UNIVERSITY OF CALIFORNIA, SAN DIEGO

Parallel Speedup Estimates for Serial Programs

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

in

Computer Science (Computer Engineering)

by

Donghwan Jeon

Committee in charge:

Professor Michael Taylor, Chair
Professor Chung-Kuan Cheng
Professor Sorin Lerner
Professor Bill Lin
Professor Steven Swanson
Professor Dean Tullsen

2012

The dissertation of Donghwan Jeon 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 mother Misook Chung,
for her endless love and devotion.

EPIGRAPH

There are only two mistakes one can make along the road to truth;
not going all the way, and not starting.
—Buddha

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help and support of the kind people around me. First and foremost, I would like to thank my incredible advisor, Professor Michael Taylor. He provided me the opportunity to work on exciting research projects, taught me how to be a better hacker, and encourage me to think big. I was extremely privileged to learn from his keen insight, endless energy, and thoughtful consideration over many years.

I am indebted to my amazing colleagues on the Kismet project with whom I collaborated. In particular, I would like to thank Saturnino Garcia. As a trustful friend and a colleague, he was always willing to help and give his best suggestions. Conversation with him allowed me to understand a problem more deeply and create a better solution. I also thank Chris Louie for his contributions in the implementation of HCPA. Former and current research group members – Anshuman Gupta, Fei Jia, Sravanthi Kota Vankata – all provided invaluable feedback to my work.

This thesis would not have been possible without my family members' support and encouragement. My parents, Youngsik Jeon and Misook Chung, devoted themselves to my education and supported me with their unconditional love. I would like to express my utmost gratitude to them. I thank my proud brother, Jonghwan Jeon, for being my lifelong friend and supporter. I also thank my cousin Chihwan Jeon, his wife Yoonae Cho, and my cute nieces – Jiye and Yoonseo. They enriched my life in San Diego with their love, smile, and delicious Korean food.

Finally, I must heartly thank my lovely wife Yooeun. Without her love, support, and patience, I would not have made it this far. She has been the great source of strength and happiness when I felt hopeless. Thank you for being with me!

Chapter 2, 3, and 5 contain 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 primary investigator and

Chapter 2 contains 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 secondary investigator and author of this paper.

Chapter 2 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 secondary investigator and author of this paper.

VITA

2001	B. S. in Computer Science and Engineering Seoul National University Seoul, Korea
2002-2005	Software Engineer MDS Technology Seoul, Korea
2005-20012	Graduate Research Assistant University of California, San Diego
2008	M. S. in Computer Science (Computer Engineering) University of California, San Diego
2012	Ph. D. in Computer Science (Computer Engineering) University of California, San Diego

PUBLICATIONS

Saturnino Garcia, Donghwan Jeon, Christopher Louie, Michael Bedford Taylor, "The Kremlin Oracle for Sequential Code Parallelization", *IEEE Micro*, *July/August 2012*.

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

Donghwan Jeon, Saturnino Garcia, Christopher Louie, Michael Bedford Taylor, "Kremlin: Like gprof, but for Parallelization", *Principles and Practice of Parallel Programming (PPoPP)*, Feb 2011.

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

Srivanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher 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.

ABSTRACT OF THE DISSERTATION

Parallel Speedup Estimates for Serial Programs

by

Donghwan Jeon

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California, San Diego, 2012

Professor Michael Taylor, Chair

Software engineers now face the difficult task of parallelizing serial programs for parallel execution on multicore processors. Parallelization is a complex task that typically requires considerable time and effort. However, even after extensive engineering efforts, the resulting speedup is often fundamentally limited due to the lack of parallelism in the target program or the inability of the target platform to exploit existing parallelism. Unfortunately, little guidance is available as to how much benefit may come from parallelization, making it hard for a programmer to answer this critical question: *"Should I parallelize my code?"*.

In this dissertation, we examine the design and implementation of Kismet, a tool that creates parallel speedup estimates for unparallelized serial programs. Our approach differs from previous approaches in that it does not require any changes or manual analysis of the serial program. This difference allows quick profitability analysis of a program, helping programmers make informed decisions in the initial stages of parallelization.

To provide parallel speedup estimates from serial programs, we developed a dynamic program analysis named hierarchical critical path analysis (HCPA).

HCPA extends a classic technique called critical path analysis to quantify localized parallelism of each region in a program. Based on the parallelism information from HCPA, Kismet incorporates key parallelization constraints that can significantly affect the parallel speedup, providing realistic speedup upperbounds. The results are compelling. Kismet can significantly improve the accuracy of parallel speedup estimates relative to prior work based on critical path analysis.

Chapter 1

Introduction

Software engineers currently face the daunting task of parallelizing their programs to take advantage of multi-core processors. These multi-core processors provide extensive parallel resources, providing the potential for greatly improved performance. However, a parallelized software is required to exploit the capabilities of these multi-core processors. This is a radical change for software engineers. Until recently, most microprocessors had only a single core. Thanks to increasing operation frequency and micro-architectural innovations fueled by new CMOS process technologies, these processors enabled exponential performance growth without any changes in the software. However, since 2005 the power wall and increasing on-chip wire delay have fundamentally changed the way processors are designed, bringing the task of parallelization to software engineers.

Unfortunately, parallelization typically relies on individual engineer's manual effort rather than automated tools. An automatic parallelizing compiler might be the ideal solution in parallelization: it analyzes the serial source code, finds parallelization opportunities, applies code transformations, and finally emits the parallel executable of the program. In reality, however, even the state of the art parallelizing compilers miss many parallelization opportunities. Because compilers have to guarantee the correctness of their output, they do not parallelize code when they cannot prove correctness, missing potential parallelization opportunities. Furthermore, many programming languages do not explicitly express the parallelism in code, which makes it even harder for an automated tool to discover and exploit

parallelism.

Manual parallelization typically requires extensive time and effort. In the first place, thinking in parallel and writing a parallel program is hard for a human. Parallel programs are also harder to test, modify, debug, and maintain due to concurrency issues such as race conditions and deadlock. Furthermore, unlike the well-established serial programming environment, the parallel programming environment is still evolving, requiring additional learning and training for programmers.

Even after extensive parallelization efforts, the resulting performance of refactored serial programs often falls short of optimistic speedups derived from the available core count. Worse-than-expected performance can be caused by several factors. First, the program may have an inherently low amount of parallelism—possibly the result of choosing an algorithm without considering its parallelizability. Second, the target system may be a poor choice for that program—the result of a mismatch between the structure of the parallelism in the program and the ability of the system to efficiently exploit it. Finally, the implementation may be poor—the result of missed parallelism opportunities or poorly executed parallelization attempts.

With the expected serious investment in engineering efforts and widely varying parallel performance, parallelization raises many risks in software engineering. Unfortunately, very little tool support is available to help parallel softwares, especially in the early stages of the parallelization. For example, if a programmer decides to parallelize a program with very limited parallelism, the programmer is likely to waste precious development time and see disappointing results. If the programmer had known that the program is parallelism-limited, their time could have been better used on serial optimization or substituting the algorithm with another one that has more parallelism.

In this thesis, we propose the design and implementation of a parallel speedup estimation tool that mitigates the risk of parallel software engineering. The tool answers the question "*Should I parallelize my code?*", allowing software developers to understand the potential benefit associated with the cost of migrating

an existing serial implementation into a parallel one. Unlike other parallel performance tuning tools that require an already parallelized program, the tool works on unmodified serial source code, helping the user in the initial stages of parallelization. Furthermore, by incorporating real-world parallelization constraints, the tool can provide parallel speedup estimates that are accurate enough for practical use.

This thesis makes the following key contributions:

- It presents Kismet, a tool that provides parallel speedup estimates from serial source code and target-specific parallelization constraints. Because Kismet automatically provides parallel speedup estimates from unmodified serial code, it requires little user efforts compared to other tools that demand pre-parallelization or user annotations [HLL10, Int].

- It introduces the use of summarization techniques on hierarchical critical path analysis (HCPA) [GJLT11], a recently proposed dynamic program analysis employed in Kismet that measures the parallelism of a program. The use of summarization techniques significantly improves the applicability of HCPA over the previous implementation of HCPA that relied on a compression technique.

- It demonstrates the effectiveness of Kismet with a wide range of benchmarks on two very different platforms: MIT Raw and AMD multi-core. With parallelism profile from HCPA and a brief description of target-specific parallelization constraints, Kismet was able to provide parallel speedup estimates close enough to guide initial stages of parallelization.

- It presents the design and implementation of Skadu, a vector shadow memory system that dramatically reduces memory and runtime overhead of hierarchical memory analysis. The effectiveness of Skadu is shown with HCPA and memory footprint analysis.

1.1 Existing Tools for Parallelization

Several tools are available to improve the productivity of parallelization. In this section, we overview existing tools that can help parallelization and discuss their merits and limitations.

Parallelizing Compiler A parallelizing compiler is the ideal solution for parallelization for a software engineer's convenience. Taking serial source code, it discovers parallelization opportunities, applies required code transformations, and finally emits the parallel executable. Unfortunately, the resulting performance is often disappointing due to missing parallelization opportunities, as shown in Kim et al. [KKL10b] and Tournavitis et al. [TWFO09]. Those missing opportunities stem from an automatic parallelizing compiler's limitations in static pointer analysis, irregular control flow, and program input dependence.

Serial Profiler Although performance profilers such as gprof [GKM82] are developed for serial program optimizations rather than parallelization, they provide useful hotspot information. By focusing on only hotspots, a programmer can make the parallelization process more productive. Unfortunately, these profilers do not quantify the parallelism of the target program and a programmer must manually estimate the profitability of parallelization for each program region, which is time-consuming and error-prone.

Critical Path Analysis Based Tools Critical path analysis (CPA) [Kum88] is a classical program analysis that quantifies the theoretical speedup upperbound. CPA analyzes dependences in data-flow style execution and reports the speedup on an ideal target platform where unlimited number of cores are available and parallelization overhead does not exist. CPA's strength lies in the quantification of parallelism regardless of the serial implementation of the program. For example, reordering two independent statements in a program does not change the result from CPA. Unfortunately, the reported number is typically overly optimistic and CPA has seen only limited use, primarily in research projects [Kum88, KMC72,

AS92, KBI+09].

Dependence Testing Based Tools Dependence testing shares similar goals of discovering parallelism with critical path analysis [Lar93, ZNJ09, KKL10b, TWFO09]. Typically a dependence testing tool monitors inter-iteration dependences at runtime and report existing dependences in the target program. Based on the work and detected dependence patterns, the tool can provide a short list of promising regions for parallelization. However, dependence testing tools have two major limitations. First, they have difficulties detecting parallelization opportunities if the serial implementation does not the program structure that the tool supports such as DOALL loops. Second, they do not quantify the amount of parallelism, making it hard to reason about the profit from parallelization.

Parallel Speedup Predictor Existing speedup predictors such as Intel Cilkview analyzer and Intel parallel advisor [HLL10, Int, KKKB12] are designed to help quick exploration of parallelization. The programmer provides annotated source code that expresses parallelizable code regions as well as parallelization strategies. After profiling the target program with the given annotations, these tools provide the estimated speedup after parallelization. Since making annotations tend to be easier than applying fully working code transformations, they can improve productivity, but still they require the programmer have a deep understanding about the target program. For example, if the software engineer working on parallelization is different from the original code writer, which is often the case, these tools do not help much until the user has enough understanding about the program to write reasonable annotations. Another interesting tool is the Intel Cilkview Scalability Analyzer [HLL10]. Unlike annotation-based tools, Cilkview accepts fully parallelized program and provides expected speedup, guiding the performance tuning of parallelized programs. However, it does not help much during the initial stages of parallelization.

```
$> make CC=kismet-cc
$> $(PROGRAM) $(INPUT)
$> kismet –openmp
Cores          1   2   4    8    16   32   64
Speedup        1   2   3.8  3.8  3.8  3.8  3.8
(est.)
```

Figure 1.1: **Kismet's User Interface.** Kismet provides an easy-to-use user interface similar to `gprof`. After compiling and executing the unmodified serial program, Kismet combines runtime profile information with target environment, producing estimated upper bounds on parallel speedups for the program.

1.2 Introducing Kismet

In this section, we introduce Kismet, a parallel speedup estimation tool. Unlike other speedup estimation tools that require already parallelized code, Kismet requires only on an unmodified, serial version of a program. Kismet first performs dynamic program analysis on the serial program to determine the work and the amount of parallelism available in each region (e.g. loop and function) of the program. Kismet localizes the parallelism of a region with a new metric called *self-parallelism*. Kismet then incorporates system constraints to calculate an approximate upper bound on the program's attainable parallel speedup. These constraints include the number of cores, synchronization overhead, cache effects, and *expressible parallelism* types (e.g. loop and task parallelism for multicore chips; instruction level parallelism for VLIW-style chips; and data level parallelism for vector machines).

Usage Model Kismet provides a simple usage model in the style of `gprof`, as shown in Figure 1.1. The program is first compiled with a drop-in compiler replacement called `kismet-cc`. `kismet-cc` inserts instrumentation code and produces an instrumented binary. Then the program runs with a representative input, which produces as a side-effect an output file containing profile information.

The user then runs `kismet`, which analyzes the output file and generates a table of estimated speedup upper-bounds for a spectrum of core counts. Because

the parallel target platform can significantly affect the achievable speedup, the user specifies the target platform (e.g. openmp in Figure 1.1) when running Kismet. Kismet evaluates a number of parallelization schemes for the target platform and then reports the highest speedup found on varying number of cores. If needed, Kismet can produce the list of regions that should be parallelized to achieve the reported speedup.

Representative Use Cases Since a user can easily run Kismet without much effort, it can be used throughout the parallelization process. Here we present three representative use cases of Kismet, but in practice, a user can flexibly apply the information Kismet provides in other scenarios.

- *Parallelize or Not:* Parallelization is hard, and the resulting performance is hard to predict. If a programmer spends time parallelizing a program with limited parallelism, he or she is likely to waste time and effort. Kismet provides the expected speedup upperbound, allowing a programmer to make an informed decision on parallelization. When little speedup is expected, the programmer might better spend their time on serial optimization of the target program.

- *Setting Parallelization Goals:* By providing the estimated speedup upperbound, Kismet helps set reasonable performance goals in parallelization, which is often time-consuming and error-prone with manual analysis. A realistic performance goal will prevent a programmer from both missing important parallelization opportunities and ineffective parallelization with little speedup.

- *Exploring Scalability:* Kismet reports the parallel performance scalability by estimating the speedup on varying number of cores. This scalability information can provide valuable insights for early parallel system design. For example, if the software in a target embedded system never gets performance improvement over four cores, the system designer can avoid using a processor

with more than four cores, potentially reducing the hardware cost as well as lowering the power consumption of the system.

1.3 Thesis Outline

The rest of the thesis is organized as follows.

- Chapter 2 presents parallelism profiling techniques based on a novel dynamic analysis called *hierarchical critical path analysis (HCPA)*. In contrast to CPA, which operates on whole programs and cannot quantify the parallelism of nested regions, HCPA quantifies the parallelism of each region (e.g. functions and loops) with a new metric called self-parallelism, allowing flexible use of the profiled information in our speedup estimation tool. HCPA is co-developed with Saturnino Garcia, but this thesis has an important contribution over the initial version of HCPA [GJLT11]. The initial version of HCPA relied on compression techniques to reduce the size of output, but its effectiveness was limited to loops where contiguous iterations share the same amount of parallelism. In contrast, our summarization-based HCPA effectively reduces the output size even with loops with irregular control flow and recursive routines, dramatically improving the applicability of HCPA.

- Chapter 3 describes how we incorporate parallelism profile and other parallelization constraints to provide practical speedup estimates. We propose the use of expressible self-parallelism (ESP) to honor each parallelization environment's limited ability to exploit parallelism. Using parallelization overhead and the program's memory locality information help provide more realistic speedup estimates. This chapter will also present the results of experimental results on two very different platforms: the MIT Raw processor and conventional multi-core systems.

- In Chapter 4, we discuss the design and implementation of Skadu, a vector shadow memory. As HCPA recursively applies CPA, which is already expensive in the original form, it incurs significant overhead in both memory

and runtime. Skadu uses a few synergistic techniques from encoding to dynamic compression, dramatically lowering the overhead of HCPA. Skadu can be also applied to other heavyweight dynamic analysis. We demonstrate the effectiveness of Skadu in both HCPA and a memory footprint analyzer.

- Chapter 5 discusses related work of the dissertation.
- Chapter 6 concludes and summarizes this dissertation.

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 "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 primary 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 secondary investigator and author of this paper. The material in this chapter is copyright

Chapter 2

Profiling Parallelism with Hierarchical Critical Path Analysis

This chapter introduces a dynamic program analysis called hierarchical critical path analysis (HCPA) that quantifies the amount of parallelism in a program. The amount of parallelism widely varies with a target program, and it is one of the major bottlenecks that limit achievable parallel speedup, as implied in the Amdahl's Law. Hence, quantifying parallelism is crucial in parallel speedup estimation. HCPA extends an existing technique called critical path analysis (CPA), which provides the theoretical parallel speedup upperbound. Unfortunately, CPA reports speedup numbers that are too optimistic for practical use. HCPA addresses major issues of CPA by incorporating the program region structure into parallelism measurement and localizing each region's parallelism with a new metric called self-parallelism.

The techniques introduced in this chapter are joint work with several collaborators. The interested reader may refer to the work of Garcia et al. [GJLT11] for the application of HCPA on providing step-by-step guidance upon manual parallelization.

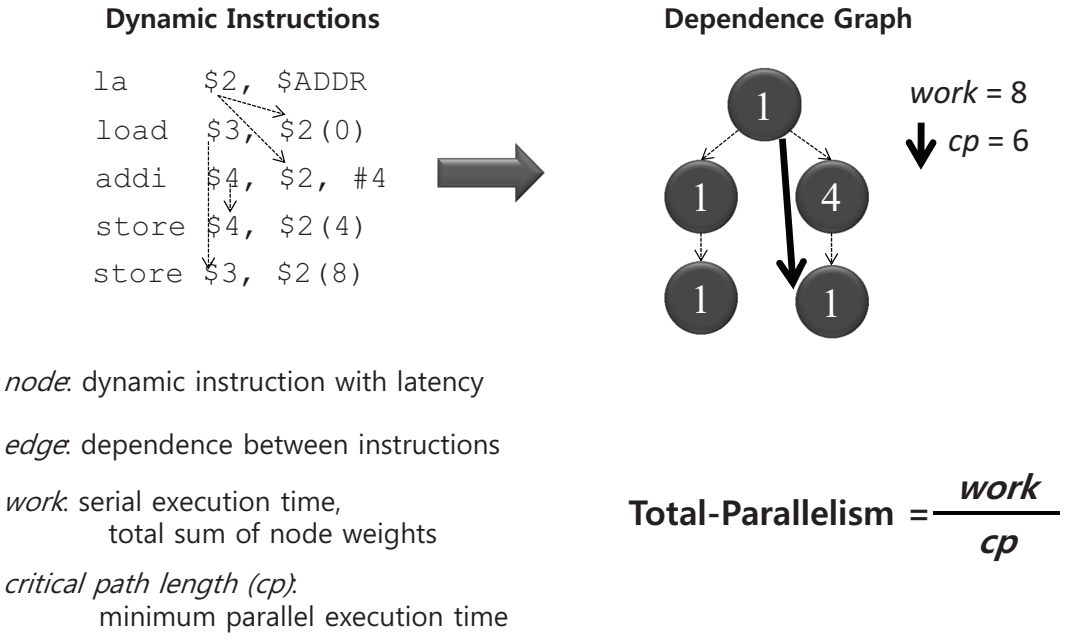


Figure 2.1: **An Example of Critical Path Analysis (CPA).** CPA computes the ideal parallel speedup without parallelizing the code by constructing the dependence graph from dynamic instructions.

2.1 Background: Critical Path Analysis (CPA)

Critical path analysis, or CPA [Kum88], is a dynamic program analysis that computes the ideal parallel speedup of a program. CPA reports the ideal speedup by analyzing serial execution of a program, without requiring parallelization. Researchers have used CPA in several projects [Kum88, KMC72, AS92, KBI+09] to identify the potential parallelism in programs.

CPA calculates the ideal speedup of code by analyzing dependences among dynamic instructions. Figure 2.1 shows how CPA works with an example. CPA first builds a dependence graph where each node represents a dynamic instruction with latency and each edge represents a register-, control-, or memory-dependence between instructions. Once the dependence graph is built, CPA finds the length of the longest path through this graph, the critical path length (cp), which represents the ideal parallel execution time of a program. Finally, CPA calculates the *total-parallelism* of the program by computing the ratio between serial execution time

Table 2.1: **Measured Speedup (16-core) and CPA Estimated Speedup.** Small correlation between CPA and measured estimates makes CPA impractical for real-world speedup estimation. Speedup was measured on 16-core Raw processor (life, is, unstruct, sha) and 16-core AMD machine (ep, is, sp).

Benchmark	**Actual Speedup**	**CPA-Estimated Speedup**	**Optimism Ratio**
ep	15.0	9,722	648×
life	12.6	116,278	9,228×
is	4.4	1,300,216	295,503×
sp	4.0	189,928	47,482×
unstruct	3.1	3,447	1,112×
sha	2.1	4.8	2.3×

(*work*) and critical path length. Total-parallelism quantifies the ideal speedup of the program when parallelized for an ideal machine where it has infinite resources and zero communication and synchronization delay.

The Promise of CPA CPA has a few major advantages in quantifying the amount of parallelism in the code. First, it does not require the parallelization of the code. Considering that manual parallelization typically requires extensive work, it is a big advantage. Also, unlike conservative static program analyses, it utilizes accurate memory dependence and control-flow information gathered at runtime, providing more accurate parallelism information of the program. Finally, CPA provides an approximate upperbound in speedup invariant of the serial expression of a program. For example, reordering two independent statements in the source code does not change CPA's output. Furthermore, typical parallelization transformations, such as loop interchange, loop fusion and loop skewing also do not affect CPA's output.

Limitations of CPA CPA has seen limited utility outside of research projects [Kum88, KMC72, AS92, KBI+09]. because it tends to be wildly optimistic. Table 2.1 contrasts CPA estimated speedups against measured speedups on 16-core

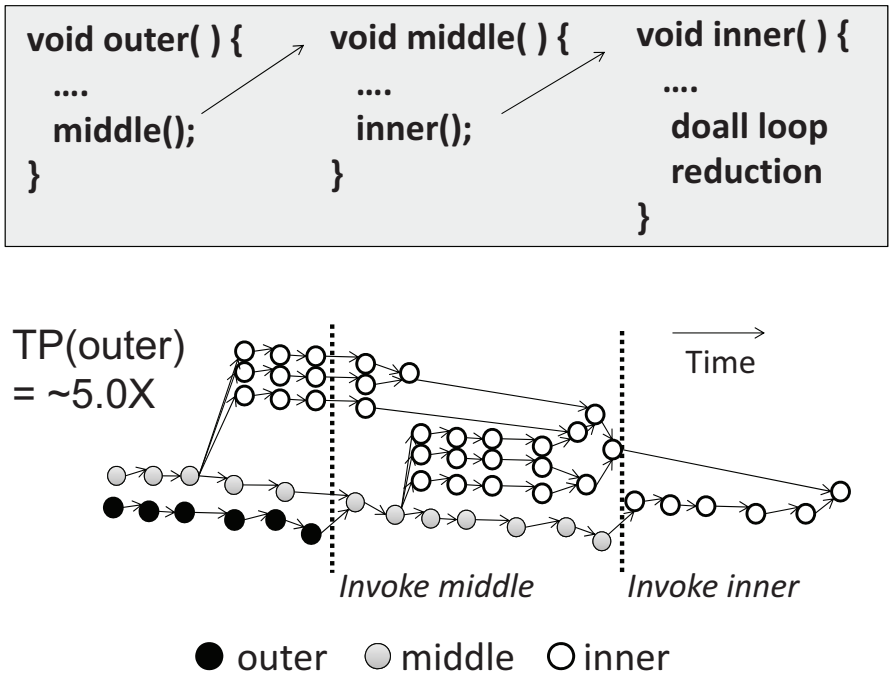


Figure 2.2: **Data-Flow Style Execution Model in CPA.** CPA calculates total-parallelism assuming an unrealistic data-flow style execution. Each circle represents a dynamic instruction and the circle's color shows which function the instruction belongs to. With the data-flow style execution model, instructions from all three functions are smeared to their earliest point at which their inputs are available.

system. As shown in the table, typical CPA reported numbers far exceed the number of available cores. What is worse, reported numbers are often uncorrelated with actual speedups attained.

CPA's optimism mainly results from two factors. First, CPA assumes an ideal execution environment - a dataflow model of execution with infinite hardware resources. Figure 2.2 shows sample source code and the corresponding dependence graph constructed by CPA. Each node represents a dynamic instruction and the color of a node shows where the instruction came from among three functions. In CPA's execution model, any instruction can be executed as soon as all of its dependencies are resolved. For instance, in Figure 2.2, instructions from `middle()` and

`inner()` are already scheduled when `outer()` invokes `middle()`. Unfortunately, this execution model does not easily map onto Von Neumann machines and imperative programming languages. For practical use, we need a program analysis that can incorporate realistic execution model and parallelization constraints.

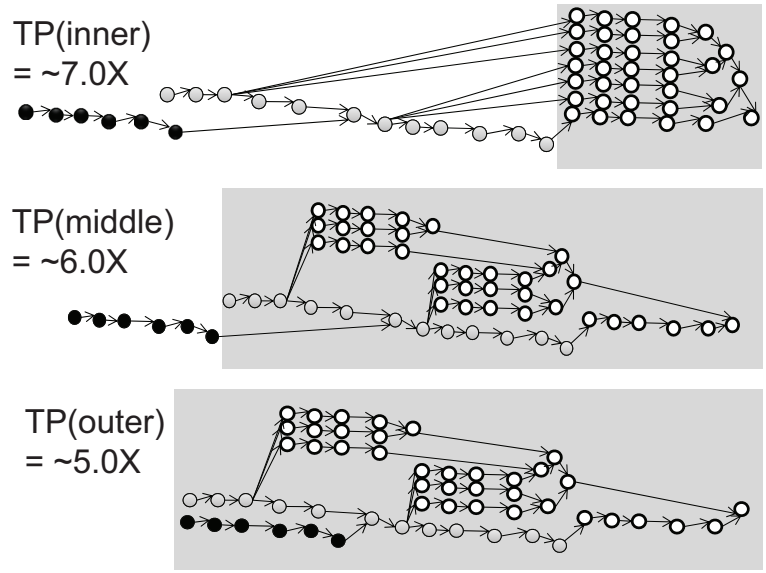
Second, CPA only reports a single total-parallelism number from a program. The total-parallelism number gives little information beyond the theoretical speedup upperbound when the whole program is ideally parallelized. In practice, however, programmers tend to focus on a few important parallelizable regions rather than the whole program due to parallelization constraints such as the limited number of available cores and non-trivial parallelization cost.

2.2 Hierarchical Critical Path Analysis

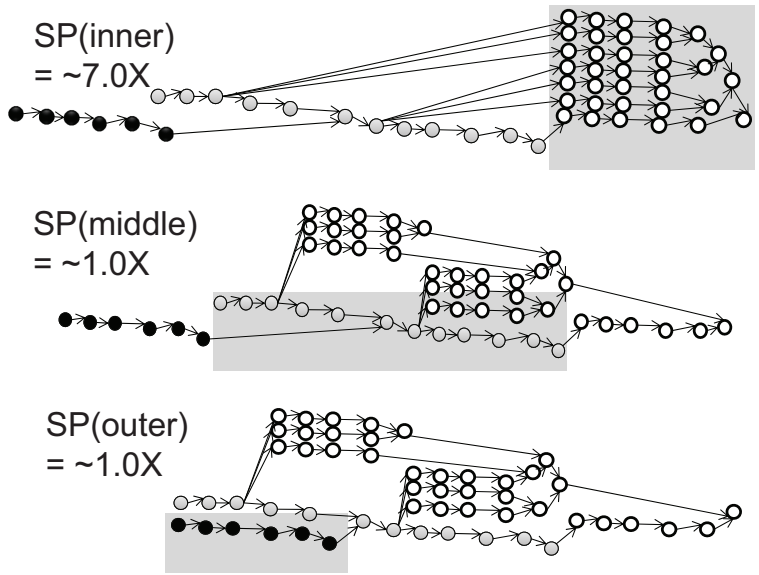
As discussed in the previous section, CPA has had limited utility in software engineering tools because of two main factors: the parallelism it reports is not indicative of the potential parallel speedup and it calculates only a single parallelism number for a single continuous program region, providing little information for parallelization in practice. To counter these limitations of CPA, we introduce hierarchical critical path analysis (HCPA). HCPA extends CPA with a hierarchical region model and provides localized parallelism information via a new parallelism metric, self-parallelism.

Unlike CPA, HCPA localizes parallelism to a specific region by independently applying CPA to each region. Figure 2.3(a) shows the result when HCPA applies CPA to each region and reports each region's total-parallelism. However, recursively applying CPA is not enough. Although the total-parallelism of each region represents the ideal speedup of each region, CPA measures parallelism that is derived from both a region *and* its children; `middle()`'s total-parallelism includes the parallelism from `inner()`, making it unclear how much parallelism `middle` contains.

HCPA further localizes parallelism to specific regions by introducing a new metric called *self-parallelism*. Self-parallelism refers to the parallelism of a region



(a) Recursively Applying CPA



(b) HCPA with Self-Parallelism

Figure 2.3: **Localizing Parallelism in HCPA.** (a) Recursively applying CPA localizes parallelism. However, a child region's parallelism is counted toward its parent's parallelism, blurring the origin of parallelism. (b) Self-parallelism further localizes parallelism to a region by excluding its children's parallelism.

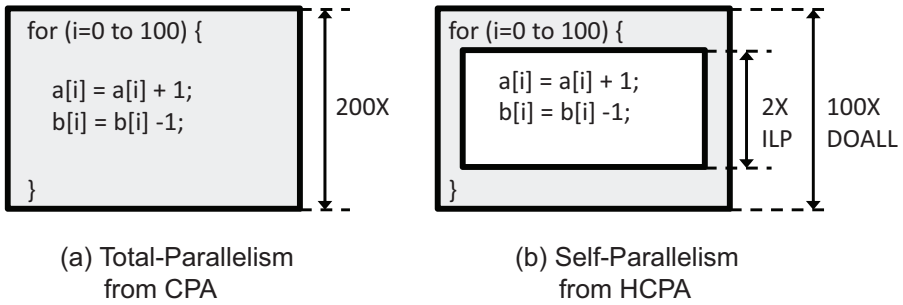


Figure 2.4: **Output Comparison between CPA and HCPA.** While CPA reports only total-parallelism value for the whole program, HCPA provides self-parallelism and parallelism type for each region, enabling better parallelization planning.

exclusive of the parallelism from its children. Figure 2.4 contrasts self-parallelism with total-parallelism. Whereas CPA's total-parallelism provides only a single parallelism number from the program, HCPA's self-parallelism provides more localized parallelism information for each region.

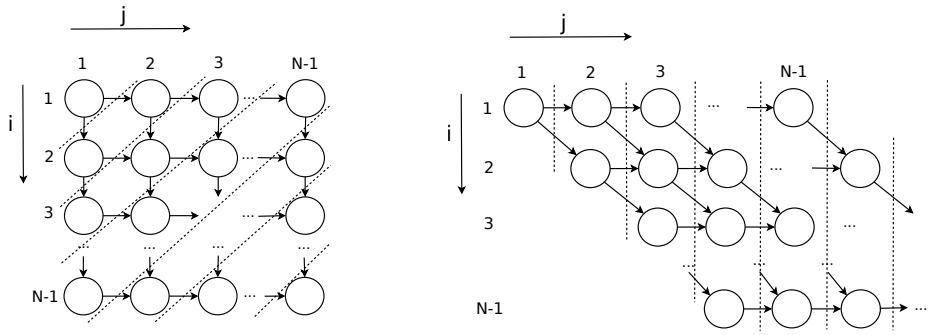
Figure 2.3(b) shows how the self-parallelism metric addresses the problem of total-parallelism in the previous example. As self-parallelism eliminates the parallelism originating from child regions, it is now clear `outer()` and `middle()` functions do not contain parallelism. By quantifying each region's exclusive parallelism, HCPA allows realistic speedup estimation where only selected regions are parallelized with realistic parallelization constraints.

HCPA's greatest strength lies in its ability to find parallelism in many forms: task-based parallelism, pipelined parallelism, skewed parallelism, data parallelism, and many forms of loop-based parallelism (including DOALL and DOACROSS) are recognized, even if the code is not currently structured to express it.

Figure 2.5 shows an example of HCPA's power in detecting parallelism– even when it is masked in the current implementation. The code in Figure 2.5a presents two challenges to the parallelizing compiler. First, the 2D array has been implemented as an array of pointers to arrays. Second, the dependence structure between updates of values in the arrays creates cross-iteration dependences in

```
1  void calc_array(int** a)
2  {
3      for(i = 1; i < N; ++i)
4          for(j = 1; j < N; ++j)
5              a[i][j] = a[i−1][j] + a[i][j−1];
6  }
```

(a) Loop with unexpressed parallelism.



(b) Before Loop Skewing

(c) After Loop Skewing

Figure 2.5: **Uncovering Masked Parallelism.** The use of critical path analysis allows HCPA 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 dependences over both loops, making it appear very serial. The iteration dependence graph in (b) shows that iterations can be grouped into independent sets, allowing parallel execution if loop skewing and interchange are used as shown in (c). Techniques relying on dependence testing would overlook this parallelism. Furthermore, the 2D array in (a) is represented as an array of pointers to arrays, thwarting a parallelizing compiler's attempt to statically analyze this section of code.

both loops. To parallelize this code requires two key analysis. First, it requires the compiler to recognize that a loop-transformation technique called loop-skewing can be applied, which restructures the loop so that execution traversals the array "diagonally" (as shown in Figure 2.5c). Second, it requires the compiler to prove, possibly using shape analysis, that none of the pointers in the first level of the array point to the same array in the second level; i.e. that there is no aliasing.

Some research compilers have implemented shape-analysis passes that could potentially decipher that the data structure is equivalent to a 2D array; and similarly some research compilers are able to automatically infer loop-skewing of static arrays. More generally, to unlock the parallelism latent in sequence programs may require that an arbitrary number of difficult analyses and transformations that must be composed. 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.

However, using runtime information, HCPA is easily able to identify and quantify the parallelism that is latent in the double-loop structure, allowing a speedup estimation tool to count this important parallelization opportunity. When parallelization the code, the programmer can work to iteratively transform the code sufficiently that the compiler or runtime system can take it the rest of the way. In contrast, weaker dynamic dependence testing-based frameworks would typically report no available parallelism because they are not able to see past the existing structure of the double loop, leading to underestimated parallel speedup.

2.3 HCPA Implementation

Although the concept of HCPA is quite straightforward as discussed in the previous section, it involves several implementation issues for the use in speedup estimation. In this section, we will focus on four major issues in HCPA implementation.

- *Designing the Region Hierarchy:* HCPA captures the self-parallelism of each region in the region hierarchy. The design of region hierarchy can fundamen-

tally impact the result of HCPA. We present a simple but effective region hierarchy design for speedup estimation.

- *Calculating Critical Path Length:* Unlike CPA that tracks the single critical path length of the whole program, HCPA must simultaneously track multiple critical paths for each region, which can be very costly. We present an array of techniques that reduces the overhead of simultaneously finding critical path lengths in multiple regions.

- *Calculating Self-Parallelism:* Self-parallelism metric represents the exclusive parallelism of a region, excluding the parallelism from its children. We calculate self-parallelism with an effective approximation.

- *Summarizing Profiled Information:* The number of dynamic regions can be extremely large, thus naively storing all the information could be prohibitive. We use a context-sensitive summarization technique to effectively and efficiently represent the profiled information.

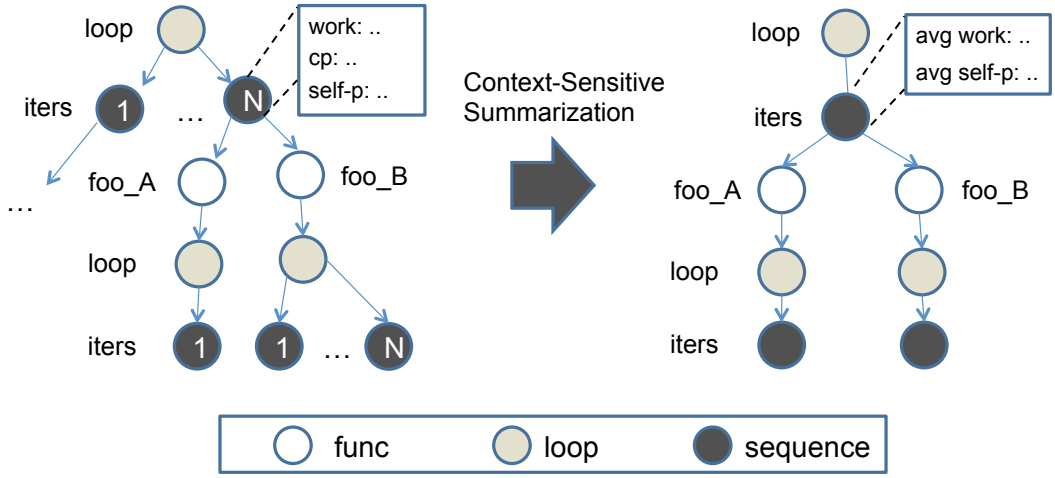
2.3.1 Designing the Region Hierarchy

HCPA uses the concept of a region to denote a region of code whose parallelism is to be measured from the time that region is entered until the time it is exited. In order for the self-parallelism metric to work, regions must obey a proper nesting structure: regions must not partially overlap, but they may nest or be siblings with the same parent region. Based on this nesting structure, we can define a dynamic region graph which shows the relationship between parent and children regions in the dynamic execution of the program.

Although more arbitrary delineations of regions are possible, we use three types of regions - *loop*, *function*, and *sequence*. These regions are designed in an attempt to quantify the parallelism of constructs that users understand well and relate directly to the process of parallelization. Loop and function regions directly map to loops and functions in the program. A sequence region can be any single-entry piece of code but we restrict sequences to two important cases: loop bodies

```
1  int main {
2      for (i=1 to N) {
3          foo(1); // callsite A
4          foo(N); // callsite B
5      }
6  }
7  void foo(int size) {
8      for(i=1 to size) {
9          // loop body
10     }
11 }
```

(a) Sample Code Fragment



(b) Dynamic Region Tree and Summarized Region Tree

Figure 2.6: **Overview of HCPA.** HCPA builds a hierarchical region structure from source code, forming a dynamic region tree at runtime. As each dynamic region is profiled, HCPA summarizes the profiled data into a summarized region tree. The summarized tree preserves context-sensitive parallelism information, exposing more parallelization opportunities.

and self-work sequences. Loop body regions form a child region for each iteration of a loop region, allowing HCPA to identify loop-level parallelism. Self-work sequence regions are sequences of code that are contained in non-leaf regions and do not have any function calls or loops. These regions may seem unintuitive but they separate different types of parallelism, improving the accuracy of speedup estimation. Self-work sequences factor out the instruction level parallelism in regions that would otherwise contain a mix of task-level parallelism (from its other children) and instruction level parallelism.

Kismet demarcates region boundaries at static instrumentation time. Kismet's instrumentor inserts function calls to the runtime, and they form a region tree at runtime. For example, the source code in Figure 2.6(a) forms a dynamic region tree as shown in Figure 2.6(b).

2.3.2 Calculating Critical Path Length

The main task of CPA, the underlying technique of HCPA, is to find the single critical path length of the whole program, which can be costly. HCPA, on the other hand, has to track the critical path length for each region, incurring much higher memory and runtime overhead.

In order to efficiently find the critical path length, HCPA independently maintains the timestamp of each operand (i.e. register and memory address) for each region. A timestamp represents the earliest execution time an instruction is available for execution with the operand. HCPA tracks the maximum timestamp value used in each region and report it as the critical path length when the region exits. All timestamps are logically initialized to zero upon entry so that a reference to an instruction outside the region will be assumed to be available immediately at the beginning of the region (i.e. time 0). Upon a dynamic instruction, HCPA takes timestamps of every source operand and control dependence, finds the maximum timestamp, and updates the timestamp of the destination operand after adding the latency of the instruction.

Although the number of dynamic regions in a program can be very large, the number of regions that need simultaneous timestamp update is limited to the

```
1  void handlerBinary(int dest, int src0, int src1, int cost) {
2      for (int depth=0; depth<active_region_depth; depth++) {
3          // calculate the updated timestamp for dest
4          Time time_control_dep = getControlDepTime(depth);
5          Time time_src0 = getRegTime(src0, depth);
6          Time time_src1 = getRegTime(src1, depth);
7          Time time_dest = max(time_control_dep, time_src0, time_src1) + cost;
8          setRegTime(dest, time_dest, depth);
9
10         // update critical path length and work
11         Region* region = getActiveRegion(depth);
12         region->cp = max(region->cp, dest);
13         region->work += cost;
14     }
15 }
```

Figure 2.7: **Runtime Instruction Handler.** Upon a dynamic instruction, HCPA calculates and updates the timestamp of the dest register for all the active regions.

number of active regions. An active region refers to a region which has entered but has not exited yet. In the dynamic region graph, the number of active regions is the same with the depth of the current dynamic region in the tree. Every active region independently manages its timestamps, making the update process similar to a vector operation.

Figure 2.7 shows a simplified runtime code that is invoked upon a dynamic instruction. The dynamic instruction takes two source registers and one destination register. Each iteration of the loop (line 2) in the code handles an active region. It first reads timestamps from two source registers and control dependence, and calculates the new timestamp by adding the cost of the instruction. Then it updates the timestamp of the destination register, and finally updates the critical path length of the region if the new timestamp value is larger than the current critical path length.

HCPA honors true dependences including register-, memory-, and control-

dependencies. Because we aim to provide speedup upperbound, HCPA ignores false dependences and easy-to-break dependences. For example, every *for loop* carries an inter-iteration dependence with its loop variable, but the dependence can be easily broken.

Register Dependence At compile time, the LLVM-based instrumentor efficiently and accurately analyze true dependencies between registers. HCPA's instrumentor inserts a function call so that the HCPA runtime uses the dependence information when it updates timestamps. Dependencies that are not true dependencies are filtered out using two main mechanisms. First, our instrumentor operates on LLVM's SSA form IR [CFR+91]. This eliminates false output (i.e. write-after-write) dependencies. Next, the instrumentor detects induction and reduction variables then breaks the false dependencies that result from them.

In order to efficiently manage timestamps, we use vector shadow register (VSR). A VSR is an array of vectors where each vector contains timestamps of a register for all the active regions. Each element of a vector represents a timestamp of an active region. Because every function independently manages a register space, HCPA allocates a VSR when the function starts and deallocates it when the function ends. In a VSR, all the vectors in a VSR share the same length - the length represents the maximum depth of a region in the dynamic region tree that might use the register associated with the vector, which can be easily analyzed at the compile time. The number of registers used in a function, in LLVM's SSA form, determines the number of vectors in a VSR.

Memory Dependence HCPA detects every memory dependence at runtime and incorporates the dependence information in its critical path length calculation. Compared to the static pointer analyses used in many parallelizing compilers, this runtime approach can handle irregular control flows and complicated pointer operations, enabling more accurate parallelism quantification.

We use vector shadow memory (VSM) to manage timestamps for memory addresses. Vector shadow memory is a variant of shadow memory, which provides efficient tagging of information to memory address space. Shadow memory is

widely used in dynamic program analyses, from memory analysis [SN05, BZ11] to computer security [CZYH06, QWL+06, XBS06].

Vector shadow memory shares a similar goal with vector shadow register. It efficiently provides an independent storage for each dynamic region in a program so that HCPA can read and update timestamps associated with each memory address. Similar to VSR, VSM consists of vectors where each vector provides storage for all the active regions.

Although vector shadow memory resembles vector shadow register in its functionality and structure, it raises serious challenges for practical use. VSM's address space is larger than VSR's register count by several orders of magnitude. If not carefully managed, VSM will incur prohibitively large memory overhead. Furthermore, the length of a vector varies in VSM, making it even more difficult to efficiently allocate and deallocate VSM's memory. Similar to the vector length in VSR, a vector's length is determined by the maximum depth of the region that accesses the vector (i.e. the address associated with the vector). However, it is undecidable at compile time as opposed to VSR's vector length that can be easily analyzed by the instrumentor. In Chapter 4, we discuss an array of techniques that can dramatically reduce the memory and runtime overhead of vector shadow memory.

Control Dependence Kismet tracks control dependencies through the use of control dependence analysis and a dynamic control dependence stack. Timestamps for control dependence are pushed to and popped from the control dependence stack whenever a control dependent region is entered and exited. A stack entry contains a vector of timestamps for every active region. For a region, this stack has monotonically increasing values from the bottom to the top, allowing HCPA to include only the topmost entry in the list of dependencies for each instruction.

Although HCPA aims to find the critical path length of each region, it also profiles other useful information for speedup prediction. For example, HCPA also determines if all of the children of a non-leaf region are independent and therefore can be executed in parallel. This information is useful for identifying DOALL

loops, which is both common and easy-to-exploit in many target platforms. To store the information, each region calculates a "P bit" using the following equation:

$$P = CP(parent) == MAX\ (CP(child_1),\ ...,\ CP(child_n))$$

where $CP(parent)$ is the critical path length of the parent and $CP(child_i)$ is the critical path length of the i^{th} child.

If all children can be executed in parallel, then the length of the critical path will simply be the length of longest critical path of all of the children. In this case, the "P bit" will be 1. Chapter 3 discusses how we leverage this in more detail.

2.3.3 Self-Parallelism Calculation

When a region exits, HCPA calculates the self-parallelism with the calculated critical path length and children's information. Since a parent region exits after all the children's execution is over, every child's profiled information is also available when a region exits.

To determine the self-parallelism of a region R, $SP(R)$, HCPA employs the following equation:

$$SP(R) = \begin{cases} \frac{\sum_{k=1}^{n} cp(child(R,k))}{cp(R)} & \text{R is a non-leaf} \\ \\ \frac{work(R)}{cp(R)} & \text{R is a leaf} \end{cases}$$

Here n is the number of children of R, $child(R,k)$ is the k^{th} child of R, $cp(R)$ is the critical path length, and $work(R)$ is the amount of work in R.

Intuitively, this equation captures the ratio of execution time between serial and parallel run of fully parallelized children. By using fully parallelized children's execution time, self-parallelism metric captures the exclusive parallelism of the parent region. Not having any children, a leaf region's self-parallelism is identical to CPA's total-parallelism.

Figure 2.8 demonstrates the calculation of SP in three non-leaf regions, one totally serial, one partially parallel (DOACROSS), and the other totally parallel

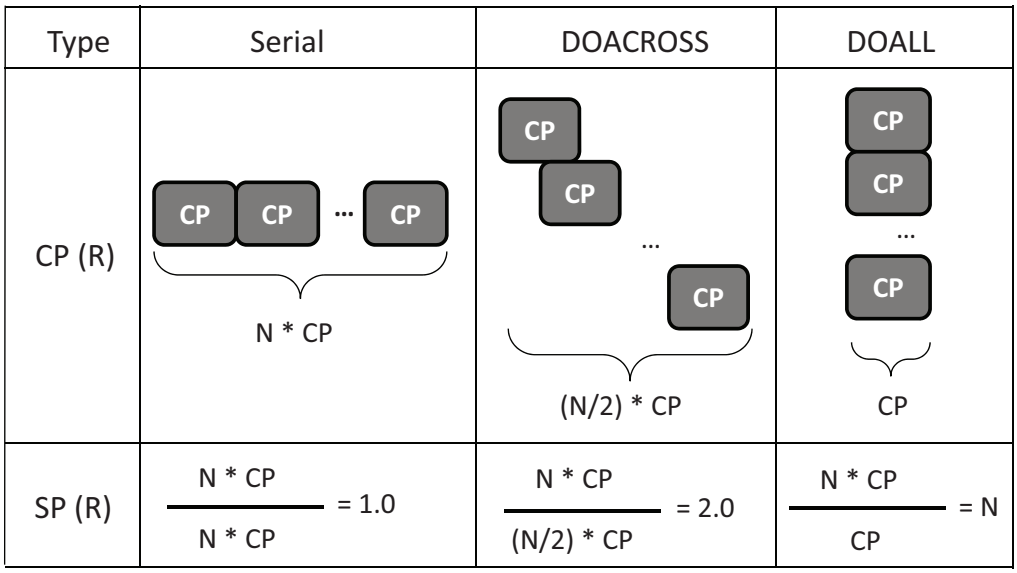


Figure 2.8: **Self-Parallelism Calculation on Regions with Varying Parallelism.** Self-parallelism computes the amount of parallelism in a parent region that is attributable to that region and not its children. The figure above shows that Kismet's self-parallelism calculation successfully quantifies parallelism across a spectrum of loop types, ranging from totally serial to partially parallel (DOACROSS) to totally parallel (DOALL). The shaded boxes are child regions, corresponding to separate iterations of the loops. The relative scheduling of child regions is indicated spatially, with time running from left to right. The self-parallelism calculation correctly quantifies parallelism in non-loop region hierarchies as well.

(DOALL). For simplicity, in the example, each iteration's critical path length cp is the same. For the serial loop, the measured $cp(R)$ will be equal to $n * cp$ and the computed self-parallelism will be $\frac{n*cp}{n*cp} = 1$, which is expected since serial dependences prevent overlapped execution of regions. For the DOACROSS loop shown, where half of an iteration can overlap with the next iteration, $cp(R)$ will be the half of the $cp(R)$ for the serial loop. Thus $SP(R)$ is $\frac{n*cp}{(n/2)*cp} = 2$. For the DOALL loop, $cp(R)$ will be equal to cp, so $SP(R)$ is $\frac{n*cp}{cp} = n$. Although we show three relatively simple cases here, this method is a good approximation of self-parallelism even with more sophisticated child region interaction.

2.3.4 Summarizing Profiled Information

HCPA produces a parallelism profile for each dynamic region that is executed. 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. We developed a summarization technique that effectively reduces the amount of profiled data while preserving context-sensitive parallelism information.

Our summarization technique combines all dynamic regions that have the same region context into a single summarized region. Figure 2.6b depicts how the runtime region tree becomes a summarized region profile. In this method, all loop iterations collapse to a single node, greatly reducing the number of regions. Each node calculates weighted averages for self-parallelism, work, and other profiled data across all dynamic regions corresponding to that node.

Kismet 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 HCPA's region hierarchy.

The example summarized region profile shown in Figure 2.6b 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 Kismet to uncover new parallelism opportunities.

To understand the merit of context-sensitive representation, consider the code in Figure 2.6b. 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. Kismet can capitalize on the split contexts, incorporating the speedup from callsite B into its estimates while ignoring callsite A. The tree structure of context-sensitive representation also allows the development of parallel execution time model when a few regions are parallelized. Details of parallel execution time model will be discussed in Chapter 3.

2.4 Experimental Results

2.4.1 Effectiveness of Self-Parallelism Metric

In order to demonstrate the merit of self-parallelism metric against CPA's total-parallelism, we examined programs in the NAS Parallel Bench (NPB) benchmark suite [BBB+91]. We measured both self-parallelism and total-parallelism of all 1953 regions in NPB and classify them based on the amount of parallelism: serial (parallelism < 1.1), moderately parallel (1.1 to 2.0), parallel (2.0 to 5.0), or very parallel (parallelism > 5.0).

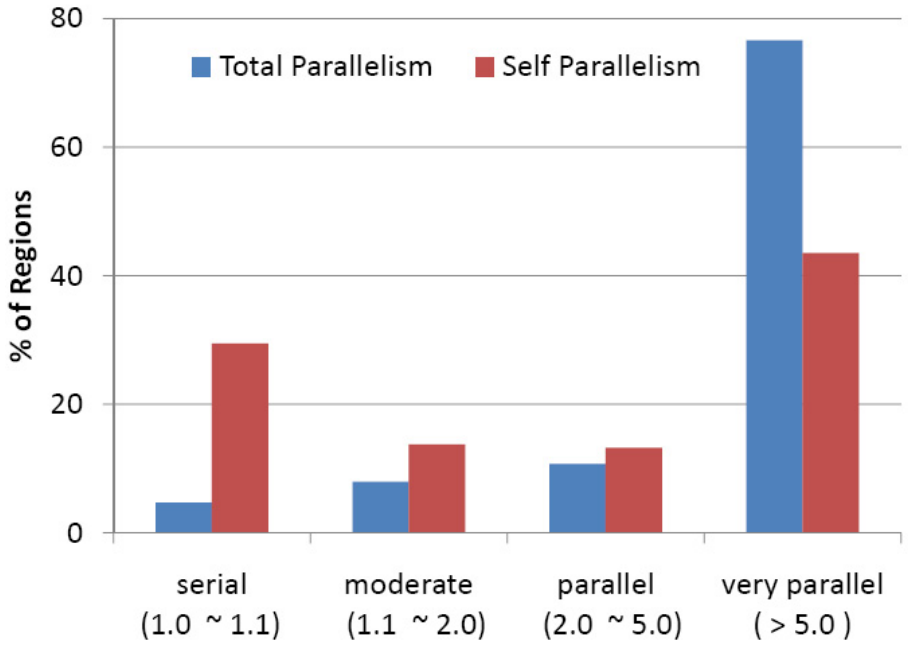


Figure 2.9: **Region Classification Based On Parallelism.** We classified all 1953 regions in the NPB benchmarks based on the amount of parallelism. Switching from total-parallelism based classification to self-parallelism based classification, 6× more regions are classified as being serial. This result highlights self-parallelism's ability to localize parallelism, filtering out false positive parallel regions in the speedup prediction.

2.4.2 Effectiveness of the Summarization Techniques

Effectiveness of the Summarization Technique To examine the effectiveness of Kismet's summarization technique, we ran NPB and SpecInt2000 benchmarks [1] with two different input sizes ('S' and 'A' for NPB, 'test' and 'ref' for SpecInt2000) and examined dynamic region counts as well as output file sizes. Figure 2.2 shows the results.

The results show that Kismet's summarization technique scales well with increasing input sizes and is effective at reducing the output file size. As expected, the dynamic region count significantly increases when we switch from small input to larger input – 463X on average. With the larger input sets, dynamic region

[1] Raw benchmarks have only a single input set.

Table 2.2: **Impact of Summarization Technique on File Size in NPB.** Switching from the small (S) to large (L) inputs causes 463× more dynamic regions to execute on average, but the output file size increases only 1.1× on average, from 77KB to 85 KB. Thus, the summarization technique is very effective in keeping output file size manageable even with large inputs.

Bench	Dynamic Region Count (Mega Regions)			Output File Size (Kilo Byte)		
Input	S	L	Ratio	S	L	Ratio
bt	4	2665	666×	102	102	1.0×
cg	38	830	22×	15	15	1.0×
ep	50	805	16×	4	4	1.0×
ft	40	1526	38×	50	50	1.0×
is	0.7	104	149×	3	3	1.0×
lu	2	2208	1104×	45	45	1.0×
mg	2	969	485×	79	79	1.0×
sp	10	7452	745×	166	167	1.0×
bzip2	846	4086	5×	62	63	1.0×
gzip	141	4477	32×	96	137	1.4×
mcf	7.8	4758	595×	19	20	1.1×
twolf	11.4	23023	2093×	260	309	1.2×
vpr	42.1	3020	72×	104	107	1.0×
mean	92	4302	**463**×	77	85	**1.1**×

profile data runs as large as several terabytes, clearly too large to be conveniently stored to disk. With HCPA, there is virtually no difference in the output file size between small and large input sets. Moreover, the summarization technique results in very modest file sizes – only 85KB on average.

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 “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 primary investigator and author of this paper.

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 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 secondary investigator and author of this paper.

Chapter 3

Predicting Speedup with Realistic Parallelization Constraints

In the previous chapter, We discussed hierarchical critical path analysis that provides the parallelism of each regions with self-parallelism metric. Although parallelism is essential for practical speedup estimation, there are other parallelization constraints that could significantly affect the achievable speedup of a target program. In this chapter, we describe the Kismet system and explain how Kismet incorporates key parallelization constraints on top of the HCPA results, providing practical speedup upperbound from unmodified source code. We also show experimental results on two very different platforms to demonstrate the effectiveness of Kismet.

3.1 Kismet System Architecture

Kismet operates in two phases; the first phase is a profiler that collects self-parallelism data with HCPA and the second phase is a speedup predictor which applies machine and system constraints. Figure 3.1 shows the overview of Kismet system.

Self-Parallelism Profiler Kismet first profiles the work and self-parallelism of each region through HCPA. As discussed in Chapter 2, self-parallelism is a new

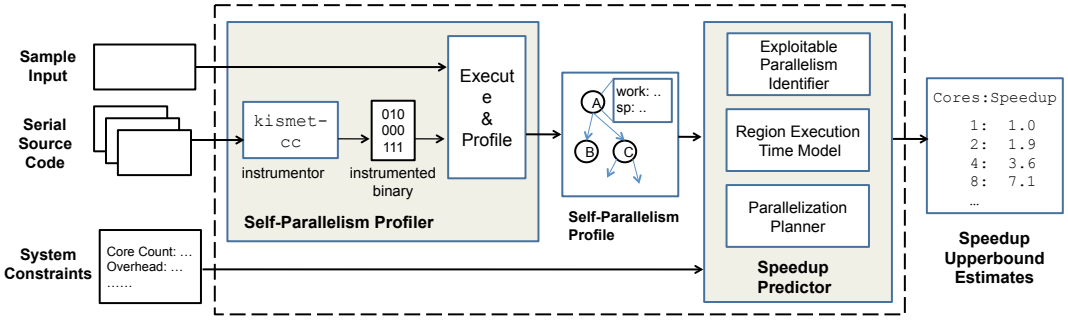


Figure 3.1: **Kismet System Architecture.** Starting with a program's source code, Kismet produces an instrumented binary by inserting profiling code. Running the instrumented binary on the sample input outputs a trace file containing both program structure and self-parallelism. Finally, the speedup predictor estimates the speedup upper bound based on the profile data. The parallelism classifier filters unexpressible parallelism for realistic speedup estimates (via the *expressible self-parallelism* filter) and the parallel execution time model incorporates hardware constraints and parallelization overhead for accurate performance prediction.

metric that represents the ideal speedup of a region when parallelized. By quantifying the parallelism of each region, Kismet supports region-based parallelization, which is similar to what a programmer does upon parallelization.

The gathering of self-parallelism information in Kismet starts with a static instrumentation phase which instruments the target program with code that implements HCPA, and ends with running the instrumented program.

The static instrumentation phase transforms the input code to support HCPA during execution of the sample input. The inserted instrumentation consists of calls to a special HCPA library, which will perform the dynamic analysis during execution. In addition to adding instrumentation for calculating critical paths, Kismet also inserts instrumentation to clearly delineate the regions of the code. Three types of regions—loops, functions, and sequence—are used, allowing HCPA to calculate each region's self-parallelism and to determine the type of parallelism in each region.

The Kismet code instrumentator utilizes LLVM's [LA04] static instrumentation infrastructure to perform a deeper level of analysis than is available with

dynamic instrumentation tools such as Valgrind [NS07b]. This allows Kismet to easily uncover the program structure and account for false dependencies introduced by loop induction variables and reduction variables. Static instrumentation also provides greater opportunity for optimizing the instrumented code in order to reduce the overhead associated with profiling.

The dynamic analysis phase begins when the instrumented binary is run with the sample input to produce per-region statistics. For each dynamic region that is executed, the dynamic analysis computes three key pieces of data: the critical path length, the amount of work done, and the self-parallelism. The data produced for each region is relatively small but the number of dynamic regions grows quickly, leading to a possibly unmanageable amount of data. Kismet improves the manageability of region data by summarizing the data as it profiles, creating *summarized region profiles*. The summarized region profiles reduce the number of recorded regions by orders of magnitude, leading to much smaller log sizes and allowing for more efficient processing in later stages of Kismet. While the reduced log size from summarization is desirable, summarization should not compromise the quality of self-parallelism information. As discussed in the previous chapter, HCPA provides rich call context-sensitive region information, helping the speedup predictor avoid underestimating the potential speedup from parallelization.

Speedup Predictor After running the instrumented binary on the sample input, Kismet has captured the underlying structure of the application in the form of the summarized region profile. The next step is to combine this information with machine and parallelization system properties in order to make a prediction.

Performance strongly depends on the target system. Kismet accepts a list of target-dependent parallelization constraints and utilizes this information to provide more accurate predictions. Typically constraints include a simple hardware specification (e.g. the number of available cores), the types of expressible parallelism by that system, and functions that quantify parallelization overheads such as synchronization. We have found that only a small number of constraints are needed to accurately predict performance. This simplifies the process of extending Kismet to new platforms. We were surprised at the ease with which our model

could support two very different parallel systems—an MIT Raw tiled processor and a 32-core AMD multicore system.

The speedup predictor contains three sub-components. The first one is a modified self-parallelism metric called *expressible self-parallelism*, or *ESP*, which filters out parallelism unexpressible by the specified target system.

The second sub-component is the *parallel execution time model*. The parallel execution time model allows the speedup predictor to estimate the parallel execution time of each program region and the whole program based on a given parallelism plan. This model incorporates self-parallelism, number of allocated cores, and parallelization overhead. Kismet also provides an extended, cache-aware parallel execution time model that considers the impact of caching on parallel execution. The parallel execution time model is used by the resource allocator to choose from completing parallelization plans and determines the final speedup numbers reported by Kismet.

The final sub-component is the *speedup planner*. An ideal parallel system will take advantage of all the expressible parallelism in a program. This desirable property is not available on most existing systems. These systems have other constraints–such as limited hardware resources, synchronization overhead, or poor support for nested parallelism – that make *expressible parallelism* not be *exploitable parallelism*. The speedup planner creates a mapping from regions to parallel resources, modeling at a high-level what the execution of the parallelized program would look like; we refer to this mapping as the parallelization plan.

3.2 Speedup Predictor

Kismet's speedup predictor attempts to find the upper bound on parallel speedup of a program by examining a spectrum of candidate parallelizations of the program on the target machine. Kismet's self-parallelism profiling provides the groundwork for calculating this speedup but it alone is not enough to determine a tight bound on speedup. In this section we will describe how Kismet processes the self-parallelism data to predict the maximum parallel speedup.

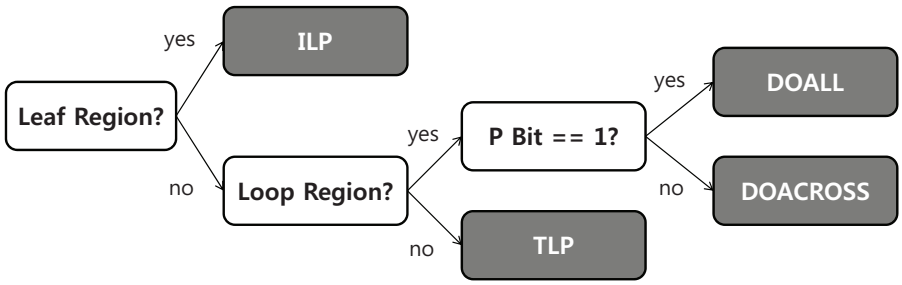


Figure 3.2: **Parallelism Identification Logic.** Kismet uses the program structure and parallelism information provided by HCPA to help classify parallelism. This figure shows the simple classification process. Kismet then uses the classification result to calculate the expressible self-parallelism (ESP). ESP quantifies the amount of expressible parallelism within a specific region of the program.

3.2.1 Expressible Self-Parallelism (ESP)

While Kismet's self-parallelism profile quantifies the parallelism in each region of the program, there is no guarantee that the parallelism will be expressible. Many systems have limitations on the type of parallelism that can effectively be expressed. Kismet transforms self-parallelism into expressible self-parallelism (ESP) in two phases. First, it classifies the type of parallelism found in each region. Second, it uses this classification to conditionally adjust self-parallelism into ESP, as follows. Regions that have self-parallelism that is unexpressible are assigned an ESP of 1. Regions with self-parallelism that is expressible have an ESP that is equivalent to their SP.

Figure 3.2 illustrates the Kismet's decision process when classifying parallelism. Kismet's region hierarchy has been designed to ensure that only leaf regions have instruction level parallelism (ILP) and that ILP is found only in leaf regions. The first step in Figure 3.2 is thus to check if the region is a leaf. If the region is not a leaf then the parallelism is either of the form of loop- or task-level parallelism. Kismet checks the region type to determine if there is a loop or a function.

Kismet further classifies loop parallelism based on whether there are cross-iterations dependencies. Loops *without* cross-iteration dependencies are classi-

fied as DOALL while those *with* cross-iteration dependencies are classified as DOACROSS. While Kismet's profile output does not contain statistics on the number of cross iteration dependencies, it does contain the information needed to quickly distinguish DOALL and DOACROSS loops. Namely, the "P bit" described in Chapter 2 indicates if all iterations are independent. Kismet examines the "P bit" for the region, classifying the region as DOALL if $P == 1$ and DOACROSS otherwise.

As with any dynamic analysis tool, Kismet's identification of parallelism is subject to differences across multiple inputs. In practice we have found that while the amount of speedup may vary slightly across multiple inputs, the Kismet classification is consistent across these same inputs.

3.2.2 Parallel Execution Time Model

Although self-parallelism is a major factor that affects the realizable speedup of a region, there are other major factors such as allocated core counts and parallelization overhead. Kismet uses a parallel execution time model that captures major factors that affect parallel execution time. With the parallel execution time model, Kismet's speedup predictor can evaluate the effectiveness of parallelization plan it produces, and reports the plan that would bring the highest speedup. We also show a cache-aware parallel execution time model that incorporates changed cache miss rates after parallelization.

Base Model The base parallel execution time model incorporates region structure, core count, and parallelization overhead in addition to self-parallelism. This model uses the following equation to determine the execution time of region R:

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

While there are different equations for leaf and non-leaf regions, they follow the same general model. The first term represents the time needed to execute the parallelized, assuming that $A(R)$ cores are allocated to that region. The top of the fraction represents the serialized execution time—the work of a leaf region, or the sum of the children's work of a non-leaf region. This time is divided by the minimum of the self-parallelism of the region ($SP(R)$) and $A(R)$. Intuitively, this means that the speedup is either fundamentally limited by the parallelism available—when $SP(R)$ is the limiting factor—or by the amount of parallel resources allocated to the region—when $A(R)$ is the limiting factor. Note that the execution time of the non-leaf regions depend on the execution time of their children; this forces a bottom-up approach to calculating the execution time of the program.

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 multicore processors. As such, Kismet 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.

Cache-Aware Model While the base model is able to accurately model benchmarks that have up to linear speedup, our results showed that some benchmarks resulted in super-linear speedup when parallelized. For example, the `cg` benchmark from the NAS Parallel Bench [BBB+91] showed significant super-linear speedup when using between 4 and 16 cores on 32-core AMD Opteron system. We found that this was a result of increasing cache size with a larger number of cores on this system, prompting us to include a cache-aware model of parallel execution time.

Kismet's cache-aware model extends the base model by including the memory service time (MST) in the calculation of $ET(R)$. MST represents time spent in memory accesses that resulted in a cache miss; it is calculated using the following

equation:

$$MST(R) = \begin{cases} \dfrac{\sum_{k=1}^{n} MST(child(R,k))}{A(R)} & \text{non-leaf} \\ \\ \dfrac{\sum_{i=1}^{depth} CMT_i(R)}{A(R)} & \text{leaf} \end{cases}$$

For both leaf and non-leaf equations, MST sums the time spent and for cache misses—either in that region for a leaf region, or among all children of a non-leaf region—in the level i cache, CMT_i, and divides this by the number of cores allocated to the region, $A(R)$. This optimistically assumes that the memory system of the target is scalable, distributing memory accesses evenly across cores so that they may be simultaneously serviced without penalty. Although it is possible to model more complicated behaviors of memory systems, this simple cache model appears to do a reasonable job of predicting superlinear speedup effects due to the memory systems.

To calculate the cache miss time at level i, Kismet uses the following equation:

$$CMT_i(R) = \sum_{i=1}^{n} MemCnt(R) * Miss_i(R, conf) * Penalty_i$$

where $MemCnt(R)$ is the number of memory accesses in region R, $Miss_i(R, conf)$ is the cache miss rate for level i, $conf$ is a specific memory configuration, and $Penalty_i$ represents the penalty for a level i cache miss. As more cores are allocated, the total cache size of $conf$ increases, potentially leading to a decrease in $Miss_i(R, conf)$.

3.3 Case Studies - Raw and Multicore

In this section, we demonstrate how Kismet can be configured to a specific platform by examining two very different platforms: the MIT Raw tiled multicore processor ("Raw") [E. 97, TKM+02, M. 04] and a conventional multicore processor ("Multicore"). Table 3.1 shows details of these two targets. We also model specific

Table 3.1: **Overview of Two Platforms** - Raw and Multicore. These two targets have different constraints in parallelization, expressible parallelism, and parallelization overhead.

Platform	Raw	Multicore
Core Type	Modified MIPS	AMD Opteron
L1 Size	32KB / Core	64KB / Core
L2 Size	-	512KB / Core
L3 Size	-	6MB / Four Cores
SW Platform	RawCC	OpenMP
Expressible Parallelism	ILP	DOALL
Non Reduction Parallelization Overhead (cycles)	$2 + 2\sqrt{N}$	250 * N
Reduction Parallelization Overhead (cycles)	$2 + 2\sqrt{N}$	500 * N

software platform because software platforms also create constraints in parallelization, affecting the speedup even on the same hardware. Specifically, we model the automatically parallelizing compiler RawCC [LBF+98] for Raw, and manual OpenMP parallelization for Multicore. For each platform, we first introduce hardware characteristics and parallelization constraints, and describe how we model parallelization overhead in parallel execution time model, and then finally describe the target-specific planning algorithm.

3.3.1 Targeting Raw in Kismet

Platform Description MIT Raw is an early tiled multi-core processor [S. 08, Tay07, GSV+10] featuring a fast, 1-cycle per hop, fine-grained scalar operand network [TLAA05]. Although many different forms of parallelism (ILP, TLP, DLP, etc) are expressible on the Raw ISA, we model the RawCC [LBF+98] parallel compiler, for which only ILP is expressible.

RawCC finds ILP in each basic block and performs space-time scheduling

to exploit it. For each instruction, the space-time scheduling determines which core executes the instruction for minimum total execution time. Inter-core data dependencies are resolved utilizing Raw's low latency network, and control flow information is broadcast across cores to ensure all cores execute the same basic block. If needed, RawCC performs loop unrolling to increase the amount of exploitable ILP in a loop.

Modeling Parallelization Overhead Two sources can incur parallelization overhead in Raw: control dependencies and data dependencies.

To ensure control dependencies are respected, RawCC broadcasts the control dependency information to all cores via Raw's static network. At the end of every basic block, each core waits for the control dependence information and branches to the specified basic block when the information arrives. The broadcast cost is $2 + 2\sqrt{N}$, where N is the number of cores. In our parallel execution time model for Raw, we approximate this overhead based on [TLAA05]: an injection latency of 2 cycles, a network diameter of $2\sqrt{N}$, and a per-hop latency of 1 cycle.

Data dependences between two instructions on different cores also incur communication overhead. Unlike with broadcasts, the cost can be hidden if the communication is not on the critical path of the execution by RawCC. As Kismet aims to bound the achievable highest speedup, Kismet does not model this overhead.

Planner Algorithm The planner algorithm takes as input the summarized region profile which includes a region tree where each node is a summarized region and each edge represents "reachable" relationship between them. As RawCC can express only ILP in a program, the planner first filters nodes with non-ILP parallelism and sets their ESP value to 1, effectively eliminating them from consideration. After ILP regions are identified, producing the plan with highest speedup is straightforward. For each ILP region R, decide A(R) that minimizes ET(R) with the given parallel execution time model. For non ILP regions, A(R) is simply set to one, representing serial execution. When A(R) is determined, parallel execution time model calculates the estimated parallel execution time of the root node with

given A(R) function, and will compute the speedup against serial execution time.

3.3.2 Targeting Multicore with OpenMP in Kismet

Platform Description The multicore platform represents conventional multicore processor systems such as Intel's Nehalem or AMD's Opteron line or processors. They use shared memory for inter-core communication and as a result the latency is significantly higher compared to Raw's low-latency networks. For the software platform, we target the popular OpenMP platform that exploits mainly DOALL parallelism. In addition to the restricting expressible parallelism to DOALL, we also disallow nested parallelization – although OpenMP supports nested parallelization, the feature is rarely used in practice as synchronization overhead is typically too large.

Modeling Parallelization Overhead OpenMP parallelization involves overhead in several aspects: thread creation, thread scheduling, reduction operations, and barrier cost. We found that thread creation cost is typically amortized with a thread pool implementation and scheduling cost is negligible when static scheduling is used. We model barrier and reduction costs since they significantly impact performance. The values chosen in Table 3.1 was taken from running the EPCC micro-benchmark [BO01] on 32-core AMD Opteron machine.

Planner Algorithm Once regions with unexpressible parallelism are filtered out, the main constraint in the Multicore planner is prohibited nested parallelization. When nested parallelization is disallowed, the planner cannot choose more than one region among regions in the path from the root node to any node in summarized region profile.

To find the optimal solution with the constraint, Kismet uses a dynamic programming algorithm. The core intuition of the algorithm is that a region should be parallelized only when the benefit of parallelization is greater than the benefit from parallelizing any set of descendant regions. The planner traverses the region profile in a bottom-up fashion, from leaf nodes up to the root node, while saving

the optimal plan P(R) at each region R. When the planner processes a new region, it compares the expected benefit of parallelizing the region against the cumulative benefit of the optimal plans of its child regions. If the benefit of parallelizing R exceeds the cumulative benefit of child regions, P(R) is set to R; otherwise P(R) is set to the union of child regions' optimal plans.

3.3.3 Kismet Usage

In this case study, we also address four commonly asked usability issues about the Kismet tool.

How Sensitive Is Kismet to Changing Inputs? Since Kismet's analysis is dynamic, it can take advantage of information that can only be extracted by observing the runtime execution of the program. This allows Kismet to find opportunities for speedup that would be undiscovered by the more conservative analyses found in parallelizing compilers. The sensitivity of a potential speedup of a program to the input varies by the underlying algorithms in the program. Although Kismet could mirror parallelizing compilers and provide more "worst-case" speedup estimates, this fails to expose the opportunities that might be available in taking advantage of input-dependent parallelization strategies. As a result, our recommended usage model is that the user run Kismet on the application multiple times, across a spectrum of representative inputs, in order to gain a deeper knowledge of this issue.

What Tasks are Performed by the User from Program to Program? Our expectation is that the maintainer of Kismet would "ship" Kismet with a library of representative machine models and planners. When the user runs Kismet, they would select via commandline parameter the machine model which most closely matches the target architecture. Thus, from the user's perspective, the tool is "push-button." Although this is clearly future work, we have also envisioned the possibility of using auto-tuner techniques (i.e. as in FFTW) to automatically calibrate these components to a new architecture, which alleviates the Kismet

maintainers of the need to update the library. Finally, as last resort, the user could extend the machine model and planner library themselves.

What is Kismet's Utility in Providing Refactoring Assistance? To be clear, Kismet does not try to make specific recommendations about how the programmer should refactor the program. Rather, it provides advanced information that helps the programmer decide a) whether it may not be worth the effort to parallelize the piece of code and b) what kind of speedup might be reasonable to aim for. The latter item may also influence the programmer's choice of transformations, but only in an indirect fashion. Although Kismet's speedup upperbounds are indeed approximate, our results show that they are rarely exceeded by actual parallelized code. A consistently low estimated speedup upperbound is a strong signal to the user that attaining speedup of the existing serial program is likely to be very challenging.

What is Kismet's Benefit Over Parallelizing Compilers? Kismet derives its key advantages over parallelizing compilers through an extension of CPA, which is a dynamic analysis not commonly used in today's parallelizing compilers. Speedup estimates provided by Kismet are likely to be higher than those attainable by a parallelizing compiler, because they are determined by empirical measurements about program parallelism rather than the ability of an automatic tool to prove properties about the program. Kismet's optimistic view of speedup attempts to take into account the programmer's greater ability to perform code transforms that would be unsafe in automatic parallelizing compilers.

3.4 Experimental Results

This section evaluates Kismet as follows. We first outline our evaluation methodology, including our selection of benchmarks and target machines. Using this methodology, we then quantify Kismet's accuracy by comparing both predicted and measured speedups from parallelization of three benchmark suites on three

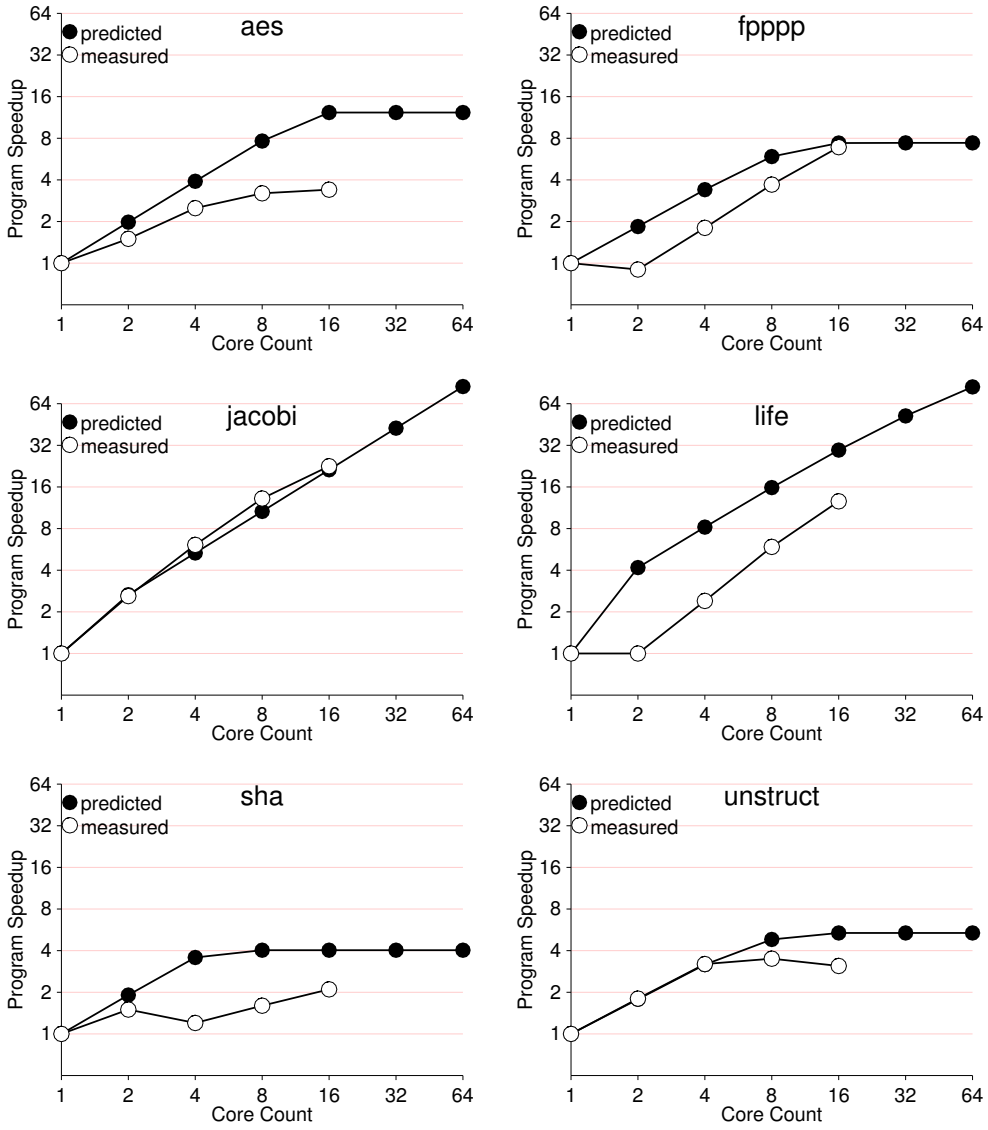


Figure 3.3: **Predicted and Measured Speedup for RAW Benchmarks on RAW hardware.** Kismet models the MIT Raw processor and RawCC, targeting the exploitation of ILP. From low- to high-parallelism benchmarks, Kismet provided appropriate upper bounds. This successful speedup prediction results from Kismet's ability to isolate ILP from other forms of parallelism based on summarizing hierarchical critical path analysis.

machine classes. Finally, we analyze the impact of novel techniques featured in Kismet: expressible self-parallelism, cache-aware prediction, and summarization.

3.4.1 Methodology

Kismet's goal is to provide realistic upper bounds on the parallel performance of serial programs. Our results will therefore focus on examining the tightness of these upper bounds on a wide range of benchmarks on several different platforms, both real and theoretical.

In our evaluation, we worked hard to address threats to validity by evaluating Kismet's performance across three very different architectures and by comparing against third-party parallelized codes from three benchmark suites, including both low and high parallelism applications.

We selected benchmarks using two primary criteria. First, the set of benchmarks needed to display a range of parallelism: from super-linear speedup down to very limited speedup. Second, the benchmarks needed to have either 1) a parallel implementation that could be used to gather real results or 2) published performance results from a variety of sources. Programs that are highly parallel tend to have a parallel implementation available while those with low amount of parallelism tend not to have parallel implementations available, possibly for reasons of vanity.

The selected benchmarks came from three benchmark suites, each targeting a different platform. Here we overview these suites, describing the amount and types of parallelism available and describing the s.pdf necessary to obtain our results.

- **Raw.** We modeled RawCC's ILP exploitation on Raw as described in Section 4.4. Kismet's estimates are compared against speedup numbers reported in [M. 04]. These benchmarks range from non-scalable to scalable.

  As mentioned before, RawCC utilizes loop unrolling to increase the amount of ILP. Unrolling also enables serial optimizations such as constant propagation and common sub-expression elimination. To control for these factors

during profiling, Kismet uses LLVM to unroll the loops before static instrumentation.

- **SpecInt2000.** SpecInt2000 benchmarks are widely known to have extremely limited parallelism. Luckily, a wide range of proposed parallelization systems—especially those using speculative parallelization—have attempted to parallelize these benchmarks, providing a fertile source of published results. We chose to examine the benchmarks from this suite that have most frequently been the target of parallelization, namely bzip2, gzip, mcf, twolf, and vpr.

  In general, these benchmarks are hard to parallelize due to complex dependence patterns in DOACROSS loops. The speedup numbers reported in literature typically required heroic code transformations, and often involved special speculative hardware support or simulation-only experiments [ZMLM08, ROR+08, PO05, KRL+10, ZL10]. To approximate the machine models in those aggressive scenarios, we modified the Multicore-OpenMP model described in Section 4.4 so that it allows the exploitation of both DOALL and DOACROSS with zero parallelization overhead. Even with these permissive settings, Kismet is able to create strong bounds.

- **NAS Parallel Bench (NPB).** In contrast to SpecInt2000, NPB [BBB+91] generally consists of benchmarks with large amounts of easy-to-exploit parallelism. We use the Multicore-OpenMP predictor targeting only DOALL parallelism with parameters for a 64-core system. We measured speedup with third-party parallelized version [Uni] of NPB, running these parallel versions on the 32-core AMD system described in Table 3.1. For all NPB benchmarks, we used the 'A' input data set during both profiling and execution of the parallel versions.

What are "Correct" Speedup Predictions? In our evaluation, we employ benchmarks that were parallelized by third-party experts. To the extent that the benchmarks have been widely used in the research community, we have a reasonable expectation that these parallelization efforts are not too far off from optimal.

To us, "correctly predict" means 1) that the actual speedup did not exceed the predicted speedup upperbound (i.e., Kismet's results correspond to actual empirical upperbounds) and 2) that the speedup experienced is close to Kismet's predictions (i.e. Kismet provides relative tight bounds.) To the extent that Kismet's bounds are not tight, it could be either due to insufficient modeling of machine constraints, or that there is remaining attainable speedup in the application.

3.4.2 Prediction Results

Raw Figure 3.3 shows predicted and measured speedup on RAW. In all benchmarks, Kismet correctly predicts the speedup trend in both high parallelism benchmarks [BFL+97] (jacobi, life) and low parallelism benchmarks (aes, fpppp, sha, unstruct).

Super-linear speedup is predicted and measured in both jacobi and life but only the former had actual super-linear speedup. These benchmarks consist mainly of DOALL loops, allowing unrolling to linearly increase the amount of ILP. In contrast, unstruct also benefits from unrolling and serial optimizations, but its loops are DOACROSS, limiting unrolling's effects and limiting scalability. For the remaining benchmarks, aes, fpppp, and sha, unrolling was ineffective as the parallelized regions were functions rather than loops.

Kismet correctly bounded the speedup for all benchmarks except jacobi, which slightly outperformed Kismet's estimates. This anomaly can be attributed to the fact that including more cores from the Raw processor increases the number of registers, leading to decreased memory system delays; Kismet did not incorporate this effect into its basic estimation model as its effect is generally negligible.

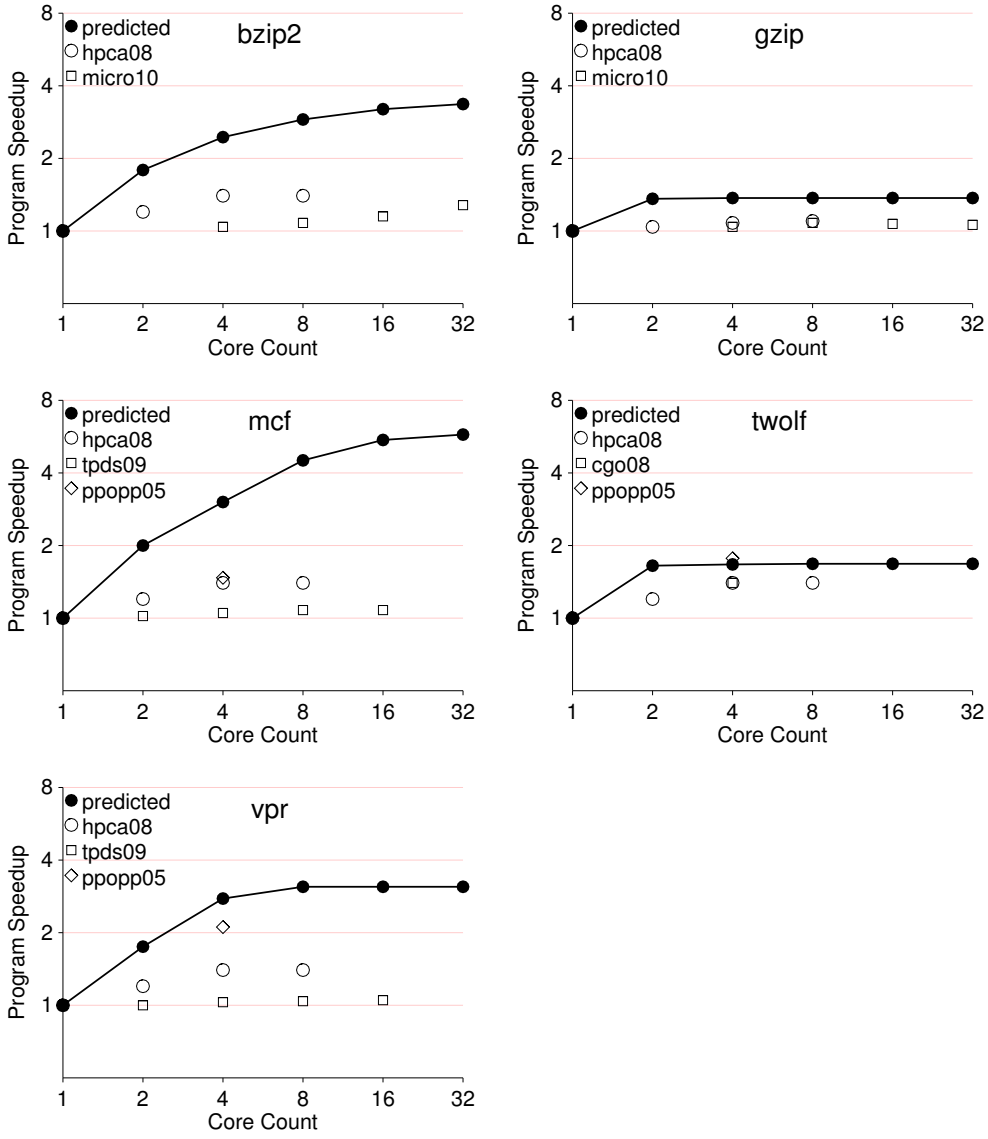


Figure 3.4: **Predicted and Reported Speedup in Low-Parallelism SpecInt2000 Benchmarks using third-party published results.** Kismet correctly captures the low parallelism in SpecInt2000 benchmarks, providing tight speedup upper bounds. Reported speedup numbers are from multiple sources that applied aggressive hardware/software techniques to extract parallelism from these benchmarks [ZMLM08, ROR+08, PO05, KRL+10, ZL10].

SpecInt2000 Figure 3.4 shows Kismet's speedup estimates and speedup numbers gathered from third-party efforts running on aggressive hypothetical hardware [ZMLM08, ROR+08, PO05, KRL+10, ZL10]. These results confirm the generally-held belief that SpecInt benchmarks are fundamentally limited in their parallelism. Kismet predicted low speedups, plateauing at a speedup of 2 to 4 for all benchmarks except mcf. The reported results conform to Kismet's upper bounds.

NAS Parallel Bench (NPB) Figure 3.5 shows predicted and measured speedups for the benchmarks in NPB. As expected,—based on the abundant, easily-exploitable DOALL parallelism of these benchmarks—Kismet estimated relatively high speedups in all benchmarks except is. The lower amount of speedup in is results from it having only a limited amount of execution spent in parallel regions.

For ep and lu, measured speedup was very close to predicted speedup. Even though the communication cost on multicore processors typically limit the scalability of benchmarks, these benchmarks' speedup continued to scale as they do not rely on inter-core communication.

cg is an interesting benchmark that exhibits super-linear speedup in both predicted and measured speedup, thanks to Kismet's cache-aware performance model. We will examine cg in more detail later in the results section.

mg and sp scale up to 8 cores, but their speedup starts to decrease from that point. The drop in performance can be attributed to shared-memory related overhead that is not captured by Kismet's parallel execution time model. These benchmarks share data across cores and a data location is written by multiple cores, greatly increasing the sharing overhead. The gap between predicted and measured performance in these benchmarks might be closed when innovations in parallel computer architecture reduce the cost of shared-memory based communication. Alternately, more advanced modeling of coherence traffic in Kismet could be of assistance.

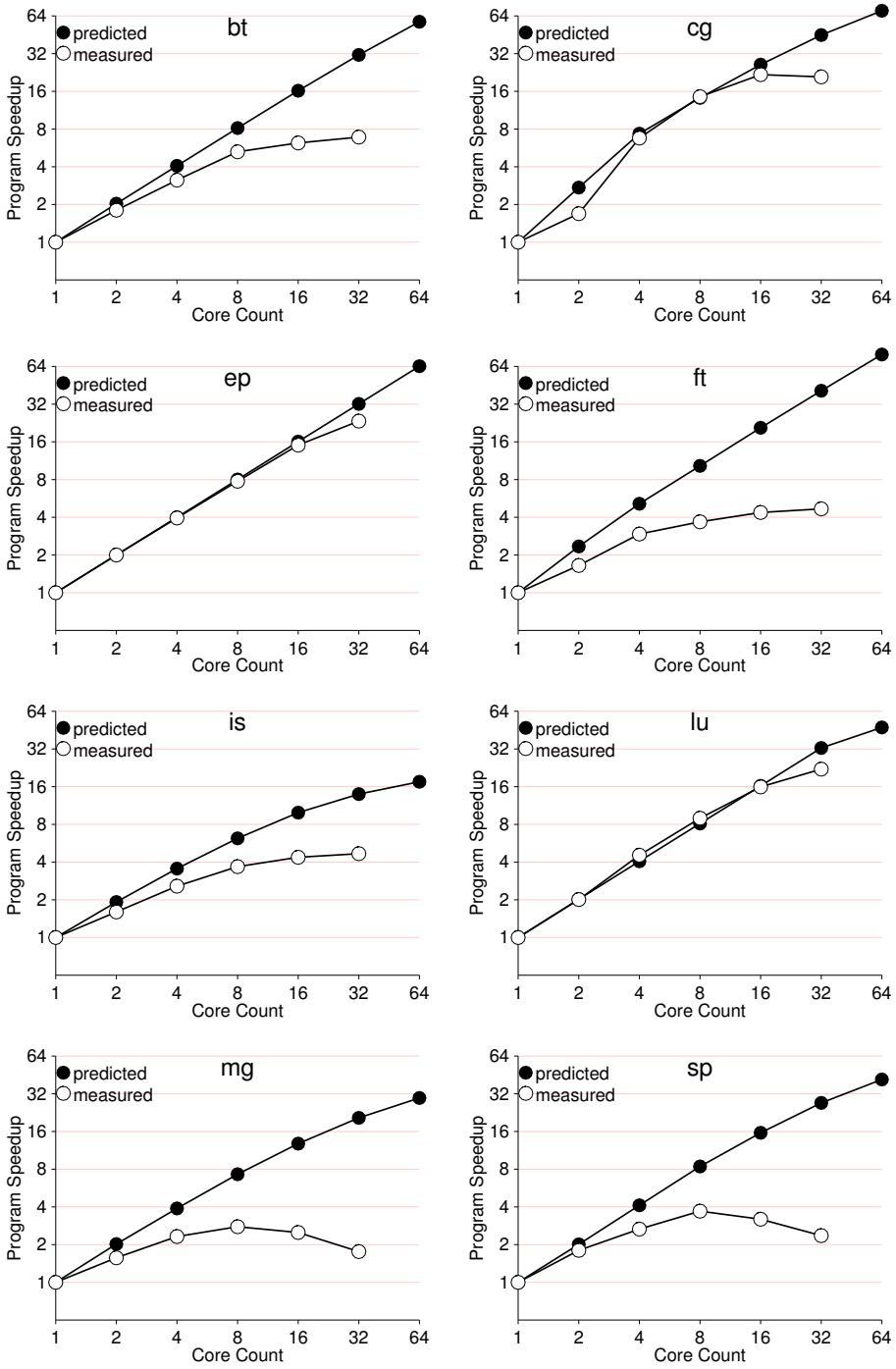


Figure 3.5: **Estimated and Measured Speedup of NAS Parallel Bench on 32-core AMD Multi-core System.**

Table 3.2: **Estimated Speedup with and without Expressible Self-Parallelism.** ESP helps the tightening of speedup estimates by providing only expressible parallelism to Kismet. In these benchmarks, ESP successfully reduced the speedup estimates by 363.2X, showing that it is indeed a central component in speedup estimation.

Benchmark		Estimated Speedup		
Suite	Name	Without ESP	With ESP	Ratio
RAW	jacobi	8649	53.81	160.7X
	life	26840	153.73	174.6X
	sha	4.81	4.71	1.0X
	fpppp	1190	98.74	12.1X
	aes	39547	150.95	262.0X
	unstruct	4416	8.22	537.2X
SpecInt2000	bzip2	17.4	3.39	5.1X
	gzip	4.27	1.37	3.1X
	mcf	67.12	5.92	11.3X
	twolf	11.35	1.68	6.8X
	vpr	15.77	3.1	5.1X
NPB	bt	161650	64.46	2507.8X
	cg	275	171.06	1.6X
	ep	93.69	38.67	2.4X
	ft	10709	151.92	70.5X
	is	565	37.53	15.1X
	lu	43845	52.98	827.6X
	mg	2478	87.35	28.4X
	sp	147873	65.18	2268.7X
Total	mean	23592	61	**363.2X**
	geomean	878	25	**34.5X**

3.4.3 Impact of Expressible Self-Parallelism (ESP)

One of HCPA's major advantages over traditional CPA is its ability to localize parallelism using the self-parallelism metric. Kismet further improves the utility of self-parallelism by introducing the concept of expressible self-parallelism (ESP), a filtering step that removes self-parallelism that is unexpressible by the target system. To quantify the impact of ESP, we compared the estimated speedup

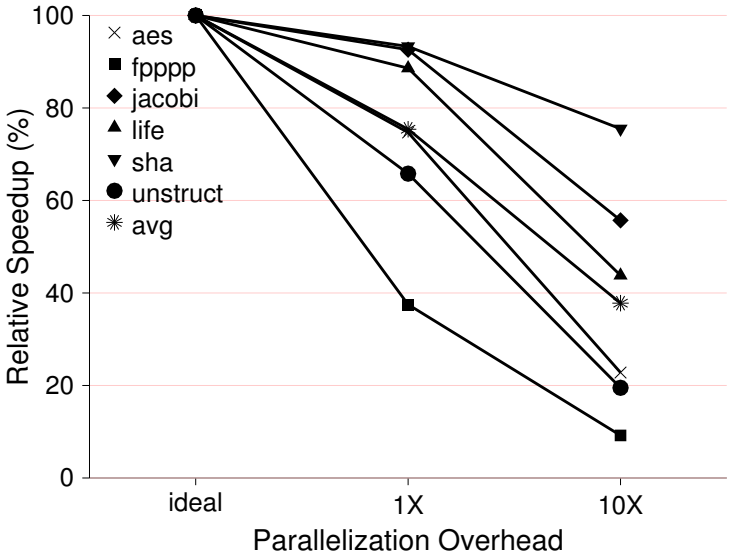


Figure 3.6: **Impact of Parallelization Overhead.** The parallelization of Raw benchmark is fine-grained and susceptible to larger parallelization overhead. To measure the impact of parallelization overhead in Raw benchmarks, we increase the parallelization overhead from zero to the default overhead, and then to 10X of the default overhead. The relative speedup drops from 100% to 75% and then to 37.8%. This experiment demonstrates the proper modeling of parallelization overhead is critical in fine-grained parallelization.

with and without ESP in all benchmarks. We assumed zero overhead and infinite cores in the speedup estimation, in order to isolate the impact from ESP from other speedup limiting factors.

Table 3.2 shows the estimated speedup number with and without ESP. By honoring only unexpressible parallelism, Kismet tightens the speedup upper bound by up to 2508X, with an average reduction in speedup of 363.2X. The results confirm that ESP is an essential part in speedup estimation system.

3.4.4 Impact of Parallelization Overhead

One important issue in parallelization is parallelization overhead. The higher the parallelization overhead is, the coarser the granularity of paralleliza-

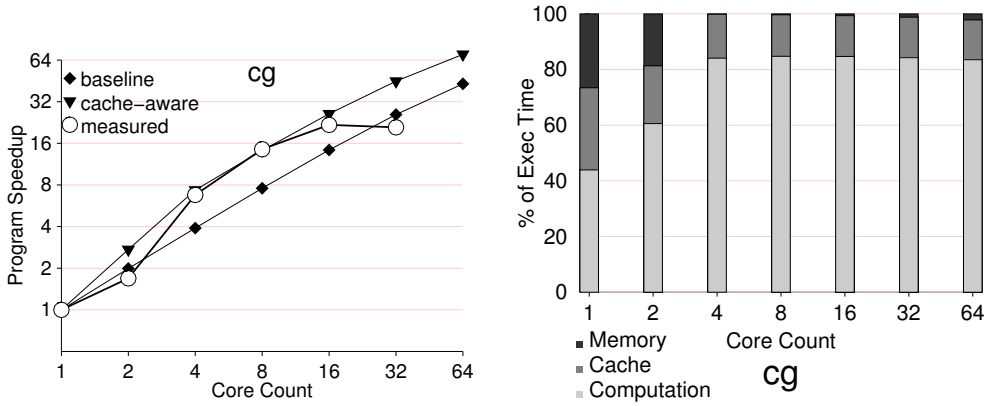


Figure 3.7: **Impact of Cache-aware Estimation in cg Benchmark.** The baseline estimation fails to predict the super-linear speedup of cg. By incorporating potentially reduced cache miss rates in a parallel execution, cache-aware estimation successfully predicts the super-linear speedup. Execution time breakdown clearly shows the time spent in cache and memory is considerably reduced from two-core to four-core execution.

tion should be to offset the overhead. In other words, overhead modeling is more important in programs with fine-grained parallelism.

To see the impact of the overhead in fine-grained parallelism, we estimated the speedup of Raw benchmarks with three different overhead cost - zero overhead, default overhead $(3 + 2 * logN)$, and 10X of the default. Among three benchmark groups, Raw is the most fine-grained.

Figure 3.6 shows the impact of parallelization overhead in estimated speedup. Even with Raw's low parallelization overhead based on highly optimized static network [TLAA05], it drops the speedup to 75% of ideal case where parallelization overhead is zero. Increasing the parallelization overhead by 10X further reduces the speedup to only 37.8% of the ideal case. This experiment explains why Raw processor designers strove to achieve a low latency in their network design.

3.4.5 Impact of Cache-aware Speedup Estimation

Cache-aware time estimation model incorporates potentially reduced cache service time caused by increased cache sizes when additional cores are used in execution. The top part of Figure 3.7 demonstrates the effectiveness of cache-aware estimation shown on the `cg` benchmark. Without cache-awareness Kismet predicts linear speedup, but measured speedup exhibits super-linear speedup. In cache-aware prediction, Kismet incorporates varying cache miss rates gathered from Cachegrind [NS07b] for each core configuration, correctly predicting super-linear speedup of `cg`.

The lower part of Figure 3.7 shows the breakdown of execution time on different number of cores. As the core count switches from one to two and from two to four, the portion of cache and memory service time is significantly reduced. When the cache miss rate does not change, the portion for cache and memory should remain the same. Indeed, switching from 1 to 4 cores, L1 cache miss rate drops from 23.3% to 6.5%, and the last level cache miss rate drops from 6% to 0.1%.

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 "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 primary investigator and author of this paper.

Chapter 4

Reducing Overhead with Efficient Vector Shadow Memory

Memory shadowing is a general technique that has been used in many different applications: from memory analysis [SN05, BZ11] to computer security [CZYH06, QWL+06, XBS06]. Memory shadowing allows metadata, referred to as a *tag*, to be associated with each memory address in a program. These tags vary in function across each dynamic program analyses. For example, tools such as MemCheck [SN05] have a single bit of metadata associated with each memory location to indicate whether that address has been properly initialized. Memory shadowing is also widely used in computer security where it performs taint tracking. Tools such as TaintTrace [CZYH06] shadow memory locations with a single bit of metadata to indicate whether a memory address has been written with data from an untrusted source.

Most recent work in memory shadowing has focused on optimizing the execution time of analyses that associate a single tag with each memory address. However, some analyses require a vector of shadow tags to be associated with each memory address. This *vector shadow memory*, or VSM, frequently causes an explosion of memory usage which makes many of these analyses impractical due to out-of-memory errors. In these systems, memory usage supplants the issue of execution time because it prevents the analysis from being run at all, let alone slowly.

This chapter examines the optimization of VSMs to enable a class of *differential dynamic analyses*, or *DDA*. Differential program analysis focus on the differences in properties between similar programs [WE03]. We focuse on differential program analyses that are dynamic and that are *auto-differential*; that is, the "similar programs" are actually just different nested regions (e.g. loops or functions) of the same program. Typically, a sub-analysis is applied to parent and child regions, and the results are summarized to eliminate the need for large log files. Later in time, the parent and child values are differenced with an abstract difference operator to produce the final result of the analysis.

As of today, auto-differential program analyses have been employed most often for guiding the optimization of programs, especially when it pertains to trying to attach performance-affecting attributes to nested regions in a piece of code. Perhaps one of the oldest such analyses is found in profilers, which quantifies the amount of program execution time spent in a program region but not its children. This analysis is simple – accumulate the time spent in a parent region, and then subtract the accumulated time in the children regions. No shadow memory is required.

Recently, the drive to aid parallelization has renewed interest in tools that can provide information that assists in the partitioning and transformation of programs to optimize their parallel execution. One such tool is Kismet,which outputs parallel speedup upperbound estimates given serial source code and a set of representative inputs. Kismet employs *Hierarchical Critical Path Analysis*, or *HCPA*, which attributes each nested program region with the amount of parallelism that is attributable to that region alone. Interestingly, HCPA is a differential form of the venerable critical path analysis [Kum88], or CPA, which traces the earliest time that every value in a program could have been produced, based on dependences. CPA's final output is the oldest such time in the program, the critical path. By summing the total work in the program, and dividing by the critical path, we get the average parallelism in the program. CPA likely is the first dynamic analyses to use shadow memory implementation in history. We say HCPA is a differential form of CPA because it applies CPA to every nested region of the program, and

then uses an approximation function to difference the CPA results and calculate the *self-parallelism*, that is, the parallelism attributable to a region independent of its children. Of course, this nested application of CPA requires a VSM to track CPA values for every operand in the context of every nested region.

Differential dynamic analyses may have many other applications but the lack of efficient frameworks for this type of analysis has likely slowed its adoption. In this chapter, we use as running examples two differential dynamic analyses that represent two extreme points for DDAs; HCPA as an example of an extremely heavy weight DDA, and another analysis, which we call *footprint analysis*, as an example of a lightweight analysis. Footprint analysis tracks the memory addresses touched by every nested region in a program. From footprint analysis, we can ascertain the memory usage of a subregion in a program, relative to its children. This is useful information for programmers trying to partition code across distributed non shared-memory systems, such as in the IBM Cell processor, or in a Non-Uniform Memory Access (NUMA) shared-memory machine, for when we want to estimate coherence bandwidth requirements, or estimate cache footprint of a region of code to prove non-interference properties.

4.0.6 Shadow Memories for Differential Dynamic Analyses

Non-differential shadow memory infrastructures such as those in [NS07b, ZBA10a, ZBA10b] need only provide one shadow tag per address in the program. In contrast, differential analyses require a vector of tags for each address, implementing the vector shadow memory. The size of this vector depends on the number of shadows that need to be tracked, which can vary with program execution (i.e, the current depth in the nested region graph determines this number.) Additionally, auto-differential analyses based on nested program regions impose an additional challenge: when execution leaves a region, a shadows needs to be deleted; and when execution enters a region, a new shadow needs to be created. A naive implementation involves scanning all of the shadow memory; instead, a more optimized implementation uses a *version ID* which identifies which dynamic region instance that the recorded data is applicable for. When the DDA goes to

access the shadow tags, it checks the version ID for each shadow to verify that the value is not stale.

This VSM implementation approach must store tags that scale as the product of the original memory footprint of the program and the number of unique regions in the program. It also must store the version IDs, which in some cases, such as storing a single bit for each memory location, can take more space than the data itself.

Table 4.1 shows the memory requirements for a naive VSM implementation implementing HPCA. As can be seen, the memory expansion factor (that is, the ratio of memory usage of the shadowed version of the program to the native version of the program), ranges from 25× to 149×. For the benchmarks listed, which have relatively small footprints as they are from older benchmarks, memory usage expands from 324 MB to 15.7 GB, which easily exceeds the amount of RAM in many software development machines. With this baseline implementation, DDAs will only be applicable if inputs sizes are drastically reduced. The typical result of a DDA using a simple VSM implementation applied to standard inputs will be an out-of-memory error. Clearly, VSM requires new techniques to address these new overheads. In this chapter we will reduce this memory expansion to a geometric mean of 5.2×.

4.0.7 Skadu's Approach

In this chapter, we present Skadu, a vectored shadow memory implementation (VSM). Skadu is the first general framework for VSM, which allows it to effectively handle differential dynamic memory analyses. Skadu takes advantage of the hierarchical structure of programs to minimize the overheads associated with VSM. This hierarchical structure allows Skadu to reduce memory overhead through sharing of shadow memory while keeping runtime overheads low. Skadu also exploits the difference in average region lifetime across levels of the region hierarchy to reduce the memory overhead of both short-lived and long-lived regions.

Skadu introduces several techniques that greatly reduce the overhead required for VSM. Skadu partitions tags according to their associated region's place-

Table 4.1: **Motivation: Vector Shadow Memory Overheads of the Hierarchical Critical Path Analysis (HCPA) Differential Dynamic Analysis.** Vector Shadow Memory by default has extremely high memory overheads, which in many cases make them intractable to run on standard workstations, unless input sizes are drastically reduced. For an assortment of memory-intensive Spec 2000 REF and NAS Parallel Bench (NPB) B inputs, the shadow memory analysis required a geometric average of 15.7 GB of memory, and causes an average memory expansion of 49× expansion makes it impractical to run standard sized inputs on many developer's machines. We reduce this memory expansion to a geometric mean of 5.2×.

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×

ment in the region hierarchy. This partitioning allows efficient sharing of shadow memory among regions in the same level of the program hierarchy. Partitioning also enables lightweight garbage collection of stale tags. Skadu utilizes a tag vector cache to keep frequently used tag vectors in a format that minimizes access time. This cache allows the user to easily trade increased memory overhead for improved performance. It serves as a kind of nursery that eliminates the need to allocate short-lived regions in long-term shadow storage. Skadu reduces the memory overhead of long-term tag storage by compressing infrequently used tags. Skadu also introduces two novel techniques for reducing the overhead associated with validating tags, a requirement stemming from the sharing of shadow memory between multiple program regions.

4.0.8 Evaluating Skadu

We have implemented two differential dynamic analyses in order to evaluate Skadu's effectiveness at reducing overheads. First, a memory footprint profiler tracks the amount of memory used by every function and loop in a program. This analysis is useful to a range of applications such as scheduling of programs on a heterogeneous multi-core processor. Second, a parallelism profiler, hierarchical critical path analysis (HCPA), determines the average amount of parallelism in every function and loop in the program. While both analyses utilize Skadu's VSM infrastructure, they demonstrate opposite ends of the analysis spectrum: the memory footprint profiler is a relatively lightweight analysis with only 1-bit tags while the heavyweight HCPA uses with 64-bit tags.

Results from our implementations of the memory footprint and parallelism profilers show that Skadu reduces memory overhead by 14.2× for memory footprint profiling and by 11.4× for HCPA versus a baseline implementation. We also examine the effect of vector caching as well as selective compression on both memory and performance overhead.

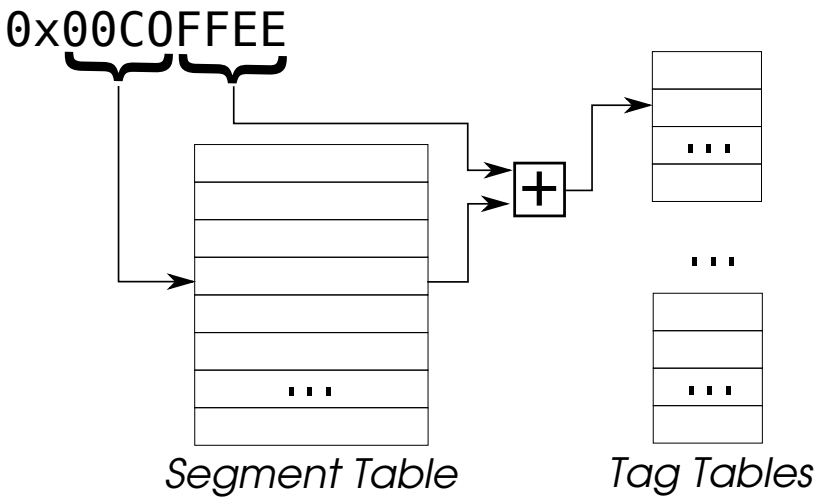


Figure 4.1: **Traditional Memory Shadowing Organization.** The memory address is used as an index into a two-level page table that contains the metadata associated with that address. To support 64-bit addresses, a three-level table may be used.

4.1 Overview and Challenges

In this section we will introduce the challenges facing vectored shadow memory (VSM). We start by introducing traditional shadow memory organization before describing the region-based differential dynamic analysis employed in this chapter. Finally, we overview the techniques that Skadu uses to create an efficient VSM framework for the differential dynamic analyses.

Traditional Memory Shadowing Technique In traditional shadow memory infrastructures, each memory address has an associated shadow memory address. Each shadow address may contain some metadata (or tag) about the associated memory address. Figure 4.1 shows a simple shadow memory organization where shadow memory is accessed via a two-level table: first the segment, then the tag. The size of the tag table entry can range from tiny to large: it is common to see one bit tags in taint tracking infrastructures while applications such as hierarchical critical path analysis introduced in Chapter 2 require 64-bit tags. While the details vary between existing memory shadowing frameworks, they generally follow this

basic organization.

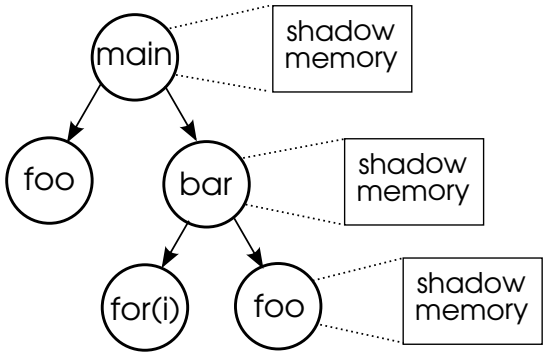
Region-Based Differential Analysis Traditional memory shadowing requires tracking only a single region of the code, usually the whole program (i.e. the `main` function). The differential analyses we consider in this chapter require separate dynamic sub-analyses to be applied to each nested region of the program. We define a region to be any single-entry piece of code but we will focus on two particular types of regions: functions and loops. During the execution of a program, the regions of a program form a natural hierarchy. Figure 4.2 demonstrates this hierarchy (shown in the form of a region tree) for an example piece of code. Each node in the region tree is a dynamic region while an edge from A to B indicates that B is a child (or subregion) of A.

In the region-based differential analysis, each region is profiled independently of the others. Conceptually, this means that each region has its own address space and therefore requires its own shadow memory address space. A naive extension of the memory shadowing shown in Figure 4.1 might be to make each entry in the tag table correspond to an array or list. However, the number of entries needed for each address depends on the number of dynamic regions in the program and therefore cannot be determined statically. Neither an array- or list-based approach are desirable in this situation. The array-based approach would either need to be radically over-allocated (if statically allocated) to ensure room for all regions or would need to be dynamically allocated, potentially leading to prohibitive performance penalties. The list-based approach suffers from the same drawbacks as the the dynamically allocated array approach.

Efficient Management of Multiple Shadow Address Spaces 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. Skadu takes advantage of this hierarchical property to minimize the memory overhead associated with multiple shadow address spaces. As shown in Figure 4.3, all regions in each level of the region tree are mapped to a single tag. In other words, shadow address space is shared amongst every region in a level.

```
1  int main() {
2      foo();
3      bar();
4      ...
5  }
6  void foo() {
7      ...
8  }
9  void bar() {
10     for(i=0..10)
11         x++;
12     foo();
13 }
```

(a) Code Snippet.



(b) Corresponding Region Tree.

Figure 4.2: **Region Hierarchy Overview.**The pseudo code in (a) results in the region tree shown in (b). Each region has an isolated shadow memory address space, as shown in (b). Skadu introduces several techniques to reduce the overhead associated with maintaining these separate address spaces.

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 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. This solution also requires additional space overhead for tracking "dirty" tags.

To avoid the cleaning costs of the naive scheme, we can use the version-based approach. This version-based approach attaches metadata to each tag to indicate which region owns that tag. Before a tag is used, the version is checked to ensure that the region accessing the tag is its owner. The tag is invalidated if the version does not match. Unfortunately, this technique requires a significant amount of space for tracking versions. Skadu introduces several novel techniques

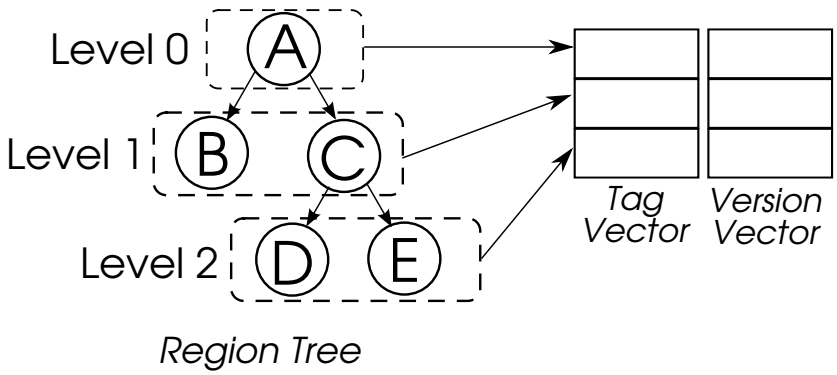


Figure 4.3: **Level-based Sharing of Shadow Memory.** The hierarchical nature of regions ensures that any level in the region tree will have at most one active region. Skadu uses this property to enable reuse of physical shadow memory space between multiple regions of the same level. This reuse requires 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. Skadu introduces two novel versioning systems that reduces the memory overhead of versioning from $O(n)$ to $O(1)$.

to minimize this cost; these techniques are described in detail in Section 4.2.

Utilizing Region Hierarchy to Reduce Overhead The definition of a region ensures that the size of regions monotonically decreases as you go from the root of the region tree to its leaves. As a result, "deep" regions tend to be much smaller than "shallow" regions and have a smaller memory footprint. Skadu leverages this property by introducing two novel memory shadowing features: level tables and tag vector caching.

Skadu introduces a level table into the basic shadow memory organization shown in Figure 4.1. This level table acts to partition tags according to the level in which their associated region resides. Each entry in the level table points to a tag table that is associated with a given level in the region tree. This organization allows Skadu to maintain the minimum number of tag tables for each active level, capturing the main benefit of the dynamic array or list-based approach without the space overhead associated with those approaches.

The level table organization has the added benefit of enabling efficient shadow memory garbage collection. Skadu includes a garbage collector that periodically scans level tables to determine which levels are no longer active. Tags in these inactive levels can be deallocated so as to reduce the memory overhead. The garbage collector allows lazy deallocation of tags, moving this deallocation away from the region exits and therefore reducing the performance overhead. Garbage collection could also done in parallel with normal analysis, further reducing runtime overhead.

While level tables help minimize the space overhead associated with the differential dynamic analysis, they could potentially add to performance overhead as multiple table traversals are required to access a single address' tag vector. Skadu minimizes this impact by introducing a tag vector cache. Skadu's cache not only k.pdf the tag vectors for frequently used memory addresses but also uses an array-based organization that minimizes access time. This cache offers a trade-off of performance and space: increased the cache size leads to increased performance. The user can exploit this trade-off to maximize performance based on the amount of memory available to them.

One further consequence of deeper regions being smaller than shallower regions is that the lifetime of tags in those deeper regions is much shorter. Skadu takes advantage of this by using a novel write-back policy. When a tag vector is evicted from the cache, the version metadata is first checked to see if any of the tags are out-of-date and therefore invalid. Invalid tags are simply discarded since they will never be used again. As a result, Skadu avoids allocating tag tables for stale tags, thus reducing the space overhead.

The tag vector cache makes the average shadow memory operation fast by storing frequently used tags in a compact, quickly accessible form. By making uncached accesses a rare event, Skadu is able to further reduce memory overhead by compressing tags residing outside of the cache. Other than a small number of recently used tags, all tags are compressed, minimizing the memory overhead while only moderately impacting performance. The number of uncompressed tags outside of the cache offers another trade-off of memory and performance: more

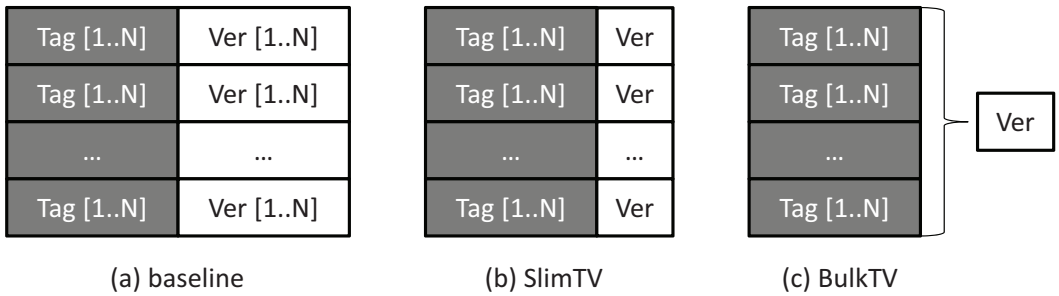


Figure 4.4: **Space Overhead of SlimTV and BulkTV.** Compared to the baseline where each level requires version information, SlimTV shares a version information for all levels. BulkTV further reduces the space overhead by sharing a single version number across a range of memory addresses.

uncompressed tags increase memory overhead but reduce the runtime penalty. Details of compression as well as the complete overview of Skadu's shadow memory architecture will be given in Section 4.3.

4.2 Efficient Tag Validation

Tag validation is an essential operation in Skadu. As described in the previous section, Skadu uses version information to determine ownership of tags. Unfortunately, naive version management is scalable neither in memory overhead nor in performance; it has $O(n)$ space and time complexity where n is the depth of the deepest region that accesses a specific memory address. This section introduces two techniques that enable efficient tag validation: Slim Tag Validation (SlimTV) and Bulk Tag Validation (BulkTV). Figure 4.4 compares the space overhead of these techniques against a baseline implementation: together they make the space requirements of tag validation almost negligible and significantly lower the runtime overhead.

4.2.1 Baseline Implementation

Our baseline implementation features a simple procedure to check tag validity that is based on a design property of the shadow memory: sharing of shadow

memory is limited to regions within the same level of the region tree. The baseline implementation utilizes this sharing property by assigning a unique ID to each region in a level. The unique IDs of all active regions are stored in a version vector associated with a tag vector whenever that memory is updated. This stored version vector is then used for tag validation on each read: if there is a mismatch between the ID of the current active region in a level and the ID for that level in the version vector, then the tag for that level is invalid.

The baseline implementation suffers from the drawback that it requires storing a version vector for every shadow memory address. This storage requirement leads to an $O(n)$ space overhead just for tag validation, where n is the depth of the region. This approach also incurs a large number of memory loads and stores from reading/writing version vectors, resulting in higher runtime overhead.

4.2.2 Slim Tag Validation (SlimTV)

Skadu introduces a new tag validation technique known as Slim Tag Validation (SlimTV). SlimTV improves upon the baseline implementation by eliminating the need to store a version vector with each tag vector; only a single value needs to be stored. This technique not only reduces the space overhead from $O(n)$ to $O(1)$ but also eliminates the excessive loads/stores associated with accessing the stored vector, greatly reducing the runtime overhead.

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. During the access of a tag vector only the ID of the most recently entered, active region is stored. The current stored ID is compared with the vector of IDs associated with the currently active regions whenever the shadow address is accessed. Active regions with IDs greater than the stored ID started *after* that region and are therefore invalid. 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.

Figure 4.5 provides an example of SlimTV's tag validation. In this example a memory address is written to in the region with version 4. This single version

number is then stored along with the tags for each active region. This same address is read later in the region with version 7, at which time the active version vector is $\langle 1,5,6,7 \rangle$. Of the active regions only the first (1) has an ID less than the stored version (4); starting from region 5, all other regions are invalid.

Theorem 1. *Suppose $V(R)$ is the active version vector of region R, the current region is R_{new}, the stored version in shadow memory is v, and the stored tag vector is T. $T[i]$ is valid if and only if $V(R_{new})[i] \leq v$.*

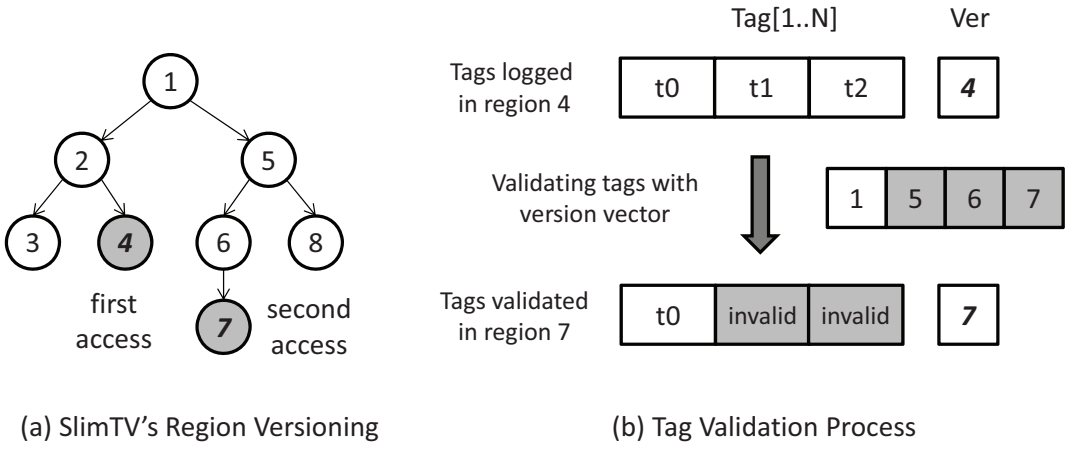
Proof. Assume for contradiction that $V(R_{new})[i] > v$ but that its associated tag is valid. Let R_{old} be the region that stored both the tag and v. Because $T[i]$ is valid, $V(R_{old})[i] == V(R_{new})[i]$; therefore $V(R_{old})[i] > v$. However, this is a contradiction because by rule v is the largest value in $V(R_{old})$. □

4.2.3 Bulk Tag Validation

Skadu's SlimTV technique reduces the memory overhead to a constant factor but this may still result in significant memory overhead. For example, when shadowing every byte of memory, the overhead incurred from an 8-byte version identifier is 8X. Skadu therefore introduces an additional technique, Bulk Tag Validation (BulkTV), that can reduce the memory overhead to a negligible amount while additionally reducing the runtime overhead.

BulkTV's key idea is to amortize the tag validation process' overhead across many addresses. BulkTV accomplishes this amortization by using only a single version number for a page of shadow memory, performing tag validation for all entries in the page whenever a single address is accessed. The effectiveness of BulkTV is clearly tied to the size of the page: the bigger the page, the bigger the benefit. For example, a modest 4KB page leads to a drastic reduction of 4096X when shadowing every byte.

BulkTV can also have a significant impact on the runtime overhead of tag validation. This impact stems from two competing factors. On one hand, there is additional overhead associated with validating a whole page, especially when only a fraction of locations in that page will be used. This factor ultimately depends



(a) SlimTV's Region Versioning (b) Tag Validation Process

Figure 4.5: **An SlimTV Example.** (a) SlimTV exploits the ordering encoded in the version ID of dynamic regions in the program. (b) Illustrates the tag validation process. Suppose a memory location is accessed in region 4 and later in region 7. After the access in region 4, tag[0:2] will be logged with the region's version number 4. When the same address is accessed in region 7, the version stack [1, 5, 6, 7] is compared against the stored version number 4. From the comparison, SlimTV detects that only region level 0 started before the previous tags were written. SlimTV therefore invalidates tag[1:2] and updates the version field in the shadow memory. SlimTV shares a version information for all levels. BulkTV further reduces the space overhead by sharing a single version number across a range of memory addresses.

on the locality of memory accesses: higher locality will lead to fewer wasted tag validations.

On the other hand, BulkTV greatly reduces overhead through a decrease in the most costly single operation in tag validation: finding the highest valid level of tags. Finding this highest valid level involves walking through the current version vector, an operation linear in the size of the vector. BulkTV still requires n comparisons for each access in the worst case but the average case requires significantly fewer comparisons. In the best case, only a single comparison is needed; this occurs when a memory location in the same page is accessed in the same region. BulkTV performs the comparisons starting at the end of the version

vector, meaning that slight differences in version number since the last access will incur significantly fewer than n comparisons. The reduction is again dependent on the locality exhibited throughout the program but our results in Section 4.5 show that only a moderate amount of locality is needed to result in a net reduction in runtime overhead.

4.3 Vectored Shadow Memory (VSM) Architecture

Traditional shadow memory infrastructures have gone to great length to minimize the runtime overhead of memory shadowing. Runtime overhead continues to be a serious concern when extending shadow memory to support vectored tags but memory overhead is potentially more limiting. Skadu introduces a novel shadow memory architecture that balances the sometime conflicting requirements of low memory and runtime overhead for vectored shadow memory. In this section we describe this novel architecture.

4.3.1 VSM Architecture Overview

Skadu's architecture separates fast, short-term shadow memory from space-efficient, long-term shadow memory. Figure 4.6a shows the interaction between the architecture's two main components, TVCache and TVStorage, which correspond to the split between short-term and long-term tag vector storage. This split allows Skadu to exploit the characteristic differences in lifetime and tag storage size across various levels of the region hierarchy.

Skadu initially places tag vectors in the TVCache, evicting them to the TVStorage only as needed. The TVCache is geared toward fast-access time; sized appropriately, it minimizes the number of accesses to the slower-access TVStorage. The TVStorage is designed for long-term storage and therefore attempts to minimize the memory overhead. It does this through a level-based storage infrastructure that facilitates lightweight garbage collection; this garbage collection

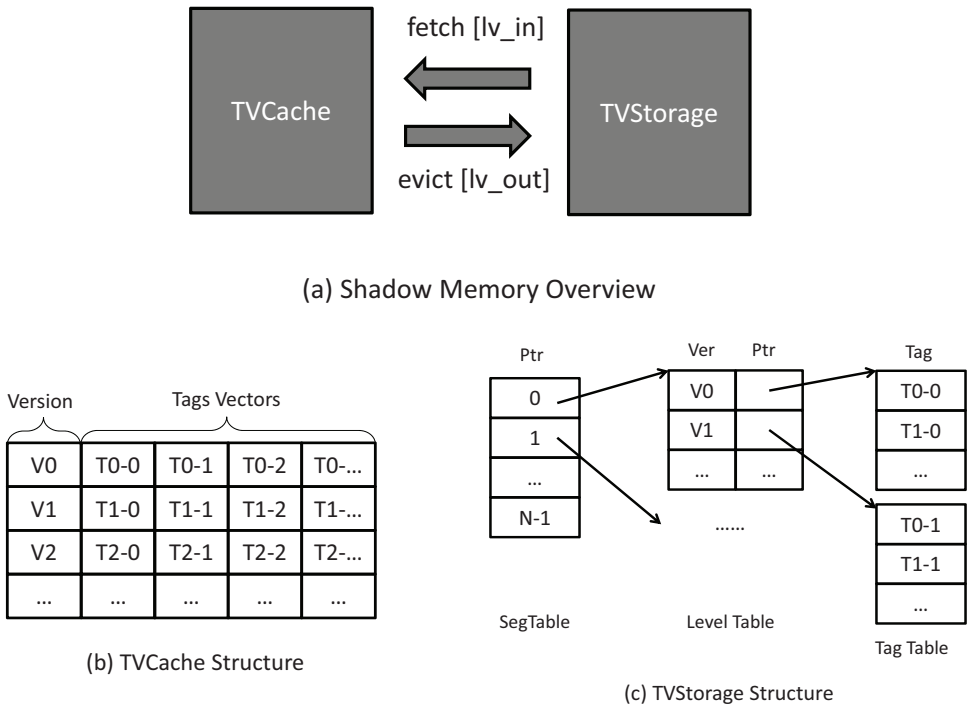


Figure 4.6: **Overview of Skadu Shadow Memory Organization.** (a) To exploit the memory footprint and liveness characteristics of hierarchical regions, Skadu uses a TVCache, reducing memory requirements and improving performance. (b) The TVCache is optimized for the performance, handling most shadow memory requests and allowing a memory-efficient organization of the TVStorage. (c) The TVStorage is optimized for low memory overhead with the addition of a level table. Paired with BulkTV, this three-level organization enables lightweight garbage collection.

reduces the runtime overhead of dynamic vector resizing. Skadu compresses tags in the TVStorage; because the TVStorage is accessed infrequently, the performance overhead of this compression is minimal. In the following sections, we will describe the components of Skadu's architecture in more detail.

4.3.2 Tag Vector Cache (TVCache)

The TVCache stores frequently used tag vectors, making the common case access time fast. These tag vectors are stored in an array format to further reduce

access time. The TVCache uses SlimTV for low-overhead tag validation but not BulkTV because "lines" in the TVCache do not contain the spatial locality required to make BulkTV profitable.

Figure 4.6b shows the structure of the TVCache–albeit slightly simplified with omitted metadata such as associated memory address and vector size. Each cache line contains the version and the tag vector associated with a memory address. All cache lines have the same vector size for better performance at the cost of possibly wasted memory. However, TVCache's memory requirement is very small compared to that of TVStorage because of the reduced address space it covers. The TVCache is direct mapped to reduce access time while still providing good hit ratios.

TVCache differs from traditional caches in that it not only improves runtime performance but also reduces memory overhead of long-term tag storage. This reduction in memory overhead is a result of the efficient write-back policy used by the TVCache. The TVCache tends to cache tag vectors long enough that short-lived regions have already exited by the time they are evicted. The TVCache validates all tags upon an evict, writing back to the TVStorage only those the are valid. This process mimics garbage collection, reducing the space used by the TVStorage.

4.3.3 Tag Vector Storage (TVStorage)

The TVStorage acts as a long-term backing store for tag vectors evicted from the TVCache. The TVCache handles most shadow memory accesses, allowing the TVStorage to focus on reducing memory rather than runtime overhead.

Figure 4.6c shows the structure of the TVStorage. The TVStorage utilizes a three-level structure that is similar to traditional shadow memory infrastructures [1] but with the novel addition of a level table. The TVStorage groups tags by their level rather than the address they shadow. This distribution of tags enables efficient garbage collection, exploiting the fact that tags become invalidated when

[1] Although not shown, this structure is easily modified to handle 64-bit addressing via an additional table before the segment table, similar to what was proposed in [ZBA10a].

regions–and therefore levels–are exited. The TVStorage also employs BulkTV: all entries in a tag table share a single version ID, which is located in the level table next to the tag table pointer.

The TVStorage organization enables invalidation of a whole tag table with only a single version number comparison. Skadu maintains a list of free tag tables: tag invalidation only requires sending off the tag table to be scrubbed and returned to the free list. This makes garbage collection extremely lightweight. Skadu employs a simple garbage collector that walks all the level tables in the TVStorage, invalidating and freeing tag tables as it goes along. This garbage collector allows Skadu to dynamically adjust the size of an address' tag vector with little-to-no performance overhead.

4.3.4 Tag Vector Compression

Skadu's TVCache-TVStorage organization facilitates the use of compression without significant overhead. The size of the TVStorage dwarfs that of the TVCache for all but the smallest programs: the TVCache is designed to be small enough to handle only the most frequently accessed addresses, leaving the TVStorage to store all other addresses. The TVStorage is the good target for compression because of its large size and relatively infrequent access.

Skadu balances the space savings of compressed tags with the performance of uncompressed tags: a small list of recently used level tables house uncompressed tag tables while all other level tables house compressed tag tables. This list of uncompressed level tables is checked whenever a line is evicted from the TVCache; if the corresponding level table is *not* in this list, it is added and one of the existing level tables is removed according to a simple "clock" eviction algorithm.

The inclusion of uncompressed level tables protects against large performance penalties during bursts of high miss rates in the TVCache. These bursts would otherwise incur decompression costs on top of the already high cost of accessing the TVStorage. Results show this method to be effective.

4.4 Case Studies

To demonstrate Skadu's effectiveness, we implemented two dynamic, region-based analyses that use vectored shadow memory: a memory footprint profiler and hierarchical critical path analysis (HCPA). The first represents a relatively lightweight application of Skadu whereas the second represents a heavyweight one. The following subsections describe these two analyses.

4.4.1 Memory Footprint Profiler

The memory footprint profiler tracks the number of memory locations accessed in each *dynamic* region and reports the average memory footprint for each *static* region. It illuminates a program's region-specific memory usage, informing memory optimizations.

Tag Format Each tag is a single bit that tracks whether or not the address has been touched by a region. This leads to a tag vector of n bits, where n is the depth of the region accessing the address. The profiler watches for the first touch of an address (i.e. tag changing from 0 to 1), incrementing a counter associated with the region when this event happens. This counter is checked when a region exits; its value then propagates to the statistics associated with the corresponding static region.

The region hierarchy leads to an inclusivity property for memory footprint analysis: if a memory address is touched in a region, it must also have been touched in all its ancestor regions. The footprint profiler exploits this property by compressing the whole tag vector into a single integer. This integer represents the shallowest level in which the address was *not* touched. This scalar representation avoids costly vector operations. This compressed format could lead to increased overhead—for example, when using an 8-bit integer to represent a the vector $\langle 1,1,0,0 \rangle$—but our results show that the size of this increase is negligible.

Efficiently Measuring Memory Footprint Each memory access triggers a check to see if the footprint of the active regions needs to be increased. This check

involves three s.pdf: tag validation, footprint update, and tag update. The tag validation step reads both the stored tag and the version from shadow memory and uses SlimTV to find the first invalid region level. The footprint update step finds and updates the range of region levels whose memory footprint should be incremented. The tag update step updates shadow memory with the new tag and version for the given address.

We use an algorithm in the footprint update step that reduces the update cost from $O(n)$ to $O(1)$. A naive algorithm would increment a counter for each level that needs to be updated, leading to an overhead of $O(n)$ for n updated regions.

Our algorithm reduces this cost through the use of a 2D array. This array contains elements, `count[maxLevel][minLevel]`, that represent the number of new memory accesses that increment minLevel to maxLevel. The footprint update increments only a element in the array that corresponds to the min and max levels to update. The profiler calculates the footprint of the region by summing all values in `count[currentLevel][...]` when a region exits and propagates these counters if $minLevel \leq currentLevel - 1$.

Implementation The memory footprint analyzer uses LLVM 2.8 [LA04] to insert functions calls into the source code that demarcate region boundaries and trigger events on memory accesses. These functions are implemented in a runtime library that is linked in at compile time. The footprint analyzer uses functions and loops as regions because they are natural, programmer-centric boundaries.

As previously mentioned, the footprint profiler compresses tag vectors into a single vector, eliminating the need for the TVCache-TVStorage organization; tag compression is still used by making the uncompressed level table list the first point of access for all accesses to shadow memory. In place of the TVCache-TVStorage architecture, we modified the traditional two-level shadow memory organization shown in Figure 4.1 to support tag validation and a 64-bit address space. Each segment table and tag table covers 4GB and 64KB of address space, respectively. Each tag is an 8-bit integer, supporting a region tree of depth 256. This was more than enough for all benchmarks we examined in our results. The footprint analyzer

supports the use of baseline tag validation, SlimTV, or BulkTV; this allowed us to examine the overheads associated with each of these techniques.

Tag tables support two separate configurations: one that tags every 4-bytes of address space and another that tags every 8-bytes of address space. The latter configuration results in less overhead (1 byte tag per 8 bytes of data or 12.5%) and is the default configuration when a tag table is created. If the analyzer detects finer granularity accesses (4-byte), it automatically switches configuration.

4.4.2 Hierarchical Critical Path Analysis

Overview As introduced in Chapter 2, hierarchical critical path analysis (HCPA) is a dynamic program analysis that computes the self-parallelism of each program region. Self-parallelism is the parallelism of a region exclusive of the parallelism of its child regions. HCPA calculates self-parallelism by performing critical path analysis (CPA) on every region of the program, utilizing the program hierarchy to determine the relationships of regions. CPA incurs a large amount of overhead as it requires every operation to be instrumented; this is required to find the *critical path* of the program, its longest set of dependent instructions.

HCPA concurrently calculates CPA on multiple regions, requiring a tag vector of n 64-bit timestamps for n active regions. The size of each tag makes memory overhead a severe issue in HCPA, much more so than the memory footprint profiler. HCPA further exacerbates the memory overhead problem by treating loop bodies as regions; this is in addition to the function and loop regions seen in the memory footprint profiler. The addition of loop bodies increases the depth of the region tree, increasing tag vector sizes and the memory overhead as a result.

HCPA operates on all instructions not just the loads and stores that were instrumented in the memory footprint profiler. This increased instrumentation greatly increases the performance overhead. HCPA does not access shadow memory on all instructions though: all non-memory operations utilize a shadow register file. This shadow register file is much smaller than shadow memory and can therefore be optimized for access time rather than space overhead in much the same way as the TVCache.

HCPA follows a three step procedure for handling loads. First, it accesses shadow memory to load in the tag vector (the timestamps) for the specified memory address. Next, it calculates the updated tag vector for the target register based on three factors: the loaded tag vector, the tag vector of control dependences, and the estimated cost of a load. Finally, it updates the shadow register file entry for the target register. The process for a store is similar except that the tag vector is initially loaded from the shadow register file and finally stored in shadow memory.

Implementation HCPA utilizes all of Skadu's techniques in order to reduce both the memory and runtime overhead. Shadow memory operations first access the TVCache to determine if the target address is available. A TVCache miss forces a load from and eviction to the TVStorage in the case of a load instruction; a miss on a store instruction simply requires an eviction to the TVStorage. All tag tables are compressed, save for those associated with a list of uncompressed level tables. If the level table associated with an evicted TVCache line is not in this list, it is added after another level table is evicted and compressed. HCPA uses a tag table size of 4KB, which is smaller than the 64KB tag tables used by the memory footprint analyzer. This smaller size reduces the runtime overhead associated with BulkTV, helping offset the increased runtime from having a variable size tag vector in HCPA.

The size of both the TVCache and the uncompressed level table list can be configured by the user. Increasing the size of either of these tends to reduce the runtime overhead at the expense of increased memory overhead.

The HCPA code also contains a lightweight garbage collector. This garbage collector walks all level tables in the TVStorage, using BulkTV to quickly find invalid tag tables and return them to the list of free tag tables. The garbage collector is activated when Skadu's dynamic memory overhead passes a threshold. Skadu adjusts this threshold based on the memory usage after garbage collection. This variable threshold avoids hysteresis effects.

Table 4.2: **Benchmark Characteristics.** We examined 12 benchmarks from three benchmark suites. These benchmarks display a wide variety of characteristics including memory usage (2MB to 434MB) and execution time (2 seconds to 2 minutes).

Benchmark		Mem. Usage (MB)	Native Runtime (Sec)	Region Depth	
Suite	Name			Footprint Profiler	HCPA
SpecInt	bzip2	189	57.1	17	25
	gzip	200	41.0	17	21
	mcf	152	90.5	48	53
	vpr	3	72.5	13	17
SpecFp	art	2	6.5	10	11
	equake	37	114	7	21
	mesa	20	120	20	26
NPB	cg	55	6.4	6	10
	ft	419	11.4	11	18
	is	68	2.0	4	7
	lu	43	82.9	6	12
	mg	434	5.6	8	13

4.5 Experimental Results

Methodology We examine the effectiveness of Skadu's proposed techniques using the two analyses described in Section 4.4: a memory footprint profiler and hierarchical critical path analysis (HCPA). Our experiments focus on both the memory and performance overheads associated with vectored shadow memory (VSM). We tracked the maximum memory overhead because it determines the minimum amount of memory required to successfully run the analysis. All measurements 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. For compression, we employed the miniLZO 2.06 library [Obe].

We examined 12 benchmarks across three benchmark suites: SpecInt 2000, SpecFP 2000, and NAS Parallel Bench (NPB) [BBB+91]. Table 4.2 characterizes each benchmark's native execution, listing runtime, memory footprint, and region depth. SpecFP and NPB benchmarks tend to have regular memory access patterns

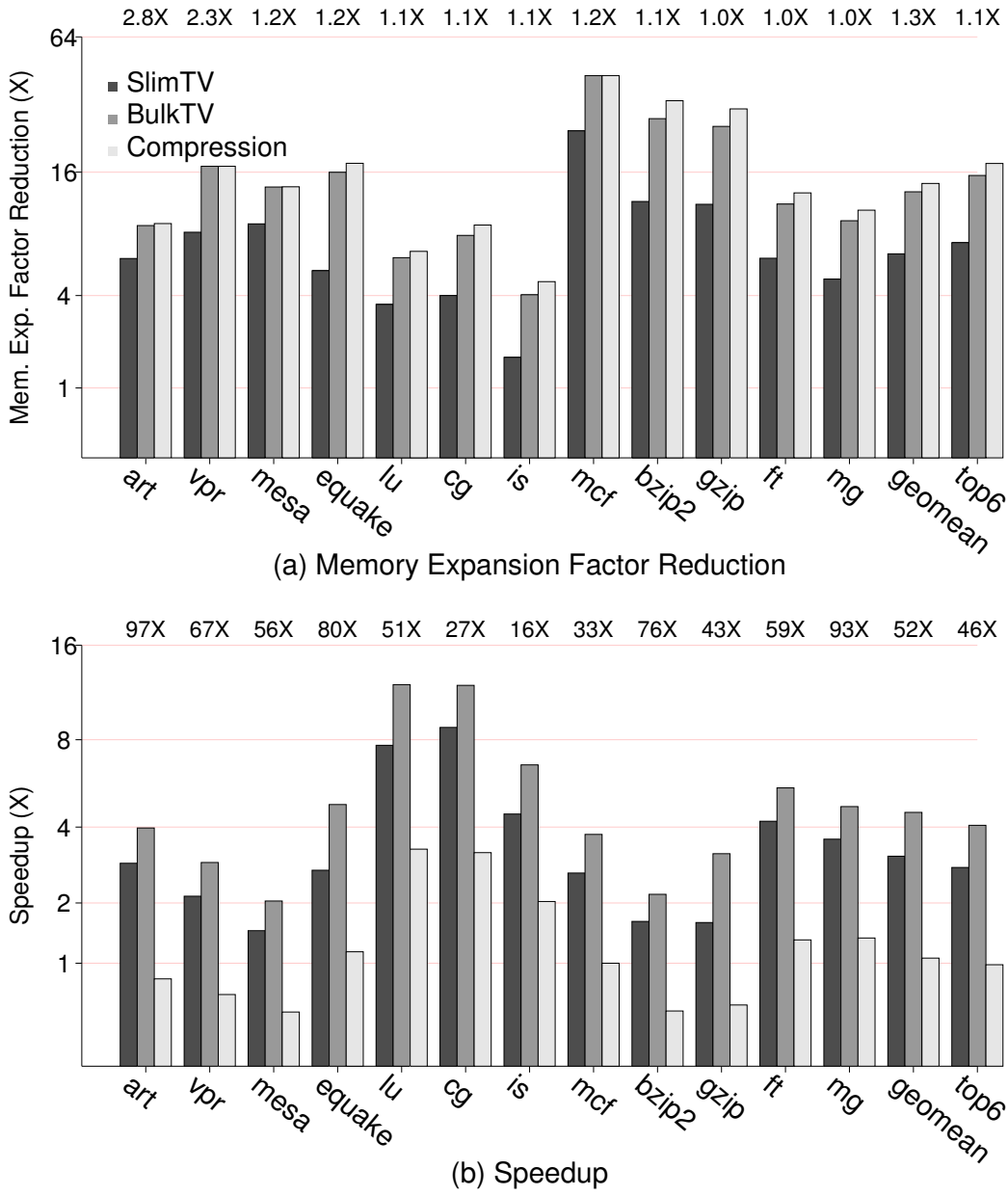


Figure 4.7: **Memory Overhead Reduction and Speedup in Footprint Profiler.** Skadu reduces the memory expansion factor from the baseline's 17.8× to 1.25× while maintaining comparable execution time. Numbers on the top represent the (a) memory expansion factor and (b) slowdown of Skadu's most aggressive memory-saving implementation compared to native execution.

and contain many dense, array-based operations. Conversely, SpecInt benchmarks have more irregular memory access patterns in addition to deeper region hierarchies. We used SpecInt and SpecFP's 'ref' input set and NPB's 'A' input set for all results.

4.5.1 Memory Footprint Profiler

As mentioned in Section 4.4, the memory footprint profiler uses only SlimTV, BulkTV, and tag compression. The footprint profiler's overheads are almost solely from tag validation, making it a good target to evaluate the impact of this process. The results are compared against those in the baseline implementation. This baseline implementation associates a version vector with every tag vector; the vector size in this baseline implementation is fixed to the deepest region level in the program.

Figure 4.7 shows the memory expansion factors and runtime overheads from the memory footprint profiler. This graph is sorted in order of increasing memory footprint. The numbers on top of the bars represent the final memory expansion factor and slowdown compared to the native execution. Skadu shows impressive reductions in the memory expansion factor of the memory footprint profiler when combining SlimTV, BulkTV, and compression. In overall, Skadu reduces the memory expansion factor by 14.2×. Benchmarks with larger memory footprints show overall better reductions, 17.5× for top six benchmarks in memory footprint.

SlimTV effectively reduces the memory expansion factor and improves performance. SlimTV's main benefits stem from its replacement of the version vector with a scalar version. These benefits will therefore be more pronounced in programs with deep region hierarchies. For example, `mcf` sees the largest reduction in memory expansion because of its region depth (48) that is more than twice the closest benchmark (20). SlimTV also speeds up the analysis by a factor of 3.1× because it eliminates the the large number of loads and stores associated with accessing version vectors.

BulkTV provides additional benefits beyond that of SlimTV. BulkTV reduces the memory overhead of tag validation from 7× (a 56-bit version for every

8-bit tag) to nearly zero (one 64 bit version per 64KB tag table). BulkTV is more effective at reducing memory expansion on programs with large memory footprints: the benefit increases as more tag tables are in use. Figure 4.7 shows this phenomenon: while the smallest (leftmost) benchmarks see little additionally benefit from BulkTV, the remaining benchmarks see significant improvements in the memory expansion factor. BulkTV also helps improve performance as explained in Section 4.2. With SlimTV and BulkTV, the geomean memory expansion factor is only 1.25× while slowdown is a manageable 12.28×.

Tag compression further reduces the memory footprint profiler's memory expansion factor. The footprint profiler maintained a list of 256 uncompressed level tables while the rest were compressed. These 256 tables covered 16MB of memory address space. This address space coverage meant that benchmarks that used less than 16MB of memory saw no benefit. Compression is therefore similar to SlimTV and BulkTV in that is sees larger benefits with larger programs. For example, `mg` receives a 13× reduction in memory, making the memory overhead almost negligible.

Compression's memory savings come at a cost: increased runtime overhead. This overhead consists of two components: the compression/decompression algorithms and the eviction algorithm used for the list of uncompressed level tables. This list uses a "clock" eviction policy [Tan07] that requires an access bit be updated every time an entry in the list is touched. This clocking cost explains the additional runtime overhead even when compression is not used (e.g. in `art`). While a simpler eviction policy may seem desirable (e.g. direct mapped cache), the higher hit ratio of the clock algorithm more than offsets its maintenance costs.

4.5.2 Hierarchical Critical Path Analysis (HCPA)

Hierarchical critical path analysis is much more costly than the memory footprint profiler in terms of both memory and performance. HCPA's baseline version results in a memory expansion factor of 59.0×, severely limiting its use outside of supercomputers and other high memory environments. HCPA utilizes Skadu's full array of techniques to rein in its overheads. The results are impres-

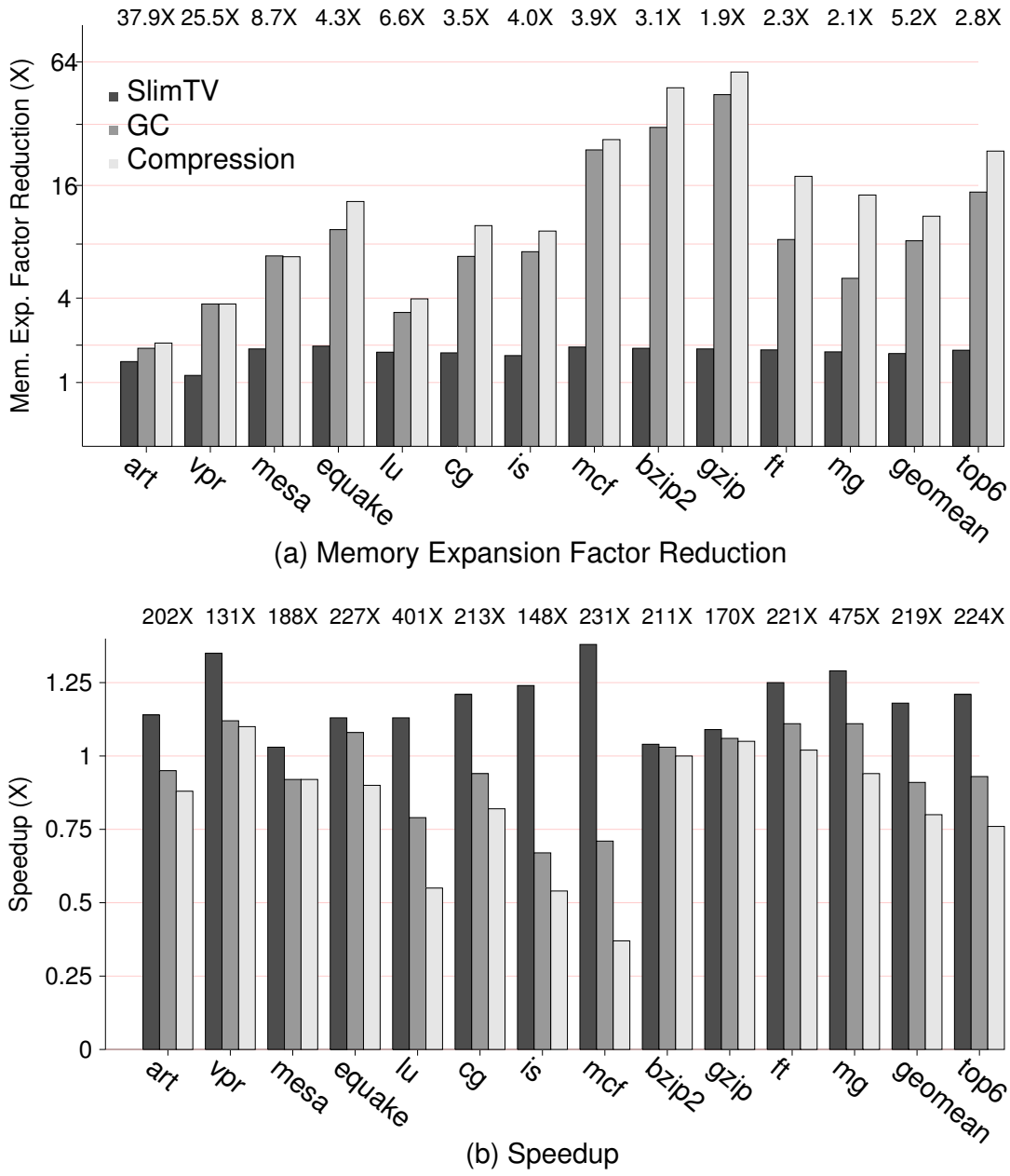


Figure 4.8: **Memory Overhead Reduction and Speedup in HCPA.** Skadu reduces HCPA's memory expansion factor by 11.2× compared to the baseline implementation at a cost of only 25% in performance overhead. Numbers on the top represent the (a) memory expansion factor and (b) slowdown of Skadu compared to native execution.

sive. Skadu reduces the a memory expansion factor to 5.2×, a reduction of 11.4× compared to the baseline implementation.

Figure 4.8 shows the memory and performance improvements from Skadu's various techniques. Benchmarks are presented in the same order as they were in Figure 4.7: in order of increasing memory footprint. The numbers on top of the bars show the memory expansion factor and slowdown vs native when using all of Skadu's techniques. We set the TVCache size to cover 1MB of address space while the list of uncompressed level tables covered 4MB. This represented a decrease in the number of uncompressed level tables compared to the memory footprint profiler. This was a result of a reduced reliance on this list to improve performance: HCPA introduces the TVCache, which greatly reduces tag table storage.

SlimTV not only reduces the memory expansion factor but also improves performance, an outcome similar to what we witnessed in the footprint profiler. HCPA's modest memory reduction of 2× stands in contrast to the footprint profiler. This difference arises because the baseline HCPA implementation's memory overhead is almost equally split between tags and tag validation; in the memory footprint profiler, almost all the overhead was a result of tag validation. This more equitable split also leads to smaller performance gains for improved tag validation in HCPA.

Skadu's TVCache-TVStorage architecture with garbage collection (labeled GC in Figure 4.8) has a significant impact on the memory expansion factor. Outside of the tiny `art` benchmark, all benchmarks benefited from this architecture. These benefits ranged from 2× (lu) to 39× (gzip) with a geomean of 5.6×. SpecInt benchmarks tend to show greater memory reductions because they move between regions more quickly than the other benchmarks. The runtime overhead is only 15% more than when using only SlimTV. Compression further reduces the memory overhead by 1.6×, increasing performance overhead by less than 20%.

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 5

Related Work

This chapter examines Kismet's related work according to four themes: parallelism profiling, performance prediction, parallel performance debugging tools, and reducing dynamic program analysis overheads.

5.1 Parallel Performance Prediction

Parallelism Profiling Approaches for parallelism-related profiling have generally fallen into two categories: critical path analysis and dependence testing.

Critical path analysis (CPA) dates back several decades, with early important works including Kumar and Austin [Kum88, AS92]. CPA approaches seek to measure the number of concurrent operations at each time step along the critical path of the program. In contrast to these approaches, Kismet's hierarchical critical path analysis is able to localize parallelism within nested program regions, and provide concrete guidance on which program regions to target. Recently, 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 Kismet's focus on estimating speedup in 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. Other works have used CPA to perform limit studies for processors that target instruction-level parallelism (ILP) [Wal91, LW92].

Dependence testing is another parallelism profiling approach that strives 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. Similar techniques are used in Alchemist [ZNJ09] and Prospector [KKL10a]. Although dependence testing and Kismet's HCPA share similar goals, HCPA 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 if significant code transformation would be required to exploit it. Dependence testing is generally more pessimistic and sensitive to existing program structure.

Performance Prediction CilkView [HLL10] and Intel Parallel Advisor's Suitability Tool [Int] are recent tools whose motivation is similar to Kismet. Like Kismet, they also predict parallel performance on a target with arbitrary number of cores. Unlike Kismet, however, CilkView and Parallel Advisor rely on the user's parallelized code—or annotations—to predict speedup. Kismet 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 multicore 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 Kismet.

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. Kismet improves upon these works by using HCPA's ability to localize parallelism; Kismet 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 Kismet'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 Kismet 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 Kismet's analytical model relatively simple, Kismet can easily incorporate these sophisticated models if needed.

Parallel Performance Debugging Tools Several systems have been developed in order to help debug the performance of pre-existing parallel programs [DRR99, AMCA+95, MCC+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 Kismet 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 Kismet; however, Kismet does not require user interaction, and uses a simplify hardware specifications to give reasonable speedup predictions of post-parallelized code.

Reducing Dynamic Program Analysis Overheads Dynamic program analyses often have huge memory and storage requirements as they can produce data for each dynamic instruction in a program that easily could run billions or trillions of instructions. To alleviate the severe memory requirements of dynamic program analysis, compression techniques have been used in whole program analysis [ZG01] and dependence analysis [KKL10b]. Initially Kismet used a compression technique similar to [GJLT11], but we found that handling more irregular programs like SpecInt necessitated the creation of Kismet's summarization-based HCPA.

In addition to memory overhead, runtime overhead is also important for practical use. Specifically for program analysis that uses shadow memory, the implementation of shadow memory significantly impacts the overall runtime as each load and store instruction will access the shadow memory. Valgrind [NS07b]'s shadow memory implementation is described in [NS07a]. Umbra [ZBA10a] and EMS64 [ZBA10b] proposed efficient shadow memory implementation for 64-bit address space, exploiting the sparse usage of memory space in 64-bit systems and cached shadow memory. Although techniques introduced in these papers can be incorporated in Kismet, Kismet's shadow memory implementation differs from other tools as it needs to efficiently store and retrieve multiple timestamps for each memory address to track the critical path of multiple region levels.

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 "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 primary investigator and author of this paper.

Chapter 6

Summary

As multi-core processors enter mainstream computing, software engineers are facing a fundamental change with parallelization. We began this dissertation by discussing why fully automatic parallelization do not work in practice and the limitations of currently available tools, leading to the need for a new speedup estimation tool. Kismet is different from existing tools in that our speedup estimation tool does not require any pre-parallelized or annotated source code. As our tool requires only unmodified serial source code, it can help a programmer making informed decisions in the early stages of parallelization, making the manual parallelization process more productive.

One of the key factors that limit achievable speedup is the amount of parallelism available in the target program. In Chapter 2, we introduced hierarchical critical path analysis (HCPA) that quantifies the amount of parallelism in each region. Unlike the original critical path analysis (CPA) that provides only the theoretical speedup upperbound, HCPA localizes the parallelism of each region with a new metric called self-parallelism, providing the basis for realistic speedup estimation. We also discussed an efficient summarization techniques that makes the huge amount of produced data at runtime manageable.

Chapter 3 described Kismet, our speedup estimation tool prototype. Besides parallelism, target-specific parallelization constraints such as expressible parallelism type, parallelization overhead, available core count, and memory locality significantly impact the achievable parallel speedup. Based on the profiled infor-

mation from HCPA and specified parallelization constraints, Kismet finds the parallelization strategy with the highest expected speedup. Our experimental results show that Kimset provided realistic speedup upperbounds on two very different target platforms: the MIT RAW and conventional multi-core processors.

Chapter 4 discussed the design and implementation of vector shadow memory (VSM). Because HCPA recursively applies CPA, which is already an expensive dynamic analysis, it can incur prohibitively expensive memory and runtime overhead. We applied a few techniques that reduces both memory and runtime overhead, dramatically reducing the overhead of HCPA to a level where most programs can be run on conventional machines. We also showed these techniques can be applied to other heavyweight memory analysis with a memory footprint analyzer.

Overall, we have shown that estimating parallel speedup from unmodified serial source code is a viable means of helping manual parallelization, which typically requires extensive efforts from a programmer. Our prototype, Kismet, allows programmers to make informed decisions in the early stages of parallelization by understanding the potential benefit from parallelization, making parallelization more productive. We have demonstrated that Kismet provides realistic speedup upperbounds on two very different target platforms with widely varying parallelization constraints. In order to help more people in parallelization and make Kismet evolve with contributions from more people, we plan to release Kismet as an open source project.

Bibliography

[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.

[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.

[BFL+97] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal. The raw benchmark suite: computation structures for general purpose computing. In *FCCM '97: Proceedings of the IEEE Symposium on FPGA-Based Custom Computing Machines*, pages 134–, Washington, DC, USA, 1997. IEEE Computer Society.

[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.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[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.

[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.

[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.

[E. 97] E. Waingold et al. Baring It All to Software: Raw Machines. *IEEE Computer*, pages 86–93, Sept 1997.

[GJLT11] Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor. Kremlin: Rethinking and rebooting gprof for the multicore age. In *PLDI '11: Proceedings of the Conference on Programming Language Design and Implementation*, New York, NY, USA, 2011. ACM.

[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.

[GSV+10] N. Goulding, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, J. Babb, M.B. Taylor, and S. Swanson. GreenDroid: A Mobile Application Processor for a Future of Dark Silicon. In *Hotchips*, 2010.

[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.

[Int] Intel. Intel Parallel Advisor 2011. .

[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.

[KKL10a] M. Kim, H. Kim, and C.K. Luk. Prospector: A dynamic data-dependence profiler to help parallel programming. In *HotPar*, 2010.

[KKL10b] 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.

[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.

[KRL+10] Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee, and David I. August. Scalable speculative parallelization on commodity clusters. In *MICRO '10: Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 3–14, 2010.

[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.

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

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, Palo Alto, California, 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.

[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.

[LW92] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *ISCA*, pages 46–57, New York, NY, USA, 1992. ACM.

[M. 04] M. B. Taylor et al. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *ISCA '04: Proceedings of the International Symposium on Computer Architecture*, Munich, Germany, Jun 2004.

[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.

[NS07a] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE '07: Proceedings of the International Conference on Virtual Execution Environments*, pages 65–74, 2007.

[NS07b] 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.

[PO05] Manohar K. Prabhu and Kunle Olukotun. Exposing speculative thread parallelism in spec2000. In *PPoPP '05: Proceedings of the ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 142–152, New York, NY, USA, 2005. ACM.

[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.

[ROR+08] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *CGO '08: Proceedings of the International Symposium*

on Code Generation and Optimization, pages 114–123, New York, NY, USA, 2008. ACM.

[S. 08] S. Bell et al. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *ISSCC '08: IEEE Solid-State Circuits Conference*, pages 88–89,598, 2008.

[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.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, 3rd edition, 2007.

[Tay07] Michael B. Taylor. *Tiled Microprocessors*. PhD thesis, Massachusetts Institute of Technology, 2007.

[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.

[TKM+02] M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, Jae-Wook Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: a computational fabric for software circuits and general-purpose programs. *Micro, IEEE*, 22(2):25 – 35, mar/apr 2002.

[TLAA05] Michael Bedford Taylor, Walter Lee, Saman P. Amarasinghe, and Anant Agarwal. Scalar operand networks. *IEEE Transactions on Parallel and Distributed Systems*, 16:145–162, Feb 2005.

[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.

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

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, New York, NY, USA, 1991. ACM.

[WE03] Joel Winstead and David Evans. Towards differential program analysis. In *WODA '03: Workshop on Dynamic Analysis*, 2003.

[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.

[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.

[ZL10] David Zier and Ben Lee. Performance evaluation of dynamic speculative multithreading with the cascadia architecture. *TPDS*, 21:47–59, Jan 2010.

[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.