplication of effort in parts of our technique (e.g., the transformation of and criteria matching for the first phase each pattern). However, this is an orthogonal aspect, since the apparent inefficiency could be offset by parallelizing these steps in our methodology while retaining the isolation of each pattern specification.

Lastly, several parallel patterns appear in fused forms in many parallel programming frameworks [171]. These special cases are combinations of multiple patterns used in chained succession (e.g., map-reduction or map-scan fusions) where one consumes the output of the previous pattern. The fused patterns tend to improve the efficiency of parallel execution by reducing communication and synchronization overheads [171]. Our approach does not contain descriptions of fused pattern variants, however, this could be remedied by:

- i) providing distinct descriptions for fused variants, or

- ii) adding decomposition stages to separate such occurrences, thus enabling their detection with the currently available descriptions.

7.3.3 User Interface Improvements

The current medium of user interaction in DCA is through low-level reports which were originally designed to assist us in identifying individual candidate loops and be easily parsed by our development utilities. As a result, this output is not well-suited for consumption by developers and used as a productivity booster. Despite this, it is possible to extend and enhance the current reports and user feedback, in similar fashion to tools such as [161, 288], to:

- i) Enable association between the results and the original source code by using source-level debugging information (e.g., a format such as DWARF [75]). This can further enable a two-way communication and update between reports and source code via an Integrated Development Environment (IDE) plugin or other external tools in the development software ecosystem.
- ii) Exploit the liveness information used in *liveness-based commutativity* by DCA to focus and filter the operations and data flow for the candidate loops presented to the user. This can occur in combination with visualizing dependence information, since this can help in validating access synchronization to shared variables.
- iii) The involvement of the user can follow a plan, be more guided and minimize the number of required user tweaks. For example, an advisor-like tool can suggest a specific set of steps to take in order to validate DCA's transformations. Such a tool could be powered by preset user configurations, heuristics or ML-based predictions.

Lastly, all the aforementioned suggestions can also be combined at various degrees to improve the overall user experience.

Bibliography

- [1] Vikas Agarwal et al. “Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures”. In: *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ISCA '00. New York, NY, USA: Association for Computing Machinery, May 2000, pp. 248–259. ISBN: 978-1-58113-232-8. DOI: [10.1145/339647.339691](https://doi.org/10.1145/339647.339691).
- [2] Alfred V. Aho, ed. *Compilers: Principles, Techniques, and Tools*. 2. ed, Pearson new intern. ed. Harlow: Pearson, 2014. ISBN: 978-1-292-02434-9 1-292-02434-8 978-1-292-02434-9.
- [3] Farhana Aleen and Nathan 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. New York, NY, USA: ACM, 2009, pp. 241–252. ISBN: 978-1-60558-406-5. DOI: [10.1145/1508244.1508273](https://doi.org/10.1145/1508244.1508273).
- [4] J. R. Allen et al. “Conversion of Control Dependence to Data Dependence”. In: *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '83. New York, NY, USA: ACM, 1983, pp. 177–189. ISBN: 978-0-89791-090-3. DOI: [10.1145/567067.567085](https://doi.org/10.1145/567067.567085).
- [5] John R. Allen and Ken Kennedy. “PFC: A Program to Convert Fortran to Parallel Form”. In: (Mar. 1982). <https://scholarship.rice.edu/handle/1911/101547>.
- [6] John Randal Allen. “Dependence Analysis for Subscripted Variable and Its Application To Program Transformations”. <https://scholarship.rice.edu/handle/1911/19045>. Thesis. 1983.

- [7] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. 1 edition. San Francisco: Morgan Kaufmann Publishers In, Mar. 2000. ISBN: 978-1-55860-286-1.
- [8] Martin Alt, Uwe Aßmann, and Hans van Someren. “Cosy Compiler Phase Embedding with the CoSy Compiler Model”. In: *Compiler Construction*. Ed. by Peter A. Fritzson. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1994, pp. 278–293. ISBN: 978-3-540-48371-7. DOI: [10.1007/3-540-57877-3_19](https://doi.org/10.1007/3-540-57877-3_19).
- [9] Saman Amarasinghe. “Why Compilers Have Failed to Support HPC Programmers and What Can We Do about It”. In: *ASCR Programming Challenges Workshop*. 2011.
- [10] Sotiris Apostolakis et al. “Perspective: A Sensible Approach to Speculative Automatic Parallelization”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, Mar. 2020, pp. 351–367. ISBN: 978-1-4503-7102-5. DOI: [10.1145/3373376.3378458](https://doi.org/10.1145/3373376.3378458).
- [11] Manuel Arenaz, Juan Touriño, and Ramon Doallo. “XARK: An Extensible Framework for Automatic Recognition of Computational Kernels”. In: *ACM Trans. Program. Lang. Syst.* 30.6 (Oct. 2008), 32:1–32:56. DOI: [10.1145/1391956.1391959](https://doi.org/10.1145/1391956.1391959).
- [12] Krste Asanovic et al. “A View of the Parallel Computing Landscape”. In: *Commun ACM* 52.10 (Oct. 2009), pp. 56–67. DOI: [10.1145/1562764.1562783](https://doi.org/10.1145/1562764.1562783).
- [13] Amir H. Ashouri et al. “A Survey on Compiler Autotuning Using Machine Learning”. In: *ACM Comput. Surv.* 51.5 (Sept. 2018), 96:1–96:42. DOI: [10.1145/3197978](https://doi.org/10.1145/3197978).
- [14] T.M. Austin and G.S. Sohi. “Dynamic Dependency Analysis of Ordinary Programs”. In: *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*. May 1992, pp. 342–351. DOI: [10.1109/ISCA.1992.753330](https://doi.org/10.1109/ISCA.1992.753330).
- [15] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. “Efficient Detection of All Pointer and Array Access Errors”. In: *SIGPLAN Not.* 29.6 (June 1994), pp. 290–301. DOI: [10.1145/773473.178446](https://doi.org/10.1145/773473.178446).

- [16] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. “Chains of Recurrences—a Method to Expedite the Evaluation of Closed-Form Functions”. In: *Proceedings of the International Symposium on Symbolic and Algebraic Computation*. ISSAC '94. New York, NY, USA: ACM, 1994, pp. 242–249. ISBN: 978-0-89791-638-7. DOI: [10.1145/190347.190423](https://doi.org/10.1145/190347.190423).
- [17] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. “Compiler Transformations for High-Performance Computing”. In: *ACM Comput Surv* 26.4 (Dec. 1994), pp. 345–420. DOI: [10.1145/197405.197406](https://doi.org/10.1145/197405.197406).
- [18] Tongxin Bai, Chen Ding, and Pengcheng Li. “Assessing Safe Task Parallelism in SPEC 2006 INT”. In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. May 2015, pp. 402–411. DOI: [10.1109/CCGrid.2015.159](https://doi.org/10.1109/CCGrid.2015.159).
- [19] D. H. Bailey et al. “The NAS Parallel Benchmarks”. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing - Supercomputing '91*. Albuquerque, New Mexico, United States: ACM Press, 1991, pp. 158–165. ISBN: 978-0-89791-459-8. DOI: [10.1145/125826.125925](https://doi.org/10.1145/125826.125925).
- [20] Roberto Baldoni et al. “A Survey of Symbolic Execution Techniques”. In: *ACM Comput. Surv. CSUR* 51.3 (July 2018), p. 50. DOI: [10.1145/3182657](https://doi.org/10.1145/3182657).
- [21] Utpal K. Banerjee. *Data Dependence in Ordinary Programs*. Urbana: Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1976.
- [22] Utpal K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. USA: Kluwer Academic Publishers, 1993. ISBN: 978-0-7923-9318-4.
- [23] Kshitij Bansal, Eric Koskinen, and Omer Tripp. “Automatic Generation of Precise and Useful Commutativity Conditions (Extended Version)”. In: *ArXiv180208748 Cs* (Feb. 2018). <http://arxiv.org/abs/1802.08748>. arXiv: [1802.08748 \[cs\]](https://arxiv.org/abs/1802.08748).
- [24] Mohamed-Walid Benabderrahmane et al. “The Polyhedral Model Is More Widely Applicable Than You Think”. In: *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*. CC'10/ETAPS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 283–303. ISBN: 978-3-642-11969-9. DOI: [10.1007/978-3-642-11970-5_16](https://doi.org/10.1007/978-3-642-11970-5_16).

- [25] Josh Berdine et al. “Shape Analysis for Composite Data Structures”. In: *Proceedings of the 19th International Conference on Computer Aided Verification*. CAV’07. <http://dl.acm.org/citation.cfm?id=1770351.1770381>. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 178–192. ISBN: 978-3-540-73367-6.
- [26] Marc Berndt and Laurie Hendren. “Dynamic Profiling and Trace Cache Generation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO ’03. San Francisco, California, USA: IEEE Computer Society, Mar. 2003, pp. 276–285. ISBN: 978-0-7695-1913-5.
- [27] A.J. Bernstein. “Analysis of Programs for Parallel Processing”. In: *IEEE Trans. Electron. Comput.* EC-15.5 (Oct. 1966), pp. 757–763. DOI: [10.1109/PGEC.1966.264565](https://doi.org/10.1109/PGEC.1966.264565).
- [28] Paul Berube, Adam Preuss, and Jose Nelson 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. New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 493–498. ISBN: 978-1-4503-0519-8. DOI: [10.1145/1958746.1958821](https://doi.org/10.1145/1958746.1958821).
- [29] Chris Birchall. *Re-Engineering Legacy Software*. Shelter Island, NY: Manning Publications Co, 2016. ISBN: 978-1-61729-250-7.
- [30] G. E. Blelloch. “Scans as Primitive Parallel Operations”. In: *IEEE Trans. Comput.* 38.11 (Nov. 1989), pp. 1526–1538. DOI: [10.1109/12.42122](https://doi.org/10.1109/12.42122).
- [31] Guy E. Blelloch. “Prefix Sums and Their Applications”. In: (2004), 1294199 Bytes. DOI: [10.1184/R1/6608579.V1](https://doi.org/10.1184/R1/6608579.V1).
- [32] William Blume et al. “Polaris: Improving the Effectiveness of Parallelizing Compilers”. In: *Languages and Compilers for Parallel Computing*. Ed. by Keshav Pingali et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1995, pp. 141–154. ISBN: 978-3-540-49134-7. DOI: [10.1007/BFb0025876](https://doi.org/10.1007/BFb0025876).
- [33] Eric Bodden et al. “Collaborative Runtime Verification with Tracematches”. In: *Proceedings of the 7th International Conference on Runtime Verification*. RV’07. Vancouver, Canada: Springer-Verlag, Mar. 2007, pp. 22–37. ISBN: 978-3-540-77394-8.

- [34] Uday Bondhugula et al. “A Practical Automatic Polyhedral Parallelizer and Locality Optimizer”. In: *SIGPLAN Not.* 43.6 (June 2008), pp. 101–113. DOI: [10.1145/1379022.1375595](https://doi.org/10.1145/1379022.1375595).
- [35] Shekhar Borkar. “Getting Gigascale Chips: Challenges and Opportunities in Continuing Moore’s Law”. In: *Queue* 1.7 (Oct. 2003), pp. 26–33. DOI: [10.1145/957717.957757](https://doi.org/10.1145/957717.957757).
- [36] Adam Bosworth et al. “Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals”. In: (Feb. 1995). <https://www.microsoft.com/en-us/research/publication/data-cube-a-relational-aggregation-operator-generalizing-group-by-cross-tab-and-sub-totals/>.
- [37] T. Brandes et al. “HPFIT: A Set of Integrated Tools for the Parallelization of Applications Using High Performance Fortran. Part I: HPFIT and the TransTOOL Environment”. In: *Parallel Computing. Environment and Tools for Parallel Scientific Computing* 23.1 (Apr. 1997), pp. 71–87. DOI: [10.1016/S0167-8191\(96\)00097-X](https://doi.org/10.1016/S0167-8191(96)00097-X).
- [38] Matthew Bridges et al. “Revisiting the Sequential Programming Model for Multi-Core”. In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 40. USA: IEEE Computer Society, Dec. 2007, pp. 69–84. ISBN: 978-0-7695-3047-5. DOI: [10.1109/MICRO.2007.35](https://doi.org/10.1109/MICRO.2007.35).
- [39] Amy Brown and Greg Wilson. *The Architecture Of Open Source Applications*. Mountain View: lulu.com, Mar. 2012. ISBN: 978-1-257-63801-7.
- [40] Juan Manuel Caamaño et al. “Full Runtime Polyhedral Optimizing Loop Transformations with the Generation, Instantiation, and Scheduling of Code-Bones”. In: *Concurr. Comput. Pract. Exp.* 29.15 (2017), e4192. DOI: [10.1002/cpe.4192](https://doi.org/10.1002/cpe.4192).
- [41] Juan Manuel Martinez Caamaño et al. “APOLLO: Automatic Speculative POLyhedral Loop Optimizer”. In: *IMPACT 2017 - 7th International Workshop on Polyhedral Compilation Techniques*. <https://hal.inria.fr/hal-01533692>. Jan. 2017, p. 8.
- [42] Cristian Cadar. *[Klee-Dev] KLEE Floating-Point Support*. <http://mailman.ic.ac.uk/pipermail/klee-dev/2020-November/002053.html>. Tue Nov 24 10:06:56 GMT 2020.

- [43] Cristian Cadar. “Targeted Program Transformations for Symbolic Execution”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, Aug. 2015, pp. 906–909. ISBN: 978-1-4503-3675-8. DOI: [10.1145/2786805.2803205](https://doi.org/10.1145/2786805.2803205).
- [44] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI’08. <http://dl.acm.org/citation.cfm?id=1855741.1855756>. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.
- [45] Cristian Cadar and Koushik Sen. “Symbolic Execution for Software Testing: Three Decades Later”. In: *Commun. ACM* 56.2 (Feb. 2013), p. 82. DOI: [10.1145/2408776.2408795](https://doi.org/10.1145/2408776.2408795).
- [46] Simone Campanoni et al. “HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing”. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization - CGO ’12*. San Jose, California: ACM Press, 2012, p. 84. ISBN: 978-1-4503-1206-6. DOI: [10.1145/2259016.2259028](https://doi.org/10.1145/2259016.2259028).
- [47] Martin C. Carlisle et al. “Early Experiences with Olden”. In: *Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Aug. 1993, pp. 1–20. ISBN: 978-3-540-57659-4. DOI: [10.1007/3-540-57659-2_1](https://doi.org/10.1007/3-540-57659-2_1).
- [48] Martin Christopher Carlisle. “Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines”. PhD Thesis. Princeton, NJ, USA: Princeton University, 1996.
- [49] Pablo De Oliveira Castro et al. “CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization”. In: *ACM Trans Arch. Code Optim* 12.1 (Apr. 2015), 6:1–6:24. DOI: [10.1145/2724717](https://doi.org/10.1145/2724717).
- [50] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. Scientific and Engineering Computation. Cambridge, Mass: MIT Press, 2008. ISBN: 978-0-262-53302-7 978-0-262-03377-0.

- [51] Arthur Charlesworth. “The Undecidability of Associativity and Commutativity Analysis”. In: *ACM Trans Program Lang Syst* 24.5 (Sept. 2002), pp. 554–565. DOI: [10.1145/570886.570889](https://doi.org/10.1145/570886.570889).
- [52] Lorenzo Chelini et al. “Progressive Raising in Multi-Level IR”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Feb. 2021, pp. 15–26. DOI: [10.1109/CGO51591.2021.9370332](https://doi.org/10.1109/CGO51591.2021.9370332).
- [53] Doreen Y. Cheng and Douglas M. Pase. “An Evaluation of Automatic and Interactive Parallel Programming Tools”. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing '91. Albuquerque, New Mexico, USA: Association for Computing Machinery, Aug. 1991, pp. 412–423. ISBN: 978-0-89791-459-8. DOI: [10.1145/125826.126052](https://doi.org/10.1145/125826.126052).
- [54] Austin T. Clements et al. “The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors”. In: *ACM Trans. Comput. Syst.* 32.4 (Jan. 2015), 10:1–10:47. DOI: [10.1145/2699681](https://doi.org/10.1145/2699681).
- [55] *COBOL Blues*. <http://fingfx.thomsonreuters.com/gfx/rngs/USA-BANKS-COBOL/010040KH18J/index.html>.
- [56] Albert Cohen, Sylvain Girbal, and Olivier Temam. “A Polyhedral Approach to Ease the Composition of Program Transformations”. In: *Euro-Par 2004 Parallel Processing*. Ed. by Marco Danelutto, Marco Vanneschi, and Domenico Laforenza. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 292–303. ISBN: 978-3-540-27866-5. DOI: [10.1007/978-3-540-27866-5_38](https://doi.org/10.1007/978-3-540-27866-5_38).
- [57] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. London : Cambridge, Mass: Pitman ; MIT Press, 1989. ISBN: 978-0-273-08807-3.
- [58] Murray Cole. “Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming”. In: *Parallel Comput* 30.3 (Mar. 2004), pp. 389–406. DOI: [10.1016/j.parco.2003.12.002](https://doi.org/10.1016/j.parco.2003.12.002).
- [59] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. “Symbolic Cross-checking of Floating-Point and SIMD Code”. In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 315–328. ISBN: 978-1-4503-0634-8. DOI: [10.1145/1966445.1966475](https://doi.org/10.1145/1966445.1966475).

- [60] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. 2nd ed. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2012. ISBN: 978-0-12-088478-0.
- [61] Emilio Coppa, Daniele Cono D’Elia, and Camil Demetrescu. “Rethinking Pointer Reasoning in Symbolic Execution”. In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. <http://dl.acm.org/citation.cfm?id=3155562.3155638>. Piscataway, NJ, USA: IEEE Press, 2017, pp. 613–618. ISBN: 978-1-5386-2684-9.
- [62] William J. Dally, Yatish Turakhia, and Song Han. “Domain-Specific Hardware Accelerators”. In: *Commun. ACM* 63.7 (June 2020), pp. 48–57. DOI: [10.1145/3361682](https://doi.org/10.1145/3361682).
- [63] Marco Danelutto et al. “Algorithmic Skeletons and Parallel Design Patterns in Mainstream Parallel Programming”. In: *Int J Parallel Prog* 49.2 (Apr. 2021), pp. 177–198. DOI: [10.1007/s10766-020-00684-w](https://doi.org/10.1007/s10766-020-00684-w).
- [64] Andrew Danowitz et al. “CPU DB: Recording Microprocessor History: With This Open Database, You Can Mine Microprocessor Trends over the Past 40 Years.” In: *Queue* 10.4 (Apr. 2012), pp. 10–27. DOI: [10.1145/2181796.2181798](https://doi.org/10.1145/2181796.2181798).
- [65] *Data Parallel C++ for Cross-Architecture Applications*. <https://www.intel.com/content/www/us/en/develop/tools/oneapi/components/dpc-compiler.html>.
- [66] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Commun. ACM* 51.1 (Jan. 2008), p. 107. DOI: [10.1145/1327452.1327492](https://doi.org/10.1145/1327452.1327492).
- [67] John Delaney. *COBOL Programmers Are Back In Demand. Seriously*. <https://cacm.acm.org/news/244370-cobol-programmers-are-back-in-demand-seriously/fulltext>. Apr. 2020.
- [68] B. Di Martino and G. Iannello. “PAP Recognizer: A Tool for Automatic Recognition of Parallelizable Patterns”. In: *4th Workshop on Program Comprehension WPC ’96*. Mar. 1996, pp. 164–174. DOI: [10.1109/WPC.1996.501131](https://doi.org/10.1109/WPC.1996.501131).
- [69] Danny Dig. “A Refactoring Approach to Parallelism”. In: *IEEE Softw.* 28.1 (Jan. 2011), pp. 17–22. DOI: [10.1109/MS.2011.1](https://doi.org/10.1109/MS.2011.1).

- [70] Chen Ding et al. “Software Behavior Oriented Parallelization”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: Association for Computing Machinery, June 2007, pp. 223–234. ISBN: 978-1-59593-633-2. DOI: [10.1145/1250734.1250760](https://doi.org/10.1145/1250734.1250760).
- [71] Johannes Doerfert, Tobias Grosser, and Sebastian Hack. “Optimistic Loop Optimization”. In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO '17. IEEE Press, 2017, pp. 292–304. ISBN: 978-1-5090-4931-8.
- [72] Johannes Doerfert et al. “Polly’s Polyhedral Scheduling in the Presence of Reductions”. In: *ArXiv150507716 Cs* (May 2015). <http://arxiv.org/abs/1505.07716>. arXiv: [1505.07716 \[cs\]](https://arxiv.org/abs/1505.07716).
- [73] Jialin Dou and Marcelo Cintra. “Compiler Estimation of Load Imbalance Overhead in Speculative Parallelization”. In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. PACT '04. USA: IEEE Computer Society, Sept. 2004, pp. 203–214. ISBN: 978-0-7695-2229-6.
- [74] Bruno Dufour et al. “Dynamic Metrics for Java”. In: *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '03. Anaheim, California, USA: Association for Computing Machinery, Oct. 2003, pp. 149–168. ISBN: 978-1-58113-712-5. DOI: [10.1145/949305.949320](https://doi.org/10.1145/949305.949320).
- [75] *Dwarf Home*. <http://dwarfstd.org/>.
- [76] Tobias J.K. Edler von Koch and Björn Franke. “Variability of Data Dependences and Control Flow”. In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2014, pp. 180–189. DOI: [10.1109/ISPASS.2014.6844482](https://doi.org/10.1109/ISPASS.2014.6844482).
- [77] Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. “Quantifying the Impact of Input Data Sets on Program Behavior and Its Applications”. In: *J. Instr. Level Parallelism* 5 (Apr. 2003). <https://pure.qub.ac.uk/en/publications/quantifying-the-impact-of-input-data-sets-on-program-behavior-and>.

- [78] Fernando A. Endo, Arthur Perais, and André Seznec. “On the Interactions Between Value Prediction and Compiler Optimizations in the Context of EOLE”. In: *ACM Trans. Archit. Code Optim.* 14.2 (July 2017), 18:1–18:24. DOI: [10.1145/3090634](https://doi.org/10.1145/3090634).
- [79] August Ernstsson et al. “SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters”. In: *Int J Parallel Prog* (May 2021). DOI: [10.1007/s10766-021-00704-3](https://doi.org/10.1007/s10766-021-00704-3).
- [80] Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. “A Survey on Thread-Level Speculation Techniques”. In: *ACM Comput. Surv.* 49.2 (June 2016), 22:1–22:39. DOI: [10.1145/2938369](https://doi.org/10.1145/2938369).
- [81] Karl-Filip Faxén et al. “Embla - Data Dependence Profiling for Parallel Programming”. In: *2008 International Conference on Complex, Intelligent and Software Intensive Systems*. Mar. 2008, pp. 780–785. DOI: [10.1109/CISIS.2008.52](https://doi.org/10.1109/CISIS.2008.52).
- [82] Paul Feautrier. “Dataflow Analysis of Array and Scalar References”. In: *Int J Parallel Prog* 20.1 (Feb. 1991), pp. 23–53. DOI: [10.1007/BF01407931](https://doi.org/10.1007/BF01407931).
- [83] Paul Feautrier. “Some Efficient Solutions to the Affine Scheduling Problem. I. One-Dimensional Time”. In: *Int J Parallel Prog* 21.5 (Oct. 1992), pp. 313–347. DOI: [10.1007/BF01407835](https://doi.org/10.1007/BF01407835).
- [84] Min Feng, Rajiv Gupta, and Iulian Neamtiu. “Effective Parallelization of Loops in the Presence of I/O Operations”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’12. New York, NY, USA: Association for Computing Machinery, June 11, 2012, pp. 487–498. ISBN: 978-1-4503-1205-9. DOI: [10.1145/2254064.2254122](https://doi.org/10.1145/2254064.2254122).
- [85] Min Feng, Changhui Lin, and Rajiv Gupta. “PLDS: Partitioning Linked Data Structures for Parallelism”. In: *ACM Trans. Archit. Code Optim.* 8.4 (Jan. 2012), 38:1–38:21. DOI: [10.1145/2086696.2086717](https://doi.org/10.1145/2086696.2086717).
- [86] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The Program Dependence Graph and Its Use in Optimization”. In: *ACM Trans Program Lang Syst* 9.3 (July 1987), pp. 319–349. DOI: [10.1145/24039.24041](https://doi.org/10.1145/24039.24041).
- [87] Allan L. Fisher and Anwar M. Ghuloum. “Parallelizing Complex Scans and Reductions”. In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. PLDI ’94. New York, NY,

- USA: ACM, 1994, pp. 135–146. ISBN: 978-0-89791-662-2. DOI: [10.1145/178243.178255](https://doi.org/10.1145/178243.178255).
- [88] Jessie Frazelle. “Chipping Away at Moore’s Law: Modern CPUs Are Just Chiplets Connected Together.” In: *Queue* 18.1 (Feb. 2020), Pages 20:5–Pages 20:15. DOI: [10.1145/3387945.3388515](https://doi.org/10.1145/3387945.3388515).
- [89] Michael A. Frumkin and Leonid V. Shabanov. “Benchmarking Memory Performance with the Data Cube Operator”. In: *Proceedings of the ISCA 17th International Conference on Parallel and Distributed Computing Systems, September 15-17, 2004, The Canterbury Hotel, San Francisco, California, USA*. 2004, pp. 165–171.
- [90] Erich Gamma et al. *Design Patterns : Elements of Reusable Object-Oriented Software*. 01 edition. Reading, Mass: Addison Wesley, Oct. 1994. ISBN: 978-0-201-63361-0.
- [91] Vijay Ganesh and David L. Dill. “A Decision Procedure for Bit-Vectors and Arrays”. In: *Proceedings of the 19th International Conference on Computer Aided Verification. CAV’07*. Berlin, Heidelberg: Springer-Verlag, July 2007, pp. 519–531. ISBN: 978-3-540-73367-6.
- [92] Pranav Garg et al. “Feedback-Directed Unit Test Generation for C/C++ Using Concolic Execution”. In: *2013 35th International Conference on Software Engineering (ICSE)*. May 2013, pp. 132–141. DOI: [10.1109/ICSE.2013.6606559](https://doi.org/10.1109/ICSE.2013.6606559).
- [93] Rakesh Ghiya and Laurie J. Hendren. “Is It a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL ’96*. New York, NY, USA: ACM, 1996, pp. 1–15. ISBN: 978-0-89791-769-8. DOI: [10.1145/237721.237724](https://doi.org/10.1145/237721.237724).
- [94] Rakesh Ghiya and Laurie J. Hendren. “Putting Pointer Analysis to Work”. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL ’98*. New York, NY, USA: ACM, 1998, pp. 121–133. ISBN: 978-0-89791-979-1. DOI: [10.1145/268946.268957](https://doi.org/10.1145/268946.268957).
- [95] Philip Ginsbach, Lewis Crawford, and Michael F. P. O’Boyle. “CAnDL: A Domain Specific Language for Compiler Analysis”. In: *Proceedings of the 27th International Conference on Compiler Construction. CC 2018*. New York, NY,

- USA: Association for Computing Machinery, Feb. 2018, pp. 151–162. ISBN: 978-1-4503-5644-2. DOI: [10.1145/3178372.3179515](https://doi.org/10.1145/3178372.3179515).
- [96] Philip Ginsbach and Michael F. P. O’Boyle. “Discovery and Exploitation of General Reductions: A Constraint Based Approach”. In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO ’17. <http://dl.acm.org/citation.cfm?id=3049832.3049862>. Piscataway, NJ, USA: IEEE Press, 2017, pp. 269–280. ISBN: 978-1-5090-4931-8.
- [97] Patrice Godefroid. “Fuzzing: Hack, Art, and Science”. In: *Commun. ACM* 63.2 (Jan. 2020), pp. 70–76. DOI: [10.1145/3363824](https://doi.org/10.1145/3363824).
- [98] Horacio González-Vélez and Mario Leyton. “A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers”. In: *Softw. Pract. Exp.* 40.12 (Nov. 2010), pp. 1135–1160. DOI: [10.1002/spe.1026](https://doi.org/10.1002/spe.1026).
- [99] Samuel Greengard. “GPUs Reshape Computing”. In: *Commun. ACM* 59.9 (Aug. 2016), pp. 14–16. DOI: [10.1145/2967979](https://doi.org/10.1145/2967979).
- [100] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. “Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation”. In: *Parallel Process. Lett.* 22.04 (Dec. 2012), p. 1250010. DOI: [10.1142/S0129626412500107](https://doi.org/10.1142/S0129626412500107).
- [101] Bolei Guo et al. “Practical and Accurate Low-Level Pointer Analysis”. In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 291–302. ISBN: 978-0-7695-2298-2. DOI: [10.1109/CGO.2005.27](https://doi.org/10.1109/CGO.2005.27).
- [102] Rajiv Gupta. “SPMD Execution of Programs with Pointer-Based Data Structures on Distributed-Memory Machines”. In: *Journal of Parallel and Distributed Computing* 16.2 (Oct. 1992), pp. 92–107. DOI: [10.1016/0743-7315\(92\)90026-J](https://doi.org/10.1016/0743-7315(92)90026-J).
- [103] Mary Hall, David Padua, and Keshav Pingali. “Compiler Research: The next 50 Years”. In: *Commun. ACM* 52.2 (Feb. 2009), p. 60. DOI: [10.1145/1461928.1461946](https://doi.org/10.1145/1461928.1461946).
- [104] Mary W. Hall et al. “Experiences Using the ParaScope Editor: An Interactive Parallel Programming Tool”. In: *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’93. San Diego, California, USA: Association for Computing Machinery, July 1993, pp. 33–43. ISBN: 978-0-89791-589-2. DOI: [10.1145/155332.155336](https://doi.org/10.1145/155332.155336).

- [105] Jingxuan He et al. “Learning to Fuzz from Symbolic Execution with Application to Smart Contracts”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 531–548. ISBN: 978-1-4503-6747-9. DOI: [10.1145/3319535.3363230](https://doi.org/10.1145/3319535.3363230).
- [106] Laurie J. Hendren, Joseph Hummell, and Alexandru Nicolau. “Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs”. In: *SIGPLAN Not.* 27.7 (July 1992), pp. 249–260. DOI: [10.1145/143103.143138](https://doi.org/10.1145/143103.143138).
- [107] L.J. Hendren and A. Nicolau. “Parallelizing Programs with Recursive Data Structures”. In: *IEEE Trans. Parallel Distrib. Syst.* 1.1 (Jan. 1990), pp. 35–47. DOI: [10.1109/71.80123](https://doi.org/10.1109/71.80123).
- [108] John L. Hennessy, David A. Patterson, and Andrea C. Arpaci-Dusseau. *Computer Architecture: A Quantitative Approach*. 4th ed. Amsterdam ; Boston: Morgan Kaufmann, 2007. ISBN: 978-0-12-370490-0.
- [109] John L. Henning. “Spec Cpu2000: Measuring Cpu Performance in the New Millennium”. In: *Computer* 33.7 (July 2000), pp. 28–35. DOI: [10.1109/2.869367](https://doi.org/10.1109/2.869367).
- [110] John L. Henning. “Spec Cpu2006 Benchmark Descriptions”. In: *SIGARCH Comput Arch. News* 34.4 (Sept. 2006), pp. 1–17. DOI: [10.1145/1186736.1186737](https://doi.org/10.1145/1186736.1186737).
- [111] W. Daniel Hillis and Guy L. Steele. “Data Parallel Algorithms”. In: *Commun. ACM* 29.12 (Dec. 1986), pp. 1170–1183. DOI: [10.1145/7902.7903](https://doi.org/10.1145/7902.7903).
- [112] Michael Hind. “Pointer Analysis: Haven’t We Solved This Problem Yet?” In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE ’01. New York, NY, USA: ACM, 2001, pp. 54–61. ISBN: 978-1-58113-413-1. DOI: [10.1145/379605.379665](https://doi.org/10.1145/379605.379665).
- [113] Mark Horowitz. “Reconfigurable Future: The Ability to Produce Cheaper, More Compact Chips Is a Double-Edged Sword.” In: *Queue* 1.7 (Oct. 2003), p. 1. DOI: [10.1145/957717.1388771](https://doi.org/10.1145/957717.1388771).

- [114] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. “A General Data Dependence Test for Dynamic, Pointer-Based Data Structures”. In: *SIGPLAN Not.* 29.6 (June 1994), pp. 218–229. DOI: [10.1145/773473.178262](https://doi.org/10.1145/773473.178262).
- [115] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. “Abstract Description of Pointer Data Structures: An Approach for Improving the Analysis and Optimization of Imperative Programs”. In: *ACM Lett. Program. Lang. Syst.* 1.3 (Sept. 1992), pp. 243–260. DOI: [10.1145/151640.151644](https://doi.org/10.1145/151640.151644).
- [116] Joseph Hummel, Alexandru Nicolau, and Laurie J. Hendren. “A Language for Conveying the Aliasing Properties of Dynamic, Pointer-Based Data Structures”. In: *Proceedings of the 8th International Symposium on Parallel Processing*. USA: IEEE Computer Society, Apr. 1994, pp. 208–216. ISBN: 978-0-8186-5602-6.
- [117] A. R. Hurson et al. “Parallelization of DOALL and DOACROSS Loops—a Survey”. In: *Advances in Computers*. Ed. by Marvin V. Zelkowitz. Vol. 45. Emphasizing Parallel Programming Techniques. Elsevier, Jan. 1997, pp. 53–103. DOI: [10.1016/S0065-2458\(08\)60706-8](https://doi.org/10.1016/S0065-2458(08)60706-8).
- [118] Oscar Ibarra, Pedro Diniz, and Martin Rinard. “On the Complexity of Commutativity Analysis”. In: *Computing and Combinatorics*. Ed. by Jin-Yi Cai and Chak Kuen Wong. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1996, pp. 323–332. ISBN: 978-3-540-68461-9. DOI: [10.1007/3-540-61332-3_166](https://doi.org/10.1007/3-540-61332-3_166).
- [119] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70. DOI: [10.1109/IEEESTD.2008.4610935](https://doi.org/10.1109/IEEESTD.2008.4610935).
- [120] Intel® C++ Compiler. <https://software.intel.com/en-us/c-compilers>. Aug. 2020.
- [121] Nikolas Ioannou et al. “Toward a More Accurate Understanding of the Limits of the TLS Execution Paradigm”. In: *IEEE International Symposium on Workload Characterization (IISWC’10)*. Dec. 2010, pp. 1–12. DOI: [10.1109/IISWC.2010.5649169](https://doi.org/10.1109/IISWC.2010.5649169).
- [122] François Irigoin, Pierre Jouvelot, and Rémi Triolet. “Semantical Interprocedural Parallelization: An Overview of the PIPS Project”. In: *ACM International Conference on Supercomputing 25th Anniversary Volume*. Munich, Germany:

- Association for Computing Machinery, June 1991, pp. 143–150. ISBN: 978-1-4503-2840-1. DOI: [10.1145/2591635.2667163](https://doi.org/10.1145/2591635.2667163).
- [123] Makoto Ishihara, Hiroki Honda, and Mitsuhsa Sato. “Development and Implementation of an Interactive Parallelization Assistance Tool for OpenMP: iPat/OMP”. In: *IEICE - Trans. Inf. Syst.* E89-D.2 (Feb. 2006), pp. 399–407. DOI: [10.1093/ietisy/e89-d.2.399](https://doi.org/10.1093/ietisy/e89-d.2.399).
- [124] Shintaro Iwasaki et al. “BOLT: Optimizing OpenMP Parallel Regions with User-Level Threads”. In: *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Sept. 2019, pp. 29–42. DOI: [10.1109/PACT.2019.00011](https://doi.org/10.1109/PACT.2019.00011).
- [125] Tim Jammer, Christian Iwainsky, and Christian Bischof. “A Comparison of the Scalability of OpenMP Implementations”. In: *Euro-Par 2020: Parallel Processing*. Ed. by Maciej Malawski and Krzysztof Rządca. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 83–97. ISBN: 978-3-030-57675-2. DOI: [10.1007/978-3-030-57675-2_6](https://doi.org/10.1007/978-3-030-57675-2_6).
- [126] Klaus Jansen and Ralf Thöle. “Approximation Algorithms for Scheduling Parallel Jobs: Breaking the Approximation Ratio of 2”. In: *Automata, Languages and Programming*. Ed. by Luca Aceto et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 234–245. ISBN: 978-3-540-70575-8. DOI: [10.1007/978-3-540-70575-8_20](https://doi.org/10.1007/978-3-540-70575-8_20).
- [127] Donghwan Jeon et al. “Kremlin: Like Gprof, but for Parallelization”. In: *SIGPLAN Not.* 46.8 (Feb. 2011), pp. 293–294. DOI: [10.1145/2038037.1941595](https://doi.org/10.1145/2038037.1941595).
- [128] Peng Jiang, Linchuan Chen, and Gagan Agrawal. “Revealing Parallel Scans and Reductions in Recurrences through Function Reconstruction”. In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’18. New York, NY, USA: ACM, 2018, 10:1–10:13. ISBN: 978-1-4503-5986-3. DOI: [10.1145/3243176.3243204](https://doi.org/10.1145/3243176.3243204).
- [129] Alexandra Jimborean et al. “Adapting the Polyhedral Model As a Framework for Efficient Speculative Parallelization”. In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’12. New York, NY, USA: ACM, 2012, pp. 295–296. ISBN: 978-1-4503-1160-1. DOI: [10.1145/2145816.2145861](https://doi.org/10.1145/2145816.2145861).

- [130] H Jin and Frumkin MA. “The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance”. In: (May 2000).
- [131] Nick P. Johnson et al. “Speculative Separation for Privatization and Reductions”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. Beijing, China: Association for Computing Machinery, June 2012, pp. 359–370. ISBN: 978-1-4503-1205-9. DOI: [10.1145/2254064.2254107](https://doi.org/10.1145/2254064.2254107).
- [132] Norman P. Jouppi et al. “In-Datcenter Performance Analysis of a Tensor Processing Unit”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. New York, NY, USA: Association for Computing Machinery, June 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: [10.1145/3079856.3080246](https://doi.org/10.1145/3079856.3080246).
- [133] Pierre Jouvelot and Babak Dehbonei. “A Unified Semantic Approach for the Vectorization and Parallelization of Generalized Reductions”. In: *Proceedings of the 3rd International Conference on Supercomputing*. ICS '89. New York, NY, USA: Association for Computing Machinery, June 1989, pp. 186–194. ISBN: 978-0-89791-309-6. DOI: [10.1145/318789.318810](https://doi.org/10.1145/318789.318810).
- [134] Shoaib Kamil et al. “Verified Lifting of Stencil Computations”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. New York, NY, USA: Association for Computing Machinery, June 2016, pp. 711–726. ISBN: 978-1-4503-4261-2. DOI: [10.1145/2908080.2908117](https://doi.org/10.1145/2908080.2908117).
- [135] Timotej Kapus and Cristian Cadar. “A Segmented Memory Model for Symbolic Execution”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, pp. 774–784. ISBN: 978-1-4503-5572-8. DOI: [10.1145/3338906.3338936](https://doi.org/10.1145/3338906.3338936).
- [136] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. “The Organization of Computations for Uniform Recurrence Equations”. In: *J. ACM* 14.3 (July 1967), pp. 563–590. DOI: [10.1145/321406.321418](https://doi.org/10.1145/321406.321418).
- [137] Arun Kejariwal et al. “On the Performance Potential of Different Types of Speculative Thread-Level Parallelism: The DL Version of This Paper Includes Corrections That Were Not Made Available in the Printed Proceedings”. In:

- Proceedings of the 20th Annual International Conference on Supercomputing*. ICS '06. New York, NY, USA: Association for Computing Machinery, June 2006, p. 24. ISBN: 978-1-59593-282-2. DOI: [10.1145/1183401.1183407](https://doi.org/10.1145/1183401.1183407).
- [138] Wayne Kelly and William Pugh. “A Framework for Unifying Reordering Transformations”. In: (Oct. 1998). <https://drum.lib.umd.edu/handle/1903/607>.
- [139] K. Kennedy, K. S. McKinley, and C. W. Tseng. “Interactive Parallel Programming Using the ParaScope Editor”. In: *IEEE Trans. Parallel Distrib. Syst.* 2.3 (July 1991), pp. 329–341. DOI: [10.1109/71.86108](https://doi.org/10.1109/71.86108).
- [140] Christoph W. Kessler. “Pattern-Driven Automatic Parallelization”. In: *Sci. Program.* 5.3 (Aug. 1996), pp. 251–274. DOI: [10.1155/1996/406379](https://doi.org/10.1155/1996/406379).
- [141] C. Kim et al. “Context-Aware Memory Profiling for Speculative Parallelism”. In: *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. Dec. 2017, pp. 328–337. DOI: [10.1109/HiPC.2017.00045](https://doi.org/10.1109/HiPC.2017.00045).
- [142] Deokhwan Kim and Martin 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. New York, NY, USA: ACM, 2011, pp. 528–541. ISBN: 978-1-4503-0663-8. DOI: [10.1145/1993498.1993561](https://doi.org/10.1145/1993498.1993561).
- [143] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. “Prospector: A Dynamic Data-Dependence Profiler to Help Parallel Programming”. In: *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism*. Berkeley, CA: USENIX Association, June 2010.
- [144] Minjang Kim, Hyesoon Kim, and Chi-Keung 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. USA: IEEE Computer Society, Dec. 2010, pp. 535–546. ISBN: 978-0-7695-4299-7. DOI: [10.1109/MICRO.2010.49](https://doi.org/10.1109/MICRO.2010.49).
- [145] James C. King. “Symbolic Execution and Program Testing”. In: *Commun ACM* 19.7 (July 1976), pp. 385–394. DOI: [10.1145/360248.360252](https://doi.org/10.1145/360248.360252).
- [146] Eric Koskinen and Kshitij Bansal. “Reducing Commutativity Verification to Reachability with Differencing Abstractions”. In: *ArXiv200408450 Cs* (Apr. 2020). <http://arxiv.org/abs/2004.08450>. arXiv: [2004.08450 \[cs\]](https://arxiv.org/abs/2004.08450).

- [147] Peter Kraft et al. “Automatic Parallelization of Sequential Programs”. In: *ArXiv180907684 Cs* (July 2018). <http://arxiv.org/abs/1809.07684>. arXiv: 1809.07684 [cs].
- [148] D. J. Kuck et al. “Dependence Graphs and Compiler Optimizations”. In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’81. New York, NY, USA: ACM, 1981, pp. 207–218. ISBN: 978-0-89791-029-3. DOI: [10.1145/567532.567555](https://doi.org/10.1145/567532.567555).
- [149] M. Kulkarni et al. “Lonestar: A Suite of Parallel Irregular Programs”. In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. Apr. 2009, pp. 65–76. DOI: [10.1109/ISPASS.2009.4919639](https://doi.org/10.1109/ISPASS.2009.4919639).
- [150] Milind Kulkarni et al. “Optimistic Parallelism Requires Abstractions”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. New York, NY, USA: ACM, 2007, pp. 211–222. ISBN: 978-1-59593-633-2. DOI: [10.1145/1250734.1250759](https://doi.org/10.1145/1250734.1250759).
- [151] Richard E. Ladner and Michael J. Fischer. “Parallel Prefix Computation”. In: *J. ACM* 27.4 (Oct. 1980), pp. 831–838. DOI: [10.1145/322217.322232](https://doi.org/10.1145/322217.322232).
- [152] Leslie Lamport. “The Parallel Execution of DO Loops”. In: *Commun ACM* 17.2 (Feb. 1974), pp. 83–93. DOI: [10.1145/360827.360844](https://doi.org/10.1145/360827.360844).
- [153] William Landi. “Undecidability of Static Analysis”. In: *ACM Lett. Program. Lang. Syst.* 1.4 (Dec. 1992), pp. 323–337. DOI: [10.1145/161494.161501](https://doi.org/10.1145/161494.161501).
- [154] J. R. Larus. “Abstract Execution: A Technique for Efficiently Tracing Programs”. In: *Softw. Pract. Exper.* 20.12 (Nov. 1990), pp. 1241–1258. DOI: [10.1002/spe.4380201205](https://doi.org/10.1002/spe.4380201205).
- [155] J.R. Larus. “Loop-Level Parallelism in Numeric and Symbolic Programs”. In: *IEEE Trans. Parallel Distrib. Syst.* 4.7 (July 1993), pp. 812–826. DOI: [10.1109/71.238302](https://doi.org/10.1109/71.238302).
- [156] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Life-long Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO ’04. <http://dl.acm.org/citation.cfm?id=977395.977673>. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. ISBN: 978-0-7695-2102-2.

- [157] Jun Li, Bodong Zhao, and Chao Zhang. “Fuzzing: A Survey”. In: *Cybersecur* 1.1 (June 2018), p. 6. DOI: [10.1186/s42400-018-0002-y](https://doi.org/10.1186/s42400-018-0002-y).
- [158] Zhen Li, Ali Jannesari, and Felix Wolf. “An Efficient Data-Dependence Profiler for Sequential and Parallel Programs”. In: *2015 IEEE International Parallel and Distributed Processing Symposium*. May 2015, pp. 484–493. DOI: [10.1109/IPDPS.2015.41](https://doi.org/10.1109/IPDPS.2015.41).
- [159] Zhen Li et al. “DiscoPoP: A Profiling Tool to Identify Parallelization Opportunities”. In: *Tools for High Performance Computing 2014*. Ed. by Christoph Niethammer et al. Cham: Springer International Publishing, 2015, pp. 37–54. ISBN: 978-3-319-16012-2.
- [160] Zhen Li et al. “Unveiling Parallelization Opportunities in Sequential Programs”. In: *J. Syst. Softw.* 117.C (July 2016), pp. 282–295. DOI: [10.1016/j.jss.2016.03.045](https://doi.org/10.1016/j.jss.2016.03.045).
- [161] Shih-Wei Liao et al. “SUIF Explorer: An Interactive and Interprocedural Parallelizer”. In: *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’99. Atlanta, Georgia, USA: Association for Computing Machinery, May 1999, pp. 37–48. ISBN: 978-1-58113-100-0. DOI: [10.1145/301104.301108](https://doi.org/10.1145/301104.301108).
- [162] Daniel Liew et al. “Floating-Point Symbolic Execution: A Case Study in N-Version Programming”. In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. <http://dl.acm.org/citation.cfm?id=3155562.3155637>. Piscataway, NJ, USA: IEEE Press, 2017, pp. 601–612. ISBN: 978-1-5386-2684-9.
- [163] Benjamin Livshits et al. “In Defense of Soundness: A Manifesto”. In: *Commun ACM* 58.2 (Jan. 2015), pp. 44–46. DOI: [10.1145/2644805](https://doi.org/10.1145/2644805).
- [164] Roberto Castañeda Lozano, Murray Cole, and Björn Franke. “Modernizing Parallel Code with Pattern Analysis”. In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’21. New York, NY, USA: Association for Computing Machinery, Feb. 2021, pp. 418–430. ISBN: 978-1-4503-8294-6. DOI: [10.1145/3437801.3441603](https://doi.org/10.1145/3437801.3441603).
- [165] Chris Mack. “The Multiple Lives of Moore’s Law”. In: *IEEE Spectr.* 52.4 (Apr. 2015), pp. 31–31. DOI: [10.1109/MSPEC.2015.7065415](https://doi.org/10.1109/MSPEC.2015.7065415).

- [166] Stanislav Manilov, Christos Vasiladiotis, and Björn Franke. “Generalized Profile-Guided Iterator Recognition”. In: *Proceedings of the 27th International Conference on Compiler Construction - CC 2018*. Vienna, Austria: ACM Press, 2018, pp. 185–195. ISBN: 978-1-4503-5644-2. DOI: [10.1145/3178372.3179511](https://doi.org/10.1145/3178372.3179511).
- [167] Aleksandr Maramzin et al. ““It Looks Like You’re Writing a Parallel Loop”: A Machine Learning Based Parallelization Assistant”. In: *Proceedings of the 6th ACM SIGPLAN International Workshop on AI-Inspired and Empirical Methods for Software Engineering on Parallel Computing Systems*. AI-SEPS 2019. New York, NY, USA: ACM, 2019, pp. 1–10. ISBN: 978-1-4503-6983-1. DOI: [10.1145/3358500.3361567](https://doi.org/10.1145/3358500.3361567).
- [168] Paul Dan Marinescu and Cristian Cadar. “Make Test-Zesti: A Symbolic Execution Solution for Improving Regression Testing”. In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE ’12. <http://dl.acm.org/citation.cfm?id=2337223.2337308>. Piscataway, NJ, USA: IEEE Press, 2012, pp. 716–726. ISBN: 978-1-4673-1067-3.
- [169] Angelo Matni et al. “NOELLE Offers Empowering LLVM Extensions”. In: *ArXiv210205081 Cs* (Feb. 2021). <http://arxiv.org/abs/2102.05081>. arXiv: [2102.05081](https://arxiv.org/abs/2102.05081) [cs].
- [170] Stina Matthiesen and Pernille Bjørn. “Why Replacing Legacy Systems Is So Hard in Global Software Development: An Information Infrastructure Perspective”. In: *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*. CSCW ’15. New York, NY, USA: Association for Computing Machinery, Feb. 2015, pp. 876–890. ISBN: 978-1-4503-2922-4. DOI: [10.1145/2675133.2675232](https://doi.org/10.1145/2675133.2675232).
- [171] Michael D. McCool, Arch D. Robison, and James Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Amsterdam: Elsevier, Morgan Kaufmann, 2012. ISBN: 978-0-12-415993-8.
- [172] Vijay Menon, Keshav Pingali, and Nikolay Mateev. “Fractal Symbolic Analysis”. In: *ACM Trans. Program. Lang. Syst.* 25.6 (Nov. 2003), pp. 776–813. DOI: [10.1145/945885.945888](https://doi.org/10.1145/945885.945888).
- [173] Samuel P. Midkiff. “Automatic Parallelization: An Overview of Fundamental Compiler Techniques”. In: *Synthesis Lectures on Computer Architecture* 7.1 (Jan. 2012), pp. 1–169. DOI: [10.2200/S00340ED1V01Y201201CAC019](https://doi.org/10.2200/S00340ED1V01Y201201CAC019).

- [174] G. E. Moore. “Cramming More Components onto Integrated Circuits, Reprinted from Electronics, Volume 38, Number 8, April 19, 1965, Pp.114 Ff.” In: *IEEE Solid-State Circuits Soc. Newsl.* 11.3 (Sept. 2006), pp. 33–35. DOI: [10.1109/NSSC.2006.4785860](https://doi.org/10.1109/NSSC.2006.4785860).
- [175] Tipp Moseley et al. “Shadow Profiling: Hiding Instrumentation Costs with Parallelism”. In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO ’07. USA: IEEE Computer Society, Mar. 2007, pp. 198–208. ISBN: 978-0-7695-2764-2. DOI: [10.1109/CGO.2007.35](https://doi.org/10.1109/CGO.2007.35).
- [176] William S. Moses et al. *Polygeist: Affine C in MLIR*. https://acohen.gitlabpages.inria.fr/impact/impact2021/papers/IMPACT_2021_paper_1.pdf. 2021.
- [177] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, Calif: Morgan Kaufmann Publishers, 1997. ISBN: 978-1-55860-320-2.
- [178] Niall Murphy. *Discovering and Exploiting Parallelism in DOACROSS Loops*. Tech. rep. UCAM-CL-TR-882. <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-882.html>. University of Cambridge, Computer Laboratory, 2016.
- [179] Niall Murphy et al. “Limits of Static Dependence Analysis for Automatic Parallelization”. In: *International Workshop on Compilers for Parallel Computing (CPC)*. Cambridge, UK, Jan. 2015.
- [180] Niall Murphy et al. “Performance Implications of Transient Loop-Carried Data Dependences in Automatically Parallelized Loops”. In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. Barcelona, Spain: Association for Computing Machinery, Mar. 2016, pp. 23–33. ISBN: 978-1-4503-4241-4. DOI: [10.1145/2892208.2892214](https://doi.org/10.1145/2892208.2892214).
- [181] Jakub Novák. “Improvements of Memory Management in KLEE”. <https://is.muni.cz/th/eazm0/>. PhD thesis. Masarykova univerzita, Fakulta informatiky, 2020.
- [182] Robert O’Callahan et al. “Engineering Record and Replay for Deployability”. In: *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC ’17. <http://dl.acm.org/citation.cfm?id=3154690.3154727>. Berkeley, CA, USA: USENIX Association, 2017, pp. 377–389. ISBN: 978-1-931971-38-6.

- [183] Kazunori Ogata et al. “Replay Compilation: Improving Debuggability of a Just-in-Time Compiler”. In: *SIGPLAN Not.* 41.10 (Oct. 2006), pp. 241–252. DOI: [10.1145/1167515.1167493](https://doi.org/10.1145/1167515.1167493).
- [184] Saahil Ognawala et al. “MACKE: Compositional Analysis of Low-Level Vulnerabilities with Symbolic Execution”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. New York, NY, USA: ACM, 2016, pp. 780–785. ISBN: 978-1-4503-3845-5. DOI: [10.1145/2970276.2970281](https://doi.org/10.1145/2970276.2970281).
- [185] Tomas Öhberg, August Ernstsson, and Christoph Kessler. “Hybrid CPU–GPU Execution Support in the Skeleton Programming Framework SkePU”. In: *J Supercomput* (Mar. 2019). DOI: [10.1007/s11227-019-02824-7](https://doi.org/10.1007/s11227-019-02824-7).
- [186] Itta Ohmura et al. “MDGRAPE-4: A Special-Purpose Computer System for Molecular Dynamics Simulations”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 372.2021 (Aug. 2014), p. 20130387. DOI: [10.1098/rsta.2013.0387](https://doi.org/10.1098/rsta.2013.0387).
- [187] Kunle Olukotun and Lance Hammond. “The Future of Microprocessors: Chip Multiprocessor’s Promise of Huge Performance Gains Is Now a Reality.” In: *Queue* 3.7 (Sept. 2005), pp. 26–29. DOI: [10.1145/1095408.1095418](https://doi.org/10.1145/1095408.1095418).
- [188] *OpenMP Specifications*. <https://www.openmp.org/specifications/>.
- [189] Guilherme Ottoni et al. “Automatic Thread Extraction with Decoupled Software Pipelining”. In: *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 38. Washington, DC, USA: IEEE Computer Society, 2005, pp. 105–118. ISBN: 978-0-7695-2440-5. DOI: [10.1109/MICRO.2005.13](https://doi.org/10.1109/MICRO.2005.13).
- [190] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. “Deferred Concretization in Symbolic Execution via Fuzzing”. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, July 2019, pp. 228–238. ISBN: 978-1-4503-6224-5. DOI: [10.1145/3293882.3330554](https://doi.org/10.1145/3293882.3330554).
- [191] Ruud van der Pas. *Using OpenMP—the next Step: Affinity, Accelerators, Tasking, and SIMD*. Scientific and Engineering Computation. Cambridge, Massachusetts: The MIT Press, 2017. ISBN: 978-0-262-53478-9.

- [192] David M. Perry et al. “Accelerating Array Constraints in Symbolic Execution”. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, July 2017, pp. 68–78. ISBN: 978-1-4503-5076-1. DOI: [10.1145/3092703.3092728](https://doi.org/10.1145/3092703.3092728).
- [193] Keshav Pingali. *Why Compilers Have Failed and What Can We Do about It*. Houston, Texas, USA, October 7 - 9, 2010.
- [194] Keshav Pingali et al. “The Tao of Parallelism in Algorithms”. In: *SIGPLAN Not.* 46.6 (June 2011), pp. 12–25. DOI: [10.1145/1993316.1993501](https://doi.org/10.1145/1993316.1993501).
- [195] Bill Pottenger and Rudolf Eigenmann. “Idiom Recognition in the Polaris Parallelizing Compiler”. In: *Proceedings of the 9th International Conference on Supercomputing*. ICS ’95. Barcelona, Spain: Association for Computing Machinery, July 1995, pp. 444–448. ISBN: 978-0-89791-728-5. DOI: [10.1145/224538.224655](https://doi.org/10.1145/224538.224655).
- [196] Prakash Prabhu et al. “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. New York, NY, USA: ACM, 2011, pp. 1–11. ISBN: 978-1-4503-0663-8. DOI: [10.1145/1993498.1993500](https://doi.org/10.1145/1993498.1993500).
- [197] S. Prema et al. “A Study on Popular Auto-Parallelization Frameworks: A Study on Popular Auto-Parallelization Frameworks”. In: *Concurrency Computat Pract Exper* 31.17 (Sept. 2019), e5168. DOI: [10.1002/cpe.5168](https://doi.org/10.1002/cpe.5168).
- [198] William 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. New York, NY, USA: ACM, 1991, pp. 4–13. ISBN: 978-0-89791-459-8. DOI: [10.1145/125826.125848](https://doi.org/10.1145/125826.125848).
- [199] X. Qu and B. Robinson. “A Case Study of Concolic Testing Tools and Their Limitations”. In: *2011 International Symposium on Empirical Software Engineering and Measurement*. Sept. 2011, pp. 117–126. DOI: [10.1109/ESEM.2011.20](https://doi.org/10.1109/ESEM.2011.20).
- [200] Jonathan Ragan-Kelley et al. “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’13. New York, NY, USA: Association for

- Computing Machinery, June 2013, pp. 519–530. ISBN: 978-1-4503-2014-6. DOI: [10.1145/2491956.2462176](https://doi.org/10.1145/2491956.2462176).
- [201] Easwaran Raman et al. “Parallel-Stage Decoupled Software Pipelining”. In: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’08. Boston, MA, USA: Association for Computing Machinery, Apr. 2008, pp. 114–123. ISBN: 978-1-59593-978-4. DOI: [10.1145/1356058.1356074](https://doi.org/10.1145/1356058.1356074).
- [202] J. Ramanujam. “Beyond Unimodular Transformations”. In: *J Supercomput* 9.4 (Dec. 1995), pp. 365–389. DOI: [10.1007/BF01206273](https://doi.org/10.1007/BF01206273).
- [203] Ram Rangan. “Pipelined Multithreading Transformations and Support Mechanisms”. PhD Thesis. USA: Princeton University, 2007.
- [204] Ram Rangan et al. “Decoupled Software Pipelining with the Synchronization Array”. In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 177–188. ISBN: 978-0-7695-2229-6. DOI: [10.5555/1025127.1026007](https://doi.org/10.5555/1025127.1026007).
- [205] Lawrence Rauchwerger and David 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. New York, NY, USA: ACM, 1995, pp. 218–232. ISBN: 978-0-89791-697-4. DOI: [10.1145/207110.207148](https://doi.org/10.1145/207110.207148).
- [206] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. “Parallelizing User-Defined Aggregations Using Symbolic Execution”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP ’15. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 153–167. ISBN: 978-1-4503-3834-9. DOI: [10.1145/2815400.2815418](https://doi.org/10.1145/2815400.2815418).
- [207] Chandan Reddy, Michael Kruse, and Albert Cohen. “Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPU”. In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. PACT ’16. New York, NY, USA: Association for Computing Machinery, Sept. 2016, pp. 87–97. ISBN: 978-1-4503-4121-9. DOI: [10.1145/2967938.2967950](https://doi.org/10.1145/2967938.2967950).

- [208] Xavier Redon and Paul Feautrier. “Detection of Scans”. In: *Parallel Algorithms Appl.* 15.3-4 (Dec. 2000), pp. 229–263. DOI: [10.1080/01495730008947357](https://doi.org/10.1080/01495730008947357).
- [209] Thomas Reps. “Undecidability of Context-Sensitive Data-Dependence Analysis”. In: *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000), pp. 162–186. DOI: [10.1145/345099.345137](https://doi.org/10.1145/345099.345137).
- [210] Colfax Research. *A Performance-Based Comparison of C/C++ Compilers*. <https://colfaxresearch.com/compiler-comparison/>. Nov. 2017.
- [211] Martin C. Rinard and Pedro C. Diniz. “Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers”. In: *ACM Trans. Program. Lang. Syst.* 19.6 (Nov. 1997), pp. 942–991. DOI: [10.1145/267959.269969](https://doi.org/10.1145/267959.269969).
- [212] Noam Rinetzky and Mooly Sagiv. “Interprocedural Shape Analysis for Recursive Programs”. In: *Compiler Construction*. Ed. by Reinhard Wilhelm. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, pp. 133–149. ISBN: 978-3-540-45306-2. DOI: [10.1007/3-540-45306-7_10](https://doi.org/10.1007/3-540-45306-7_10).
- [213] Kamil Rosiak et al. “Analyzing Variability in 25 Years of Industrial Legacy Software: An Experience Report”. In: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B*. SPLC ’19. New York, NY, USA: Association for Computing Machinery, Sept. 2019, pp. 65–72. ISBN: 978-1-4503-6668-7. DOI: [10.1145/3307630.3342410](https://doi.org/10.1145/3307630.3342410).
- [214] Philipp Ruemmer and Thomas Wahl. “An SMT-LIB Theory of Binary Floating-Point Arithmetic”. In: *Satisfiability modulo Theories (SMT)*. 2010.
- [215] Karl Rupp. *42 Years of Microprocessor Trend Data* | Karl Rupp. <https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>. Feb. 2018.
- [216] Richard M. Russell. “The CRAY-1 Computer System”. In: *Commun. ACM* 21.1 (Jan. 1978), pp. 63–72. DOI: [10.1145/359327.359336](https://doi.org/10.1145/359327.359336).
- [217] Christos Sakalis et al. “Splash-3: A Properly Synchronized Benchmark Suite for Contemporary Research”. In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Apr. 2016, pp. 101–111. DOI: [10.1109/ISPASS.2016.7482078](https://doi.org/10.1109/ISPASS.2016.7482078).
- [218] Juan Salamanca, Jose Nelson Amaral, and Guido Araujo. “Using Hardware-Transactional-Memory Support to Implement Thread-Level Speculation”. In: *IEEE Trans. Parallel Distrib. Syst.* 29.2 (Feb. 2018), pp. 466–480. DOI: [10.1109/TPDS.2017.2752169](https://doi.org/10.1109/TPDS.2017.2752169).

- [219] Yukinori Sato, Yasushi Inoguchi, and Tadao Nakamura. “Whole Program Data Dependence Profiling to Unveil Parallel Regions in the Dynamic Execution”. In: *2012 IEEE International Symposium on Workload Characterization (IISWC)*. Nov. 2012, pp. 69–80. DOI: [10.1109/IISWC.2012.6402902](https://doi.org/10.1109/IISWC.2012.6402902).
- [220] Prateek Saxena et al. “Loop-Extended Symbolic Execution on Binary Programs”. In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ISSTA '09. New York, NY, USA: Association for Computing Machinery, July 2009, pp. 225–236. ISBN: 978-1-60558-338-9. DOI: [10.1145/1572272.1572299](https://doi.org/10.1145/1572272.1572299).
- [221] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. “All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”. In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 317–331. ISBN: 978-0-7695-4035-1. DOI: [10.1109/SP.2010.26](https://doi.org/10.1109/SP.2010.26).
- [222] Manuel Selva et al. “Building a Polyhedral Representation from an Instrumented Execution: Making Dynamic Analyses of Nonaffine Programs Scalable”. In: *ACM Trans Arch. Code Optim* 16.4 (Dec. 2019). DOI: [10.1145/3363785](https://doi.org/10.1145/3363785).
- [223] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C”. In: *SIGSOFT Softw. Eng. Notes* 30.5 (Sept. 2005), pp. 263–272. DOI: [10.1145/1095430.1081750](https://doi.org/10.1145/1095430.1081750).
- [224] Sangmin Seo, Gangwon Jo, and Jaejin Lee. “Performance Characterization of the NAS Parallel Benchmarks in OpenCL”. In: *2011 IEEE International Symposium on Workload Characterization (IISWC)*. Nov. 2011, pp. 137–148. DOI: [10.1109/IISWC.2011.6114174](https://doi.org/10.1109/IISWC.2011.6114174).
- [225] John Shalf. “The Future of Computing beyond Moore’s Law”. In: *Philos Trans A Math Phys Eng Sci* 378.2166 (Mar. 2020), p. 20190061. DOI: [10.1098/rsta.2019.0061](https://doi.org/10.1098/rsta.2019.0061).
- [226] John M. Shalf and Robert Leland. “Computing beyond Moore’s Law”. In: *Computer* 48.12 (Dec. 2015), pp. 14–23. DOI: [10.1109/MC.2015.374](https://doi.org/10.1109/MC.2015.374).
- [227] Hongzhang Shan et al. “A Programming Model Performance Study Using the NAS Parallel Benchmarks”. In: *Sci. Program.* 18.3-4 (Jan. 2010), pp. 153–167. DOI: [10.3233/SPR-2010-0306](https://doi.org/10.3233/SPR-2010-0306).

- [228] Yakun Sophia Shao et al. “Co-Designing Accelerators and SoC Interfaces Using Gem5-Aladdin”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–12. DOI: [10.1109/MICRO.2016.7783751](https://doi.org/10.1109/MICRO.2016.7783751).
- [229] Ridwan Shariffdeen et al. “Concolic Program Repair”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. New York, NY, USA: Association for Computing Machinery, June 2021, pp. 390–405. ISBN: 978-1-4503-8391-2. DOI: [10.1145/3453483.3454051](https://doi.org/10.1145/3453483.3454051).
- [230] David E. Shaw et al. “Anton 2: Raising the Bar for Performance and Programmability in a Special-Purpose Molecular Dynamics Supercomputer”. In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2014, pp. 41–53. DOI: [10.1109/SC.2014.9](https://doi.org/10.1109/SC.2014.9).
- [231] Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. “Oil and Water Can Mix: An Integration of Polyhedral and AST-Based Transformations”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 287–298. ISBN: 978-1-4799-5500-8. DOI: [10.1109/SC.2014.29](https://doi.org/10.1109/SC.2014.29).
- [232] Stelios Sidiroglou-Douskos et al. “Managing Performance vs. Accuracy Trade-Offs with Loop Perforation”. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: Association for Computing Machinery, Sept. 2011, pp. 124–134. ISBN: 978-1-4503-0443-6. DOI: [10.1145/2025113.2025133](https://doi.org/10.1145/2025113.2025133).
- [233] Michael Šimáček. “Symbolic-Size Memory Allocation Support for Klee”. <https://is.muni.cz/th/mdedh/?lang=en>. PhD thesis. Masaryk University, Faculty of Informatics, 2018.
- [234] Andreas Simbürger and Armin Größlinger. “On the Variety of Static Control Parts in Real-World Programs: From Affine via Multi-Dimensional to Polynomial and Just-in-Time”. In: *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*. Ed. by Sanjay Rajopadhye and Sven Verdoolaege. Jan. 2014.

- [235] Kevin Smith and Bill Appelbe. “Interactive Conversion of Sequential to Multi-tasking FORTRAN”. In: *Proceedings of the 3rd International Conference on Supercomputing*. ICS ’89. Crete, Greece: Association for Computing Machinery, June 1989, pp. 225–234. ISBN: 978-0-89791-309-6. DOI: [10.1145/318789.318814](https://doi.org/10.1145/318789.318814).
- [236] Byoungro So, Sungdo Moon, and Mary W. Hall. “Measuring the Effectiveness of Automatic Parallelization in SUIF”. In: *Proceedings of the 12th International Conference on Supercomputing*. ACM, July 1998, pp. 212–219. ISBN: 978-0-89791-998-2. DOI: [10.1145/277830.277876](https://doi.org/10.1145/277830.277876).
- [237] Kevin Streit et al. “Sambamba: A Runtime System for Online Adaptive Parallelization”. In: *Compiler Construction*. Ed. by Michael O’Boyle. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 240–243. ISBN: 978-3-642-28652-0. DOI: [10.1007/978-3-642-28652-0_13](https://doi.org/10.1007/978-3-642-28652-0_13).
- [238] Yulei Sui, Peng Di, and Jingling Xue. “Sparse Flow-Sensitive Pointer Analysis for Multithreaded Programs”. In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO ’16. Barcelona, Spain: Association for Computing Machinery, Feb. 2016, pp. 160–170. ISBN: 978-1-4503-3778-6. DOI: [10.1145/2854038.2854043](https://doi.org/10.1145/2854038.2854043).
- [239] Yulei Sui and Jingling Xue. “SVF: Interprocedural Static Value-Flow Analysis in LLVM”. In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. New York, NY, USA: ACM, 2016, pp. 265–266. ISBN: 978-1-4503-4241-4. DOI: [10.1145/2892208.2892235](https://doi.org/10.1145/2892208.2892235).
- [240] Herb Sutter. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. <http://www.gotw.ca/publications/concurrency-ddj.htm>. 2005.
- [241] *Test-Suite Guide — LLVM 12 Documentation*. <https://llvm.org/docs/TestSuiteGuide.html>.
- [242] *The ACE Companies*. <http://www.ace.nl/>.
- [243] Neil Thompson and Svenja Spanuth. *The Decline of Computers As a General Purpose Technology: Why Deep Learning and the End of Moore’s Law Are Fragmenting Computing*. SSRN Scholarly Paper ID 3287769. Rochester, NY: Social Science Research Network, Nov. 2018. DOI: [10.2139/ssrn.3287769](https://doi.org/10.2139/ssrn.3287769).

- [244] Chen Tian, Min Feng, and Rajiv Gupta. “Supporting Speculative Parallelization in the Presence of Dynamic Data Structures”. In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’10. New York, NY, USA: Association for Computing Machinery, June 2010, pp. 62–73. ISBN: 978-1-4503-0019-3. DOI: [10.1145/1806596.1806604](https://doi.org/10.1145/1806596.1806604).
- [245] Fernando G. Tinetti and Mariano Méndez. “Fortran Legacy Software: Source Code Update and Possible Parallelisation Issues”. In: *SIGPLAN Fortran Forum* 31.1 (Mar. 2012), pp. 5–22. DOI: [10.1145/2179280.2179281](https://doi.org/10.1145/2179280.2179281).
- [246] Georgios Tournavitis. “Profile-Driven Parallelisation of Sequential Programs”. <https://www.era.lib.ed.ac.uk/handle/1842/5287>. Thesis. Edinburgh, Scotland, UK: University of Edinburgh, June 2011.
- [247] Georgios Tournavitis and Björn Franke. “Semi-Automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism Using Profiling Information”. In: ACM Press, 2010, p. 377. ISBN: 978-1-4503-0178-7. DOI: [10.1145/1854273.1854321](https://doi.org/10.1145/1854273.1854321).
- [248] Georgios Tournavitis et al. “Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping”. In: *SIGPLAN Not.* 44.6 (June 2009), pp. 177–187. DOI: [10.1145/1543135.1542496](https://doi.org/10.1145/1543135.1542496).
- [249] Ross Albert Towle. “Control and Data Dependence for Program Transformations.” PhD Thesis. Champaign, IL, USA: University of Illinois at Urbana-Champaign, 1976.
- [250] David Trabish et al. “Chopped Symbolic Execution”. In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. New York, NY, USA: ACM, 2018, pp. 350–360. ISBN: 978-1-4503-5638-1. DOI: [10.1145/3180155.3180251](https://doi.org/10.1145/3180155.3180251).
- [251] Konrad Trifunovic et al. “GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation”. In: *GCC Research Opportunities Workshop (GROW’10)*. <https://hal.inria.fr/inria-00551516>. Jan. 2010.
- [252] Marek Trtík. “Symbolic Execution and Program Loops”. <https://is.muni.cz/th/t52nv/>. PhD thesis. Masarykova univerzita, Fakulta informatiky, 2014.

- [253] Peng Tu and David Padua. “Automatic Array Privatization”. In: *Languages and Compilers for Parallel Computing*. Ed. by Utpal Banerjee et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1994, pp. 500–521. ISBN: 978-3-540-48308-3. DOI: [10.1007/3-540-57659-2_29](https://doi.org/10.1007/3-540-57659-2_29).
- [254] *uClibc*. <https://www.uclibc.org/>.
- [255] Abhishek Udupa, Kaushik Rajan, and William Thies. “ALTER: Exploiting Breakable Dependences for Parallelization”. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. New York, NY, USA: ACM, 2011, pp. 480–491. ISBN: 978-1-4503-0663-8. DOI: [10.1145/1993498.1993555](https://doi.org/10.1145/1993498.1993555).
- [256] Robert Utterback et al. “Processor-Oblivious Record and Replay”. In: *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’17. New York, NY, USA: Association for Computing Machinery, Jan. 2017, pp. 145–161. ISBN: 978-1-4503-4493-7. DOI: [10.1145/3018743.3018764](https://doi.org/10.1145/3018743.3018764).
- [257] Neil Vachharajani et al. “Speculative Decoupled Software Pipelining”. In: *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. Sept. 2007, pp. 49–59. DOI: [10.1109/PACT.2007.4336199](https://doi.org/10.1109/PACT.2007.4336199).
- [258] H. L. A. van der Spek, E. M. Bakker, and H. A. G. Wijshoff. “SPARK00: A Benchmark Package for the Compiler Evaluation of Irregular/Sparse Codes”. In: *ArXiv08053897 Cs* (May 2008). <http://arxiv.org/abs/0805.3897>. arXiv: [0805.3897 \[cs\]](https://arxiv.org/abs/0805.3897).
- [259] Harmen L. A. van der Spek, C. W. Mattias Holm, and Harry A. G. Wijshoff. “A Compilation Framework for the Automatic Restructuring of Pointer-Linked Data Structures”. In: *High-Performance Scientific Computing: Algorithms and Applications*. Ed. by Michael W. Berry et al. London: Springer, 2012, pp. 97–122. ISBN: 978-1-4471-2437-5. DOI: [10.1007/978-1-4471-2437-5_4](https://doi.org/10.1007/978-1-4471-2437-5_4).
- [260] Harmen L.A. van der Spek, Erwin M. Bakker, and Harry A.G. Wijshoff. “Characterizing the Performance Penalties Induced by Irregular Code Using Pointer Structures and Indirection Arrays on the Intel Core 2 Architecture”. In: *Proceedings of the 6th ACM Conference on Computing Frontiers*. CF ’09. New York, NY, USA: ACM, 2009, pp. 221–224. ISBN: 978-1-60558-413-3. DOI: [10.1145/1531743.1531779](https://doi.org/10.1145/1531743.1531779).

- [261] Hans Vandierendonck, Sean Rul, and Koen 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. New York, NY, USA: ACM, 2010, pp. 389–400. ISBN: 978-1-4503-0178-7. DOI: [10.1145/1854273.1854322](https://doi.org/10.1145/1854273.1854322).
- [262] Rajeshwar Vanka and James Tuck. “Efficient and Accurate Data Dependence Profiling Using Software Signatures”. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO ’12. San Jose, California: Association for Computing Machinery, Mar. 2012, pp. 186–195. ISBN: 978-1-4503-1206-6. DOI: [10.1145/2259016.2259041](https://doi.org/10.1145/2259016.2259041).
- [263] C. Vasiladiotis et al. “Loop Parallelization Using Dynamic Commutativity Analysis”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Feb. 2021, pp. 150–161. DOI: [10.1109/CGO51591.2021.9370319](https://doi.org/10.1109/CGO51591.2021.9370319).
- [264] Chris Vasiladiotis. *Iterator Recognition Code Repository*. <https://github.com/compor/IteratorRecognition>. July 2019.
- [265] B. Veltman, B. J. Lageweg, and Jan Karel Lenstra. “Multiprocessor Scheduling with Communication Delays”. In: R 9018 (Jan. 1990). <https://ir.cwi.nl/pub/5713>.
- [266] Tobias J. K. Edler von Koch et al. “Towards a Compiler Analysis for Parallel Algorithmic Skeletons”. In: *Proceedings of the 27th International Conference on Compiler Construction*. CC 2018. New York, NY, USA: ACM, 2018, pp. 174–184. ISBN: 978-1-4503-5644-2. DOI: [10.1145/3178372.3179513](https://doi.org/10.1145/3178372.3179513).
- [267] Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. “Modeling Optimistic Concurrency Using Quantitative Dependence Analysis”. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’08. New York, NY, USA: Association for Computing Machinery, Feb. 2008, pp. 185–196. ISBN: 978-1-59593-795-7. DOI: [10.1145/1345206.1345234](https://doi.org/10.1145/1345206.1345234).
- [268] Steven Wallace and Kim Hazelwood. “SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance”. In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO ’07. USA: IEEE

- Computer Society, Mar. 2007, pp. 209–220. ISBN: 978-0-7695-2764-2. DOI: [10.1109/CGO.2007.37](https://doi.org/10.1109/CGO.2007.37).
- [269] Amos Waterland et al. “ASC: Automatically Scalable Computation”. In: *SIG-PLAN Not.* 49.4 (Feb. 2014), pp. 575–590. DOI: [10.1145/2644865.2541985](https://doi.org/10.1145/2644865.2541985).
- [270] Mark Weiser. “Program Slicing”. In: *IEEE Trans. Softw. Eng.* SE-10.4 (July 1984), pp. 352–357. DOI: [10.1109/TSE.1984.5010248](https://doi.org/10.1109/TSE.1984.5010248).
- [271] Andreas Wilhelm et al. “Tool-Based Interactive Software Parallelization: A Case Study”. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP ’18. Gothenburg, Sweden: Association for Computing Machinery, May 2018, pp. 115–123. ISBN: 978-1-4503-5659-6. DOI: [10.1145/3183519.3183555](https://doi.org/10.1145/3183519.3183555).
- [272] Michael E. Wolf and Monica S. Lam. “A Data Locality Optimizing Algorithm”. In: <https://doi.org/10.1145/113446.113449>. Association for Computing Machinery, May 1991.
- [273] Michael Wolfe. “Advanced Loop Interchanging”. In: *International Conference on Parallel Processing, ICPP’86, University Park, PA, USA, August 1986*. IEEE Computer Society Press, 1986, pp. 536–543.
- [274] Michael Wolfe. *High Performance Compilers for Parallel Computing*. 1 edition. Redwood City, Calif: Pearson, June 1995. ISBN: 978-0-8053-2730-4.
- [275] Michael Wolfe. “The Tiny Loop Restructuring Research Tool”. In: *Proc 1991 Intl Conf Parallel Process*. Vol.II (1991). <https://ci.nii.ac.jp/naid/10000032836/>, pp. 46–53.
- [276] Michael Joseph Wolfe. *Optimizing Supercompilers for Supercomputers*. Cambridge, MA, USA: MIT Press, 1990. ISBN: 978-0-262-73082-2.
- [277] Peng Wu, Arun Kejariwal, and Călin Cașcaval. “Compiler-Driven Dependence Profiling to Guide Program Parallelization”. In: *Languages and Compilers for Parallel Computing*. Ed. by José Nelson Amaral. Vol. 5335. http://link.springer.com/10.1007/978-3-540-89740-8_16. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 232–248. ISBN: 978-3-540-89739-2 978-3-540-89740-8.
- [278] Baowen Xu et al. “A Brief Survey of Program Slicing”. In: *SIGSOFT Softw. Eng. Notes* 30.2 (Mar. 2005), pp. 1–36. DOI: [10.1145/1050849.1050865](https://doi.org/10.1145/1050849.1050865).

- [279] Hui Xu et al. “Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs”. In: *IEEE Trans. Dependable Secure Comput.* (2018), pp. 1–1. DOI: [10.1109/TDSC.2018.2866469](https://doi.org/10.1109/TDSC.2018.2866469).
- [280] Paraskevas Yiapanis. “High Performance Optimizations in Runtime Speculative Parallelization for Multicore Architectures”. [https://www.research.manchester.ac.uk/portal/en/theses/high-performance-optimizations-in-runtime-speculative-parallelization-for-multicore-architectures\(1d980957-a08d-49f0-8f7f-d5c35bae38b8\).html](https://www.research.manchester.ac.uk/portal/en/theses/high-performance-optimizations-in-runtime-speculative-parallelization-for-multicore-architectures(1d980957-a08d-49f0-8f7f-d5c35bae38b8).html). Ph.D. University of Manchester, 2013.
- [281] Paraskevas Yiapanis et al. “Optimizing Software Runtime Systems for Speculative Parallelization”. In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013), 39:1–39:27. DOI: [10.1145/2400682.2400698](https://doi.org/10.1145/2400682.2400698).
- [282] Hongtao Yu and Zhiyuan Li. “Fast Loop-Level Data Dependence Profiling”. In: *Proceedings of the 26th ACM International Conference on Supercomputing. ICS '12*. San Servolo Island, Venice, Italy: Association for Computing Machinery, June 2012, pp. 37–46. ISBN: 978-1-4503-1316-2. DOI: [10.1145/2304576.2304584](https://doi.org/10.1145/2304576.2304584).
- [283] Hongtao Yu and Zhiyuan 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*. Minneapolis, MN, USA: Association for Computing Machinery, July 2012, pp. 23–33. ISBN: 978-1-4503-1454-1. DOI: [10.1145/2338965.2336756](https://doi.org/10.1145/2338965.2336756).
- [284] Vadim Zaytsev. “Open Challenges in Incremental Coverage of Legacy Software Languages”. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience. PX/17.2*. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 1–6. ISBN: 978-1-4503-5522-3. DOI: [10.1145/3167105](https://doi.org/10.1145/3167105).
- [285] Xiangyu Zhang, Armand Navabi, and Suresh 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*. USA: IEEE Computer Society, Mar. 2009, pp. 47–58. ISBN: 978-0-7695-3576-0. DOI: [10.1109/CGO.2009.15](https://doi.org/10.1109/CGO.2009.15).

- [286] Song Zhao, Yixin Bian, and Sensen Zhang. “A Review on Refactoring Sequential Program to Parallel Code in Multicore Era”. In: *Proceedings of 2015 International Conference on Intelligent Computing and Internet of Things*. Jan. 2015, pp. 151–154. DOI: [10.1109/ICAIOT.2015.7111558](https://doi.org/10.1109/ICAIOT.2015.7111558).
- [287] Chuan-Qi Zhu and Pen-Chung Yew. “A Scheme to Enforce Data Dependence on Large Multiprocessor Systems”. In: *IEEE Trans. Softw. Eng.* SE-13.6 (June 1987), pp. 726–739. DOI: [10.1109/TSE.1987.233477](https://doi.org/10.1109/TSE.1987.233477).
- [288] Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. “Visual Program Manipulation in the Polyhedral Model”. In: *ACM Trans. Archit. Code Optim.* 15.1 (Mar. 2018), 16:1–16:25. DOI: [10.1145/3177961](https://doi.org/10.1145/3177961).