UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Practical Oracle for Sequential Code Parallelization

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Saturnino Garcia, Jr.

Committee in charge:

Professor Michael Taylor, Chair
Professor Peter Asbeck
Professor Chung-Kuan Cheng
Professor Sorin Lerner
Professor Steven Swanson

2012

The dissertation of Saturnino Garcia, Jr. is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____
Chair

University of California, San Diego

2012

DEDICATION

To my wife, for her endless love and support.

EPIGRAPH

The greatest obstacle to discovery is not ignorance–it is the illusion of knowledge.

—Daniel J. Boorstin

TABLE OF CONTENTS

Signature Page . . . iii

Dedication . . . iv

Epigraph . . . v

Table of Contents . . . vi

List of Figures . . . ix

List of Tables . . . x

Acknowledgements . . . xi

Vita and Publications . . . xiv

Abstract of the Dissertation . . . xvi

Chapter 1 Introduction . . . 1
1.1 Taxonomy of Parallelization Tools . . . 3
1.1.1 Overview of Parallelization Stages . . . 4
1.1.2 Existing Parallelization Tools . . . 5
1.2 A Practical Oracle for Parallelization . . . 7
1.3 Thesis Organization . . . 9

Chapter 2 The Cost of Inefficient Parallelization . . . 12
2.1 Example Parallelization Methodology . . . 12
2.2 User Study . . . 14
2.2.1 Setup . . . 15
2.2.2 Benchmark Analysis . . . 16
2.2.3 Impact on Program Speedup . . . 18
2.2.4 Time Spent On Critical Regions . . . 20
2.2.5 Threats to Validity . . . 22
2.2.6 Conclusions . . . 23

Chapter 3 System Overview . . . 26
3.1 Usage Model . . . 26
3.2 System Architecture . . . 30
3.3 Limitations of Kremlin and other Dynamic Analyses . . . 33

Chapter 4 Planning-Aware Parallelism Discovery . . . 35
4.1 Requirements of Planning-Aware Discovery . . . 35
4.2 Background: Critical Path Analysis . . . 36
4.3 Hierarchical Critical Path Analysis . . . 38
4.3.1 Defining a Region . . . 39
4.3.2 Calculating Critical Path with Shadow Memory . . . 41
4.3.3 Introducing Hierarchy into Shadow Memory . . . 44
4.3.4 Summarizing Dynamic Regions . . . 47
4.4 Identifying Local Parallelism . . . 48
4.4.1 Initial Approach: Parallelism Charts . . . 49
4.4.2 Self-Parallelism . . . 51
4.5 Evaluation . . . 57

Chapter 5 From Parallelism to Parallelization Plan . . . 62
5.1 Defining Parallelism Planning . . . 62
5.2 Estimating Parallel Execution Time . . . 64
5.3 Identifying Parallelism Types . . . 66
5.4 Planner Personalities . . . 67
5.4.1 OpenMP Planning Personality . . . 68
5.4.2 OpenCL Planning Personality . . . 70
5.4.3 Cilk++ Planning Personality . . . 72
5.4.4 Developing Additional Planner Personalities . . . 73
5.5 Experimental Evaluation . . . 74
5.5.1 Methodology . . . 75
5.5.2 Comparing Plan Size . . . 76
5.5.3 Performance Comparison . . . 77
5.5.4 Effectiveness of Region Prioritization . . . 79
5.5.5 Influences on Plan Size . . . 80
5.5.6 Initial GPGPU Planning Results . . . 81

Chapter 6 Improving Kremlin's Practicality . . . 85
6.1 Efficient Shadow Memory Organization . . . 86
6.2 Static Partial Evaluation of CPA . . . 91
6.3 Evaluation . . . 94
6.3.1 Shadow Memory Optimization . . . 95
6.3.2 Static Partial Evaluation of CPA . . . 98

Chapter 7 Related Work . . . 100
7.1 Parallelism Discovery . . . 100
7.2 Parallelism Planning . . . 102
7.3 Performance Prediction . . . 103
7.4 Shadow Memory Design . . . 105
7.5 Parallel Performance Debugging Tools . . . 106

Chapter 8 Summary 108

Bibliography 112

LIST OF FIGURES

Figure 1.1: A Taxonomy of Parallelization Tools 4

Figure 2.1: Speedup vs. Time Graphs for All Users (sift) 20

Figure 3.1: Kremlin's Usage Model 27
Figure 3.2: Overview of the Kremlin System Architecture 30

Figure 4.1: Localizing Parallelism 37
Figure 4.2: HCPA Hierarchical Region Model and Summarization 40
Figure 4.3: Traditional Shadow Memory Organization 41
Figure 4.4: Calculating Parallel Time with Shadow Memory 42
Figure 4.5: Shadow Memory and Region Hierarchy 44
Figure 4.6: Level-based Sharing of Shadow Memory Tags 46
Figure 4.7: Parallelism Chart for MPEG Encoder 50
Figure 4.8: Self-Parallelism Scenarios 54
Figure 4.9: Self-Parallelism with Pipeline Parallelism 55
Figure 4.10: Uncovering Hidden Parallelism with Self-Parallelism 56
Figure 4.11: Classification of Regions Based on Total- and Self-Parallelism . 58
Figure 4.12: Percentage of Regions Parallelized as a Function of Parallelism and Work 59

Figure 5.1: Shortcomings of Greedy Planning 69
Figure 5.2: Evaluating the Performance of Kremlin-based Parallelization . . 77
Figure 5.3: Effectiveness of Region Prioritization 78
Figure 5.4: Effects of Factors on Plan Size 81

Figure 6.1: Efficient Shadow Memory Organization 87
Figure 6.2: Exploring Optimization Possibilities 91

LIST OF TABLES

Table 1.1: Auto-Parallelization Performance 5

Table 2.1: Speedup and Recommendation Rank of Critical Regions 17
Table 2.2: Speedup Achieved and Time Spent 19
Table 2.3: Time Spent (%) Parallelizing Critical Regions 21

Table 4.1: Region Key for MPEG Encoder Benchmark 49

Table 5.1: Evaluating Plan Size 76
Table 5.2: Marginal Benefit of Region Prioritization 80
Table 5.3: OpenCL Planning Results 82

Table 6.1: Shadow Memory Overheads for HCPA 86
Table 6.2: Memory Usage with Optimized Shadow Memory 95
Table 6.3: Performance Impact of Optimized Shadow Memory 97
Table 6.4: Speedup From Static Partial Evaluation of CPA 98

ACKNOWLEDGEMENTS

This thesis would not have been possible if not for the support of a great many people. My advisor, Michael Taylor, provided invaluable feedback on my research and created an environment that allowed me to blossom as a researcher. His advice allowed me to keep the proper perspective on research and life through the inevitable ups and downs of my studies.

Others have also provided me great guidance–both formal and informal–during my career. In particular, I would like to thank the following people for their advisory roles: the members of my thesis committee; my former advisor, Alex Orailoglu; and my undergraduate advisors at Drexel University, Moshe Kam and Kapil Dandekar.

The work presented in this thesis was a close collaboration with several highly talented individuals. I am especially thankful to my colleague and friend, Donghwan Jeon, with whom I enjoyed many a delightful conversation. I would also like to acknowledge the hard work and intelligence of Chris Louie, who I was lucky enough to mentor over several years. Of course, I would be remiss if I did not acknowledge the valuable feedback received from my current and former officemates as well as the members of the architecture and programming systems groups here at UCSD.

Finally, I offer my heartfelt thanks to my family for their unwavering belief in me. Throughout my life they have always encouraged and supported me while I have pursued my dreams, however crazy they may have seemed.

Chapters 1, 2, 3, 4, 5, and 7 contain material from "Kremlin: Rethinking and Rebooting gprof for the Multicore Age", by Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor, which appears in *PLDI '11: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. The dissertation author was the primary investigator and author of this paper.

copyrighted component of this work in other works.

Chapter 4 contains material from "Bridging the Parallelization Gap: Automating Parallelism Discovery and Planning", by Saturnino Garcia, Donghwan Jeon, Chris Louie, Srivanthi Kota-Venkata, and Michael Bedford Taylor, which appears in *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2010. The dissertation author was the primary investigator and author of this paper.

VITA

2005	B. S. in Computer Engineering Drexel University Philadelphia, Pennsylvania
2005-2012	Graduate Research Assistant University of California, San Diego
2006-2008,2011	Teaching Assistant University of California, San Diego
2007	M. S. in Computer Science University of California, San Diego
2011	Instructor University of California, San Diego
2011-2012	Master Teaching Assistant University of California, San Diego
2012	Ph. D. in Computer Science University of California, San Diego

PUBLICATIONS

Saturnino Garcia, Donghwan Jeon, Chris Louie, Michael Bedford Taylor, "The Kremlin Oracle for Sequential Code Parallelization", *IEEE Micro*, *To appear*.

Donghwan Jeon, Saturnino Garcia, Chris Louie, Michael Bedford Taylor, "Kismet: Parallel Speedup Estimates for Serial Programs", *Proceedings of ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, October 2011.

Saturnino Garcia, Donghwan Jeon, Chris Louie, Michael Bedford Taylor, "Kremlin: Rethinking and Rebooting gprof for the Multicore Age", *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2011.

Donghwan Jeon, Saturnino Garcia, Chris Louie, Michael Bedford Taylor, "Parkour: Parallel Speedup Estimates for Serial Programs", *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, May 2011.

Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddharth Nath, Vikram Bhatt, Jonathan Babb, Steven Swanson, Michael Bedford Taylor, "The GreenDroid mobile application processor: An architecture for silicons dark future", *IEEE Micro*, March/April 2011.

Saturnino Garcia, Donghwan Jeon, Chris Louie, Sravanthi Kota Venkata, Michael Bedford Taylor, "Kremlin: Like gprof but for Parallelization", *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2011.

Jack Sampson, Ganesh Venkatesh, Nathan Goulding, Saturnino Garcia, Steven Swanson, Michael Bedford Taylor, "Efficient complex operators for irregular code", *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2011.

Saturnino Garcia, Donghwan Jeon, Chris Louie, Sravanthi Kota Venkata, Michael Bedford Taylor, "Bridging the Parallelization Gap: Automating Parallelism Discovery and Planning", *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, June 2010.

Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, Michael Bedford Taylor, "Conservation Cores: Reducing the Energy of Mature Computations", *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2010.

Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Chris Louie, Saturnino Garcia, Serge Belongie, Michael Bedford Taylor, "SD-VBS: The San Diego Vision Benchmark Suite", *Proceedings of IEEE International Symposium on Workload Characteristics (IISWC)* October 2009.

Saturnino Garcia, Alex Orailoglu, "Making DNA Self-Assembly Error Proof: Attaining Small Growth Error Rates Through Embedded Information Redundancy", *Proceedings of Design, Automation, and Test in Europe Conference & Exhibition (DATE)*, April 2009.

Saturnino Garcia, Alex Orailoglu, "Online Test and Fault-Tolerance for Nanoelectronic Programmable Logic Arrays", *Proceedings of the International Symposium on Nanoelectronic Architectures (NANOARCH)*, June 2008.

Gustave Anderson, Leonardo Urbano, Gaurav Naik, David Dorsey, Andrew Mroczkowski, Donovan Artz, Nicholas Morizio, Andrew Burnheimer, Kris Malfettone, Dan Lapadat, Evan Sultanik, Saturnino Garcia, Max Peysakhov, William Regli, Moshe Kam, "A Secure Wireless Agent-based Testbed", *Proceedings of IEEE International Information Assurance Workshop (IWIA)*, April 2004.

ABSTRACT OF THE DISSERTATION

A Practical Oracle for Sequential Code Parallelization

by

Saturnino Garcia, Jr.

Doctor of Philosophy in Computer Science

University of California, San Diego, 2012

Professor Michael Taylor, Chair

With the relatively recent switch from single- to multi-core processors, parallelism now plays a much larger role in maximizing program performance. This switch calls for converting the existing serial implementations of programs into parallel implementations in order to ensure scalable performance on future generations of processors. While automated tools to perform this conversion have been developed, the resulting performance often significantly lags behind that of manually parallelized code. This gap in performance has led researchers to develop tools that ease the manual parallelization process. These tools have greatly simplified the later stages of parallelization, but they provide no assistance with one of the primary questions faced by programmers: *"Which parts of this program should I spend time parallelizing?"*.

In this dissertation we examine the design and implementation of Kremlin, a practical oracle for the parallelization of sequential programs. Kremlin predicts the outcomes of parallelization in order to guide the programmer towards regions of the program that will be most fruitful for parallelization. Kremlin accomplishes this task by extending a classic technique, critical path analysis, to make it practical for

two often-overlooked phases of parallelization: parallelism discovery and planning. This oracle requires only unmodified serial source code, a representative set of inputs, and simple system parameters such as the number of cores to produce a parallelization plan that prioritizes regions by their potential parallel speedup. Our results highlight Kremlin's utility as a practical oracle: parallelization guided by Kremlin results in fewer program regions being parallelized (1.57×, on average) in order to achieve peak parallel performance.

Chapter 1

Introduction

The emergence of multi-core processors over the past decade has been a disruptive force in software engineering. For a half century before the switch to multi-core, software engineers benefited from a stable hardware interface that presented the abstraction of a single processing core. The single-core abstraction was not the only abstraction available to programmers but it has been the dominant one; abstractions for multiprocessors (e.g. simultaneous multiprocessing) have a long history but were largely ignored by mainstream software engineers as multiprocessor systems have until recently been the exception rather than the rule.

The abstraction of a single core processor hid many of the changes made to computer architecture during this time. This has allowed software engineers to see exponential increases in the performance of their programs as new processors were developed–all without any changes to their code. While multi-core processors support the single-core abstraction, the free lunch appears to be over: software engineers can continue to write code as they always did but the performance of this code will no longer see exponential improvements from generation to generation.

Much of the performance available in multi-core processors is tied to the abundance of available hardware parallelism. Computer architects continue to look for ways to use this hardware parallelism to benefit single-thread programs [KST10, KLW+04, TT11], but unlocking the full potential of hardware parallelism will require the use of software parallelism.

Software parallelism comes in many forms, both within a program and

across multiple programs. Parallelism across programs requires little from programmers but is limited in its utility: in many contexts the number of concurrent, active programs is limited, placing an upper bound on the number of cores that are useful. This type of parallelism also does not provide scalable performance, only attempting to minimize the performance degradation that occurs when multiple programs are concurrently executing. Far more important for scalable performance is the other type of parallelism, that which comes from within the program itself.

Parallel programming has a rich history, but the scarcity of multiprocessor systems has limited parallel programming to a small number of domains, e.g. scientific computing. Parallel programming has gained a reputation as being extremely difficult, which is understandable given the difficulties humans have with thinking about multiple concurrent events: humans are notoriously poor multitaskers. Parallel programming requires a unique set of skills, which a vast majority of programmers have largely ignored as decades of exponential improvements in single-threaded performance have minimized their importance. The rise of multicore processors has changed this equation, forcing mainstream programmers to question how to exploit the parallelism that may be present in their existing serial programs.

Automatic parallelizing compilers such as Polaris [BDE+02], SUIF [HAA+96], and RawCC [LBF+98] offer a fully automated approach to program parallelization. These approaches are obviously desirable for software engineers but the performance of the code they generate often pales in comparison to the code generated by a skilled human. These fully automated tools are hampered by the ambiguity that arises from a lack of semantic information during static analysis. This lack of information forces these compilers to be conservative in order to maintain program correctness. Researchers have looked at the possibility of allowing the programmers to annotate programs with semantic information but these approaches include a large, possibly non-trivial manual component and have not seen widespread adoption by software engineers.

Programmers who wish to obtain maximum parallel performance have little recourse other than manual parallelization. With this situation in mind there has

been a push to develop tools to aid in manual parallelization. One area that has seen notable success is this regard is the development of language extensions or parallel libraries. OpenMP [DM98] and Cilk++ [Lei09] are two examples. Both of these language additions make it trivial to express parallelism in many instances, requiring the addition of only a single line of code in some cases. These approaches have the potential to increase programmer productivity when moving from earlier, low-level approaches such as `pthreads` in much the same way that moving from assembly languages to high-level programming languages increased productivity in the 1950's and 60's.

Researchers have also developed tools to help the parallel programmer debug both the correctness and performance [TMC09, HLL10, AL90] of their parallel programs. While all of these tools are of assistance to software engineers, they do not form a complete picture of the parallelization process. These tools rely on the basic assumption that programmers have knowledge of which parts of the program have parallelism and should be parallelized. These are non-trivial tasks, even for moderately-sized software projects. What are needed are tools to help the programmer with these critical but often overlooked aspects of parallelization.

In the following section, we will discuss a taxonomy of parallelization tools that we developed to illuminate the end-to-end parallelization process and to identify weaknesses in the existing set of tools. The insights gained from creating this taxonomy led us to the creation of Kremlin, a practical oracle for the parallelization of sequential programs. This oracle predicts the outcome of parallelization, providing guidance to the programmer during the initial phases of parallelization.

1.1 Taxonomy of Parallelization Tools

Figure 1.1 presents our taxonomy for parallelization tools. This taxonomy contains five basic phases that are needed to convert the serial implementation of a program into a parallel implementation.

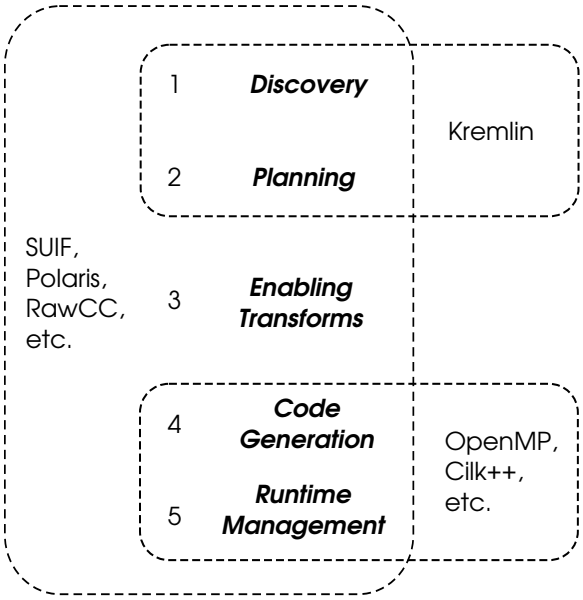


Figure 1.1: **A Taxonomy of Parallelization Tools.** The taxonomy categorizes parallelization tools based on which of five fundamental parallelization stages they assist with. Automatic parallelizing compilers like Polaris [BDE+02] and SUIF [HAA+96] attempt to perform all five without programmer assistance, while tools like OpenMP, Cilk++ [Lei09], and X10 [SSvP07] focus on the last two. We have developed Kremlin, a tool that focus on the first two stages.

1.1.1 Overview of Parallelization Stages

Parallelization starts with *Parallelism Discovery* where the program is inspected to determine which of its regions contains parallelism. Parallelism comes in many forms, not all of which will be exploitable on any given system. This stage must therefore also be capable of distinguishing between different types of parallelism so as to give a more accurate picture of the program's potential. The second stage is that of *Parallelism Planning*, which is where decisions must be made regarding which regions of the program should be parallelized. Planning for an ideal system is trivial: simply parallelize every region that has parallelism, prioritizing those regions with a combination of high coverage and high parallelism. Planning for real systems is unfortunately non-trivial. The planner must account for the exploitability of specific types of parallelism, limitations on the amount of parallel resources available, and overhead introduced from parallelization. The third stage

Table 1.1: **Auto-Parallelization Performance.** For the Rodinia [CBM+09] benchmark suite, Intel's `icc` compiler successfully auto-parallelized only 17% of the program regions that were parallelized by humans despite the relative simplicity of the benchmarks. Further analysis revealed that over half of the regions were not auto-parallelized because of dependencies that could not be resolved statically, a key limitation of parallelizing compilers.

Benchmark	Parallelized (Human)	Parallelized (`icc`)	**Success Rate (`icc`)**
backprop	2	0	**0%**
bfs	2	0	**0%**
heartwall	1	0	**0%**
hotspot	2	1	**50%**
kmeans	1	0	**0%**
lavaMD	1	0	**0%**
lud	2	0	**0%**
nn	1	0	**0%**
nw	2	0	**0%**
pathfinder	1	0	**0%**
particlefilter	10	2	**20%**
srad	2	2	**100%**
streamcluster	2	0	**0%**
Total	29	5	**17%**

in our taxonomy is *Enabling Transforms*. This stage involves transforming the program to exploit the parallelism that was detected in the first stage. The transforms required here vary widely in their complexity; the simplest cases require only privatizing some variable or arrays while more complex cases may require intricate transformations such as loop skewing and interchange. The final two stages of parallelization, *Code Generation* and *Runtime Management*, deal with generating parallel code and providing the runtime environment in which the parallelism is exploited with minimal overhead.

1.1.2 Existing Parallelization Tools

Fully Automated Tools As shown in Figure 1.1, automatic parallelizing compilers such as Polaris [BDE+02], SUIF [HAA+96], and RawCC [LBF+98] attempt

to automate all stages of our taxonomy. This approach eliminates the need for programmers to modify their programs, but it often results in performance that pales in comparison to manual approaches.

Parallelizing compilers have the onus of ensuring correctness so they must prove the safety of any transforms they perform, mainly using only static program analysis. Proving correctness is not possible in many cases without the benefit of additional runtime or semantic information to resolve ambiguities. Table 1.1 demonstrates the difficulty a state-of-the-art parallelizing compiler, `icc`, has in proving the safety of even simple transformations. `icc` could parallelize only 17% of the regions that were manually parallelized despite all required program transformations having already been manually performed. Our analysis revealed that over 50% of the regions failed to be parallelized because `icc` could not statically resolve potential dependencies. Tournavitis et al [TWFO09] also demonstrated the poor performance of `icc` on the generally high-parallelism benchmarks in NPB [BBB+91].

Programmer-Oriented Tools The often lackluster performance of these completely automated approaches has given rise to an alternative approach, one centered around providing automated tools focusing on specific parts of the taxonomy. As previously mentioned, tools such as OpenMP, Cilk++ [Lei09], X10 [SSvP07], and Fast Track [KBDZ09] have greatly eased the process of indicating parallel regions of the code. These tools correspond to the final two stages of our taxonomy. Researchers have spent considerably less time on tools for the first three stages. While some tools have been developed for automating the enabling transforms stage [DME09, WST09], this stage is the most difficult and benefits most from manual intervention. It is therefore of little surprise that few tools have been developed for this stage; what is surprising is the lack of tools for the first two stages, discovery and planning.

Existing tools for parallelism discovery tend to rely either on critical path analysis (CPA) [Kum88, HSHZ09, RVVYS10] or on dependence testing [Lar93, ZNJ09, KKL10], both of which are poorly suited as precursors for parallelism planning. CPA is useful for quantifying the amount of parallelism within a program but

is overly optimistic, leading to inaccurate estimates of the potential speedup from parallelization. CPA also looks only at the program as a whole, severely limiting its utility in quantifying the impact of parallelizing only parts of the program. Dependence testing reports which regions can be executed concurrently but is closely tied to program structure, leading latent parallelism to go unreported. Dependence testing also does not quantify parallelism, making it difficult to estimate parallel speedup in all but the most trivial cases.

Parallelism planning has largely been overlooked by the research community. While automatic parallelizing compilers implicitly do planning, their planning algorithms are generally not applicable to programmer-oriented planning. As previously discussed, planning is a non-trivial task that must take into account many complex, interacting factors. Without automated tools for this task, programmers are often forced into an *ad hoc* planning methodology that reduces their productivity and can lead to suboptimal parallel performance.

1.2 A Practical Oracle for Parallelization

Software engineers must confront the following question at the very beginning of parallelization: "*What parts of the program should I spend time parallelizing?*". Answering this question manually often requires years of parallel programming experience in addition to a detailed understanding of the program's code structure and dependencies. Neither of these are easy to come by so programmers are forced to consult one or more tools for guidance.

Programmers attempt to act as *seers*, interpreting the results of existing tools to determine a plan for parallelization. Unfortunately, these tools offer mostly vague or misleading guidance. The confusing nature of these tools' results springs from a number of sources. First, these tools may have incomplete knowledge of parallelization. For example, programmers often repurpose serial profilers (e.g `gprof`) for parallelization planning, despite the fact that high coverage does not correlate with high parallelism. Second, these tools may guide programmers down dead-end paths or fail to alert them of important opportunities. For example, pro-

grammers consulting critical path analysis (CPA) or dependence testing tools are likely to waste time on serial regions or to ignore parallel regions masked by serial implementations. Finally, these tools may lack sufficient foresight. For example, tools that rely on greedy algorithms to select regions to parallelize may suggest a parallelization whose benefit is later negated by another suggested parallelization.

Instead of forcing programmers to act as seers, we should aim to provide them with a tool that acts as a parallelization *oracle*. We have designed Kremlin to meet this goal, and act as a *practical* oracle for parallelization of sequential programs. As a parallelization oracle Kremlin predicts the extraordinarily complex outcomes of parallelization with uncanny precision, offering its users an ordered parallelization plan to guide them through the regions of the program they should parallelize. Unlike other tools, Kremlin is a practical tool for parallelization: Kremlin makes its predictions using only serial source code, a sample input, and a simple description of the target platform.

Kremlin builds upon critical path analysis, but introduces several novel techniques to make CPA suitable as a basis for a practical oracle: a new type of dynamic program analysis called *hierarchical critical path analysis* (HCPA); the concept of *self-parallelism*; a lightweight method for approximating self-parallelism; and the concept of a *planning personality* that tailors a parallelization plan to a specific target system. These contributions will be discussed in detail throughout the rest of this dissertation.

Kremlin has shown to be effective at reducing the number of program regions that need to be parallelized, with its recommendations leading to an average 1.57× fewer parallelized regions when compared to an expert, third-party implementation. This reduction in regions comes with little-to-no impact on performance compared to the third-party implementation, and in several cases greatly exceeds third-party performance. Results also show that Kremlin accurately orders region recommendations, with an average of 86.4% of the total parallel speedup available after completing only the first half of recommendations.

1.3 Thesis Organization

The rest of this thesis will have the following organization. In Chapter 2 we will examine the ineffective parallelization methodology that many programmers employ today. This chapter will also present the results of a user study that we performed to quantify the cost of inefficient parallelism planning. This user study used an early prototype of Kremlin; we will discuss how the results of this study helped shape the direction of Kremlin as a practical oracle.

In Chapter 3 we will look at Kremlin's usage model and present a high-level system overview. This overview will tie together Kremlin's individual components, which will be described in further detail in the subsequent chapters.

Chapter 4 presents the techniques we developed for planning-aware parallelism discovery. This chapter will first provide background on critical path analysis (CPA), a classic technique for quantifying the amount of parallelism in a program. We will then look at extending CPA to make it useful as part of a practical oracle, culminating with two novel contributions: hierarchical critical path analysis (HCPA) and self-parallelism. HCPA enables analysis of individual regions of a program (e.g. functions and loops), analyzing all regions in only a single pass. Self-parallelism is a new metric that quantifies the amount of parallelism in a program region, excluding the parallelism that comes from regions nested below it. HCPA allows Kremlin to closely approximate self-parallelism without the intractable requirement to store the complete program dependency graph. The results section in this chapter will look at well self-parallelism filters serial regions that would otherwise be considered parallel. We will also look at how well self-parallelism correlates with the regions of a program that were parallelized by third-party experts, showing that self-parallelism matches well with a "ground truth" for parallelizability.

Chapter 5 describes how we can transition from parallelism discovery to parallelism planning. This chapter will begin by describing the problem of parallelism planning both formally and informally. Discussion will then move to two critical components in predicting the result of parallelization: determining which regions contain exploitable types of parallelism and estimating the time of regions

after parallelization. We will then examine how system-specific constraints can be accounted for during planning by introducing the concept of a planning personality. Our discussion will include several proposed planning personalities and the process of developing additional personalities. Results in this chapter will show the effectiveness of Kremlin as a practical oracle. We will look at Kremlin's ability to reduce the number of regions and how that affects the speedup from parallelization. We will also examine Kremlin's ability to accurately prioritize its recommendation, specifically focusing on how much speedup is attainable after implementing a limited percentage of Kremlin's recommendations.

One of Kremlin's main contributions, HCPA, builds upon CPA and is similarly a heavyweight dynamic analysis. This heavyweight nature imposes challenges to the practicality of Kremlin on many systems: high runtime overhead and/or memory overhead could limit Kremlin's utility on all but the highest-end systems. Chapter 6 will look at two separate approaches to limiting both of these types of overhead in HCPA. The first techniques is a novel shadow memory architecture that makes the common case fast and the uncommon cases space-efficient. The second technique utilizes static program analysis to partially evaluate critical paths. Results in this chapter show that the first technique reduces memory overhead –while not greatly increasing runtime overhead– to the point where even standard laptops can evaluate programs using sizable inputs. Results also show that the second technique can significantly reduce the runtime overhead while leaving the memory overhead unchanged.

We will discuss related work in Chapter 7 before summarizing our findings and offering concluding remarks in Chapter 8.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under CAREER Award 0846152, by NSF Awards 0725357, 0846152, and 1018850, and by a gift from Advanced Micro Devices.

This chapter contain materials from "Kremlin: Rethinking and Rebooting

gprof for the Multicore Age", by Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor, which appears in *PLDI '11: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. The dissertation author was the primary investigator and author of this paper.

Chapter 2

The Cost of Inefficient Parallelization

Our introduction described how the process of parallelization was hindered by a lack of tools for several important stages of parallelization, namely parallelism discovery and parallelism planning. In this chapter we will take a look at how the parallelization methodology currently used by many programmers leads is highly inefficient and can lead to poor parallel performance. We will also look at a user study we performed after having developed an early prototype of Kremlin. This user study helped us quantify the impact of inefficient parallelization methodology and provided insight for Kremlin's later designs.

2.1 Example Parallelization Methodology

Despite recent research into parallel programming tools, many programmers still rely on a relatively painful methodology that employs serial profiling tools such as `gprof` in order to direct their parallelization activities. The process starts with a serial hotspot list, which ranks regions by the amount of time spent inside them. This list effectively becomes the order that they examine the functions to improve their performance.

It is at this point that the process gets especially onerous. The programmer starts leafing through the code trying to puzzle through the dependencies in the

code, and the granularity at which to try to exploit it. Since the programmer has no indication of whether a hotspot is parallelizable, they frequently give up before they are able to recognize subtle but large parallelism opportunities, or they spend excessive amounts of time fruitlessly modifying serial parts of the code. Alternatively, even if parallelism does exist, it may not be large enough to yield speedup, or when combined with lower coverage, the overall speedup may not justify the effort. Finally, interference between nested parallel regions may prevent speedup.

We examined the feature_tracking benchmark from the San Diego Vision Benchmark Suite (SD-VBS) [KVAJ+09] to illustrate the shortcomings of this coverage-based approach to parallelizing programs. The programmer is typically interested in exploiting either loop-based parallelism (e.g. with OpenMP) or task-based parallelism (e.g. with Cilk++). These types of parallelism require examining the loops and functions in a program so we profiled feature_tracking to determine the work coverage of its loops and functions. The resulting profile data was then used to sort the loops and functions from largest to smallest coverage.

To demonstrate the lack of correlation between coverage and parallelizability, we manually analyzed the top 20 hotspots in feature_tracking to determine which of them could be parallelized. Of these top 20 regions, over 50% were either serial (3 functions, 3 loops), contained very limited parallelism (4 functions), or required significant program restructuring to exploit the available parallelism (1 loop). The remaining regions (9 loops) were easily parallelizable, requiring only minor code transformations such as privatization.

Our analysis suggests that relying on `gprof` or similar profiling tools will lead to less than optimal results. The programmer would be required to analyze a large number of regions that are not parallelizable. This inefficiency results from favoring execution coverage over the other factor in parallelizability: the amount of parallelism available.

While useful, tools that quantify parallelism are alone not quite enough. Rather, parallelization also requires planning tools that help process this information and apply both parallel programming system and machine constraints. With

the ability to positively identify the existence of parallelism, and also to prioritize regions, users can invest their time more productively, attacking the correct portions of the program.

2.2 User Study

Our own experience with parallelization led us to conclude that current parallelization methodologies–like the one described earlier–are inefficient and can result in poor parallel implementations. This qualitative assessment led us to create an early version of Kremlin that used critical path analysis to quantify the parallelism in every function and loop in the program. This parallelism info was then used to roughly estimate the ideal performance of parallelizing a function or loop and was therefore subsequently used for rudimentary parallelism planning.

We used this early version of Kremlin in a user study that was designed to quantify the impact of parallelism discovery and planning on the parallelization process[1]. More specifically, the user study was designed to answer the following two research questions:

1. Would Kremlin users be able to achieve significant speedups sooner than the non-Kremlin users?

2. Would Kremlin users spend more time focusing on regions with the largest potential speedup than non-Kremlin users?

In order to answer these questions, we set out to measure both the order in which the participants attempted parallelization and the outcome of their efforts. User study participants included seven graduate students in a parallel architecture course at the University of California, San Diego.

[1]This user study was approved by the UC San Diego Institutional Review Board, Project #100056.

2.2.1 Setup

The user study was spread over three class assignments, with each assignment asking them to parallelize one benchmark from SD-VBS using Cilk++. Users were given access to a supercomputing cluster at the San Diego Supercomputer Center where they would have access to AMD computing nodes with up to 32 cores. To ensure a uniform work environment, they were provided with a Makefile system with a small number of commands to control compilation, execution, and debugging of their parallel programs. The result of all commands were logged so that we could accurately retrace the steps they took. After each run of their program, a snapshot of their code was saved so that we could recreate the changes they made as they worked on the assignments.

Both of our research questions relied on knowledge of the time spent on various tasks; accurate accounting of time was therefore of the utmost importance. For each command the participants ran, users were asked to enter in the amount of time they had spent actively working on the assignment since they last reported. Using this data, we could determine how long they spent on each task. It is well-known that self-reported data can be inaccurate so we did not rely solely on this data for recreating the time spent on each task. We instead bolstered the self-reported data by incorporating some automatic data that was available from the logging we did with each command. This hybrid approach is similar to the one described in [HBZ+05]. To further increase our understanding of their efforts, we also prompted users to optionally describe their work activities since their last reported time.

The study was split across three class assignments. The first assignment acted as a primer for parallel programming using Cilk++ and for using the parallel programming environment we created. The students were given an hour-long demonstration of how to use our programming environment as well an introduction to using Cilk++ for parallelization. The students were also given a tutorial on effectively using Kremlin before their first required use of the tool.

We split students into two groups, groups A and B, for the final two assignments: one with access to the early Kremlin prototype and another without

access. The group without access to Kremlin was the control group that allowed us to factor out the variation in difficulty of parallelizing across the different programs. The control group switched between the assignments two and three: group A was the control for assignment two while group B was the control for the assignment 3. Both groups were given access to `gprof` as a tool to help them plan for which regions to parallelize.

We formed the two groups in the following way. At the end of the first assignment we analyzed the users' logged data for the amount of time spent and the speedup achieved. We used this data to compute a performance-to-work ratio, which we used to rank each student's performance. We used this ranking to split the students into two groups of roughly equal skill.

2.2.2 Benchmark Analysis

Before looking at the results of the user study, we will analyze each of the three benchmarks in detail to identify the regions that were the most critical in achieving the best parallel implementation.

We parallelized the programs to get a reference version with the best speedup, and noted which functions and loops were required to be parallelized in this optimal implementation. We refer to these regions as the "critical" regions. The benchmarks were of limited size, allowing us to be confident that we had achieved the best speedup possible. After gathering the results, we checked the students' implementations for any possible speedups that we missed in the reference implementation but did not find any.

Table 2.1 details the regions that were found to be critical for each program. Also given in this table is the speedup that could be achieved by parallelizing this region and any that are listed above it for that program. This number formed a bound on what we believe to be the best speedup attainable using only parallel programming transformations.

Disparity Disparity was the first benchmark that was assigned and was also the simplest to understand, containing less than 500 lines of code. Based on our

Table 2.1: **Speedup and Recommendation Rank of Critical Regions.** Critical regions were those regions found to be essential in obtaining maximum parallel performance. These regions were not the only ones that benefited from parallelization but parallelizing other regions in addition to these would not lead to higher speedups. Also shown are the order in which these regions were recommended by the early Kremlin prototype and `gprof` (based on both self- and total-time).

			Recommendation Rank		
Benchmark	Function	Speedup	Kremlin	gprof self-time	gprof total-time
disparity	getDisparity	3.5	1	14	2
tracking	script_tracking	1.3	1	NA	1
	calcPyrLKTrack	1.2	5	14	4
	calcGoodFeature	2.1	4,10	17	10
sift	imsmooth	4.1	1,2	1	4
	sift	6.4	3	2	2
	fSetArray	8.4	5	4	6
	diffss	10.1	4	3	5

analysis we found that parallelizing a single function, `getDisparity`, was enough to achieve the best speedup available. `getDisparity` consisted of a loop where the image disparity was found at many different levels. The disparity in each iteration could be calculated independently of the other iterations and thus most of the work in the loop could be done in parallel. However, at the end of each iteration, the current disparity value was checked against the best value so far. The loop could be parallelized with modest effort by privatizing the intermediate disparity arrays and distributing the comparison into another (sequential) loop afterwards.

Scale-Invariant Feature Transform (SIFT) Our analysis determined the four most important regions in the SIFT program to be `imsmooth`, `sift`, `fSetArray`, and `diffss`. The `imsmooth` function contained two loops which were easily parallelized with the `cilk_for` primitive but which could achieve a speedup of over 4×. The `fSetArray` and `diffss` functions had a similar format and difficulty but only one of the loops was important for performance. The `sift` function, although offering ample potential speedup was much harder to parallelize.

Feature Tracking For the feature tracking algorithm, we identified three critical functions: `script_tracking`, `calcPyrLKTrack`, and `calcGoodFeature`. The for loop in the `script_tracking` function processed each frame based on the previous frame's output. Thus, it had a strong sequential component. However, each image is preprocessed before the features are located. This preprocessing step could be done in parallel for all images since it does not depend on previous images. Refactoring the code such that the preprocessing occurred before the main for loop therefore offered a significant potential speedup (1.36×). Identifying and refactoring the main for loop was a non-trivial task with only three students successfully accomplishing this task.

The loop in the `calcPyrLKTrack` function offered similar speedup potential but involved much easier code transformations. To successfully parallelize this loop, several variables needed to be privatized. All but one of the students was able to successfully parallelize this region.

Finally, the `calcGoodFeature` function also offered a significant potential speedup–albeit much smaller than `script_tracking` or `calcPyrLKTrack`. This function was trivially parallelizable, requiring the addition of only two `cilk_for` keywords to the loops in the function.

2.2.3 Impact on Program Speedup

One of the questions we wished to answer was if access to Kremlin would effect the speedups obtained by the user. We might expect access to the tools to have one or both of the following effects:

1. Improved performance at the end of the parallelizing process.

2. Decreased time required to obtain the largest possible speedup.

To test whether the final performance improved as a result of using Kremlin, we tabulated the final speedup time for all users and calculated the average value for Kremlin users and the average for non-Kremlin users. Table 2.2 shows this result. While the Kremlin users had slightly better performance than non-users

Table 2.2: **Speedup Achieved and Time Spent.** Kremlin did not have a strong impact on the final speedup obtained by the participants: tracking was nearly the same (1.8 with Kremlin, 1.7 without), while sift had a significant difference (7.2 with, 9.2 without) due mostly to the lack of effort (and subsequent poor performance) of user 139. The lack of difference can also be explained by the limited complexity of the benchmarks compared with the time given to parallelize them.

	tracking		sift	
User	Speedup	Time	Speedup	Time
139	1.6	235	2.4	88
280	2.3	1195	7.8	279
579	1.2	176	6.9	274
911	1.6	405	11.5	375
143	2.2	602	7.4	184
249	1.6	528	11.4	401
371	1.4	737	8.7	297
Kremlin	1.8	622.3	7.2	254
non-Kremlin	1.7	502.7	9.2	294
Average	1.7	554	8.0	271.1

for tracking, the difference (1.8 vs 1.7) was not significant considering the small sample size. In sift, the average for Kremlin users is well below that of the non-users (7.2 vs 9.2) but performance of user 139 was clearly an outlier that skewed the average. The average excluding this user rises to 8.8, but the difference is still too small lead to conclusive results.

To determine if Kremlin was able to decrease the time required to obtain the maximum speedup, we looked at each users speedup as a function of the amount of time they worked. If Kremlin did make an impact, we would expect the slope of Kremlin users to be larger than that of non-Kremlin users. Figure 2.1 shows the speedup vs time graphs for each of the users on the sift benchmark. Also indicated in this figure is the time spent parallelizing critical regions (light gray), time spent on non-critical regions (dark gray) and time spent on sequential optimization (white). The top 4 users (139, 280, 579, and 911) were Kremlin users. While there is no clear trend on the slopes of Kremlin vs non-Kremlin, it

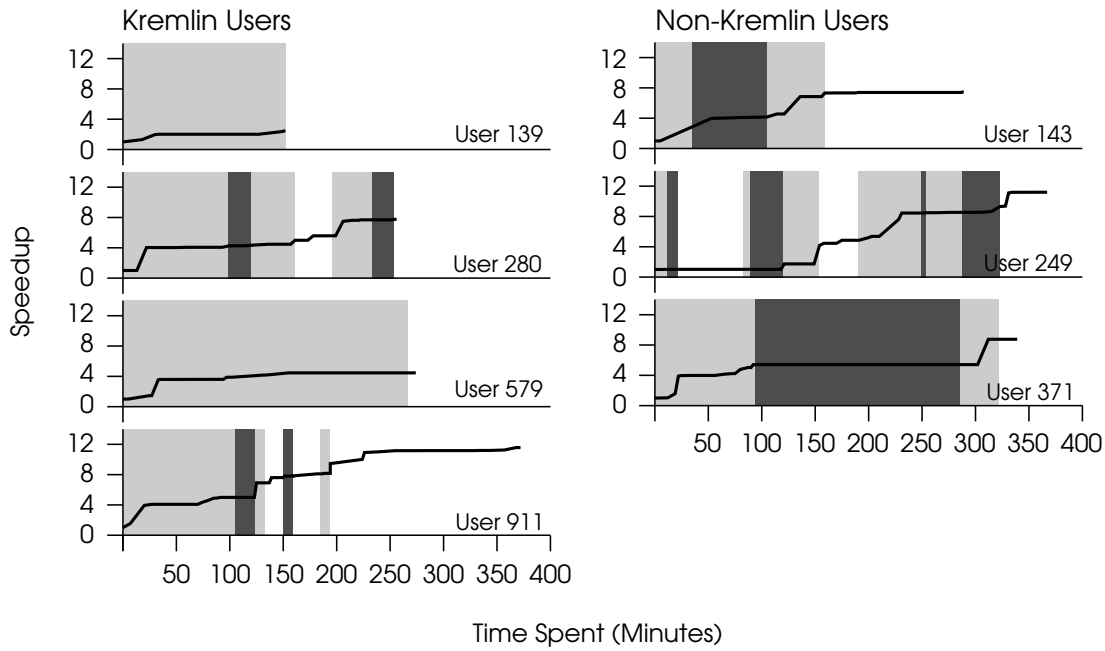


Figure 2.1: **Speedup vs Time Graphs (sift).** The graphs show the speedup obtained by each user as a function of the amount of time spent. Also shown is a breakdown of the periods of time spent on different types of regions; light gray indicates time spent in critical regions, dark gray indicates time spent in non-critical regions, and while indicates time spent doing serial optimization. The graphs indicate that Kremlin users spent much more of their time parallelizing critical regions than the non-Kremlin users.

is clear to see that the times were spent very differently between the two groups. Kremlin users had much more time in the critical regions (gray) than the others. Unfortunately the time spent in the regions does not seem to indicate the end success of the parallelization efforts or we would have seen a clearly steeper slope for Kremlin users.

2.2.4 Time Spent On Critical Regions

The other question which we wished to answer with the user study was whether Kremlin would direct users to spend more time working on parallelizing regions that offered the most speedup (i.e. the critical regions). To answer this question, we looked at the amount of time spent working on each function and calculated the percentage of this time that was spent working on critical regions.

Table 2.3: **Time Spent (%) Parallelizing Critical Regions.** The average time spent working on parallelizing "critical" regions greatly differed between Kremlin and non-Kremlin users. The Kremlin group spent an average of 83.9% (group A) and 83.6% (group B) of their time working in critical regions on the tracking and sift benchmarks, respective. This number dropped to 48.6% and 51.8% for the group without access to Kremlin.

		% Time Spent (Avg.)	
Benchmark	Critical Region	Group A	Group B
disparity	getDisparity	18.5	21.2
tracking	script_tracking	55.4	23.9
	calcPyrLKTrack	26.7	18.5
	calcGoodFeature	1.7	6.1
	Total	**83.9**	48.7
sift	imsmooth	12.0	29.4
	sift	30.3	41.5
	fSetArray	2.4	3.2
	diffss	7.0	13.5
	Total	51.9	**87.7**

Based on the quality of Kremlin's recommendations in Table 2.1 and its ability to predict speedup we know that Kremlin users had a good idea of what were the critical regions. We can therefore hypothesize that Kremlin users will show a clear advantage in the percentage of time spent working on critical regions.

Table 2.3 overviews the percentage of time each group spent working on each of the "critical" regions as well as the total time spent across all of these regions. To control for the variation caused by the time users worked on serial optimization rather than parallelization, we did not include the time spent doing serial optimization in the total time spent. Across both assignments, non-Kremlin users spent approximately 50% of their time working on the critical regions (48.7% and 51.9% for tracking and sift, respectively). In contrast, Kremlin users spent roughly 85% of their time working on these critical regions (83.9% and 87.7% for tracking and sift). This clearly shows that Kremlin was able to focus users on the regions that mattered most and is strong evidence for our hypothesis.

2.2.5 Threats to Validity

Having discussed the results of our user study, we will now examine how we addressed threats to the validity of our findings. In particular, we were concerned with threats to three types of validity: construct validity, internal validity, and external validity. Construct validity concerns whether our hypotheses are the best explanation of the results. Internal validity concerns whether the independent variables are responsible for the changes in the dependent variables. External validity concerns whether our results apply to a broader population than those involved in the study.

Construct Validity We faced one major threat to construct validity in this study. We needed to ensure that the speedup measured was the speedup obtained from parallelization and not from some other optimization. In the first two assignments we did not advise the participants against doing serial optimization and therefore several students achieved significant program speedup from serial optimization. In the third assignment we instructed the students to focus on parallel optimization as their grades would depend on that aspect of their speedup. However, as there was common code shared among the three programs used, some students used their old serial optimization on the final assignment. We controlled for this in two ways: we factored out the sequential time when calculating the percentage of time spent in critical regions; and we clearly labeled time spent on activities other than parallelization in the individual speedup vs. time graphs.

Internal Validity One possible threat to the internal validity of our study was misuse of the planner. Based on written reports and logged data, we found that some students did not use the tool as we had intended. In general, we found several major ways in which they deviated. First, because there was some code reuse between assignments, some students spent time re-implementing the same changes they had made in previous assignments. Often this meant that they optimized regions that were either not recommended or were only weakly recommended by the Kremlin. Another common way in which they deviated was by favoring gprof

results over Kremlin. `gprof` orders functions by the time spent inside of them. This was often misleading though as either the parallelism was too low in these functions or the parallelism was too fine-grained and did not offer significant performance gains. Finally, some students seemed to favor parallelizing on a first come, first serve basis. As a result, some students tried parallelizing functions that were called from the main function because those were the first they encountered. As a result of these deviations, we could not successfully test our initial hypothesis that Kremlin users would achieve significant speedups faster than non-Kremlin users.

External Validity One threat to the external validity of our study was the complexity of the programs that were parallelized in the study. These programs had a relatively small amount of code (less than 1000 lines of code) spread across 10-15 source files. A significant portion of the files had the same basic structure: two DOALL loops with one level of nesting each. Furthermore, several of these files were common files that were reused across the three assignments. This worked to limit the areas in which the students had to look for parallelism opportunities. The eight days given for each assignment was ample time for them to identify the major opportunities for exploiting parallelism. As a result of the limited complexity of these programs and the time they had to work on them, we did not expect Kremlin users to have significantly better performance than non-Kremlin users. However, we expect that Kremlin will have a more noticeable impact on more complex applications as its ability to focus users on the most critical regions becomes more important. Other threats to external validity include the limited parallel programming background of the participants and the lack of more sophisticated parallel performance measurement tools to help determine bottlenecks in performance.

2.2.6 Conclusions

While we were unable to show that the early Kremlin prototype was able to reduce the time needed to parallelize a program, our results indicate that users benefited from have a parallelism planner. To understand the strengths and weaknesses of early Kremlin prototype, we asked for qualitative feedback from the

users in the form of a survey of short answers. The combination of quantitative and qualitative feedback was able to provide us valuable insight that we would carry forward into Kremlin's later designs.

We noticed that despite having a planner, study participants often chose to ignore its advice. Students indicated that they were often unable to quickly determine the "trick" to successfully parallelizing a recommended region; when a recommendation looked too difficult to quickly parallelize, they simply moved on to a region that they thought we be easier. This initial confusion about how to parallelism was likely a direct result of a lack of guidance about the type of parallelism available in a region. The type of transformations needed to exploit parallelism is often directly tied to the type of parallelism. For example, DOALL parallelism in loops commonly require privatization of variables to eliminate false dependencies and loop fission to remove serial parts of a loop.

Student participants had access to parallelism charts to help determine the type of parallelism but student surveys indicated that they did not find them very useful overall. The deficiencies found in parallelism charts led us to re-examine our parallelism metrics, prompting the eventual creation of the new *self-parallelism metric* and techniques to help identify the type of parallelism. We will discuss the strengths and weaknesses of parallelism charts in Section 4.4.1, contrasting them with the self-parallelism metric described in Section 4.4.2. Section 5.3 discusses how later versions of Kremlin can identify the type of parallelism in a region.

We also noticed that students would occasionally continue working on a region even after obtaining the maximum possible speedup for that region. This likely resulted from the confusion caused by the rudimentary speedup estimates in the early Kremlin prototype. We addressed this shortfall partially through the creation of the self-parallelism metric and an improved parallel time estimation model, but also through the introduction of *planning personalities*. Planning personalities allow for tailor the planner to the specifics of a target machine and greatly enhance the quality of plans. Section 5.2 discusses our more advanced model of parallel execution time while Section 5.4 discusses planning personalities and describes several we have created.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under CAREER Award 0846152, by NSF Awards 0725357, 0846152, and 1018850, and by a gift from Advanced Micro Devices.

This chapter contain materials from "Kremlin: Rethinking and Rebooting gprof for the Multicore Age", by Saturnino Garcia, Donghwan Jeon, Christopher Louie, and Michael Bedford Taylor, which appears in *PLDI '11: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. The dissertation author was the primary investigator and author of this paper.

Chapter 3

System Overview

Kremlin is designed to be a practical oracle for sequential code parallelization. To be considered practical, Kremlin must be both simple to use and accurate, guiding programmers to the most important parts of the program with as little manual intervention as possible.

In this chapter we will look at Kremlin's basic usage model and the underlying system architecture that supports this model. Kremlin's system architecture introduces several new techniques, including hierarchical critical path analysis (HCPA), self-parallelism, and planning personalities. These techniques will be introduced in this chapter to provide insight into their interrelations, but detailed descriptions of these techniques and their implementations will be delayed until subsequent chapters.

3.1 Usage Model

Kremlin's user interface presents a simple three step usage model for obtaining a parallelization plan. Kremlin's usage model takes inspiration from `gprof`, with simplicity and clarity of results paramount. Figure 3.1 demonstrates this usage model. Kremlin starts with unmodified, serial source code and produces an instrumented binary. The user then runs this binary with its normal inputs to produce a dynamic parallelism profile. This profile is used by Kremlin's parallelism planner, along with a specified planning personality (OpenMP in the example), to

```
$> make CC=kremlin-cc
$> ./srad 100 0.5 502 458 image.pgm
$> kremlin srad --model=openmp --num_cores=4

Cores   1 2  4  8  16     32     64
Speedup 1 2  4  8  15.89  31.58  62.35
(est.)
     File (lines)      Cov.  (%)  Self-P    Iters.    TimeRed.(%)
1    srad.c (262-296)  70.25      458.0     458.0     52.69
2    srad.c (306-325)  24.25      458.0     458.0     24.20
3    srad.c (247-251)  5.29       502.0     502.0     3.96
4    srad.c (226-227)  0.09       229916.0  229916.0  0.07
5    srad.c (342-343)  0.04       229916.0  229916.0  0.03
...  ...               ...        ...       ...       ...
```

Figure 3.1: **Kremlin's Usage Model** Kremlin's three-step usage model is inspired by `gprof` [GKM82]: first the program is compiled with `kremlin-cc`, then the program is executed with its normal inputs, and finally the parallelization planner is run with the desired planning options. The planner allows the user to specify system-specific constraints (e.g. OpenMP on an 4-core processor); the planner orders regions according to decreasing expected parallelization benefit, allowing software engineers to first target the most important regions.

produce the parallelization plan. This parallelization plan ultimately helps answer the question, "Which parts of this program should I spend time parallelizing?" by listing the regions that should be parallelized, in the order that they should be parallelized.

Figure 3.1 shows the parallelization plan for the srad benchmark [CBM+09] as it would be displayed to the user. The plan presents an ordered list of regions for the programmer to parallelize. A region can be any single-entry sequence of instructions but Kremlin's recommendations focus on functions and loops as they are the most relevant during manual parallelization. The details of region formation will be discussed further in Section 4.3.1.

Components of Kremlin's Parallelization Plan Kremlin outputs several key pieces of data for each of its recommended regions: region location, coverage, self-parallelism, number of subregions, and the reduction in execution time if the

region is parallelized. Region location allows the user to quickly identify which region is being recommended (both filename and line numbers). Coverage refers to the percentage of serial execution time spent within a region. Self-parallelism is a new metric we have formulated to represent the amount of parallelism in a region, exclusive of any parallelism contained within nested regions. Together the coverage and self-parallelism fundamentally limit the impact of parallelizing a region in accordance with Amdahl's Law. In other words, speedup from parallelizing a specific region is governed by the following inequality:

$$speedup \leq \frac{1}{(1 - C(R)) - \frac{C(R)}{SP(R)}} \tag{3.1}$$

where $C(R)$ and $SP(R)$ are the coverage and self-parallelism of the region R, respectively. The time reduction given for each region provides an estimation of the impact on the whole program execution time from parallelizing the specified region. This estimation takes into account the equation above as well as other system-specific parallelization overheads and limitations, as we will discuss further in Section 5.2.

The number of subregions indicates how many regions are directly contained within the recommended region. This value is helpful when compared to the self-parallelism. Self-parallelism excludes any parallelism that comes from within subregions, and therefore measures the amount of parallelism available from executing subregions in parallel. When the number of subregions matches the self-parallelism, it is an indication that all subregions can be executed in parallel. Conversely, when the number of subregions greatly exceeds self-parallelism, it is an indication that few (if any) subregions can be executed in parallel. Parallelization tends to be much easier in the former case than in the latter so the ratio of self-parallelism to the number of subregions can quickly give the programmer a sense of the difficulty in parallelizing a region. Sections 4.4.2 and 5.3 will provide more insight into the relationship between these ratios and their impact on parallelization.

Acting on Kremlin's Recommendations Once the programmer has Kremlin's parallelization plan, the basic usage model is that they visit these regions

of code in the specified order and determine how to expose the underlying parallelism that was detected by Kremlin. Kremlin orders the parallelization plan in decreasing order of estimated time reduction, which takes into account factors such as self-parallelism, coverage, and parallelization overhead. The plan contains only those regions that are expected to meet a minimum speedup threshold; the programmer can expect to obtain nearly all the performance benefits possible if they parallelize all the regions in the list.

Kremlin's parallelism discovery builds upon critical path analysis, which quantifies the raw amount of parallelism available by analyzing only the true data and control dependencies in the program. Because Kremlin looks only at true dependencies, it can reduce many complex forms of parallelism into raw parallelism. This allows Kremlin to uncover parallelism of nearly all forms, including: loop-based parallelism such as DOACROSS and DOALL; pipeline parallelism between loops and functions; instruction level parallelism; and thread and task-level parallelism. Any of these forms of parallelism may be present in any given benchmark but the planner selects only those types that the planner deems profitable.

Exposing the parallelism detected by Kremlin may require user transformations such as: privatization; loop restructuring, fusion and interchange; insertion of OpenMP, Cilk++, or similar constructs; and refactoring of code and data structures to eliminate false sharing and contention. These transformations range in difficulty from trivial to difficult; some may require less than an hour of work while others may require many hours. As we will show in the Chapter 5, Kremlin is able to significantly reduce the number of regions that must be parallelized, thereby significantly reducing the total effort needed to parallelize the program.

Kremlin also provides a mechanism whereby the user can specify a set of regions that are too difficult to parallelize and rerun the planner, which recomputes the optimal plan excluding those regions. This exclusion list feature was not utilized while generating the results presented throughout the rest of this dissertation.

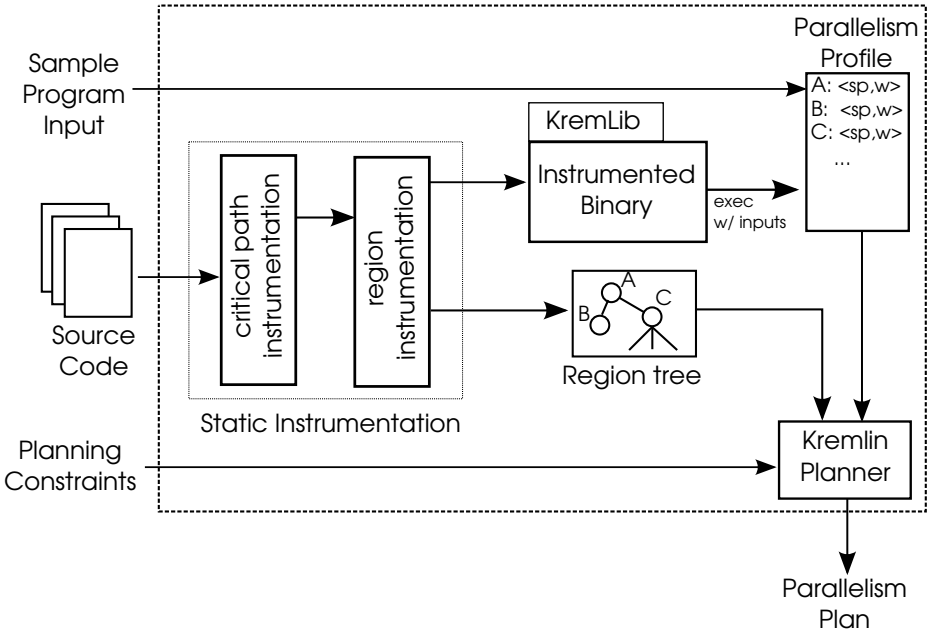


Figure 3.2: **Overview of Kremlin System Architecture.** Starting with a program's source code, Kremlin statically instruments the code to insert the proper profiling code and extract the region structure (i.e. region graph) from the program. Running the instrumented binary with a sample input produces a parallelism profile for each of the program regions. Combined with the region graph, the parallelism profile is used by the parallelism planner to provide the user with a specific list of regions to parallelize (the parallelization plan).

3.2 System Architecture

Figure 3.2 shows Kremlin's internal system architecture. Kremlin consists of several distinct phases: static instrumentation, linking and execution, and planning. In this section we will briefly overview each of these stages to provide an understanding of the interaction between them; more detailed descriptions of these stages will come in the following chapters.

Static Instrumentation Kremlin's parallelism discovery stage uses a new type of analysis known as hierarchical critical path analysis (HCPA). HCPA requires that critical path analysis be performed separately on all regions of the program. To meet this requirement, Kremlin introduces two instrumentation stages: critical path instrumentation and region instrumentation. The first sets up the profiling

infrastructure required to quantify parallelism via critical path analysis while the second helps uncover the program's structure and localize parallelism results to specific regions.

Both stages of the discovery phase utilize LLVM's [LA04] static instrumentation infrastructure. Static instrumentation has two important benefits over dynamic instrumentation. First, it allows for a deeper analysis of the program since the full program source is available. In our experience, tasks such as identifying induction variables, reduction variables, and region boundaries are challenging in dynamic infrastructures such as Valgrind [NS07] but are easy when performed statically. Second, by statically inserting instrumentation, Kremlin can heavily optimize the code to produce a more efficient instrumented binary. This helps to lower the overhead associated with the heavyweight analysis infrastructure required for measuring the amount of parallelism in every region of the program. Kremlin performs this optimization after instrumentation occurs so that it does not taint the analysis.

During critical path and region instrumentation, Kremlin inserts calls to instrumentation functions that calculate the critical paths of the program and track region entries and exits. These instrumentation functions are implemented inside the KremLib library. Section 4.3 provides more details on Kremlin's hierarchical critical path analysis. These functions maintain data structures which track dynamic control and data dependencies as the instrumented binary executes. Since the analysis is hierarchical, it simultaneously tracks these values across many nested regions. Section 4.3.2 provides further details on how Kremlin calculates critical paths at runtime, and Section 4.3.3 describes how we do this across many regions at the same time.

Linking and Execution Kremlin next links in the KremLib instrumentation library to produce the instrumented binary. When run, the instrumented binary also produces a parallelism profile output file–in addition to its normal outputs–that contains parallelism information for each dynamic instance of a program region. This information includes each region's total amount of work as well as its self-parallelism, a metric describing the amount of parallelism in that region, ex-

cluding any that came from subregions. Section 4.4.2 discusses self-parallelism in more detail.

The resulting parallelism profile contains results for an unbounded number of dynamic regions; without compression, this profile could easily contain terabytes of data. Kremlin uses a summarizing technique that represents the set of program regions as a tree, collapsing all dynamic regions with the same context into a single node in the tree. This technique not only greatly reduces the size of the profile, it provides context-sensitive results that can improve the quality of a parallel implementation. Kremlin's summarized region tree also provides a means for combining the results from multiple runs of the program, each with different inputs. This feature can help mitigate the chief limitation that Kremlin shares with all dynamic analyses: its dependence on specific inputs. Section 4.3.4 describes this summarizing technique in more detail.

Kremlin Planner With the parallelism profile and summarized region tree produced by the discovery phase, Kremlin can begin to create an effective plan to utilize the parallelism in the program. As we have seen, Kremlin crafts an ordered plan for the programmer that describes which regions should be parallelized. Kremlin uses planning personalities that incorporate both target- (e.g. OpenMP) and machine-specific parameters (e.g. the number of cores) in order to improve accuracy. Section 5.4 describes several planning personalities that we have created.

Kremlin models the execution time after parallelization using a combination of factors that are both target-independent–such as the coverage and self-parallelism–and target-dependent–such as the number of cores available and the type of parallelism that is exploitable. The estimated execution time can be used to compare the potential benefit of competing parallelization plans. Section 5.2 discusses this parallel execution time model.

3.3 Limitations of Kremlin and other Dynamic Analyses

Kremlin utilizes dynamic program analysis and therefore comes with the same inherent limitations associated with these types of analyses. Dynamic information depends on the input and therefore does not necessarily predict the program's execution with other inputs. This does not imply that behavior will vary widely across different inputs, but it also does not preclude behavior from varying. It is therefore critical for the user to choose inputs wisely, preferably utilizing multiple inputs and merging their results. Kremlin supports aggregating data from multiple runs, providing a means for users to increase their confidence in its results.

Kremlin extends critical path analysis (CPA). CPA-based tools also cannot predict the amount of parallelism available in alternative algorithms. CPA instead can be used to identify highly-serial regions of the program that could potentially benefit a change in algorithm; the programmer could use this information to guide their search for more parallel algorithms.

Unnecessary dependencies that are still true dependencies can limit CPA-based tools. These dependencies commonly manifest themselves through induction and reduction variables; Kremlin recognizes these common dependencies and automatically breaks them in order to mask their effects.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under CAREER Award 0846152, by NSF Awards 0725357, 0846152, and 1018850, and by a gift from Advanced Micro Devices.

This chapter contain materials from "Kremlin: Rethinking and Rebooting gprof for the Multicore Age", by Saturnino Garcia, Donghwan Jeon, Christopher Louie, and Michael Bedford Taylor, which appears in *PLDI '11: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and imple-*

mentation. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Planning-Aware Parallelism Discovery

The design of a practical oracle requires that both parallelism discovery and planning work in harmony to provide effective guidance during manual parallelization. In other words, parallelism discovery must become planning-aware. Kremlin extends critical path analysis to make it suitable for planning-aware parallelism discovery, introducing two main techniques to achieve this goal: *hierarchical critical path analysis* (HCPA), and lightweight approximation of a new metric known as *self-parallelism*.

In this chapter we will examine the design and implementation of both HCPA and self-parallelism. While these techniques are described in terms of their relevance as components of a practical oracle, they are joint work with several collaborators and also serve as the common framework for other systems. These other systems are outside of the scope of this thesis but the interested reader may refer to the work of Jeon et al [JGLT11] for details of how they serve as the basis for parallel performance prediction.

4.1 Requirements of Planning-Aware Discovery

Planning-aware parallelism discovery needs to produce results that are appropriate for use during parallelism planning. Planning places two major require-

ments on parallelism discovery. First, it requires that the amount of parallelism be quantified. Without quantification, the planner cannot estimate the impact of parallelization and therefore cannot effectively compare different plans. Second, it requires localized parallelism information. Without this localized information, the planner cannot model the process of iterative improvement that manual parallelization entails.

As we discussed in the introduction, existing parallelism discovery tools rely on one of two techniques: critical path analysis (CPA) or dependence testing. These two techniques differ in their goals—CPA to quantify parallelism, dependence testing to identify independent parts of the program—but they share a common shortcoming: their results are poorly suited for use during parallelism planning. Despite this joint shortcoming, CPA is better suited as a basis for planning-aware discovery than dependence testing because CPA can quantify parallelism. Kremlin's key contributions to parallelism discovery, HCPA and self-parallelism, help overcome CPA's chief limitation: its lack of localized parallelism information.

4.2 Background: Critical Path Analysis

One promising approach for quantifying parallelism is to use a *critical path analysis* [Kum88], or *CPA*. CPA is a dynamic analysis that finds the string of dependencies that forms a lower bound on the execution time (the critical path) of a piece of code. The critical path in turn creates an approximate upper bound on the parallelism available, with the ideal parallel implementation performing all non-critical operations in parallel with the critical path operations. The work and critical path define the average amount of parallelism available –which we refer to as the *total-parallelism*–according to the following equation:

$$p = \frac{work}{length_{cp}} \tag{4.1}$$

The basic premise behind parallelism discovery tools that employ critical path analysis [Kum88, KMC72, AS92, KBI+09] is to evaluate the application's potential for parallelization under relatively optimistic assumptions based on ob-

```
for(i=win..rows-win) {
    for(j=win..cols-win) {
        currLambda = lambda[i][j];
        ...
        for(k=0..nFeatures) {
            if(features[2][k] < currLambda) {
                ...
                features[0][k] = j;
                features[1][k] = i;
                features[2][k] = currLambda;
            }
        }
    }
}
```

Figure 4.1: **Localizing Parallelism.** In this nested loop from the fillFeatures function in feature_tracking, only the innermost loop (over induction variable k) is parallel. Traditional CPA would erroneously report parallelism in the outer loops because they contain the innermost.

servation of the program's dynamic execution. Most parallelizing compilers, in contrast, must take relatively pessimistic views because they are responsible for guaranteeing correctness. For example, parallelizing compilers may not be able to prove that two pointers do not alias, while a critical path analysis will at least report that it did not observe such dependencies in the actual execution of the program. The basic idea is to elevate to the user awareness of the at least circumstantial evidence of parallelism in the program, so that users can apply their understanding of the real application constraints (as opposed to what is encoded in program source) and refactor to exploit the parallelism.

Unfortunately, traditional critical path analysis has not found widespread use as a parallelism quantification tool for parallel programmers because it has one important limitation: it cannot localize the parallelism to a particular level of the

nested hierarchy of a program's regions. This limitation is illustrated by a code snippet from the feature_tracking benchmark from the San Diego Vision Benchmark Suite [KVAJ+09], shown in Figure 4.1. In this example, only the innermost loop is parallel. Traditional CPA would only detect that parallelism exists somewhere among the three loops, not just the innermost.

This fundamental limitation of CPA manifests itself in a way that makes CPA's results impractical for planning. Because CPA cannot localize parallelism, it overestimates the amount of parallelism in various parts of the program. For example, CPA would report significant amounts of parallelism in the outer loops in Figure 4.1 when there is none. This makes planning inaccurate as it appears as though parallelizing the outer loops (e.g. by adding an OpenMP pragma before the loop) will be profitable when it will not. As we will see, even if we modify CPA to look at each region individually (instead of the program as a whole), the lack of localized parallelism information will still mislead the planner.

4.3 Hierarchical Critical Path Analysis

Traditional critical path analysis suffers from a lack of knowledge about the structure of the program. This lack of knowledge leads to confusion about the source of parallelism detected in the program. Parallelism discovery needs to first be aware of the structure of the program and the relationships between various parts of the program before it can identify the exact sources of parallelism in a program.

Kremlin introduces a new form of analysis, *hierarchical critical path analysis* (HCPA), that profiles the parallelism in every region of the program and uncovers the structure of the program. HCPA provides the basis for efficiently approximating self-parallelism, a new metric we have defined to quantify localized parallelism. HCPA is a region-based analysis–which we will define in the proceeding subsection–and it is also hierarchical, taking advantage of the program hierarchy not only to localize parallelism but also to analyze multiple regions efficiently.

In this section we will discuss the design and basic implementation of HCPA,

starting with the basics of performing CPA with shadow memory before moving on to discuss the changes required to perform HCPA with shadow memory. However, before we begin further discussion we will first define what a region is and why the choice of regions is critical to our analysis.

4.3.1 Defining a Region

We use the concept of a region to denote a piece of code whose parallelism is to be measured from the time that region is entered until the time it is exited. Regions must obey a proper nesting structure: regions must not partially overlap, but they may nest or be siblings with the same parent region. This nesting structure gives rise to a dynamic region tree which shows the relationship between parent and children regions in the dynamic execution of the program. We can leverage this nesting structure to localize the amount of parallelism in each region, using a new metric called self-parallelism.

Although more arbitrary delineations of regions are possible, HCPA defines several types of regions: functions, loops, loop bodies, and self-work sequences. Function and loop regions correspond well to program constructs with which the programmer is familiar. These two types of regions have the added benefit of being the source of most parallelism in the program: task-based parallelism is often focused on functions while the myriad of loop-based parallelism types obviously originates from loops. These two region types also play a central role in most existing manual parallelization systems, further underscoring their importance for parallelism discovery and planning.

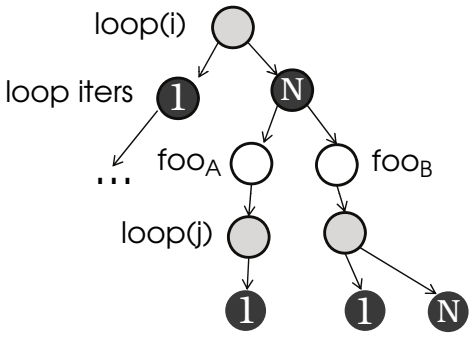
The remaining types of HCPA regions were developed as a method for enhancing Kremlin's ability to classify different forms of parallelism. Loop body regions for a child region for each iteration of a loop region, allowing us to identify loop-level parallelism. Self-work sequence regions force instruction level parallelism (ILP) to childless regions, clearly delineating ILP from other forms of parallelism.

Figure 4.2 shows an example of how HCPA transforms code into a hierarchy of dynamic regions. The code in Figure 4.2a becomes the region tree shown in 4.2b, representing the relations between the function, loop, and loop body re-

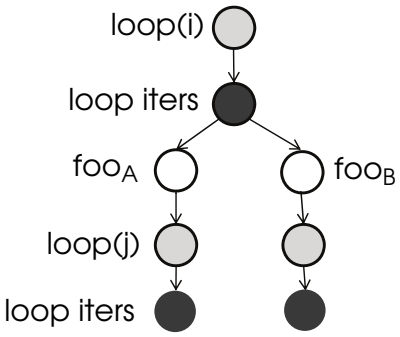
```
for (i=1 to N) {
    foo(1); // callsite A
    foo(N); // callsite B
}
```

```
void foo(int size) {
    for(i=1 to size) {
        ...
        // loop body
        ...
    }
}
```

(a) Sample Code Fragment



(b) Dynamic Region Tree



(c) Summarized Region Profile

Figure 4.2: **HCPA's Hierarchical Region Model and Summarization.** At runtime the code in (a) forms the region tree shown in (b) based on HCPA's function, loop, loop body, and sequence regions. HCPA will calculate the CPA recursively for each dynamic region. Dynamic regions sharing the same context will be summarized into a single node, resulting in the tree shown in (c). Loop body regions are collapsed into a single node while separate calls to `foo` have separate nodes to indicate differing contexts. This context-based approach can lead to more efficient parallelization as different contexts can contain different amounts of parallelism..

gions in the code. This dynamic region tree is later compacted into a summarized, context-sensitive form to reduce log size output and provide more precise parallelism profiling. This summarizing technique will be discussed in more detail in Section 4.3.4.

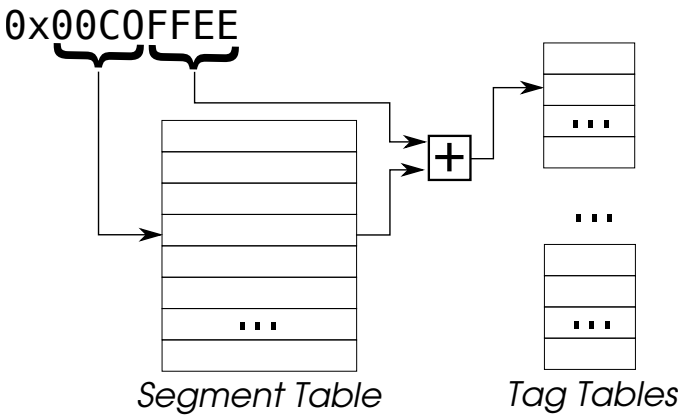


Figure 4.3: **Traditional Shadow Memory Organization.** The memory address is used as an index into a two-level page table that contains the metadata associated with that address. To support 64-bit addresses, a three-level table may be used. This multi-level architecture is similar to that of a page table, and allows only active subsets of the memory address space to have allocated shadow memory tags.

4.3.2 Calculating Critical Path with Shadow Memory

Critical path analysis calculates parallelism by quantifying both the amount of work done and the minimum time needed to do that work (i.e. the length of the critical path). The ratio of work to critical path length indicates the average number of instructions that can be executed in parallel in the ideal case. Kremlin efficiently determines both of these values through the use of shadow memory [NS07, ZBA10a].

Figure 4.3 shows a traditional shadow memory layout. Shadow memory provides metadata storage for each memory location, allowing each address to be "tagged". This metadata has been used for a wide range of dynamic program analyses, with applications ranging from memory analysis [SN05, BZ11] to computer security [CZYH06, QWL+06, XBS06].

Kremlin employs shadow memory to help calculate the "parallel time" of each dynamic instruction, the earliest time the result of that instruction will be available. This parallel time depends on the set of instructions required to produce the operands for the instruction as well as the time needed to perform the instruction. Parallel time takes into account only true data and control dependencies,

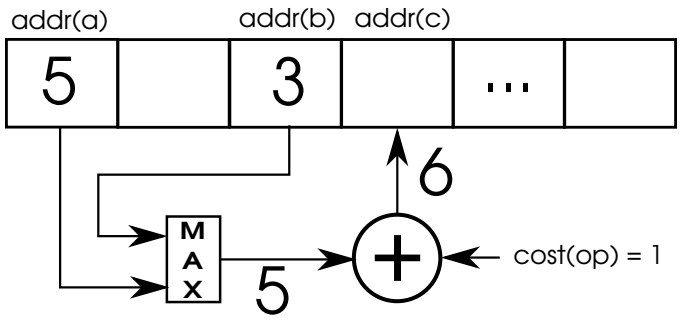


Figure 4.4: **Calculating Parallel Time with Shadow Memory.** In this example the operation `c = a + b` triggers a shadow memory update. Shadow memory stores the "parallel time" for each address. To calculate the parallel time of the value being written (`c`), the times for the operands (`a` and `b`) must be read from shadow memory, compared (`max`), and added to the cost of the operation before storing the result back in shadow memory. Kremlin introduces several optimizations to minimize the overhead associated with using shadow memory.

taking advantage of the single static assignment form of LLVM's intermediate representation to eliminate false dependencies such as anti- and output dependencies.

Kremlin stores the parallel time into shadow memory, tagging the address associated with the value being written by the instruction. This organization leads to a simple four step process for each instruction, as shown in Figure 4.4. Kremlin can easily capture the length of the critical path by tracking the largest parallel time that is stored, as this corresponds to the instruction that required the longest chain on dependencies to compute (i.e. the critical path).

Kremlin utilizes an array of techniques to reduce both the runtime and memory overhead associated with employing shadow memory. While we will hold off discussion of some of these techniques until later, one bears mentioning at this point: shadow register tables for local variables. Shadow memory generally requires a multi-level structure similar to that of a page table so that only actively used parts of the address space are shadowed, as shown in Figure 4.3. This layout sacrifices performance for space efficiency, requiring multiple pointer dereferences to access the metadata for a specific address. While local variables have a corresponding stack address, the load-store architecture implicit in the LLVM intermediate representation means that most operations are performed on registers rather than memory locations. Kremlin analyzes each function for the number of

used registers and allocates a shadow register table that stores tags associated with these registers. The shadow register file is optimized for speed by being directly addressable, greatly reducing the time to access local variables.

Resolving False and Easy-to-Break Dependencies A major challenge for any critical path analysis infrastructure is to mitigate the effects of false and easy-to-break dependencies. Many of these false dependencies, such as unnecessary reuse of a variable, are eliminated by the use of SSA form in LLVM's IR. However, the easy-to-break dependencies associated with induction and reduction variables are more challenging. These types of dependencies can create the false impression that a parallel region is actually serial. The induction variable for a loop can create the false impression of inter-iteration dependencies and therefore mask available parallelism. Similarly, reduction variables appear to require serial execution when the exact ordering of operations on these variables is not important. Kremlin breaks these dependencies by statically identifying induction and reduction variables and utilizing a special shadow memory update rule that ignores the dependency on their old value.

Managing Control Dependencies Kremlin performs static control dependence analysis to identify which values a basic block is control dependent upon. Unfortunately, static analysis cannot fully resolve all control dependencies. This shortcoming is demonstrated in the following code snippet:

```
if(x == 0) { rotate(img); }
```

If the condition is true, then the comparison of `x` will be a control dependence for all instructions executed in `rotate`. Because `rotate` may occur in other contexts where this control dependency does not exist, the dependence on `x` must be resolved dynamically.

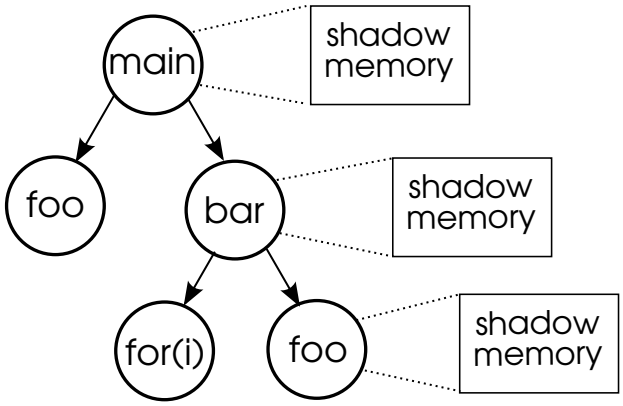
Kremlin handles control dependencies through the use of a control dependence stack similar to one proposed in [XZ07]. Kremlin pushes a dependency onto the control stack at the beginning of a control dependent region, popping it off when exiting that region. The times stored in the control stack can only increase:

```
int main() {
    foo();
    bar();
    ...
}

void foo() {
    ...
}

void bar() {
    for(i=0..10)
        x++;
    foo();
}
```

(a) Code Snippet.



(b) Corresponding Region Tree.

Figure 4.5: **Shadow Memory and Region Hierarchy.** Each node in (b) is a dynamic region that requires a unique set of shadow tags. Each region's tags must be isolated when simultaneously profiling multiple regions in order to avoid incorrect profiling results.

the parallel time for a control dependency is dependent on all active control dependencies so the times must strictly increase. Kremlin leverages this property, incorporating control dependencies by checking only the top of the stack.

4.3.3 Introducing Hierarchy into Shadow Memory

Traditional critical path analysis is a flat analysis: only a single region of the program is examined, typically the `main` function. Kremlin's hierarchical analysis requires that we examine all dynamic regions of the program. The naïve method of running the program once for each region is impractical as even trivially small programs can contain an exorbitant number of regions because of loops. The

alternative approach, examining all regions in a single run of the program, presents an interesting problem of resource isolation.

Much like separate executing processes maintain a separate address space, each dynamic region requires a separate shadow memory address space. Figure 4.5b demonstrates this idea using the dynamic region tree obtained from the code in Figure 4.5a. When entering a region, all previous work and dependencies should be invisible to the newly created region while remaining visible in the appropriate ancestor regions. This situation clearly calls for separate tags for each region.

To implement shadow memory for the evolving set of dynamically nested regions as the program runs, each location in the shadow memory and register tables is associated with a vector of parallel times rather a single time. This vector expands when a region is entered and shrinks when a region is exited.

An efficient representation for the vector of parallel times is critical for reasonable profiling performance. Two factors conspire to make an efficient representation difficult to achieve. First, only a (possibly small) subset of all active shadow memory locations will be used by a region. It therefore does not make sense to allocate a new spot in the set when we enter a region: most of them will simply not be used and the time needed to allocate would be large. Second, nesting of loops can lead to frequent entering and exiting of regions. The implication is that upon exiting a region, we cannot tolerate lengthy data cleanup times that would be associated with allocating or deallocating the values in the set that are specific to a region.

The key insight for efficient shadow address space management is that there is at most one active region in any given level in the region tree. Kremlin takes advantage of this hierarchical property to minimize the memory overhead associated with multiple shadow address spaces. As shown in Figure 4.6, all regions in each level of the region tree are mapped to a single tag. In other words, every region in a level shares the same shadow address space.

Sharing of shadow address spaces could potentially lead to one region polluting the address space of another. It is possible to clean the “dirty” tags after

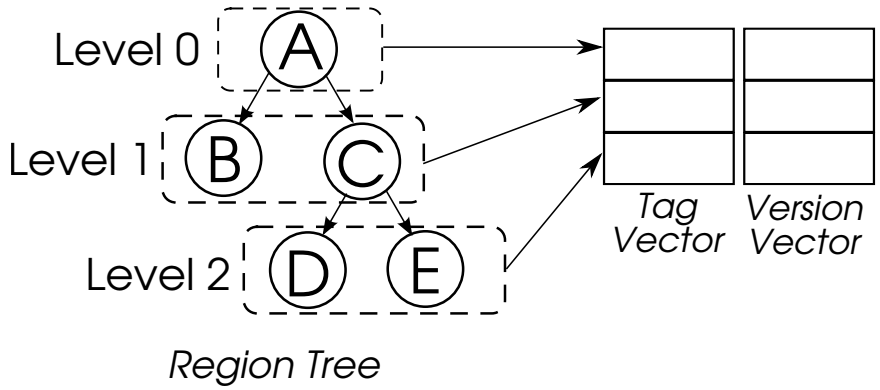


Figure 4.6: **Level-based Sharing of Shadow Memory Tags.** The hierarchical nature of regions ensures that any level in the region tree will have at most one active region. Kremlin uses this property to enable reuse of physical shadow memory space between multiple regions of the same level. This reuse requires that tags be validated to ensure that stale metadata is not used (e.g. not using region B's metadata for region C). Each tag has an version associated with it to determine the region in which it is valid.

exiting a region but this is likely to incur a significant performance penalty. This penalty is especially onerous for regions that are entered and exited rapidly, such as deeply nested loops.

We can avoid the cleaning costs of the naïve scheme by instead using a version-based approach. Kremlin assigns a unique ID to every dynamic region; this ID is then stored along with each tag (i.e. parallel time) whenever shadow memory is updated. Kremlin compares the ID of the current region with the stored ID whenever reading a tag; if the version matches the value is valid, otherwise it is invalidated by overwriting it with the value of 0. This process is analogous to an operating system's use of process IDs to isolate the memory of each process.

The downside of this version-based approach is that is requires a significant amount of space for tracking versions, essentially doubling the memory overhead associated with profiling. We have developed several techniques that mitigate this overhead, which we will discuss in Chapter 6.

4.3.4 Summarizing Dynamic Regions

The number of dynamic regions quickly grows as nested loops with many iterations are executed. This large amount of regions poses practical challenges not only in the size of the profile output but also in the runtime of algorithms that need to analyze this data.

The initial version of Kremlin used a dictionary-based compression algorithm to reduce the profile size. However, this type of compression performs poorly on programs with loop iterations that vary in their work or critical path length as was often the case for irregular programs such as those found in SpecInt. This approach also failed to utilize context-sensitivity, which we will see can be critical in producing a plan that will lead to the highest possible parallel speedup.

Summarizing Technique Kremlin combines all dynamic regions that have the same region context into a single summarized region. Figure 4.2 depicts how the runtime region tree (4.2b) becomes a summarized region profile (4.2c). In this method, all loop body regions (i.e. iterations) collapse to a single node, greatly reducing the number of regions. Each node calculates weighted averages for parallelism, work, and other profiled data across all dynamic regions corresponding to that node.

Kremlin maintains a `current` pointer that tracks the summary node that corresponds to the current dynamic region. When a new region is entered it updates the `current` pointer to one of its children node based on statically assigned callsite ID information. If there is no corresponding node, it creates a new summary node and updates the `current` pointer. When a region exits, the region's profiled information is added to the current node and the pointer returns to the parent node. This process is similar to the call context tree described in [ABL97] but modified for Kremlin's region hierarchy.

Using Context Sensitivity to Improve Planning The example summarized region profile shown in Figure 4.2c contains two nodes for the same function (`foo`) from what appears to be the same context. This corresponds to two separate calls

from the same loop. While this increases the number of nodes in the summarized profile, it allows Kremlin to uncover new parallelism opportunities.

To understand the merit of context-sensitive representation, consider the code in Figure 4.2a. When the loop in function `foo` is parallel and N is large, the parallelism of this loop significantly differs between callsites A and B. Callsite A's loop will always have a self-parallelism of 1, providing no benefit to parallelism and likely causing slowdown due to synchronization overhead. Callsite B's loop will have a self-parallelism of N and would likely be a good candidate for parallel refactoring. Kremlin can capitalize on the split contexts, incorporating the speedup from callsite B into its estimates while ignoring callsite A. If all nodes corresponding to the same static region have similar parallelism stats, they can be merged into a single node by the planner.

Using Context Sensitivity to Ease Aggregation As with any tool based on dynamic analysis, Kremlin's results depend on the inputs used. This limitation can be partially avoided by running with multiple times with distinct inputs and comparing results. Kremlin's region summarizing technique provides a convenient way to aggregate results from multiple runs into a single combined report.

The ability to aggregate results arises from HCPA's use of unique identifiers for each program callsite. Kremlin stores these unique IDs in the summarized region so it is trivial to map regions in one summarized profile to corresponding regions in a separate profile. This mapping can be utilized to combine the results of multiple profiles into a single profile. When a region is found in multiple profiles, the results from the corresponding regions in each profile can be averaged to produce the result in the combined profile.

4.4 Identifying Local Parallelism

The parallelism trace outputted by HCPA contains all the information necessary to determine the parallelism of each dynamically executed region as well as to recreate the dynamic region structure of the program (i.e the region tree). However, further analysis is needed to localize the parallelism to specific regions

Table 4.1: **Region Key for MPEG Encoder Benchmark.** Regions listed here appear in the parallelism chart shown in Figure 4.7.

Region	Source	Lines	Function
A	motion.c	208-220	ptmotion_estimation
B	motion.c	211-220	ptmotion_estimation
C	putpic.c	376-612	ptputpict
D	putseq.c	257-518	putseq
E	putseq.c	94-125	thread_work
F	quantize.c	105-137	ptquant
G	transfrm.c	176-233	pttransform
H	transfrm.c	249-305	ptitransform

of the program. This is because the nesting structure of regions causes parallelism from a region to percolate up to its ancestor regions. This percolation of parallelism between nested regions makes it difficult to understand if a high parallelism score indicates that a region, its children, or both have parallelism.

As we discussed at the beginning of this chapter, isolation of parallelism is critical to effective planning. In this section we will discuss two approaches that we developed to identify local parallelism. The first approach was used in our user study and provided a visualization of the program structure and total parallelism of each region. This approach relied on the user to manually infer the amount of parallelism based on the visualization and total parallelism. The second approach resulted from insights gained from our user study and our own experiences with the first approach. This newer approach provides an automated method for identifying local parallelism, specifically in the form of a new metric called self-parallelism.

4.4.1 Initial Approach: Parallelism Charts

Our initial approach to localizing parallelism involved creating parallelism charts like the one shown in Figure 4.7. A parallelism chart plots the total parallelism of dynamic regions (y-axis) over the course of the program's execution (x-axis). Parent regions encompass children in the parallelism chart (e.g. region A is a parent of region B), providing the user with an opportunity to see the relation-

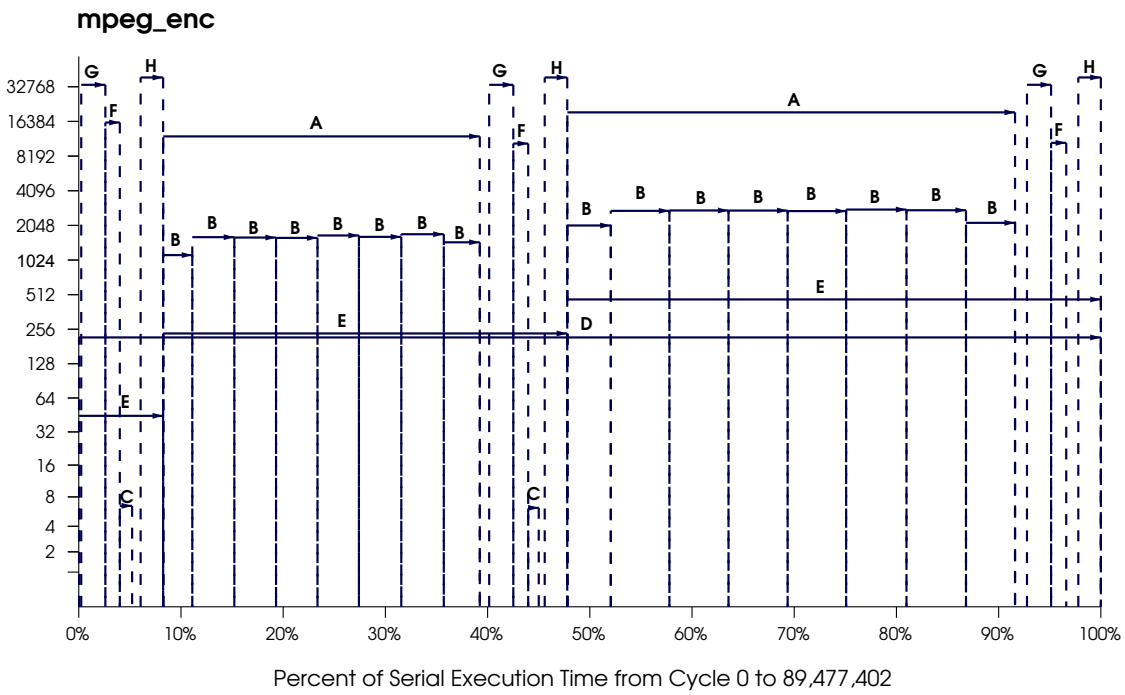


Figure 4.7: **Parallelism Chart for MPEG encoder.** Our initial approach to localizing parallelism was the creation of parallelism charts. These charts plot the total (or ideal) parallelism of dynamic program regions (y-axis) as the program executes (x-axis). They allow the inference of the amount of parallelized in specific cases such as with highly parallel regions. The chart shown here shown here for the MPEG Encoder benchmark from the ALP Benchmark Suite [SLA+07], with accompanying region key in Table 4.1. The chart implies a potential speedup of 8× for Region A based on the relationship between its 8 subregions' parallelism and its parallelism. Unfortunately it is poor at inferring the localized parallelism in more complex cases such as Region E.

ships between regions in the hierarchy. This chart inspection method works well in specific situations, allowing the user to infer the amount of parallelism specific to a region. For example, region A's contains roughly 8× the amount of parallelism than that of it's 8 children, implying that 8-way parallelism exists in region A.

There are several major drawbacks of this parallelism charts approach. First, this approach does not scale well to larger programs with many more regions. This limitation could be slightly mitigated by making the charts interactive and allowing users to "zoom in" to specific parts of the program but this may still not work for extremely long running regions with many subregions. Second, par-

allelism charts work well for inferring the amount of parallelism in highly parallel regions (e.g. region A) but can lead to confusion for more complicated expressions of parallelism. This limitation is evident in region E in Figure 4.7, which has a significant amount of parallelism but which contains only 4 subregions—3 of which have much higher levels of parallelism. The parallelism chart does not give an indication of how much parallelism is available in region E alone. Finally, the parallelism charts still relies on expert interpretation to understand the program's parallelism. While the charts present the parallelism in a more compact and intuitive way, they are not fundamentally different than the manual approach of having a programmer look through the program to identify potential parallelism.

Our experience with parallelism charts led us to the understanding that any successful approach to identifying the true amount of parallelism in a region would need two key characteristics. First, it would need to be completely automated to avoid requiring expert knowledge to interpret results. Second, it would need to be presented in a simplified form that was not ambiguous and handled complex parallelism expressions (e.g. pipeline parallelism) in addition to simple cases (e.g. doall loops). These requirements led us to the development of self-parallelism as a key metric for quantifying localized parallelism.

4.4.2 Self-Parallelism

Self-Parallelism is a new metric that we developed to quantify the amount of parallelism in a region, exclusive of the parallelism in that region's subregions. Self-parallelism can also be thought of as the ideal speedup that is possible from parallelizing a region without modifying its subregions. For example, the self-parallelism of a loop with independent iterations is equal to the number of loop iterations (i.e. subregions); in the ideal case, parallelizing that loop would result in a speedup equal to the number of iterations because all iterations are executed concurrently rather than in sequence.

Self-parallelism is conceptually similar to *self-time* in `gprof` and similar profilers. Self-time starts with the total time of a region and factors out time in subregions. Self-parallelism starts with the total parallelism of a region and factors

out the parallelism in subregions.

Self-parallelism differs from self-time in that is non-trivial to calculate. Self-time benefits from the additive nature of time, requiring only a simple subtraction to factor out a subregion's contributions. Conversely, self-parallelism relies on the more complex relationship between the parallelism of region and subregion, requiring more than simple arithmetic operators to determine the true value.

We can conceptually view the process of determining self-parallelism as the process of "dividing out" the parallelism from subregions but only in the simplest cases would division result in the exact value. The exact value of self-parallelism can only be determined if we have the whole dependency graph available to inspect and unravel, which unfortunately is too large to store for any non-trivial program. To make the task of determining self-parallelism tractable, we need a way to approximate this value using easy to summarize values such as the total amount of parallelism in each region and the relationships between regions.

Approximating Self-Parallelism We leverage a key observation to help us approximate self-parallelism: parallelism arises from work that is off of the critical path. This observation provides the insight for another key observation: work outside of the critical path will result in parallelism in all ancestor regions. This second observation is the basis for our approximation equation; by eliminating the non-critical path work of all a region's children, we can suppress the expression of that parallelism in the region itself. Eliminating non-critical path work of subregion can be achieved by simply replacing the work of each subregion, which includes both critical path and non-critical path work, with the critical path length of the subregion. All that is needed for our approximation of self-parallelism is tracking the critical path length of every dynamic region and a knowledge of the program's hierarchical structure (i.e. the dynamic region tree).

With these insights in mind Kremlin uses the following equation to approximate $SP(R)$, the self-parallelism of a region R:

$$SP(R) = \frac{\sum_{k=1}^{n} cp(child(R,k)) + SW(R)}{cp(R)} \tag{4.2}$$

where n is the number of children of R, $child(R, k)$ is the k^{th} child of R, and $cp(Q)$ is the critical path length of region Q. $SW(R)$ represents the amount of work that is performed exclusively in region R (i.e. self-work).

Equation 4.2 bears a strong resemblance to the calculation of total parallelism in Equation 4.1. This is by construction as the two metrics follow the same intuition: the numerator in both represents a measure of work while the denominator is the ideal parallel time of the region.

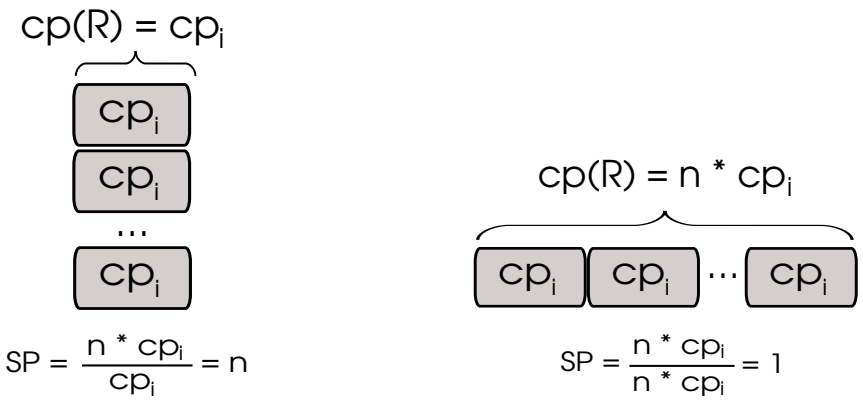
The self-work of a region R is calculated using the following equation:

$$SW(R) = work(R) - \sum_{k=1}^{n} work(child(R, k)) \tag{4.3}$$

We can however simplify the calculation of $SP(R)$ by ensuring that all $SW(R)$ is always 0. This can be accomplished by ensuring that all work is done is leaf-regions, which is possible through the use of the self-work sequence regions we described in Section 4.3.1.

Summarizing Self-Parallelism Kremlin calculates self-parallelism each dynamic instance of a region but must combine these separate instances into a single value for the summarized version of that region. This summarization could use the average self-parallelism value but this value could be misleading. For example, consider when there is one short running (i.e. low work) but highly parallel (i.e. high self-parallelism) instance and one long running but mostly serial instance. Averaging self-parallelism would make it appear as though there is significant speedup to be had from parallelizing the region; in reality the speedup would be limited because only a short amount of the total work for the region is spent in the highly parallel instance.

Kremlin instead takes a weighted average of self-parallelism. The idea behind this approach is to calculate the "parallel work" for each dynamic instance. This parallel work represents the ideal time to execute the parallelized region and is calculated using the equation $work_p = \frac{work}{self-parallelism}$. Kremlin keeps a running total of both $work$ and $work_p$ for each summarized region: after execution terminates, the weighted average of self-parallelism can be calculated according to the



(a) Parallel Subregions (b) Serial Subregions

Figure 4.8: **Self-Parallelism (SP) Scenarios.** SP identifies the parallelism local to a region by relating its critical path to the sum of its subregions' critical paths and its self-work. Shown in the example are SP calculations for two regions; (a) one whose subregions can execute concurrently, (b) one whose subregions must execute serially.

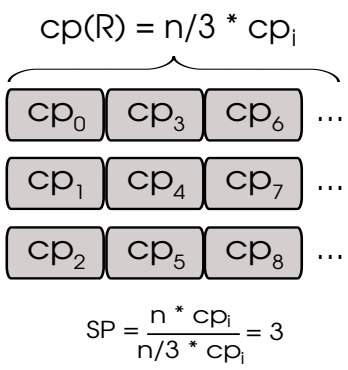
equation $SP_{weighted}(R) = \frac{work(R)}{work_p(R)}$. This weighted value will be the value used for the region during parallelism planning. For the previous example we examined, this will lead to a self-parallelism that is much closer to that of the long running, less parallel instance.

Illustrating Self-Parallelism's Effectiveness To illuminate the effectiveness of self-parallelism, we will examine the self-parallelism in several scenarios. We will start by looking at two opposite ends of the parallelism spectrum: a region where all subregions must be executed in sequence (i.e. a serial region) and a region whose subregions can all be executed concurrently (i.e. a totally parallel region). Figure 4.8 illustrates both of these cases.

For the parallel region in Figure 4.8a, its measured critical path will be equal to a single child (i.e. $cp(R) = cp_i$). Thus, the computed self-parallelism will be $\frac{n*cp_i}{cp_i} = n$; this is as expected because its parallelism is equal to the number of children. Now consider a parent whose children must be executed completely serially (Figure 4.8b). In this case, the measured $cp(R)$ will be equal to $n * cp_i$ and therefore the computed self-parallelism will be $\frac{n*cp_i}{n*cp_i} = 1$; again, this is expected

```
for ( i = 3..N−1)
    a[ i ] = 2∗a[ i −3]
```

(a) Loop with Pipeline Parallelism.



(b) Semi-parallel Subregions.

Figure 4.9: **Self-Parallelism with Pipeline Parallelism.** Self-parallelism is able to identify pipeline parallelism, expressed in (a) as a DOACROSS loop. In this example, a cross-iteration dependency length of 3 allows three separate strands of computation to proceed concurrently.

because we cannot overlap execution of the children.

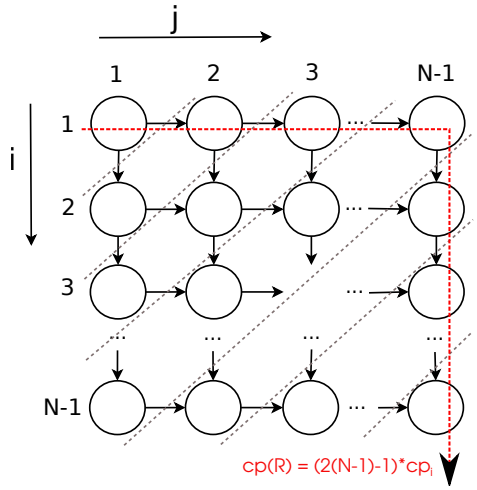
We will now move on to a slightly more complex example where subregions partially overlap, allowing only a limited amount of parallelism. This situation is often referred to as either DOACROSS–in the context of loop-based parallelism–or pipeline parallelism–in the context of task-based parallelism. Figure 4.9 illustrates this case. The loop in Figure 4.9a has a cross-iteration dependency that spans three loop iterations. Figure 4.9b shows that three separate strings of dependencies could be executed in parallel. The self-parallelism of this region is $\frac{n*cp_i}{\frac{n}{3}*cp_i} = 3$, which matches our intuition.

Finally, we will look an even more complex example to demonstrate the power of self-parallelism. In this final example (shown in Figure 4.10) a significant amount of parallelism is hidden by the strongly serial implementation in Figure 4.10a. The dependency graph in Figure 4.10b shows that iterations can be grouped into independent, diagonal sets that can be executed in parallel. The critical path of the code goes through $2(n-1)-1$ iterations so the self-parallelism of the outer loop is $\frac{(n-1)^2*cp_i}{(2n-3)*cp_i} \approx \frac{n}{2}$.

This final example exemplifies many of the benefits that Kremlin has over other techniques. Techniques that rely on dependency testing rather than critical

```
void calc_array(int** a) {
  for(i = 1; i < N; ++i)
    for(j = 1; j < N; ++j)
      a[i][j] = a[i-1][j] + a[i][j-1];
}
```

(a) Loop with unexpressed parallelism.



(b) Iteration Dependency Graph

Figure 4.10: **Uncovering Hidden Parallelism with Self-Parallelism.** Self-parallelism's underlying reliance on critical path analysis allows it to uncover parallelism even when masked by a serial implementation. The code in (a) shows a nested loop operating on a 2D array with cross-iteration dependencies over both loops, making it appear very serial. The iteration dependence graph in (b) shows that iterations can be grouped into independent, diagonal sets, allowing parallel execution if loop skewing and interchange are used. The critical path goes through $2(N-1)-1$ iterations, leading the self-parallelism of the region to be $\frac{(n-1)^2 * cp_i}{(2n-3) * cp_i} \approx \frac{n}{2}$.

path analysis for locating parallelism would miss this parallel region because of its initial serial expression. Automatic parallelizing compilers would have difficulty with this code because the array is passed using an array of pointers to arrays, requiring complicated shape analysis to uncover the parallelism. In general, uncovering parallelism could require an arbitrary number of complex analyses. Because of complexity and runtime issues, modern compilers are not able to compose all

of these heroic tasks simultaneously into one coherent analysis and transformation framework.

4.5 Evaluation

As we have discussed earlier, traditional critical path analysis leads to confusion about the source of parallelism through a program. This confusion is a result of ignoring the structure of the program, which leads many serial regions to appear parallel because they have subregions with substantial amounts of parallelism.

We examined all eight benchmarks in the NAS Parallel Bench (NPB) benchmark suite [BBB+91] in order to determine self-parallelism's ability to accurately quantify the amount of parallelism available in a region. These benchmarks are known for having abundant amounts of parallelism but with many regions that do not contain parallelism. The nesting structure of parallel regions contained within serial regions provides an opportunity to assess a parallelism metric's ability to localize parallelism to specific regions.

We began by classifying all 1953 regions in NPB into one of four categories according to the amount of parallelism available: serial (parallelism < 1.1), moderately parallel (1.1 to 2.0), parallel (2.0 to 5.0), or very parallel (parallelism $>$ 5.0). We used Kremlin to calculate both the total and self-parallelism numbers for all regions so that we could perform the classification.

Figure 4.11 shows the results of this classification. Self-parallelism identified approximately $6\times$ as many regions as being serial when compared to total-parallelism. There was also a corresponding large drop in the number of regions classified by self-parallelism as being very parallel.

The data from Figure 4.11 is alone not enough to determine whether self-parallelism correlates well with ability to exploit parallelism. To determine this, we determined what percentage of regions in several ranges of parallelism (total- and self-) were parallelized in the third-party parallel implementation of the benchmarks. A correct classification for a parallelism metric would show that regions classified with low levels of parallelism would rarely be parallelized while a larger

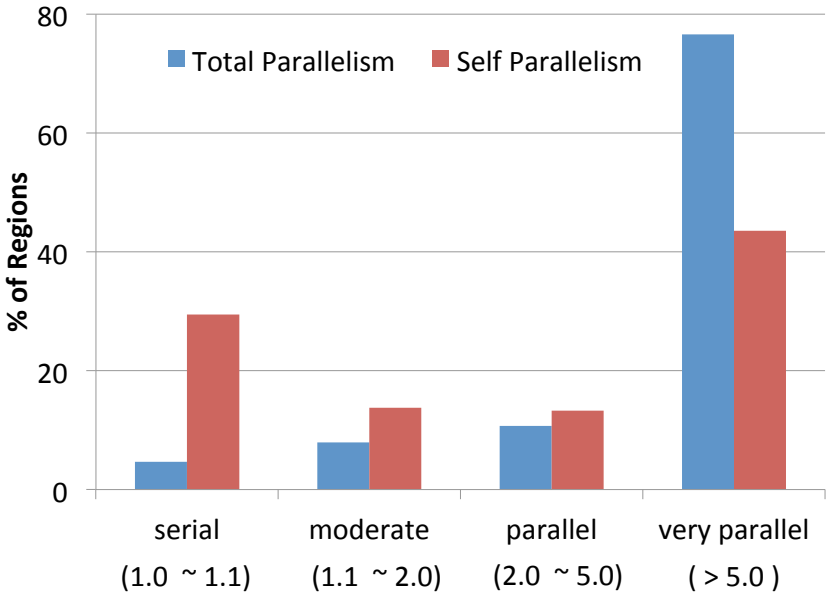


Figure 4.11: **Classification of Regions Based On Total- and Self-Parallelism.** All 1953 regions in the NPB benchmark suite were classified based on their parallelism, both total and self. Self-parallelism quantifies the parallelism attributable to a specific region. In contrast, total parallelism includes parallelism inherited from a region's children. Self-parallelism identifies 6× more regions as being serial than total-parallelism. A corresponding drop is seen in the number of regions classified by self-parallelism as being very parallel. The large number of changes in classification emphasizes the importance of self-parallelism in helping to avoid "false positives," regions that are classified as parallel but are serial.

percentage of regions with high parallelism would be parallelized.

Figure 4.12 charts the percentage of regions for both total- and self-parallelism as a function of both parallelism and work coverage. For both total- and self-parallelism, regions with parallelism less than 5.0 are rarely parallelized. This indicates that the regions that self-parallelism classifies as having low parallelism do indeed have low levels of parallelism: they are rarely exploited. Conversely, a much larger percentage of regions that self-parallelism classifies as being very parallel have been parallelized. This indicates that self-parallelism performs better at classifying regions as being very parallel.

While Figure 4.12 shows that self-parallelism is better at predicting whether

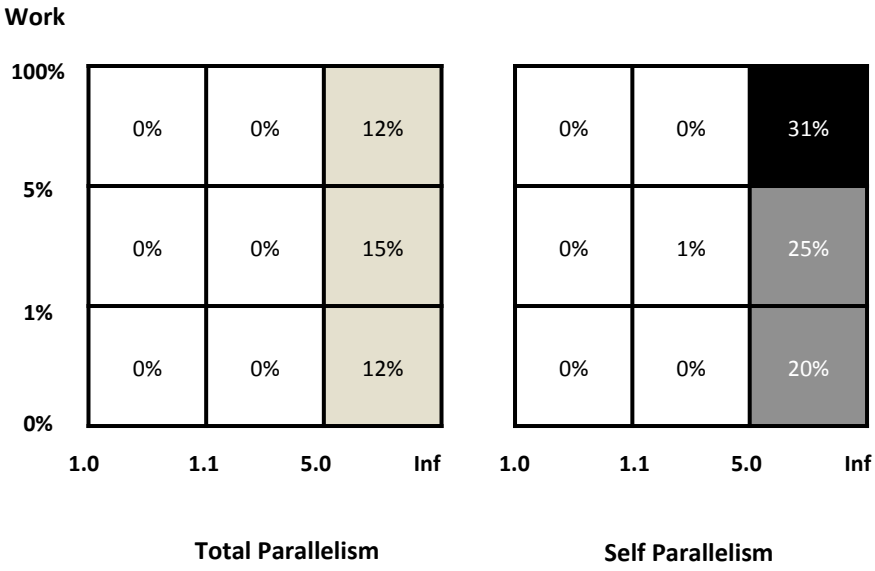


Figure 4.12: **Percentage of regions parallelized as a function of parallelism and work.** All 1953 regions were classified based on their parallelism and work coverage. The chart shows the percentage of regions inside each classification range that were parallelized in by third-party experts. None of the regions with SP of less than 1.1 were parallelized, indicating that SP correctly classified serial regions (Figure 4.11). The percentage for "very parallel" regions (parallelism $>$ 5.0) SP is approximately double that of the same classification with TP. Manual analysis of the non-parallelized regions with SP $>$ 5.0 and work $>$ 5% (i.e. upper right bucket) revealed that most of these regions were also parallel but were not exploited because of region-nesting or infrequently occurring dependencies.

a region will be parallelized, it leaves one open question: why isn't the percentage higher? To determine if self-parallelism was correct in classifying these regions, we examined the 41 regions classified as being in the upper right square for self-parallelism (SP $>$ 5.0, work $>$ 5%) but that were *not* exploited in the third-party version. Of these 41 regions, 9 contained parallelism that was difficult to exploit (e.g. because of difficult-to-resolve WAW dependencies). The remaining 32 had easily exploitable parallelism but we found that either an ancestor or descendant of the region was already parallelized and thus excluded those regions from being parallelized. This re-emphasizes the importance of incorporating region structure information during planning: it is possible that parallelization of a less frequent

region negates the benefit of parallelizing a region with a larger work coverage.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under CAREER Award 0846152, by NSF Awards 0725357, 0846152, and 1018850, and by a gift from Advanced Micro Devices.

This chapter contains material from "The Kremlin Oracle for Sequential Code Parallelization", by Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor, which is set to appear in *IEEE Micro*. The dissertation author was the primary investigator and author of this paper.

This chapter contains material from "Bridging the Parallelization Gap: Automating Parallelism Discovery and Planning", by Saturnino Garcia, Donghwan Jeon, Chris Louie, Sravanthi Kota Venkata, and Michael Bedford Taylor, which appears in *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2010. The dissertation author was the primary investigator and author of this paper.

Chapter 5

From Parallelism to Parallelization Plan

Kremlin's parallelism discovery phase provides the user with the amount of self-parallelism in each region but this alone does not provide the actionable information needed to start parallelization. In this chapter we will look at the process of moving from a parallelism profile to a customized parallelization plan. We'll start by defining parallelism–both informally and formally–before discussing important related issues such as estimating the time of a region after it has been parallelized and identifying the type of parallelism available in a region. Next, we will introduce the concept of *planner personalities* and discuss several planners that target different systems. Finally, we will evaluate how our planner performs by comparing Kremlin's plans to actual parallel programming transformations performed by expert, third-party programmers.

5.1 Defining Parallelism Planning

Parallelism planning ultimately aims to maximize parallel speedup while minimizing the amount of programmer effort. This ultimate goal is not the only goal though as human factors play a large role in forming a good parallelism plan. For example, large, complex programs are likely to require a lengthy parallelization process; it is therefore important that a parallelization plan enable an iterative

process of parallelizing the program region-by-region, starting with the regions with the largest potential benefits.

We will informally define parallelism planning as the problem of producing a sequence of program regions for the programmer to parallelize, ordered according to their expected impact on program execution. Self-parallelism and work provide the basis for creating an effective parallelism plan but additional factors will also impact planning. One such constraint is the risk of over-parallelizing the program: since parallel execution often incurs some overhead in terms of work and/or resource contention, expressing more parallelism than there are cores available to exploit can result in slowdown. For instance, we found that in our experimental setup using OpenMP it was seldom profitable to parallelize a child region of a region that had already been parallelized. Another constraint is that synchronization and data movement costs in the system often affect the smallest parallel region that can attain speedup.

We now formally define the problem of parallelism planning so that we may later solve it algorithmically. Let R_T be the set of regions in a program's summarized region tree T. A parallelism plan creates a tuple $\langle R_P, < \rangle$ that defines both the subset of regions that should be parallelized (R_P) and a relation ($<$) that is true for $A < B$ iff A should be parallelized before B. While it is possible to define this problem in terms of the static region graph G rather than the region tree T, the region tree format enables context-sensitive planning as discussed in Section 4.3.4. The use of a tree also simplifies our planning algorithms; in general, algorithms that work on trees are less complex than those that must handle arbitrary graphs.

A set of constraints, C, will help to define both R_P and $<$. Examples of these constraints range from architecture-specific constraints (e.g. the number of cores available), to language-specific constraints (e.g. inability to express pipeline parallelism in OpenMP), and even human factors (e.g. the desire to achieve large speedups as soon as possible). These constraints are combined to form a personality for the planner. A planner personality may range from detailed (e.g. fine-grained parallelism on a 100-core Tilera machine) to general (e.g. coarse-grained parallelism), depending on the goal of the user; detailed planners will have better

performance on targeted systems while broader personalities will have more robust performance across a broader range of machines.

5.2 Estimating Parallel Execution Time

Parallelism planning requires a model of the impact of various decisions in order to be successful. Kremlin models parallel execution time by leveraging the self-parallelism calculated during parallelism discovery along with other major factors in performance such as the number of available cores, the synchronization overhead of parallelization, and the types of parallelism that can successfully exploited by the target system. With this model, Kremlin can evaluate the effectiveness of many potential parallelization plans, allowing the planner to choose the plan that will lead to the largest speedup.

Kremlin uses the following equation to calculate the parallel execution time of a region R:

$$ET(R) = \begin{cases} \dfrac{\sum_{k=1}^{n} ET(child(R,k))}{min(SP(R), A(R))} + O(R) & \text{non-leaf} \\ \\ \dfrac{work(R)}{min(SP(R), A(R))} + O(R) & \text{leaf} \end{cases} \tag{5.1}$$

where $A(R)$ is the number of cores allocated to R, $child(R,k)$ is the k^{th} child region of R, and $O(R)$ is the parallelization overhead of R.

$SP(R)$ denotes the amount of self-parallelism that is available to be exploited by the system, which is not necessarily the same as the self-parallelism calculated during parallelism discovery. Some systems can profitably exploit only a limited set of parallelism types (e.g. data parallelism on GPUs). In the following section, we will describe how we might determine the type of parallelism available in a region. Kremlin can use this parallelism type information to set $SP(U) = 1$ for any region U that contains unexploitable parallelism.

Leaf and non-leaf regions are handled similarly despite the use of separate equations for each type. The first term for both cases represents the ideal parallelized time of the region when $A(R)$ cores are allocated to it while the second

term ($O(R)$) corrects for the overhead associated with parallelization. The numerator of the first term represents the amount of serial work to be done, either the work in the region itself (if the region is a leaf) of the combined execution times of the region's children. The denominator represents potential speedup available from parallelization. This speedup is limited either by the self-parallelism or the allocated core count, meaning that speedup is limited either fundamentally by the amount of parallelism available in that region or by the amount of parallel resources available in the system.

The second term, $O(R)$ models target-dependent parallelization overhead. Parallel execution typically involves overhead from several sources: thread management, synchronization, communication, etc.. As a result, the overhead factor is highly target-dependent. For example, the synchronization operation takes less than 20 cycles in the MIT Raw processor but takes several thousand cycles on shared memory multi-core processors. As such, Kremlin allows target-dependent customization of $O(R)$ by accepting parallelization constraints. This overhead function directly impacts the parallelization granularity as the amount of work in a region should offset parallelization overhead for a profitable parallelization.

Execution time is defined recursively, with a non-leaf region requiring the execution time of its children. Calculating the execution time of the overall program (i.e. the `main` function) is a recursive process, with leaf regions acting as the base cases. In practice, execution time is calculated in a bottom-up manner starting at the leaves.

Kremlin's basic model for parallel execution time ignores many factors that can affect parallel performance. For example, we showed that modeling the influence of caches on performance leads to more accurate estimates for program speedup [JGLT11]. However, our experience shows us that increasing the accuracy of program speedups does not fundamentally change the results of planning; the results of aforementioned work are therefore beyond the scope of this thesis.

5.3 Identifying Parallelism Types

While Kremlin's self-parallelism profile quantifies the parallelism in each region of the program, there is no guarantee that the parallelism will be expressible. As we alluded to earlier, many systems have limitations on the type of parallelism that can expressed. In this section we will describe a procedure for identifying the type of parallelism available so that we may indicate to our planner that certain regions will not be profitable to exploit.

As previously described, Kremlin's region hierarchy has been designed to ensure that only leaf regions have self-work; any parallelism inside of the leaves is therefore instruction level parallelism (ILP). ILP is typically handled automatically by some combination of the compiler and hardware so most planners will consider the self-parallelism of all leaf regions to be 1 (i.e. not parallelizable by the programmer).

Kremlin checks the region type of non-leaf regions to determine if the parallelism is either loop- or task-based—loop regions being guaranteed to be the former and function regions being the latter. Loop regions with parallelism typically have either an embarrassing amount of parallelism (e.g. DOALL loop) or a small, fixed amount of parallelism (e.g. DOACROSS loops). Embarrassingly parallel loops are fairly easy to detect as the critical path the loop is the same as the critical path of the longest iteration (as shown in Figure 4.8a). Loops with small amounts of fixed parallelism are usually also easy to detect as their self-parallelism is only a small fraction of the total number of iterations of the loop. Systems such as OpenMP are efficient only at exploiting embarrassingly parallel loops: Kremlin is therefore able to cater to their needs in planning by setting the self-parallelism of all other loops to 1.

Function regions with parallelism are similar to loops in that they tend to be either completely parallel (e.g. in task-level parallelism) or have a small, fixed amount of parallelism (e.g. pipeline parallelism). Kremlin can differentiate between the two in the same manner as it handled loops, by comparing critical path lengths of a region with its subregions or comparing the self-parallelism with the number of subregions.

While the above cases cover most regions, there is still the possibility that a region will have a scalable amount of parallelism but not be embarrassingly parallel. The example shown in Figure 4.10 demonstrates this point as the self-parallelism of the outer loop scales with the number of iterations (N) but is not equal to the number of iterations. Kremlin can identify these cases using a simple heuristic: if the ratio of self-parallelism to the number of subregions is higher than some constant c it considers the loop to contain scalable parallelism. This heuristic can of course fail in degenerate cases where there is a large, fixed amount of parallelism in a region; to detect this case, the user could run Kremlin on inputs of varying size and note if there is any change in the self-parallelism for the questionable region.

5.4 Planner Personalities

We can view parallelization as a special type of performance optimization. Optimizations often vary in effectiveness from system to system; the same optimization that brings a large benefit on one processor may even be harmful on another processor. Just as no compiler would blindly apply all possible optimizations for every target, no competent parallel programmer would attempt to parallelize every region of the program that has parallelism. This realization ultimately led us to develop the concept of the *planner personality*.

Planner personalities brings the concept of target-specific optimizations to parallelism planning. Each planner personality summarizes the salient characteristics of a defined target, providing an algorithm that maps the parallelism and program structure uncovered in parallelism discovery to an ordered parallelization plan that is tailored for the specified target. The specified target can be as detailed as desired, with highly detailed specifications providing more accurate plans at the cost of reduced applicability to other systems.

In this section we will discuss several planning personalities: one for the OpenMP environment, a prototype for the OpenCL environment, and one targeting Cilk++. We will also provide some insight gained from the development of these personalities and briefly talk about the process of developing additional

personalities.

5.4.1 OpenMP Planning Personality

OpenMP is a popular parallel programming environment with a strong focus on parallelization of loops. Programmers insert `pragma` statements into their source code and the OpenMP compiler generates the necessary threaded code for them to run. While it does support nested parallelism, the overhead is often too high for it to be effective: the number of execution contexts available is often not enough to handle the extra threads that are spawned and thus the cost of spawning new threads is never amortized. Furthermore, OpenMP requires the programmer to transform loops into parallel (i.e. DOALL) loops in order to achieve good performance.

Kremlin's OpenMP planner takes into account the major constraints associated with OpenMP. The planner disallows nested parallel regions to avoid the performance penalty we observed on our experimental setup. OpenMP supports reduction variables in parallel loops, but they have significant overheads [BO01]. We found that the amount of work in a region should be large enough to amortize these costs. For instance, reduction-based loops in the SPEC OMP2001 benchmarks art and ammp have too little work to overcome overheads. On the other hand, ep, from the NAS Parallel Benchmarks [BBB+91] ("NPB"), has a reduction-based main function that should be parallelized because it has ample work.

Based on these constraints for OpenMP planning, we can formulate the problem as follows. Given a summarized region tree T, select a set of regions to parallelize, R, such that in any path, P, in T there is at most one node in R (i.e. $|P \cap R| \leq 1$). The non-nesting requirement means that all regions in R will have all n available cores allocated to it (i.e. $A(r \in R) = n$). The optimal solution will minimize the time required to execute the program.

OpenMP Planning Algorithm A naive algorithm for determining which regions to parallelize would be to repeatedly select the region with the largest potential speedup among all regions considered for parallelization. When a region is

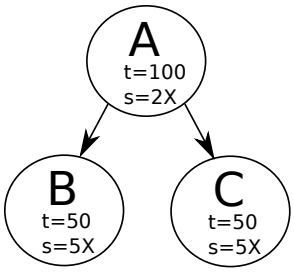


Figure 5.1: **Shortcomings of Greedy Planning.** Using a greedy planning algorithm can lead to sub-optimal results. In the region graph shown here, region A has the largest potential speedup, a potential cost reduction of 50 (2× speedup, 100 → 50). However, each of regions B and C have a reduction of 40 (5×, 50 → 10) for a total reduction of 80 when combined together. A greedy solution would pick region A, leading to a sub-optimal result.

selected by the planner, any region that can reach or can be reached from a selected region would then be excluded from consideration to avoid nested parallelization. In some cases, this algorithm may lead to optimal results but in many cases it is suboptimal. For example, a parent region might have the highest single potential speedup, but collectively, a set of its child regions could offer a higher combined speedup. A greedy algorithm would select the parent, precluding the more optimal solution of selecting the set of child regions. Specifically, this problem was observed in two of the NPB benchmarks: ft and lu.

Figure 5.1 demonstrates how this greedy approach to planning can lead to suboptimal results. In this example, parallelizing region A will lead to a reduction in execution time of 50 (2×, 100 → 50). However, both regions B and C can each be reduced by 40 (5×, 50 → 10) for a total reduction of 80. It is therefore advantageous to parallelize those two regions rather than immediately go for the largest single speedup (from region A).

Kremlin's OpenMP planner employs a dynamic-programming algorithm to find the optimal set of regions to parallelize. This algorithm works as follows. Kremlin breaks down the problem of finding the optimal set of regions R for the whole program (i.e. the root region of the summarized region tree T) into the problem of finding the optimal set of regions for each subtree rooted at one of the root node's children. Kremlin combines the solutions of these subproblems by

taking advantage of the fact that the subtrees of the children do not overlap. This non-overlapping property means there are only two possible choices to be made: either the root region is selected or the union of all subproblem solution sets is selected. If the root region is selected, no other regions can be selected because all other regions are nested under that region. If the root region isn't selected, all subproblem solutions can be combined because the non-overlapping property of each child subtree guarantees that there is no path from a region in on subtree to another. Kremlin uses the parallel time execution model described earlier to determine which of the options will lead to a lower execution time and should therefore be selected.

Kremlin can recursively apply the algorithm just described to each child of the root region to find the optimal solution for the subtree rooted at that region. This recursion will stop when we reach the leaves of the tree as their are no subtrees to consider at that point: the only choice would be to parallelize the region or not.

The preceding algorithm provides the set of of regions that should be parallelized but does not provide an parallelization ordering. Kremlin produces an ordering by setting the core allocation count for all regions to 1 (i.e. not parallelized) then iteratively selecting only one region $r \in R$ to parallelize (i.e. setting $A(r) = n$) and noting the decrease in execution time. The times are then placed in decreasing order to produce a plan that recommends the regions with the largest speedups first.

5.4.2 OpenCL Planning Personality

Graphic processing units (GPU) have recently received considerable attention as general purpose computation devices (i.e. GPGPUs). GPGPUs are now programmable via platforms such as OpenCL and NVIDIA's Compute Unified Device Architecture (CUDA). These platforms allow programmers to exploit the massive data-parallel processing power of GPGPUs, and have led to the rapid adoption of GPGPUs in fields such as scientific computing.

We have developed an OpenCL-based, GPGPU planning personality prototype to help programmers realize the potential of GPGPUs. This personality

builds upon our OpenMP planner as both OpenMP and OpenCL share some common traits. First, both environments work best with embarrassingly parallel region so both use Kremlin's ability to filter out other types of parallel regions. Second, both OpenMP and OpenCL have a limitation on their ability to exploit nested parallelism. In OpenMP this limitation results from the significant overhead associated with nested parallelism. In OpenCL this limitation is fundamentally tied to the programming model: each OpenCL kernel is self-contained and therefore cannot exploit another kernel for additional parallelism.

Our OpenCL personality deviates from our OpenMP personality mainly in the way it estimates parallel execution time. GPGPUs often have hundreds of processing cores, greatly increasing the potential speedup over OpenMP environments where most systems are limited to eight or fewer cores. GPGPUs also differ from OpenMP environments in both the size and nature of parallelization overhead. These devices support hundreds of lightweight threads, each operating mostly independently of one another; this greatly reduces the overhead associated with synchronization on GPUs. Conversely, the OpenCL model introduces a new type of overhead: the overhead required to transfer data from a "host" device (normally a traditional CPU) to the "guest" (e.g. a GPU) that is executing the parallel kernel. This overhead is especially large for discrete GPUs where data must be transferred over a PCI bus that have far less bandwidth than the internal CPU buses. Our OpenCL planner takes these differences into account by appropriately setting $O(R)$ in Equation 5.1.

We plan to continue refining our OpenCL planning personality to more accurately model the OpenCL environment. For example, the data transfer overhead between host and guest scales with the size of data rather than the number of threads working on it. We plan to create a dynamic analysis pass that examines the working set size of each region of the program; this profiling data can then be passed to the planner to more accurately set $O(R)$. This data transfer overhead can also affect planning in other ways; for example, if two high-coverage regions are separated by a low coverage region that has the same working set, additional overhead can be avoided by also recommending the low-coverage region be parallelized

soon after the high-coverage regions.

5.4.3 Cilk++ Planning Personality

Cilk++ [Lei09] is a popular parallel programming environment that extends C++, supporting both fork-join style parallelization and parallel loops. In contrast to OpenMP, Cilk++ leverages very light-weight threads and a work-stealing-based scheduler, helping programmers exploit both nested and finer-grained parallelism.

Kremlin was originally developed with a Cilk++ planner, which was used in the user study described in Chapter 2. This early version of Kremlin did not employ region summarizing or have a concept of self-parallelism. This meant that the parameters of planning were much different than the current version of Kremlin and therefore that version of the Cilk++ planner is no longer relevant.

Cilk++ lacks a large, established benchmark suite, which has hindered our ability to perform a quantitative evaluation of potential Cilk++ planning personalities. We are still however able to posit an approach to planning for Cilk++ planning, which we will now briefly describe.

Unlike OpenMP, Cilk++ does not incur significant overhead for nested parallelism so the limitation on nested parallelism is no longer valid. We are still, however, limited by the number of cores available. Rather than selecting a binary yes/no as to whether a region will be parallelized, we now must decide how many parallel resources (i.e. cores) to allocate to each region. If we let $A(R)$ denote the number of cores allocated to a region R then we have the following constraint: for any path P in the summarized region tree T the following constraint must be met:

$$\prod_{R \in P} A(R) < N \tag{5.2}$$

where N is the total number of cores available. This constraint comes about because core usage is multiplicative between parent and child: if the parent is given i cores and the child is given j cores then there will be i simultaneous children requiring j cores each, a total of $i * j$ cores.

For each region R we must decide how to split the cores between parent and

child in order to minimize execution time while still not violating the constraint in Equation 5.2. However we do not know if an ancestor of R will be better suited to use the cores so we must determine the best allocation for a range of possible core counts, likely from 1 core to N cores.

We can calculate the execution time of R ($ET(R)$) using Equation 5.1 by setting $A(R) = C$ and setting the ET of each child ($ET(child)$) to the time when $A(child)$ is $\frac{N}{C}$. If we work in a bottom-up fashion, we can create a table of "best" execution times for each node for any given value of C and then use that table to determine the $ET(child)$ when calculating $ET(R)$.

While the algorithm above would give us the best allocation of cores to regions, it would not give us an ordering of regions. The best ordering will ultimately depend on the objective of the programmer. The simplest method is to start with all regions allocated only a single core. Next, we find the region R that leads to the smallest execution time when allocating cores to only that region. We would then leave that region with $A(R)$ cores and repeat the process with the remaining regions.

5.4.4 Developing Additional Planner Personalities

Planning personalities provide an avenue for the user to tailor planning recommendations to different systems. Underlying the development of new planning personalities is a fundamental tension between accuracy and portability. The designer of a planning personality must decide the level of architectural independence that is part of the personality. Architectural independence is a desirable property for portability, allowing the planning results to be useful over a wide range of systems, but may need to be sacrificed in order to attain sufficient accuracy.

The development of the OpenMP and Cilk++ personalities provided some insight into the portability-accuracy trade-off. These personalities required that we model only fundamental parameters of the parallel machines: synchronization costs, types of exploitable parallelism, and the profitability of nested parallel regions. These parameters are likely to port well to other parallelization systems, reducing the work necessary to develop new planners for these other systems. Our

first two personalities required algorithms to handle both nesting (Cilk++) and non-nesting (OpenMP) parallelism environments. We expect these two algorithms to provide the basis for many different personalities, as demonstrated by our re-use of the OpenMP planning algorithm for the our GPGPU personality.

We found that while machine-specific parameters such as cache size, page size, and memory bandwidth do influence parallel performance, and influence how code should be transformed, they have limited impact on the set of regions that should be parallelized. These machine-specific parameters therefore are of greater import during the *Enabling Transform* stage of parallelization than during the *Planning* stage.

5.5 Experimental Evaluation

We evaluated Kremlin's planning abilities using all 8 programs in the NAS Parallel Benchmarks (NPB) [BBB+91] and all 3 C-language programs in the SPEC OMP2001 benchmark suite. For NPB, we used the third-party OpenMP manually-parallelized version of these programs [omn] as a point of comparison for Kremlin's ability to create an effective parallelization plan. For SPEC OMP2001, we ran our tool on the corresponding serial versions of the programs in the SPEC 2000 benchmark suite, and then compared Kremlin's plans against those parallelized by humans in the SPEC OMP2001 versions. For art and ammp, SPEC OMP versions benefit from serial optimizations compared to their SPEC 2000 counterparts [TWFO09]. To exclude the effect of serial optimizations, we applied those optimizations on the SPEC 2000 code before running Kremlin. Our evaluation included only third-party benchmarks that have preexisting parallel versions to facilitate comparison and to make our results more credible. The programs vary greatly in terms of speedup (1.5x to 25.89x, Figure 5.2), but low coverage, low parallelism, parallelization overhead, and other factors significantly reduce the percentage of regions that are good candidates for parallelization (Figure 5.4).

One might expect that iterative, "trial and error" manual parallelization would do significantly better than Kremlin, because the user has the benefit of

performing iterative runtime measurements as they incrementally parallelize the program. We found that parallelization with Kremlin came surprisingly close in terms of performance on all but two benchmarks, and in those cases, it did much better. At the same time, it achieved these results with substantially smaller numbers of regions that needed to be parallelized.

5.5.1 Methodology

We first ran Kremlin on the unmodified, serial versions of the benchmarks to generate a parallelism plan for each program. The resulting plan was used to create a parallelized version of the serial program. In cases where Kremlin's parallelism plans recommended regions that had also been parallelized in the third-party, manually-parallelized version of the benchmark ("MANUAL"), we reused the parallelized regions from the MANUAL version. This allowed us to control for variances in performance that could result from slightly different parallel implementations of the same region.

To generate and evaluate parallelism plans, the 'W' input set was used for NPB benchmarks while the 'train' input was used for SPEC OMP. Kremlin relies on dynamic analysis and therefore may be affected by varying inputs. To test for input-related sensitivities, we reused the parallelized program based on the 'train' input parallelism plan to measure the speedup numbers for SPEC OMP benchmarks with the larger 'ref' input. We found that Kremlin-based parallelization remained equally competitive on both input sizes, despite requiring a much smaller set of parallelized regions.

Program performance was tested on 32-core system (8 $\times$ AMD 8380 Quad-core processors) with 256GB of memory running on the Linux 2.6.18 Kernel. Programs were compiled with `gcc` version 4.1 with OpenMP and -O3 flags specified. We executed the programs using configurations of 1, 2, 4, 8, 16, and 32 cores. As is typical for these kinds of systems, performance can decline as locality effects start to trump the benefits due to parallelization. For each parallel version, we determined the configuration with the best performance and report that number.

Table 5.1: **Evaluating Plan Size.** Kremlin offers significantly smaller plan sizes (1.57× on average) than the MANUAL implementation.

Benchmark	MANUAL	Kremlin	Overlap	**Reduction**
ammp	6	3	2	**2.00x**
art	3	4	1	**0.75x**
equake	10	6	6	**1.67x**
bt	54	27	27	**2.00x**
cg	22	9	9	**2.44x**
ep	1	1	1	**1.00x**
ft	6	6	5	**1.00x**
is	1	1	0	**1.00x**
lu	28	11	11	**2.55x**
mg	10	8	7	**1.25x**
sp	70	58	47	**1.21x**
Overall	211	134	116	**1.57x**

5.5.2 Comparing Plan Size

Kremlin seeks to focus the programmer's efforts on a small subset of regions that have the most potential for speedup from parallelization. To test Kremlin's effectiveness in this regard, we compared Kremlin's recommended regions ("plans") to the set of regions that were parallelized in the third party-parallelized version of the benchmark suite, referred to as MANUAL. Figure 5.1 provides this plan size comparison.

Across all of the regions in the benchmarks, the MANUAL version included 1.57× more regions than the plan provided by Kremlin. For small benchmarks (e.g. ep, ft, and is) there was little room for improvement, but larger, more complex benchmarks showed larger savings compared to the average. At the extreme end, lu's manually-parallelized plan size was 2.55× the size of Kremlin. The programmer using Kremlin would have had far fewer regions to parallelize than the original third party programmers. [1]

[1] Programmer effort metrics for the *Enabling Transforms* part of parallelization is clearly a hard problem. We have also explored other metrics, like lines of code, as proxies for programmer effort in Kremlin, since it could perhaps be a better proxy for parallelization complexity. However, our impression from the benchmarks is that, at least, for OpenMP, region count is a better, albeit

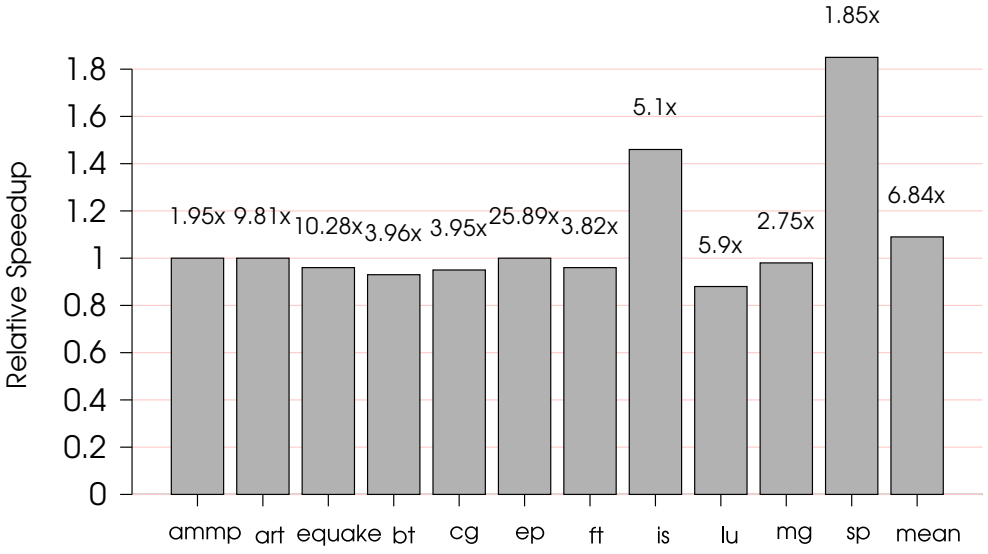


Figure 5.2: **Evaluating the Performance of Kremlin-based Parallelization.** Even though Kremlin proposes a substantially smaller number of regions to a user (Table 5.1), this graph shows that the resulting performance is generally quite close to the MANUAL parallelized versions, ranging from 12% slower to 85% faster. Note that Kremlin formulated its plans solely by examining the execution of the unmodified serial code. In order to reduce the experimental effects of different effort levels and different programmers for hand tuning, we evaluated the plans for Kremlin by using the parallelized code regions in the manually-parallelized version. In the case of SP and IS, Kremlin's recommendations were significantly different, so we had to manually apply those optimizations.

5.5.3 Performance Comparison

Next, we evaluated the speedup of parallelized versions based on Kremlin's parallelism plan against the MANUAL version. Figure 5.2 shows the results of this comparison. The Kremlin version of sp and is performed significantly better (1.85×, 1.46×) than MANUAL as Kremlin was able to identify parallelism that was missed in the MANUAL version. In this case, Kremlin recommended a coarse-grained parallelization, requiring privatization and refactoring. Other benchmarks saw a slight degradation in performance, averaging about 3.8%. Kremlin generally selected the same regions as MANUAL, but decided to stop earlier because of the

imperfect, approximation of programmer effort.

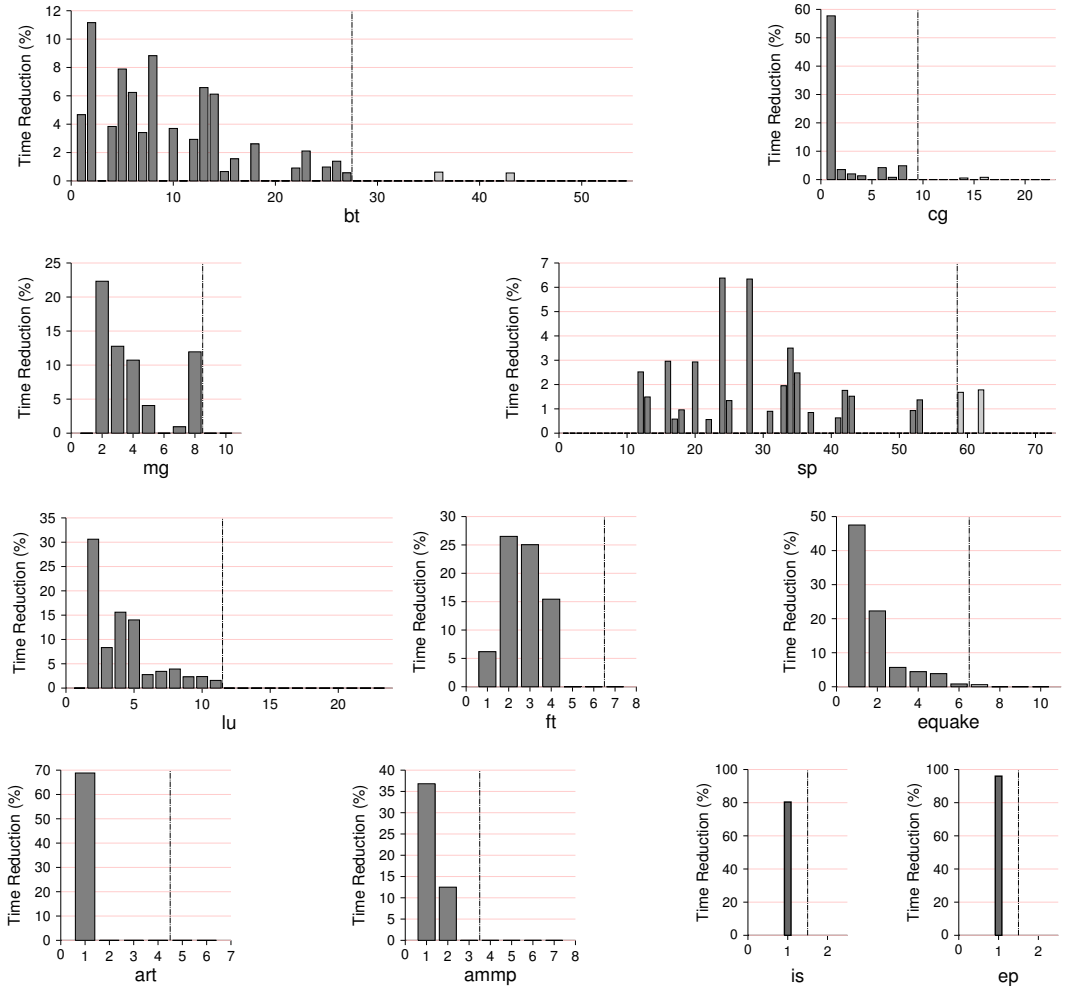


Figure 5.3: **Effectiveness of Region Prioritization.** Kremlin provides a list of regions prioritized by their estimated speedup so that users can maximize their productivity. The graphs above show the marginal decrease in execution time, relative to the original program run time, as each region in Kremlin plan is parallelized. We also included regions that were filtered out in the Kremlin plan but were chosen to be parallelized by the expert third party (MANUAL). These regions are shown to the right of the dotted line. As the graphs illustrate, little benefit came from regions that were parallelized by the third-party but that were not suggested by Kremlin.

diminishing returns.

To gain additional insight, Figure 5.3 shows the marginal benefit attained by applying each of the recommendations, in order, from Kremlin's plans. Also shown in the graphs are the marginal benefits of regions parallelized in MANUAL

but *not* recommended by Kremlin (regions to the right of the dotted line).

In a large majority of cases, regions not recommended by Kremlin but parallelized by MANUAL provide negligible benefit. Additionally, we can see that, although Kremlin's plans are well-prioritized overall, the incremental contribution of a parallelizing a region can be somewhat noisy. For instance, in several cases, the second recommended region attains a much higher incremental speedup than the first recommended region—this is because as more of the program is parallelized, less data migration happens in the NUMA machine. Often it is groups of regions that must be parallelized before any speedup is observed.

Overall, Kremlin does an excellent job of eliminating regions that offer little benefit. Even for those few regions that were eliminated by Kremlin but had some marginal benefit, the benefits are slight. Given the savings in the number of regions parallelized by Kremlin, we suspect that the programmer could easily make up the difference by applying serial optimizations rather than attempting to parallelize the additional regions.

5.5.4 Effectiveness of Region Prioritization

An important aspect of planning is to ensure not only that the regions with the most benefit are selected but also that they are prioritized correctly. The planner attempts to place regions with the largest benefit at the beginning of the plan. Meeting this goal maximizes the productivity of the programmer by focusing their efforts where they are most valuable.

To evaluate the effectiveness of the ordering produced by Kremlin, we measured the fraction of total realized execute time reduction attained by following increasing portions of Kremlin's plans, including the first 25%, first 50%, first 75%, and all 100% of the plan. We would expect well-prioritized plans to generally produce monotonically decreasing benefits for each additional fraction that is added. As shown in Table 5.2, Kremlin's plans are well-prioritized. The first 25% of the plans average 56.2% of the benefit, the next 25% averages 30.2% of the benefit, while the following 25% yields 9.2%, and the last 25% yields 4.4% of the benefit.

Table 5.2: **Marginal Benefit of Region Parallelization.** A well-prioritized parallelism plan will show decreasing marginal benefits as more of the recommended regions are parallelized. This table shows the average marginal benefit of 25% increments in the fraction of regions parallelized. The final row shows that a majority (56.2%) of benefit comes from the first 25% of regions with the following intervals showing decreasing average marginal benefits. This suggests that Kremlin's parallelism planner is effective at region prioritization.

	Fraction of Kremlin Plan Applied			
Benchmark	**First 25%**	**First 50%**	**First 75%**	**All 100%**
ammp	74.7 %	100.0 %	100.0 %	100.0 %
art	100.0 %	100.0 %	100.0 %	100.0 %
equake	82.5 %	89.2 %	99.0 %	100.0 %
bt	48.9 %	85.8 %	92.2 %	100.0 %
cg	84.9 %	86.7 %	93.5 %	100.0 %
ep	100.0 %	100.0 %	100.0 %	100.0 %
ft	44.7 %	78.9 %	100.0 %	100.0 %
is	100.0 %	100.0 %	100.0 %	100.0 %
lu	45.8 %	84.0 %	95.4 %	100.0 %
mg	35.6 %	73.0 %	79.5 %	100.0 %
sp	9.6 %	62.1 %	94.5 %	100.0 %
average benefit	56.2 %	86.4 %	95.6 %	100.0 %
marginal average benefit	**56.2 %**	**30.2 %**	**9.2 %**	**4.4 %**

5.5.5 Influences on Plan Size

Next, we evaluated how plan size is reduced as additional information is taken into account. The factors that we looked at were work coverage, self-parallelism, and usage of the full OpenMP planner personality. Figure 5.4 illustrates the impact of each of these factors on the programs. Programmers that take into account only work information (e.g. a `gprof`-based approach) would be left with an average of approximately 59% of the total regions to analyze and attempt to parallelize. With the addition of self-parallelism information for each region, the average plan size is cut to 25.4% of all regions. Finally, when using the

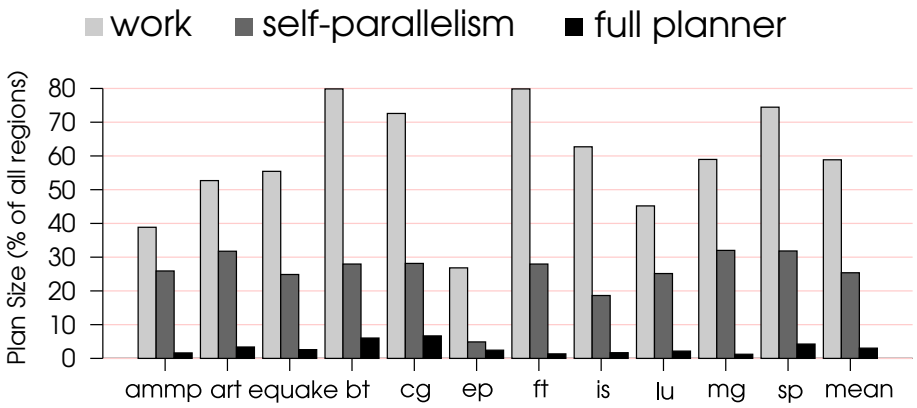


Figure 5.4: **Effects of Factors on Plan Size.** Plans based only on work coverage comprised 58.9% of all regions on average. Using self-parallelism to eliminate low parallelism regions cut this more than half (25.4%, on average). Finally, using the full OpenMP planner personality, the plan size was reduced to only 3.0% of the total regions.

full planner an average of only 3.0% of the regions are included in the plan. As we have shown in Figure 5.2, despite only parallelizing a fraction of the regions, Kremlin achieves performance that is comparable to the highly-tuned MANUAL version.

5.5.6 Initial GPGPU Planning Results

We described in Section 5.4.2, we have created a prototype GPGPU planning personality for OpenCL environments. To test the effectiveness of this prototype, we examined the programs in version 2.0.1 of Rodinia [CBM+09], a benchmark suite for heterogeneous programming environments. The benchmarks initially targeted an NVIDIA GTX 280 GPU running CUDA but were later ported to the OpenCL programming environment.

Table 5.3 shows our initial results for OpenCL planning on the Rodinia benchmarks. Kremlin's recommendations exactly matched those of the third-party version on 8 of the 12 (i.e. 66%) analyzed benchmarks.[2] In two cases (cfd and

[2]The lud and particlefilter benchmarks could not be analyzed due to runtime errors.

Table 5.3: **OpenCL Planning Results.** Preliminary results show that Kremlin is effective at selecting parallelization regions for code in the OpenCL-based parallelization of the Rodinia benchmark suite. In many cases, Kremlin matched the third-party human manual parallelization exactly. On two benchmarks, cfd and srad, Kremlin found low coverage for regions that had been turned into kernels, suggesting wasted parallelization efforts.

Benchmark	OpenCL Kernels (Manual)	Kremlin Recommended	Overlap
backprop	2	2	100%
bfs	2	2	100%
cfd	4	3	75%
heartwall	1	1	100%
hotspot	2	2	100%
kmeans	2	2	0%
lavaMD	1	1	100%
nn	1	1	100%
nw	2	3	66%
pathfinder	1	1	100%
srad	5	3	60%
streamcluster	1	1	100%
Total	24	22	79%

srad), Kremlin found regions of low coverage (i.e. coverage $< 3\%$ of total execution) that had been turned into OpenCL kernels, potentially leading to wasted programming effort. In another benchmark (nw), Kremlin was able to identify a missed opportunity for exploiting data-level parallelism on a high-coverage (38%) region.

In the future we plan to parallelize each benchmark according to Kremlin's recommendations to establish the "ground truth" for regions that did not match.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under CAREER Award 0846152, by NSF Awards 0725357, 0846152, and 1018850, and by a gift from Advanced Micro Devices.

This chapter contain materials from "Kremlin: Rethinking and Rebooting gprof for the Multicore Age", by Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor, which appears in *PLDI '11: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. The dissertation author was the primary investigator and author of this paper.

This chapter contains material from "Kismet: parallel speedup estimates for serial programs", by Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor, which appears in *OOPSLA '11: Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. The dissertation author was the secondary investigator and author of this paper.

This chapter contains material from "The Kremlin Oracle for Sequential

Code Parallelization", by Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor, which is set to appear in *IEEE Micro*. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Improving Kremlin's Practicality

Kremlin's parallelism discovery phase relies on a new dynamic program analysis known as hierarchical critical path analysis (HCPA). HCPA is similar to traditional CPA in that it is a heavyweight analysis; unlike CPA however, HCPA analyzes every region of the program separately rather than analyzing only a single region. This additional requirement places an even larger burden on an already heavyweight analysis; without proper design, both the runtime and memory overheads can quickly make HCPA impractical. In Chapter 4, we discussed basic techniques for managing the runtime and memory overheads of HCPA. These techniques include concurrent analysis of multiple regions–to reduce runtime–and sharing of shadow memory between multiple levels–to reduce memory overhead.

In this chapter we will look at two advanced techniques we developed to further reduce the overheads associated with HCPA: a novel, space-efficient shadow memory organization, and static partial evaluation of critical path analysis. The latter technique takes advantage of Kremlin's static instrumentation architecture (introduced in Chapter 3) while the former leverages key properties of HCPA to produce a shadow memory that makes the common case fast and uncommon cases space-efficient. These two techniques reduce HCPA's requirements to the point where most laptops can run Kremlin on sizable inputs.

The work in this chapter is part of a wider effort to make Kremlin more efficient and therefore more accessible to a wide range of programs and systems. The dissertation author was the secondary investigator of this work. The descriptions

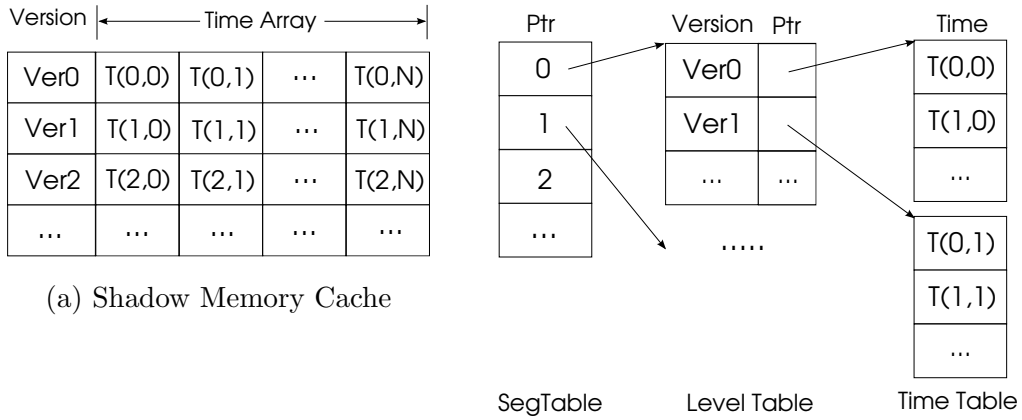
Table 6.1: **Shadow Memory Overheads for HCPA.** Kremlin's initial implementation of HCPA resulted in high memory overheads. This table shows that the average memory expansion overhead for a selection of Spec2000 ("ref" inputs) and NAS Parallel Bench (NPB, "B" inputs) is 49×, resulting in an average memory usage of 15.7GB. This limited its utility in standard workstations, which typically do not have this much memory. New techniques have allowed us to reduce this memory expansion factor to 5.2×, making Kremlin practical on a wider range of machines.

Suite	Benchmark	**W/ Shadow Memory (GB)**	Native Memory (GB)	Memory Expansion Factor
Spec	bzip2	**28.2**	.189	149×
	mcf	**16.0**	.152	105×
	gzip	**21.7**	.200	109×
NPB	sp	**8.0**	.316	25×
	mg	**13.0**	.449	29×
	cg	**14.4**	.427	34×
	is	**13.9**	.384	36×
	ft	**66.0**	1.683	39×
Geomean		**15.7**	.324	49×

contained within this chapter are intended to summarize the techniques in the context of Kremlin's goal of being a practical parallelization oracle. The results are also focused on this objective. For a fuller description of Kremlin's novel shadow memory architecture and additional results related to these techniques, the reader is referred to the forthcoming doctoral dissertation by Donghwan Jeon.

6.1 Efficient Shadow Memory Organization

While runtime overhead is likely the most salient characteristic of a heavyweight analysis infrastructure, the memory overhead is potentially more damaging to such an infrastructure. If the memory overhead is large, it can cripple the system by severely limiting the size of inputs that can successfully be run, the system on which the infrastructure can be used, or both. Table 6.1 demonstrates the



(a) Shadow Memory Cache

(b) Shadow Memory Backing Store

Figure 6.1: **Efficient Shadow Memory Organization.** Kremlin's improved shadow memory organization utilizes a direct-mapped cache (a) to make the common case fast. Only when valid shadow data needs to be evicted does it get written to the space-efficient but slower access backing store (b). Two novel techniques, SlimTV and BulkTV, make this organization possible by greatly reducing the amount of memory overhead required for validation of data. While not shown, compression is used for all but the most recently used time tables to further reduce memory overhead.

scale of this problem. On a set of Spec2000 and NPB benchmarks, the average memory expansion factor (i.e. the ratio of instrumented to native memory usage) for the baseline implementation HCPA was 49×. This resulted in an average minimum required memory size of 15.7GB, an amount far exceeding what is typical in contemporary computer workstations.

Traditional memory shadowing infrastructures have largely overlooked the issue of memory overhead because they have not required hierarchical analysis. Hierarchical analysis calls for profiling multiple regions simultaneously, which as we have previously seen, requires multiple shadow tags per address. To address the issue of memory overhead, we have introduced a novel memory shadowing infrastructure that leverages the key properties of hierarchical analysis to greatly reduce memory overhead while not sacrificing runtime overhead.

Figure 6.1 illustrates Kremlin's improved memory shadowing infrastructure. This infrastructure attempts to make the common case fast while all other cases are handled in a space-efficient manner. Kremlin makes the common case fast by

providing a small, direct-access cache for commonly used addresses, as shown in Figure 6.1a. Kremlin reduces memory overhead by employing a novel, three-level backing store for infrequently accessed addresses, as shown in Figure 6.1b.

Kremlin's shadow memory cache differs from traditional caches in that it is not a subset of its backing store: it is only when a line needs to be evicted that it is placed in the backing store. This policy resembles that of a *write-back* policy for traditional caches but Kremlin further optimizes storage by first validating the evicted data; any tags that are invalid (i.e. have were written by a region that has already finished) are not written out. This policy is extremely efficient as our experience shows that a cache line that is being evicted is usually sufficiently old that most, if not all, of its tags are invalid.

Kremlin's space-efficient, three-level backing store is enables by two key techniques: slim and bulk tag validation. Slim tag validation is also utilized by the shadow memory cache. While not the focus of this thesis, below are descriptions of these two techniques.

Slim Tag Validation (SlimTV) SlimTV replaces the version vector of HCPA's baseline implementation (shown in Figure 4.6) with a single version, reducing the space overhead of version management from $O(n)$ to $O(1)$ where n is the number of levels. This change also leads to a reduction in runtime overhead as the number of loads/stores required for each operation is greatly reduced.

SlimTV relies on the key insight that unique IDs can be used to create a total ordering of all regions in the region tree. SlimTV assigns IDs to regions in the order in which they begin; a larger ID indicates that a region began *after* the region with a smaller ID. HCPA's hierarchical nature also ensures that a region's children will all have larger IDs. Kremlin constantly maintains a vector of IDs associated with the set of regions that is currently active; newly entered regions push their ID to the back of this vector while exiting a regions causes the last entry in this vector to be removed.

SlimTV sports a validation process that uses a single stored ID to determine the deepest level whose data is still valid. Writing to memory results in tagging the associated address with the ID of the deepest active region (i.e. the last ID in the

vector). Reading from memory triggers a validation procedure to determine which of the stored "parallel time" tags can be safely used. This validation checks which IDs in the vector of active IDs are less than the ID stored with the address being read; all other active IDs are from regions that began after the tags were written so these regions should ignore the stored tags. SlimTV reduces the problem of tag validation to finding the minimum region level with an invalid tag: active regions at deeper levels must have started later and therefore are also invalid.

Bulk Tag Validation (BulkTV) While SlimTV greatly reduces the overhead introduced by version management, it still requires a non-trivial amount of overhead. For example, when shadowing every byte of memory, the 8-byte version ID used by HCPA will result in 8× memory expansion.

BulkTV reduces tag validation memory expansion by amortizing validation overhead across a range of memory addresses. This amortization is accomplished by using only a single version ID for a page of shadow memory, performing tag validation for all entries in the page whenever a single address is accessed. Figure 6.1b shows how this is implemented in Kremlin; each level table entry contains a single version, which corresponds to the version for all entries in the pointed-to tag table. The effectiveness of this technique is clearly tied to the size of the page with bigger pages producing bigger benefits. For example, a modest 4KB page leads to a drastic reduction of 4096X when shadowing every byte.

BulkTV works in concert with SlimTV. Reading from or writing to a tag table triggers the same validation process that happens on a SlimTV read, the difference being that each level has its own tag table so invalidation requires invalidating a range of addresses rather than a single address. Invalidating a tag table is a relatively lightweight operation as Kremlin maintains a free list of tag tables, which can be easily switched in by simply changing the pointer in the level table. Invalidated tables can be scrubbed (i.e. all values returned to 0) and then placed on the free list.

As with SlimTV, BulkTV can also have an impact on runtime overhead. BulkTV can affect runtime overhead in two major ways. First, BulkTV can potentially increase the time required for correcting invalid tags; what previously

involved simply writing a 0 into a vector now involves pulling a free table from a list and resetting the pointer to that table. The impact of this effect will depend on the locality exhibited by the program as higher locality means fewer invalidations and therefore lower overhead.

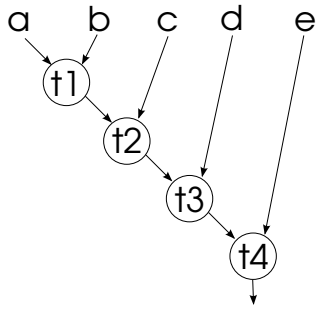
Second, BulkTV greatly reduces the overhead associated with the costly tag validation procedure. SlimTV requires examining up to n entries in the active ID vector to determine the deepest region that is still valid; BulkTV still requires this lookup and has the same worst (n comparisons) and best case (1 comparison) but the average case tends to have fewer required comparisons. The improved average case is a result of leveraging locality. When similar addresses are accessed by the same or a closely related region, the later accesses benefit from the fact that the version ID is updated by the earlier accesses; most active IDs should still be valid and therefore only one or two IDs will need to be checked before we find the first valid ID—as long as we start from the end of the vector.

Other Benefits of Kremlin's Shadow Memory Architecture In addition to benefiting from SlimTV and BulkTV, Kremlin's improved shadow memory architecture lends itself to two other major techniques: compression and garbage collection. Compression greatly reduces memory usage but can quickly lead to large runtime overheads if not utilized correctly. Kremlin's split cache-storage architecture keeps frequently accessed data in the cache, freeing up the possibility of compressing the backing store. Kremlin keeps only a small subset of tag tables in an uncompressed state, with the rest being compressed for storage and then decompressed when needed again. Kremlin places newly evicted cache entries in uncompressed tables as they are the most likely to be used again. A simple "clock" algorithm is used to determine which page will be compressed should the number of uncompressed tag tables reach its limit.

Kremlin employs a simple garbage collector that scans all active level tables and determines which of them are no longer valid. Invalidated tables are either scrubbed and then sent to the free list or simply deallocated if there are already enough entries in the free list.

```
def foo(a, b, c, d, e):
    t1 = a + b;
    t2 = t1 + c;
    t3 = t2 + d;
    t4 = t3 + e;
    return t4;
```

(a) Code example.



(b) Dependency graph.

Figure 6.2: **Exploring Optimization Possibilities.** The dependency graph for the code example shows that most of the values (`t1`,`t2`,`t3`) are intermediate values. Our initial HCPA implementation calculated and stored each of these intermediate values but memory bandwidth can be reduced by expressing outputs as functions of inputs, eliminating many loads and stores in the process.

6.2 Static Partial Evaluation of CPA

Chapter 4 described our basic approach to instrumenting a program to enable the implementation of hierarchical critical path analysis. This approach provides a clean abstraction for calculating the critical path, but it also completely places the burden of calculating critical path on the runtime environment. This section will describe a technique for moving some of that burden to static analysis time, which can significantly reduce the runtime overhead of HCPA.

As we have discussed, Kremlin calculates critical path lengths with the assistance of shadow memory. Our initial approach required the following 4 steps for every binary instruction (e.g. add) in the program, each required ℓ times to account for all ℓ active program regions:

1. Read stored availability time for each input from shadow register.
2. Perform a `max` operation to determine the latest available time.
3. Add the cost of the performed operation (e.g. add or mul) to the maximum time calculated.
4. Store the result into the shadow register associated with the operation.

Critical path analysis requires steps 2 and 3 to calculate the critical path length but the remaining steps are simply overhead associated with keeping calculated "parallel times" in shadow memory. The additional steps produce significant runtime overhead as they tax the memory system with many loads and stores. However, many of these loads and stores are unnecessary.

Figure 6.2 demonstrates how many shadow register accesses can be avoided. Figure 6.2b represents the dependency graph of the code snippet shown in Figure 6.2a. Variables `t1`, `t2`, and `t3` are intermediate values that are used only as part of the calculation of the output, `t4`. The times calculated for these variables would ideally be passed directly from source to destination, avoiding reads from and writes to shadow registers.

We can statically analyze a program to identify intermediate values and avoid storing temporary values for them. This analysis is keyed by the realization that we can express the availability times of all outputs solely as a function of the inputs. The dependency graph in Figure 6.2b provides the insight necessary to understand how this is possible. The graph in this example has 5 inputs (`a` - `e`) and a single output (`t4`). We can see that the critical path is limited to 5 possible paths:

- Starting at a and going through all four nodes.
- Starting at b and going through all four nodes.
- Starting at c and going through the final three nodes.
- Starting at d and going through the final two nodes.
- Starting at e and going through only the final node.

We cannot statically determine which of these paths is the correct one but we can simplify the calculation of the output time to the following:

$$\begin{aligned} time_{t4} = & max(time_a + costs(t1, t2, t3, 4), time_b + costs(t1, t2, t3, t4), \\ & time_c + costs(t2, t3, t4), time_d + costs(t3, t4), time_e + costs(t4)) \end{aligned} \tag{6.1}$$

where $time_X$ is the parallel time of value x (stored in shadow memory) and `costs()` is simply the sum of the cost of each operation. The number of `max` operations and additions has not changed; as discussed earlier, this is a requirement of CPA. What has changed is the number of reads and writes to shadow registers: from 8 reads (2 for each instruction) down to 5 (1 for each input) and from 4 writes (1 for each instruction) down to 1 (for the lone output). These savings are multiplied by ℓ to account because each of the ℓ active levels obtains this savings.

The savings of this technique is directly tied to the number of values that are considered inputs and outputs. Kremlin optimizes the number of inputs by classifying as inputs only values that cannot be statically determined. These values include function arguments, values returned from function calls, loads, and LLVM ϕ-nodes. Function arguments are required because each function may have multiple call sites and therefore the source of the function arguments is not known until runtime. This break at function boundaries also forces values returned from function calls to be considered inputs. Loads are required because alias analysis is imprecise and we therefore cannot determine what values previously wrote to the address being pointed to. ϕ-nodes are required because by definition their value is one of several options, being finally determined only during runtime.

Similarly to inputs, Kremlin limits the values considered to be outputs to those that cannot be statically determined. These values include function return values, values passed as parameters during a function call, and stores. Also similar to inputs, these requirements all stem from limitations of interprocedural analysis (the first two) and pointer analysis (stores).

The downside of this approach is that it may possibly result in additional, redundant computations. For example, consider the scenario when one of the intermediate values in Figure 6.2a (e.g. `t3`) is used by an additional output (e.g. `t5`). This will result in the `max` calculations for `t4` and `t5` sharing many of the same terms. It would be computationally more efficient to calculate the time for `t3` and reuse that value when calculating the outputs. However, that would also increase the required memory bandwidth because of additional reads and writes to the shadow register for non-output values.

While we would like to find the optimal set of intermediate values that should be stored to minimize total runtime, this optimization problem is non-trivial. It may in fact be NP-complete as the dependency graphs can be arbitrarily complex. We plan to look into this issue in the future; however, our results show that runtime overhead often significantly decreases even without trying to minimize redundant computations. This performance boost leads us to believe that memory bandwidth is ultimately the limiting factor in our runtime overhead.

6.3 Evaluation

Methodology We examined the effectiveness of both the novel shadow memory architecture and the static partial evaluation of CPA by looking at both the memory and runtime overheads. For memory overhead we tracked the maximum memory required because it determines the minimum amount of memory necessary for Kremlin to successfully complete. Measurements for evaluating Kremlin's unique shadow memory architecture were performed on a 32-core system (8X AMD Opteron 8380 Quad-core processors) with 256GB of memory running on the Linux 2.6.18 kernel. Measurements for evaluating static partial evaluation of CPA were performed on an 8-core system (2X Intel Xeon E5530 Quad-core processors) with 24GB of memory running on the Linux 2.6.18 kernel. For compression, we employed the miniLZO 2.06 library [Obe].

Our evaluation looked at 12 benchmarks across three benchmark suites: SpecInt 2000, SpecFP 2000, and NAS Parallel Bench (NPB) [BBB+91]. Benchmarks were selected from these suites so long as they had a native memory footprint of 1MB or greater, were written in C, and had a runtime of less than 12 hours. The minimum footprint requirement was selected so that memory overheads were representative of scaling cost rather than the modest fixed overhead of shadow memory. We used SpecInt and SpecFP's 'ref' input set–as it was the largest available–and NPB's 'A' input set–as larger inputs can require several GB of memory natively–for all results.

The selection of three benchmark suites allowed for a range of types of

Table 6.2: **Memory Usage with Optimized Shadow Memory.** Kremlin's optimized shadow memory reduces the memory expansion for shadow memory to an average of 10.16× without compression and 8.65× with compression. Without these optimizations a system with 28.182GB of memory would be required to run all the analysis; with these optimizations, the maximum system memory requirements are reduced to 2.51GB (without compression) or 0.963GB (with compression).

Benchmark		Memory Usage (GB)				Expansion Factor	
Suite	Name	Native	Baseline	All w/o Compr.	All Opts	All w/o Compr.	All Opts
SpecInt	bzip2	0.189	28.182	0.911	0.586	4.82	2.44
	gzip	0.200	21.759	0.488	0.380	2.44	1.90
	mcf	0.152	15.988	0.666	0.593	4.38	3.90
	vpr	0.003	0.282	0.077	0.077	25.50	25.50
SpecFP	art	0.002	0.157	0.083	0.076	41.29	37.90
	equake	0.037	2.113	0.222	0.159	6.00	4.30
	mesa	0.020	1.204	0.174	0.174	8.71	8.70
NasPB	cg	0.055	1.538	0.224	0.154	5.09	3.50
	ft	0.419	17.125	2.027	0.963	4.84	2.30
	is	0.068	2.546	0.350	0.270	5.14	4.00
	lu	0.043	1.124	0.343	0.284	7.97	6.60
	mg	0.434	13.051	2.510	0.911	5.78	2.10
	max	0.434	28.182	2.510	0.963	41.29	37.90
	avg	0.134	8.756	0.673	0.386	10.16	8.65

programs. SpecFP and NPB benchmarks tend to have regular memory access patterns and contain many dense, array-based operations. Conversely, SpecInt benchmarks have more irregular memory access patterns in addition to deeper region hierarchies.

6.3.1 Shadow Memory Optimization

Evaluating Memory Requirements Kremlin's shadow memory optimization primarily focus on reducing the memory overhead associated with performing hierarchical critical path analysis. Table 6.2 shows the memory requirements when various combinations of optimizations are used. The required memory for the

implementation without any optimizations (our "baseline") averages 8.576GB–an amount that even most new desktop machines would struggle to support–but goes as high as 28.182GB–an amount usually available only in supercomputers. A combination of SlimTV, BulkTV, and garbage collection (labeled "All w/out Compr.") bring this down to an average of 673MB with a maximum of 2.51GB, both reasonable to expect available on even on many laptops sold today. Kremlin's ability to compress the time tables further reduces the requirements to 963MB and 386MB for the worst and average case, respectively.

Table 6.2 also shows the memory expansion factors for the benchmarks. Kremlin's optimized shadow memory requires an average of only 10.16× if compression *isn't* used or 8.65× if compression *is* used. With these techniques, it is possible to run programs requiring approximately 800MB or 1GB natively on a modest system with 8GB of system memory, depending on whether or not compression is used. If even larger inputs are required, Kremlin's region summarizing opens the possibility of performing multiple runs, each with a subset of levels in the region tree instrumented, and quickly combining their results to achieve whole-program coverage.

Evaluating Runtime Overhead As previously discussed, Kremlin's shadow memory optimizations can affect runtime overhead in both positive and negative ways. We measured the slowdown relative to native execution time with the same combinations of optimizations used in our evaluation of memory requirements in order to quantify their impact on performance.

Table 6.3 presents the performance results for our shadow memory optimizations. The baseline implementation (i.e. no optimizations) led to an average slowdown of 193×. The combination of SlimTV, BulkTV, and garbage collection resulted in an average slowdown of 198×, an increase of only 2.6% over the baseline implementation. While this increase is small, we would expect even better performance with a slightly more optimized version of these techniques. For example, our implementation of garbage collection pauses program execution while shadow memory is cleaned. We would expect a multi-threaded implementation of garbage collection to significantly improve performance.

Table 6.3: **Performance Impact of Optimized Shadow Memory.** Despite being optimized for greatly reduced memory requirements, Kremlin's optimized shadow memory infrastructure leads to roughly equal performance on average when compression is not used. Compression adds an average of 20% overhead in addition to the other optimizations. This overhead is a small price to pay for the ability to run in memory constrained systems or with larger input sizes.

Benchmark		Native Runtime (sec)	Slowdown			
Suite	Name		Baseline	All w/o Compr.	All Opts	From Compr.
SpecInt	bzip2	57.1	211	205	211	1.03
	gzip	41.0	179	168	170	1.01
	mcf	90.5	85	120	231	1.92
	vpr	72.5	144	129	131	1.02
SpecFP	art	6.5	178	187	202	1.08
	equake	114	204	189	227	1.20
	mesa	120	173	188	188	1.00
NasPB	cg	6.4	175	186	213	1.15
	ft	11.4	225	203	221	1.09
	is	2.0	80	119	148	1.24
	lu	82.9	220	279	401	1.44
	mg	5.6	447	403	475	1.18
	max	90.5	447	403	474	1.92
	avg	50.8	193	198	235	1.20

Compression causes the average slowdown to increase to an average of 235×, an increase of 20% compared to the optimizations without compression. This increase is most notable in mcf, a program notorious for having poor memory locality. This poor locality leads to an increase in the number of active time tables and therefore a greater number of compressions and decompressions that need to be performed. Surprisingly, the reduction in memory expansion from using compression on mcf (11%) is poor in comparison to its increased runtime overhead (92%). This appears to be a degenerate case as the average increase in compression's runtime overhead (20%) compares much more favorably to the average reduction in memory expansion (14.9%). Similarly to the other optimizations, compression's runtime overhead could be reduced via some simple techniques.

Table 6.4: **Speedup From Static Partial Evaluation of CPA.** Combining static partial evaluation of CPA with our novel shadow memory architecture resulted in an average of 1.17× and 1.16× speedup (without and with compression enabled) compared to using only the novel shadow memory. The speedup from this technique is large enough to offset the slowdown from using compression. lu offered the largest speedup, largely because its source code contained many lines that resulted in a large number of LLVM IR temporary values. Even at its worst (in gzip), static partial evaluation did not result in appreciable slowdown.

Benchmark		Speedup	
Suite	Name	All w/o Compr.	All Opts
SpecInt	bzip2	1.06	1.06
	gzip	1.01	0.99
	mcf	1.07	1.07
	vpr	1.14	1.13
SpecFP	art	1.02	1.05
	equake	1.13	1.21
	mesa	1.15	1.19
NasPB	cg	1.00	1.15
	ft	1.15	1.13
	is	1.27	1.19
	lu	1.78	1.56
	mg	1.21	1.20
	avg	1.17	1.16

6.3.2 Static Partial Evaluation of CPA

Our evaluation of static partial evaluation of CPA focused on the runtime overhead when using this technique in addition to our novel shadow memory architecture. This technique should have minimal impact on memory overhead because it does not affect the size of the shadow memory; loads and stores are considered to be unresolvable statically and therefore none are removed or added during optimization.

Table 6.4 shows the speedup from applying static partial evaluation of CPA on top of our novel shadow memory architecture, both with and without compression enabled. The average speedup without compression and with compression was

1.17× and 1.16×, respectively. This slight difference is most likely due to the increased percentage of time spent in shadow memory operations compared to other operations: as previously mentioned, the number of loads and stores remains the same because their values cannot be determined statically. The speedup obtained when using compression is enough to offset the slowdown from compression in our novel shadow memory architecture.

Static partial evaluation of CPA resulted in a range of speedups across the benchmarks. At its worst, this technique did not result in an appreciable slowdown (gzip displayed a negligible 1% slowdown with compression). This confirms our intuition that the decrease in memory bandwidth is at least large enough to offset any increase in time from redundant computations. At its best, static partial evaluation resulted in significant speedup: lu was 78% faster without compression and 56% faster with compression. lu is an almost ideal candidate for this technique; its source code revealed many lines of code were of the form $X = a_1$ op a_2 op a_3 .. op a_n. These lines of code become $n - 1$ binary operator instructions in LLVM's intermediate representation. Static evaluation of CPA ensures that the $n - 2$ temporary values are not stored, and because they are not used elsewhere there are no redundant computations to offer slowdown. The gzip had few such lines of code, and therefore did not result in speedup when this technique was applied.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under CAREER Award 0846152, by NSF Awards 0725357, 0846152, and 1018850, and by a gift from Advanced Micro Devices.

Chapter 7

Related Work

7.1 Parallelism Discovery

Approaches for parallelism-related profiling have generally fallen into two categories: critical path analysis (CPA) and dependence testing. Critical path analysis dates back several decades, with early important works including [Kum88, AS92]. These approaches measured the number of concurrent operations at each time step along the critical path of the program. More recent work includes application of CPA to Java [HSHZ09] as well as a modified CPA for the purpose of function-level parallelism in Java programs [RVVYS10]. Unlike these approaches, Kremlin's hierarchical critical path analysis is able to localize parallelism within nested program regions, and provide concrete guidance on which program regions to parallelize.

Allen et al. [ABC+88] performed static analysis of Fortran programs in an attempt to automatically identify the correct granularity of parallelism for a target architecture. Kremlin is also able to identify the proper granularity of parallelism through the use of self-parallelism and planning personalities. However, the work in [ABC+88] was limited to structured, Fortran code; Kremlin is able to work with unstructured code that contains pointers which cannot be analyzed statically. Furthermore, Kremlin focuses on enabling the user to parallelize complicated code with which automatic parallelizing compilers have traditionally struggled.

Kulkarni et al. [KBI+09] used a critical path based analysis to bring insight

into the parallelism inherent in the execution of irregular algorithms. In contrast to Kremlin's focus on localizing parallelism to concrete code regions via HCPA, Kulkarni's approach attempts to transcend the details of the implementation and to quantify the amount of latent parallelism in irregular programs that exhibit amorphous data parallelism.

Cilkview [HLL10] is a recent tool that takes an already-parallelized Cilk++ program and estimates how that program's performance will change as the number of cores is increased. Similar to Kremlin, Cilkview leverages runtime information, and analyzes runtime dependencies in the program. However, Cilkview examines dependencies between pre-parallelized threads in a work-queuing runtime system rather than between instructions.

Another approach to parallelism-related profiling has been to use dependence testing to uncover the dependencies between different regions in the program. `pp` [Lar93] is an early important work that proposed hierarchical dependence testing to estimate the parallelism in loop nests. Notable recent works include Alchemist [ZNJ09] and SD3 [KKL10], which reduces runtime and memory overhead of dependence testing through the use of parallelization and compression. Although dependence testing and Kremlin's HCPA share similar goals, Kremlin focuses on localizing and quantifying parallelism across many different, nested program regions rather than establishing independence of pre-existing regions. As a result, it can identify more nuanced forms of parallelism even though significant transformation is required to expose it. Dependence testing is generally more pessimistic and sensitive to existing program structure.

A number of works have used dependence testing to determine the probability that specific dependencies will occur [WKC08, vPBC08]. DProf [WKC08] uses a compiler to identify may dependencies and then determine the probability that these dependencies will occur. von Praun et al. [vPBC08] introduced the *dependence density* metric to describe the probability that two random tasks would have a dependency. Both of these approaches target optimistic concurrency such as TLS or transactional memory.

The main difference between Kremlin's parallelism discovery and depen-

dence testing frameworks is in the stage of parallelization (Figure 1.1) that profiling targets. Kremlin's parallelism discovery is meant to quantify the parallelism in a fashion that is not as strongly tied to the program's current structure, exposing hidden sources of parallelism. In contrast, dependence testing-based approaches are more aligned with the enabling transforms stage of parallelization as they enable identifying specific changes that need to be made to enable parallelism. In the absence of discovery and planning tools, [ZNJ09] orders regions by total execution time. An interesting possibility would be to augment [ZNJ09]'s approach with the improved analysis provided by Kremlin. ParaScope [KMT91] used static analysis to expose difficult-to-analyze dependencies to the user so that they could circumvent them via refactoring.

HCPA initially used a compression scheme that resembled whole-program path compression schemes [ZG01]. We achieved much higher compression levels because we did not need to store information about the relative ordering of child subregions. We have replaced this compression with one that better handles multiple calling contexts and has a much higher compression ratio on similar-but-not-identical regions.

7.2 Parallelism Planning

The task of parallelism planning has been mostly overlooked in the context of manual parallelization. Outside of manual parallelization, automatic parallelizing compilers such as SUIF [HAA+96] and Polaris [BDE+02] implicitly perform planning. Because these tools do not target user-assisted parallelization, their planning phases focus on finding thresholds for profitable exploitation.

Speculative parallelization systems [SBV95, RP95] have created new opportunities for compilers to exploit parallelism even in the face of difficult-to-analyze code, or infrequent dependencies that result in overly conservative execution. These systems typically have a memory speculation system, often in special hardware but sometimes in software, which removes the burden of proving the correctness of potential parallel transformations, allowing the compiler to

focus on selecting the transformations that maximize performance. TLS compilers [CO03, DLL+04, LTC+06, RP95, ZMLM08, TFNG08] and often use dynamic critical path or dependence testing analyses in order to establish regions which are likely to be profitable for TLS-style execution. Kremlin's HCPA can be used in a complementary manner by providing a way to guide programmers in restructuring their code to improve parallelism for execution on TLS. This can enable an even larger class of transformations than these systems natively support.

Recent work by Tournavitis et al. [TWFO09] provides a semi-automated approach to parallelization. This approach automates parallelism discovery using a form of dependence testing and uses machine learning to pick a set of regions to be parallelized. The selected regions are automatically annotated with OpenMP pragma statements. The presumption is that the user will verify the correctness of the parallelization. While this approach has promise, it is limited by the compiler's ability to perform the *Enabling Transforms* phase of parallelization. In contrast, Kremlin has a more optimistic view of parallelism and is able to report regions with parallelism even if the compiler is not up to the task of exploiting it automatically.

Systems such as SUIF Explorer [LDB+99] and CAPO-Paraver [JJLG03] share Kremlin's focus on empowering the user during parallelization. SUIF Explorer's novelty focuses around its use of static interprocedural program analysis including pointer analysis and slicing; its use of dynamic analysis is very briefly described but appears to detect the absence or presence of memory dependencies within loops, and to provide time profiles for regions. CAPO-Paraver extends the CAPO parallelizing compiler to allow it to insert instrumentation that helps the user understand the load balancing properties of parallelized code.

7.3 Performance Prediction

CilkView [HLL10], Parallel Prophet [KKKB12], and Intel Parallel Advisor's Suitability Tool are recent tools whose motivation is similar to Kremlin. Like Kremlin, they also predict parallel performance on a target with arbitrary number of cores. Unlike Kremlin, however, these tools rely on the user's parallelized code—

or annotations—to predict speedup. Kremlin minimizes user's efforts in prediction by automatically detecting parallelism in the serial program.

Simulation has been used to predict the performance of processors and systems that are still in development. In this case, a parallel version of the program exists, but the machine itself is not available to run it. ManySim [ZIM+07] is one such simulator that was designed to evaluate the performance potential and scalability of large-scale multi-core processors. GEMS [MSB+05] is a full-system functional simulator for multiprocessors. It separates the simulation from the timing models, allowing them build a detailed memory system timing simulator rather than focus on basic functional simulation. However, simulators still require code that has been parallelized for these systems, unlike Kremlin.

A number of works have looked at the limits of parallelism and their impact on performance. Theobald et al [TGH92] examined the "smoothability" of a program's parallelism, i.e. the ability to which a program's parallelism could be equally spread throughout the program's entire execution to ensure high utilization on a constrained multiprocessor. Rauchwerger et al [RDN93] also looked at the ability to map ideal parallelism to a constrained processor, introducing the concept of *slack* to describe the ability of parallelism to be pushed to later parts of the program. Kremlin improves upon these works by using HCPA's ability to localize parallelism; Kremlin can examine the effect of parallelizing specific regions of the program in order to gain a better estimate of the program's parallel performance.

There have been several efforts to predict serial performance [OH00, Loh01, HPE+06, KS04]. In theory, these predictions could be combined with Kremlin's speedup predictions to predict the parallel execution time of a program.

Several works have looked at predicting the scalability of parallel programs based on their performance on a small number of processors [BRL+08, ZCZ10]. Barnes et al [BRL+08] looked at several techniques for extrapolating performance of MPI programs, including one that measured the global critical path. Zhai et al [ZCZ10] avoid performance extrapolation to predict performance; instead, they use deterministic replay to measure sequential time of each process using only a single node. Again, these systems differ from Kremlin in that they predict performance

based on an existing parallel implementation.

Hill and Marty [HM08] recently proposed a simple performance analytical model, extending Amdahl's law. Their model assumes future processors include different types of cores and each program region can choose the more appropriate core based on its workload. Chung and Mai [CMHM10] further improved Hill and Marty's model with heterogeneous chip including ASIC, FPGA, and GPU. Although we kept Kremlin's analytical model relatively simple, Kremlin can easily incorporate these sophisticated models if needed.

7.4 Shadow Memory Design

Wide applicability of shadow memory has led to a wide range of shadow memory architectures. Some of these approaches use only a single-level implementation [SBN+97, CZYH06, BFW03], relying on assumptions about the size of address space (e.g. 32-bit addresses) and often allocating half of the address space for shadow memory. This single-level approach is not robust: it often fails in the face of programs that make assumptions about memory placement and often clash with operating systems which have assumptions about object locations. MemCheck [SN05], pinSEL [NPP+06], and an array of tools [New05, MW07, NM03] built using Valgrind [NS07] use a two-level translation table similar to the one shown in Figure 4.3. This approach works well for 32-bit address spaces but does not scale well to 64-bit spaces. Recent work has expanded this basic structure to three-levels to better support 64-bit address spaces [ZBA10a, ZBA10b].

While Kremlin's shadow backing store uses a three-level address translation organization similar to that of Umbra [ZBA10a], Kremlin's overall architecture is optimized to meet the needs of region-based analysis and vectored shadow memory operations. Kremlin introduces novel shadow memory features such as a shadow memory cache, level tables, garbage collection, and tag compression. These additions are unnecessary in traditional memory shadowing applications but are critical in meeting the exacting demands of region-based analysis.

A number of tools propose specialized hardware to reduce the overhead

of memory shadowing [VRSP07, ZTZ07, NG09]. These proposals are effective at reducing overhead but they add inflexibility to the shadow memory infrastructure. These tools overwhelmingly focus on a single application of memory shadowing and are therefore not general frameworks. Furthermore, they are also targeted towards traditional shadow memory.

7.5 Parallel Performance Debugging Tools

Several systems have been developed in order to help debug the performance of pre-Kremlin- parallel programs [DRR99, AMCA+95]. SvPablo provided an integrated viewing and instrumentation environment that allowed performance debugging of MPI programs. Adve et al [AMCA+95] performed similar analysis on data parallel FORTRAN. Paradyn [MCC+95] automatically searches for performance problems in long running programs by dynamically instrumenting the program. Martonosi et al [MFH96] were able to examine the performance of the cache system with very little overhead by integrating performance monitoring into existing cache-coherence mechanisms. These systems could be used in concert with Kremlin to help determine why actual performance does not match the predicted bound on program performance. SUIF Explorer [LDB+99] uses static and dynamic analyses to understand parallel-execution related properties, much like Kremlin; however, Kremlin does not require user interaction, and uses a simplify hardware specifications to give reasonable speedup predictions of post-parallelized code.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under CAREER Award 0846152, by NSF Awards 0725357, 0846152, and 1018850, and by a gift from Advanced Micro Devices.

This chapter contain materials from “Kremlin: Rethinking and Rebooting gprof for the Multicore Age”, by Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor, which appears in *PLDI ’11: Proceedings of the 32nd*

ACM SIGPLAN conference on Programming language design and implementation. The dissertation author was the primary investigator and author of this paper.

This chapter contains material from "Kismet: parallel speedup estimates for serial programs", by Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor, which appears in *OOPSLA '11: Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications.* The dissertation author was the secondary investigator and author of this paper.

Chapter 8

Summary

The switch from single- to multi-core processors has fundamentally changed the way software engineers will achieve scalable performance. We began this dissertation by discussing how the scalable performance on multi-core processors requires software to exploit the parallelism available inside of a program. We discussed how fully automated approaches to parallelization are fundamentally limited and result in performance that often pales in comparison to that of a manually parallelized implementation. This fundamental limitation has led to the creation of programmer-centric tools to ease certain parts of the parallelization process.

We examined a taxonomy of parallelization tools that underscored how the initial stages of parallelization, parallelism discovery and parallelism planning, currently lack practical tools. This gap in the parallelization toolchain forces programmers to consult a number of oracles to answer one of the primary questions of parallelization, "*What parts of this program should I spend time parallelizing?*". The oracles are impractical for a number of reasons, giving rise to the need for a practical oracle for parallelization. Throughout the rest of the dissertation, we discuss the design and implementation of Kremlin, one such practical oracle.

In Chapter 2, we examined the parallelization methodology currently employed by many programmers. This methodology is highly efficient because it relies on the impractical oracles first introduced in Chapter 1. In this chapter we describe a user study we performed with an early prototype of Kremlin. The results of the user study underscored the importance of a practical oracle to guide

users through parallelism discovery and planning. In this study, users without a practical oracle spent roughly 50% of their time working on performance critical regions while users with access to the early Kremlin oracle spent roughly 85% of their time in these critical regions. Our results from this study also guided the further development of Kremlin as a practical oracle, having focused our attention on automated tools for quantifying the type and amount of parallelism within specific program regions.

Existing parallelism discovery tools rely on one of the following two general techniques: critical path analysis and dependence testing. Unfortunately, neither of these two techniques are suited as precursors for parallelism planning. In Chapter 4, we examined creating a planning-aware parallel discovery tool. Kremlin builds upon the critical path analysis, but extends to make it more realistic its results more practical for us in parallelism planning. In this chapter, we looked at two of our core contributions: hierarchical critical path analysis (HCPA) and the self-parallelism metric. HCPA provides localized parallelism info, which enables iterative planning by allowing evaluation of only partially parallelized programs. The results in this chapter showed that our self-parallelism metric more accurately classifies the amount of parallelism in each region, e.g. leading to 6× more regions being identified as serial as compared to the total-parallelism metric from the overly optimistic CPA. Our results also show that self-parallelism aligns much more closely with the parallelizability of a region, with a nearly 2× increase in the likelihood that regions classified as being highly parallel are parallelized.

In Chapter 5 we examined parallelism planning. We described how Kremlin is able to model the parallel execution time of a program and to use self-parallelism and the program structure to infer the type of parallelism in each region of the program. We looked at three separate planning personalities (OpenMP, OpenCL, and Cilk++), which tailor planning results to a specific target system. Results show that Kremlin's planning ability is able to reduce the number of regions that need to be parallelized by 1.57×. This reduction does not come at the cost of poor performance: in two cases Kremlin's plan led to greatly improved performance while on the remaining cases, Kremlin's succinct plan led to speedup was within

4% of the performance for an optimized, expert, third-party implementation. Our results in this chapter also showed that Kremlin is able to prioritize regions well, with an average of 86.4% of performance coming after implementing only the first half of the plan and only 4.4% improvement coming from the final quartile of recommendations.

Kremlin's effectiveness as a parallelization oracle relies in large part on a HCPA, a heavyweight dynamic analysis. Naively implemented, HCPA is impractical because of extreme memory and/or runtime overheads. In Chapter 6, we looked at two optimizations to reduce the overhead associated with HCPA. The first technique, a novel shadow memory architecture, provides not only fast access to shadow memory for most accesses but also space-efficient storage for infrequently used memory. The second technique, partial static evaluation of critical paths, offloads part of the runtime calculation of critical path lengths to compile time through the use of static analysis. Our results in this chapter show that the first technique reduces the average memory requirement of HCPA from by an average of 22.7× (8.756GB to 386MB) at a cost of only 21.8% slowdown; when performance is more critical, compression can be turned off and the slowdown reduces to 2.6% while memory reduction reduces to 13× (average: 673MB). This reduction makes Kremlin practical even for lower-end systems such as laptops. Results also show that enabling second technique improves performance by an average of 1.17× without compression enabled, and 1.16× with compression.

Overall, we have shown that Kremlin is indeed a practical oracle for the parallelization of sequential code. Kremlin provides a simple-to-use tool that allows programmers to understand the parts of the program on which they should focus their parallelization efforts. Kremlin's HCPA implementation allows efficient discovery of parallelism throughout the program, providing the basis for efficient parallelism planning. Kremlin's results can be customized to the system environment to allow highly relevant results, no matter what the target is. We plan on releasing Kremlin as an open source tool so that both researchers and practitioners can benefit from its capabilities.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under CAREER Award 0846152, by NSF Awards 0725357, 0846152, and 1018850, and by a gift from Advanced Micro Devices.

Bibliography

[ABC+88] F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh, and V. Sarkar. A framework for determining useful parallelism. In *ICS '88: Proceedings of the International Conference on Supercomputing*, pages 207–215. ACM, 1988.

[ABL97] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, New York, NY, USA, 1997. ACM.

[AL90] Thomas E. Anderson and Edward D. Lazowska. Quartz: a tool for tuning parallel program performance. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '90, pages 115–125. ACM, 1990.

[AMCA+95] V.S. Adve, J. Mellor-Crummey, M. Anderson, J-C. Wang, D. A. Reed, and K. Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *SC '95: Proceedings of the ACM/IEEE conference on Supercomputing*, 1995.

[AS92] Todd Austin and Gurindar S. Sohi. Dynamic dependency analysis of ordinary programs. In *ISCA '92: Proceedings of the International Symposium on Computer Architecture*, pages 342–351, 1992.

[BBB+91] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The nas parallel benchmarks summary and preliminary results. In *Supercomputing, 1991. Supercomputing '91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 158 –165, nov. 1991.

[BDE+02] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, W.M. Paek, Y. Pottenger, L. Rauch-

werger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, Aug 2002.

[BFW03] Michael Burrows, Stephen Freund, and Janet Wiener. Run-time type checking for binary programs. In *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 90–105. Springer Berlin / Heidelberg, 2003.

[BO01] J. Mark Bull and Darragh O'Neill. A microbenchmark suite for OpenMP 2.0. *SIGARCH Computer Architecture News*, 29:41–48, Dec 2001.

[BRL+08] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. A regression-based approach to scalability prediction. In *ICS '08: Proceedings of the International Conference on Supercomputing*, pages 368–377, 2008.

[BZ11] D. Bruening and Qin Zhao. Practical memory checking with dr. memory. In *CGO '11: International Symposium on Code Generation and Optimization*, pages 213 –223, 2011.

[CBM+09] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44 –54, oct. 2009.

[CMHM10] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *MICRO '10: Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 225–236, Washington, DC, USA, 2010. IEEE Computer Society.

[CO03] Michael K. Chen and Kunle Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *ISCA '03: Proceedings of the International Symposium on Computer Architecture*, pages 434–446. ACM, 2003.

[CZYH06] W. Cheng, Qin Zhao, Bei Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Computers and Communications, 2006. ISCC '06. Proceedings. 11th IEEE Symposium on*, pages 749 – 754, june 2006.

[DLL+04] Zhao H. Du, Chu C. Lim, Xiao F. Li, Chen Yang, Qingyu Zhao, and Tin F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *PLDI '04: Proceedings of the Conference on Programming Language Design and Implementation*, pages 71–81. ACM, 2004.

[DM98] L. Dagum and R. Menon. Openmp: An industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, jan-mar 1998.

[DME09] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *ICSE '09: Proceedings of the International Conference on Software Engineering*, pages 397–407. IEEE Computer Society, 2009.

[DRR99] L.A. De Rose and D.A. Reed. SvPablo: A multi-language architecture-independent performance analysis system. In *ICPP '99:International Conference on Parallel Processing*, pages 311 –318, 1999.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126. ACM, 1982.

[HAA+96] M. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, Shih-Wei Liao, and E. Bu. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Aug 1996.

[HBZ+05] Lorin Hochstein, Victor R. Basili, Marvin V. Zelkowitz, Jeffrey K. Hollingsworth, and Jeff Carver. Combining self-reported and automatic data to improve programming effort measurement. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 356–365. ACM, September 2005.

[HLL10] Y. He, C. Leiserson, and W. Leiserson. The Cilkview Scalability Analyzer. In *SPAA '10: Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, pages 145–156, 2010.

[HM08] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41:33–38, July 2008.

[HPE+06] Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K. John, and Koen De Bosschere. Performance prediction based on inherent program similarity. In *PACT '06: Parallel Architectures and Compilation Techniques*, 2006.

[HSHZ09] C. Hammacher, K. Streit, S. Hack, and A. Zeller. Profiling java programs for parallelism. In *IWMSE '09: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, pages 49–55, 2009.

[JGLT11] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Kismet: parallel speedup estimates for serial programs. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 519–536. ACM, 2011.

[JJLG03] Gabriele Jost, Haoqiang Jin, Jesus Labarta, and Judit Gimenez. Interfacing computer aided parallelization and performance analysis. In *Proceedings of the 2003 international conference on Computational science*, ICCS'03, pages 181–190, Berlin, Heidelberg, 2003. Springer-Verlag.

[KBDZ09] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast track: A software system for speculative program optimization. In *CGO '09: Proceedings of the International Symposium on Code Generation and Optimization*, pages 157–168. IEEE Computer Society, 2009.

[KBI+09] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In *PPoPP '09: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–14, 2009.

[KKKB12] Minjang Kim, Pranith Kumar, Hyesoon Kim, and Bevin Brett. Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model. In *IPDPS '12: Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*, 2012.

[KKL10] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. SD3: A scalable approach to dynamic data-dependence profiling. *MICRO '10: Proceedings of the International Symposium on Microarchitecture*, 0:535–546, 2010.

[KLW+04] D. Kim, S.S.-W. Liao, P.H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J.P. Shen. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 27 – 38, march 2004.

[KMC72] D.J. Kuck, Y. Muraoka, and Shyh-Ching Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, Dec. 1972.

[KMT91] K. Kennedy, K. S. McKinley, and C. W. Tseng. Interactive parallel programming using the parascope editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, 1991.

[KS04] Tejas S. Karkhanis and James E. Smith. A first-order superscalar processor model. In *ISCA '04: Proceedings of the International Symposium on Computer Architecture*, pages 338–, Washington, DC, USA, 2004. IEEE Computer Society.

[KST10] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. Software data spreading: leveraging distributed caches to improve single thread performance. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 460–470, 2010.

[Kum88] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9):1088–1098, Sep 1988.

[KVAJ+09] S. Kota Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M.B. Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *IISWC '09: Proceedings of the IEEE International Symposium on Workload Characterization*, pages 55–64. IEEE Computer Society, 2009.

[LA04] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75 – 86, march 2004.

[Lar93] J. R. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Trans. Parallel Distrib. Syst.*, 4(7):812–826, 1993.

[LBF+98] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a Raw machine. In *ASPLOS '98: International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–54, Oct 1998.

[LDB+99] Shih-Wei Liao, Amer Diwan, Robert P. Bosch, Jr., Anwar Ghuloum, and Monica S. Lam. SUIF Explorer: an interactive and interprocedural parallelizer. In *PPoPP '99: Proceedings of the ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 37–48, New York, NY, USA, 1999. ACM.

[Lei09] Charles E. Leiserson. The Cilk++ concurrency platform. In *DAC '09: Proceedings of the Design Automation Conference*, pages 522–527, 2009.

[Loh01] Gabriel Loh. A time-stamping algorithm for efficient performance estimation of superscalar processors. In *SIGMETRICS*, pages 72–81, New York, NY, USA, 2001. ACM.

[LTC+06] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits porogram structure. In *PPoPP '06: Proceedings of the ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 158–167. ACM, 2006.

[MCC+95] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[MFH96] Margaret Martonosi, David Felt, and Mark Heinrich. Integrating performance monitoring and communication in parallel computers. In *SIGMETRICS*, pages 138–147, 1996.

[MSB+05] Milo Martin, Daniel Sorin, Bradford Beckmann, Michael Marty, Min Xu, Alaa R. Alameldeen, Kevin Moore, Mark Hill, and David Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33:92–99, Nov 2005.

[MW07] Arndt Mhlenfeld and Franz Wotawa. Fault detection in multithreaded c++ server applications. *Electronic Notes in Theoretical Comp Sci*, 174(9):5 – 22, 2007.

[New05] James Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS '05: Proceedings of the Network and Distributed System Security Symposium*, 2005.

[NG09] Vijay Nagarajan and Rajiv Gupta. Architectural support for shadow memory in multiprocessors. In *VEE '09: Proceedings of the International Conference on Virtual Execution Environments*, pages 1–10, New York, NY, USA, 2009. ACM.

[NM03] Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. *Electronic Notes in Theoretical Comp Sci*, 89(2):149 – 170, 2003.

[NPP+06] Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn, and Brad Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *SIGMETRICS '06*, pages 216–227, New York, NY, USA, 2006. ACM.

[NS07] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the Conference on Programming Language Design and Implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[Obe] Markus Oberhumer. LZO Data Compression Library. http://www.oberhumer.com/opensource/lzo/.

[OH00] David Ofelt and John L. Hennessy. Efficient performance prediction for modern microprocessors. In *SIGMETRICS*, pages 229–239, New York, NY, USA, 2000. ACM.

[omn] NAS Parallel Benchmarks 2.3; OpenMP C. http://www.hpcc.jp/Omni/.

[QWL+06] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[RDN93] Lawrence Rauchwerger, Pradeep K. Dubey, and Ravi Nair. Measuring limits of parallelism and characterizing its vulnerability to resource constraints. In *MICRO '93: Proceedings of the international symposium on Microarchitecture*, pages 105–117, 1993.

[RP95] L. Rauchwerger and D. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *PLDI '95: Proceedings of the Conference on Programming Language Design and Implementation*, pages 218–232, New York, NY, USA, 1995. ACM.

[RVVYS10] A. Rountev, K. Van Valkenburgh, Dacong Yan, and P. Sadayappan. Understanding parallelism-inhibiting dependences in sequential java programs. In *ICSM '10: Proceedings of the International Conference on Software Maintenance*, pages 1–9, Sept 2010.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, November 1997.

[SBV95] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the International Symposium on Computer Architecture*, pages 414–425. ACM, 1995.

[SLA+07] Ruchira Sasanka, Man L. Li, Sarita V. Adve, Yen K. Chen, and Eric Debes. Alp: Efficient support for all levels of parallelism for complex media applications. *ACM Transactions on Architecture and Code Optimization*, 4(1), 2007.

[SN05] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[SSvP07] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: concurrent programming for modern architectures. In *PPoPP '07: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 271, 2007.

[TFNG08] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO '08: Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 330–341. IEEE Computer Society, 2008.

[TGH92] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. On the limits of program parallelism and its smoothability. In *MICRO '92: Proceedings of the International Symposium on Microarchitecture*, pages 10–19. IEEE Computer Society Press, 1992.

[TMC09] Nathan R. Tallent and John M. Mellor Crummey. Effective performance measurement and analysis of multithreaded applications. In *PPoPP '09: Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 229–240, 2009.

[TT11] Hung-Wei Tseng and D.M. Tullsen. Data-triggered threads: Eliminating redundant computation. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 181 –192, feb. 2011.

[TWFO09] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F. P. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI '09: Proceedings of the ACM SIGPLAN Conference on Programming Language Design And Implementation*, pages 177–187, 2009.

[vPBC08] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP '08: Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 185–196, 2008.

[VRSP07] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. *High-Performance Computer Architecture, International Symposium on*, 0:273–284, 2007.

[WKC08] Peng Wu, Arun Kejariwal, and Călin Caşcaval. Compiler-driven dependence profiling to guide program parallelization. In *LCPC '08: Languages and Compilers for Parallel Computing*, pages 232–248, 2008.

[WST09] J. Wloka, M. Sridharan, and F. Tip. Refactoring for reentrancy. In *FSE '09: Proceedings of the ACM Symposium on the Foundations of Software Engineering*, pages 173–182. ACM, 2009.

[XBS06] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[XZ07] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *ISSTA '07: Proceedings of the International Symposium on Software Testing and Analysis*, pages 185–195, 2007.

[ZBA10a] Q. Zhao, D. Bruening, and S. Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *CGO '10: Proceedings of the IEEE/ACM international symposium on Code Generation and Optimization*, pages 22–31, 2010.

[ZBA10b] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Efficient memory shadowing for 64-bit architectures. In *ISMM '10: Proceedings of the International Symposium on Memory Management*, Toronto, Canada, Jun 2010.

[ZCZ10] Jidong Zhai, Wenguang Chen, and Weimin Zheng. Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. In *PPoPP '10: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 305–314, 2010.

[ZG01] Youtao Zhang and Rajiv Gupta. Timestamped whole program path representation and its applications. In *PLDI '01: Proceedings of the*

ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 180–190, 2001.

[ZIM+07] Li Zhao, R. Iyer, J. Moses, R. lllikkal, S. Makineni, and D. Newell. Exploring Large-Scale CMP Architectures Using ManySim. *IEEE Micro*, 27(4):21 –33, July 2007.

[ZMLM08] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *HPCA '08: Proceedings of the International Symposium on High Performance Computer Architecture*, 2008.

[ZNJ09] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *CGO '09: Proceedings of the International Symposium on Code Generation and Optimization*, pages 47–58. IEEE Computer Society, 2009.

[ZTZ07] Pin Zhou, R. Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *HPCA '07: International Symposium on High Performance Computer Architecture*, pages 121 –132, 2007.