Iowa State University
Digital Repository

Graduate Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

2011

Phase-based tuning: better utilized performance asymmetric multicores

Tyler Sondag
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
Sondag, Tyler, "Phase-based tuning: better utilized performance asymmetric multicores" (2011). *Graduate Theses and Dissertations*. 10416.
https://lib.dr.iastate.edu/etd/10416

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

Phase-based tuning: better utilized performance asymmetric multicores

by

Tyler Sondag

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:

Hridesh Rajan, Major Professor

Morris Chang

Soma Chaudhuri

Jack Lutz

Joseph Zambreno

Iowa State University

Ames, Iowa

2011

Copyright © Tyler Sondag, 2011. All rights reserved.

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

ACKNOWLEDGMENTS ix

ABSTRACT x

CHAPTER 1. Introduction 1

1.1 Problems Facing Efficient Use of AMPs 2

1.2 Phase-based Tuning 3

1.2.1 Compile Time Grouping and its Problems 5

1.2.2 Lazy Grouping and its Benefits 6

1.3 Benefits of Phase-based Tuning 7

1.4 Organization 8

CHAPTER 2. Phase-based Tuning with Compile Time Grouping 10

2.1 Static Phase Transition Analysis 11

2.1.1 Static CFG Annotation 12

2.1.2 Phase Transition Marking 19

2.1.3 Determining block types 22

2.2 Dynamic Analysis and Tuning 23

CHAPTER 3. Analysis and Instrumentation Framework 26

3.1 Disassembly 26

3.2 Analysis 27

3.2.1 Control Flow Analysis 27
3.2.2 Abstract Interpretation 27
3.3 Instrumentation 29
CHAPTER 4. Evaluation of Phase-based Tuning 31
4.1 Experimental Setup 32
4.1.1 System Setup 32
4.1.2 Workload Construction 33
4.2 Space and Time Overhead 33
4.2.1 Space Overhead 34
4.2.2 Time Overhead 36
4.2.3 Core Switches 39
4.3 Throughput 41
4.3.1 IPC threshold 41
4.3.2 Clustering error 42
4.3.3 Minimum instruction size 43
4.4 Fairness 44
4.5 Analysis of Trade-offs 46
4.6 Summary of Results 46
CHAPTER 5. Practical and Future Use of Phase-based Tuning 48
5.1 Applicability to Multi-threaded Programs 48
5.2 Changing Application and Core Behavior 49
5.3 Floating Point Emulation and IPC 50
5.4 Scalability for many-cores 50
CHAPTER 6. Lazy Grouping 51
6.1 Compile Time – Computing Similarity Metrics 52
6.1.1 Instruction Latency 53

6.1.2 Instruction Cache 54
6.1.3 Data Cache 54
6.1.4 Instruction Level Parallelism (ILP) 55
6.1.5 Loop Size 56
6.2 OS/Machine Install Time – Network Training 56
6.3 Program Install Time – Grouping 59
6.4 Summary of Benefits 59

CHAPTER 7. Evaluation of Lazy Grouping 60
7.1 Experimental setup 60
7.1.1 System Setup 60
7.1.2 Computing Actual Behavior for Evaluation 61
7.1.3 Accuracy Metrics 62
7.2 Compile Time vs. Lazy Grouping 63
7.2.1 Universal Compile Time Grouping 63
7.2.2 Target Machine Specific Compile Time Grouping 64
7.3 Accuracy of Lazy Grouping 66
7.4 Space Overhead 72
7.5 Lazy Grouping in Phase-based Tuning 72
7.5.1 Impact of Lazy Grouping on Overheads 73
7.5.2 Impact of Lazy Grouping on Speedup and Fairness 75
7.6 Summary 76

CHAPTER 8. Related Work 77
8.1 Related Work: Phase-based Tuning 77
8.2 Related Work: Lazy Grouping 79
8.2.1 Behavior Similarity 79
8.2.2 Machine Learning for Optimization 80

CHAPTER 9. Future Work 82

9.1 Potential Applications of Phase-based Tuning 82

9.1.1 JIT Optimizations 82

9.1.2 Power Consumption 83

9.2 Potential Applications of Lazy Grouping 84

9.2.1 Guide Optimization 84

9.2.2 Benchmark Selection 85

9.3 Small Improvements 85

9.3.1 Phase-based Tuning 85

9.3.2 Lazy Grouping 86

CHAPTER 10. Conclusion 88

BIBLIOGRAPHY 89

LIST OF TABLES

Table 4.1 Space overhead of phase marks 35

Table 4.2 Time overhead of phase-based tuning 37

Table 4.3 Core switches per benchmark 39

Table 4.4 Phase-based tuning fairness 45

Table 7.1 Core types used for lazy grouping 61

Table 7.2 Lazy grouping accuracy per configuration 67

LIST OF FIGURES

Figure 1.1 Overview of phase-based tuning 4
Figure 1.2 Compile time similarity dimension problem 6

Figure 2.1 Overview of phase transition analysis 12
Figure 2.2 Interval type summarization algorithm 14
Figure 2.3 Interval summarization illustration 15
Figure 2.4 Loop summarization illustration 17
Figure 2.5 Loop type summarization algorithm 18
Figure 2.6 Lookahead based phase marking 21
Figure 2.7 Lookahead based reduction of phase marks 22
Figure 2.8 Algorithm for preferred core assignment 24

Figure 3.1 Conversion from address to labels 26
Figure 3.2 Abstract interpretation example 28
Figure 3.3 Binary instrumentation example 30

Figure 4.1 Space overhead of phase marks 36
Figure 4.2 Time overhead of phase marks 38
Figure 4.3 Average cycles between core switches 40
Figure 4.4 Throughput with variable IPC threshold (basic block technique) . . 42
Figure 4.5 Throughput with variable IPC threshold (interval technique) 42
Figure 4.6 Impact of similarity precision 43
Figure 4.7 Throughput per technique 43

Figure 4.8 Speedup vs fairness: average time vs. max stretch 47

Figure 6.1 Lazy grouping overview 51

Figure 6.2 Lazy grouping: compile and program install time 53

Figure 6.3 Lazy grouping: OS install time 57

Figure 7.1 Compile time vs lazy grouping 64

Figure 7.2 Compile time grouping portability 65

Figure 7.3 Lazy group distribution (single core) 68

Figure 7.4 Lazy group distribution (AMPs) 69

Figure 7.5 Per benchmark lazy grouping (i7-atom) 71

Figure 7.6 Lazy grouping space overhead 72

Figure 7.7 Phase-based tuning space overhead with lazy grouping 73

Figure 7.8 Phase-based tuning time overhead with lazy grouping (per benchmark) 74

Figure 7.9 Phase-based tuning time overhead with lazy grouping (summary) . . 75

ACKNOWLEDGMENTS

I would like to thank everyone who over the past several years has helped me with my studies and guided me in the right directions.

First and foremost, I would like to thank my adviser Dr. Hridesh Rajan for being a great mentor over the past years. It has truly been an honor to be his student. His advice and guidance have helped me grow tremendously as a researcher and as a person. I sincerely appreciate all the time, advice, and patience he has given me over the past five years.

I would also like to thank Dr. Kian Pokorny and Dr. Jim Feher who inspired me to go to graduate school, to pursue a career in research, and gave great advice throughout my undergraduate career and since.

Thanks to the members of the Laboratory for Software Design at ISU for their help throughout my graduate studies and for their constructive criticism and suggestions during all stages of my work. I also thank Patrick Carlson and Paul Murphy for their help developing and debugging parts of the analysis and instrumentation framework used for evaluating the ideas in this thesis.

I would like to thank my parents Brenda and Dale for their love and support throughout everything. Without them none of this would be possible. There is no way I can put into words how grateful I am to them. My brother Trevor for sparking my interest in mathematics at an early age and for being a great role model. My brother Trent letting me practice (and improve) my teaching and presentation skills on him.

Finally, I thank my wife Kasey for being supportive and understanding (especially before deadlines) and encouraging me when I needed it most. For being my best friend, and distracting me when I needed it most.

ABSTRACT

The latest trend towards performance asymmetry among cores on a single chip of a multicore processor is posing new challenges. For effective utilization of these performance-asymmetric multicore processors, code sections of a program must be assigned to cores such that the resource needs of code sections closely matches resource availability at the assigned core. Determining this assignment manually is tedious, error prone, and significantly complicates software development. To solve this problem, this thesis describes a transparent and fully-automatic process called *phase-based tuning* which adapts an application to effectively utilize performance-asymmetric multicores. The basic idea behind this technique is to statically compute groups of program segments which are expected to behave similarly at runtime. Then, at runtime, the behavior of a few code segments is used to infer the behavior and preferred core assignment of all similar code segments with low overhead. Compared to the stock Linux scheduler, for systems asymmetric with respect to clock frequency, a 36% average process speedup is observed, while maintaining fairness and with negligible overheads.

A key component to phase-based tuning is grouping program segments with similar behavior. The importance of various similarity metrics are likely to differ for each target asymmetric multicore processor. Determining groups using too many metrics may result in a grouping that differentiates between program segments based on irrelevant properties for a target machine. Using too few metrics may cause relevant metrics to be ignored thereby considering segments with different behavior similar. Therefore, to solve this problem and enable phase-based tuning for a wide range of a performance-asymmetric multicores, this thesis also describes a new technique called *lazy grouping*. Lazy grouping statically (at compile and install times) groups program segments that are expected to have similar behavior. The basic idea is to use extensive

compile time analysis with intelligent install time (when the target system is known) group assignment. The accuracy of lazy grouping for a wide range of machines is shown to be more than 90% for nearly all target machines and asymmetric multicores.

CHAPTER 1. Introduction

Increases in single processor performance through frequency scaling has hit a wall [66]. Improving performance for this class of processors is prohibitively expensive due to design complexity, space constraints, and heat dissipation [29]. Therefore, replicating cores has become commonplace in recent years in order to continue improving performance as suggested by Moore's law [29].

Many recent processor designs have been homogeneous multicore processors. These homogeneous designs have some drawbacks [4, 5, 31, 44, 53, 61]. Replicating fast complex cores results in fewer cores and thus less performance for highly parallel programs but good single-threaded performance. Replicating simpler less powerful cores results in improved performance for highly parallel programs, but reduced serial performance.

Amdahl's law states that the serial portion of the program limits the speedup achievable by parallelism [60]. Thus, recently researchers and vendors have advocated the need for heterogeneous multicore processors to achieve the best of both types of systems (high throughput for both parallel and sequential workloads) [4, 5, 31, 44, 53, 61]. A common trend is to have a few fast cores to improve serial performance, but many slow cores to achieve high parallel performance [31]. Other attractive options include specialized processors such as vector processors or GPUs to accelerate certain portions of execution [26, 39]. There is also a potential to eliminate excess hardware capabilities from certain cores in the system (e.g. dedicated floating point units, out-of-order execution, etc.) to use less power, reduce space, and decrease heat. Aside from improving throughput for multiple types of workloads, a major advantage is that heterogeneous multicore processors decrease power consumption and use less die space

compared to homogeneous multicores [44]. For these same reasons, they are also seen as a cost efficient alternative for supercomputing (especially for large data centers where power and heat are important concerns) [58].

Single-ISA performance-asymmetric multicore processors (AMPs) [4, 5, 31, 44, 53, 61] are an important class of these heterogeneous multicores. All cores in an AMP support the same instruction set, however, they are heterogeneous in terms of performance characteristics such as clock frequency, cache size, in-order vs. out-of-order execution, etc [31, 45, 61].

The technical contribution of this work is a hybrid (static and dynamic) program analysis and optimization technique called *phase-based tuning* for effectively utilizing AMPs. In this section, some of the major problems facing efficient use of AMPs are discussed. Then, an overview of phase-based tuning is given. Finally, benefits of phase-based tuning are outlined.

1.1 Problems Facing Efficient Use of AMPs

There are several problems that arise with the introduction of AMPs. First, application performance becomes unpredictable and scalability becomes increasingly difficult to ensure [6]. Additionally, programming languages are typically not designed to handle the increased complexity of heterogeneous multicore processors and programmers are not trained to understand their programs' behavior and hardware characteristics [16]. Furthermore, program behavior and/or system design may not be known statically. This necessitates new scheduling techniques to take full advantage of these new architectures [9].

In general, for effective utilization of AMPs, code sections of a program must be executed on cores such that the resource requirements of a section closely match the resources provided by the core [43, 53]. To match the resource requirements of a code section to the resources provided by the core, both must be known.

First, let us consider having the programmer manually perform such a mapping. This manual tuning has at least three problems.

1. First, the programmer must know the runtime characteristics of their code as well as the details of the underlying asymmetry. This increases the burden on the programmer. Further, both the resource requirements of processes and the resources provided by cores cannot be determined statically (code behavior may change with inputs, core characteristics may change between different platforms or if the workload on the system changes [49, 12]). This is troubling since utilization is heavily influenced by the accuracy of this knowledge [43, 53, 49].

2. Second, with multiple target AMPs this manual tuning must be carried out for each AMP, which can be costly, tedious, and error prone.

3. Third, as a result of this manual tuning a custom version must be created for each target AMP, which decreases re-usability and creates a maintenance problem. Further, the performance asymmetry present in the target AMP may not be known during development.

Next, consider an automated (or semi-automated) static technique. Such a technique has several problems. First, both the resource requirements of processes and the resources provided by cores cannot be determined statically (as discussed above). Also, like a manual approach, unknown target AMPs creates a significant problem.

Finally, consider a dynamic technique. At runtime, there is an advantage of being able to observe behavior of the program on each core type. Unfortunately, great care must be taken to avoid excessive overhead which can easily overshadow gains achieved by such a technique.

Effective utilization of AMPs is a major challenge. Finding techniques for automatic tuning is critical to address this challenge and to realize the full potential of AMPs [49].

1.2 Phase-based Tuning

To help solve the problem of effective utilization of AMPs, this thesis introduces a novel program analysis technique called *phase-based tuning* for matching resource requirements of

code sections to the resources provided by the cores of AMPs. Phase-based tuning builds on a well-known insight that programs exhibit phase behavior [35, 37, 51, 64, 73, 79, 82]. That is, programs go through phases of execution that show similar runtime characteristics compared to other phases [7, 18, 20, 21, 30, 75]. Based on this insight, phase-based tuning has two parts: static analysis which identifies likely *phase-transition points* (where runtime characteristics are likely to change) between code sections, and a lightweight dynamic analysis that determines section-to-core assignment by exploiting a program's phase behavior. The static analysis results are used to generate standalone binaries in which each phase-transition point is instrumented with a tiny code fragment for dynamic analysis. These fragments contain analysis code as well as phase information. This phase information reduces dynamic analysis costs by using the runtime behavior of previously executed sections to make future assignments.

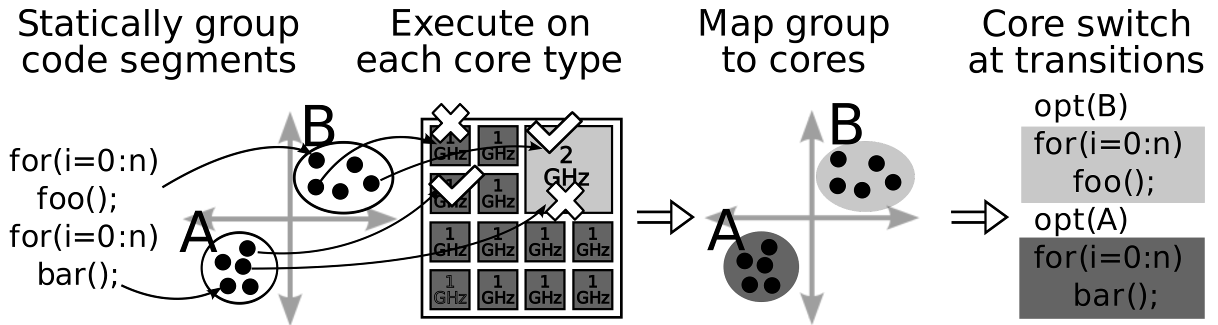

Figure 1.1 Overview of phase-based tuning. Segments are grouped statically. At runtime, a few segments from each group are run on each core type and their behavior is monitored. Using this observed behavior of only a few segments, entire groups are assigned to cores.

Figure 1.1 illustrates a simple overview of phase-based tuning. On the far left there is a simple function which contains two loops with different behaviors that are better suited for the different cores in the target AMP. Thus, these two loops are placed into separate *groups* (labeled A and B) which contain other code segments which should have similar resource requirements. Next, at runtime, a few program segments from each group are executed on each core type. After this, the expected best fit in terms of core type for the group is determined

(groups become shaded) and all segments in the group are mapped to this core type. Finally, the two loops considered initially now know which core they are likely to be best suited for and may make the decision to execute themselves on these cores. It is important to note that phase-based tuning did not require these loops to be executed on either core type.

Statically grouping similar program segments is a key component of phase-based tuning. Here, a brief overview of *lazy grouping*, a novel static (compile+install) similarity grouping technique is described. First, the limitations of a purely compile time grouping technique are shown. Then, lazy grouping, and how it solves these problems is discussed.

1.2.1 Compile Time Grouping and its Problems

A simple compile time technique could be used to statically group similar program segments. This technique could work as follows. First, behavior metrics (e.g., instruction latency, cache behavior) are statically computed for each program segment. For example, in Figure 1.1, each loop has two metrics calculated for it thus placing each loop somewhere in a two dimensional space. Using these metrics, all segments are grouped using the k-means clustering technique [55]. The figure shows two groups marked A and B. The goal is that all segments in a group have similar behavior.

For simple asymmetry such as difference in clock frequency, such compile time technique works remarkably well, however, as the variety of asymmetry increases, a more sophisticated approach is needed. For many different types of target machines, the analysis must compute metrics for several aspects of behavior (e.g. caching, instruction level parallelism, etc) and variations on each of these metrics (e.g. different cache sizes). Thus, there are many metrics to consider.

Such a compile time technique has no knowledge of the relevance of metrics for the target machines and thus will create the same groups for all machines. A metric (e.g. instruction level parallelism) may only be relevant for a subset of target machines, however, compile time grouping will differentiate based on this metric for all machines. Thus, it will likely create sep-

arate groups which do not differ on many target machines. Consider the example in Figure 1.2. On the left, a single metric relevant for this machine is used resulting in three groups. On the right, an additional metric is added which is not strongly relevant for this machine. In this case, this extra dimension hurts rather than helps. Total groups were tripled but those which vary only in this dimension are similar. Such differentiation may cause excessive overhead for phase-based tuning.

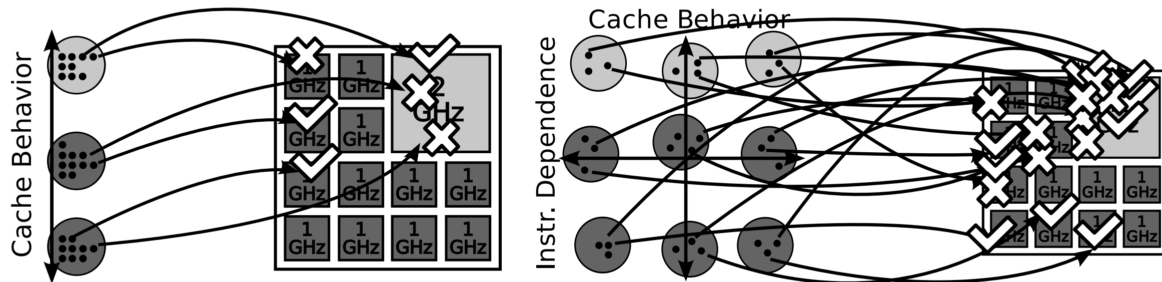

Figure 1.2 Left: Runtime behavior of phase-based tuning using one similarity metric. Right: The same process but with two metrics. The figure shows that with more metrics, benefits likely decrease.

Adding metrics is desirable in order to tackle more diverse types of AMPs. However, adding metrics into the existing approach creates too many groups, many of which may not differ. Thus, a new technique is needed to handle a significant amount of metrics but produce few, accurate groups.

1.2.2 Lazy Grouping and its Benefits

To address these problems, a novel technique is presented for accurately grouping similar program segments without running them. The basic idea is as follows. At compile time, several program behavior metrics are statically computed. At *program install time* (when the program is put on the target machine) code segments are assigned to groups using a neural network which takes as input the similarity metrics computed at compile time. This neural network is created and trained at *OS or machine install time* once for each machine. This approach

is called “lazy” since it waits till the target machine is known to compute the grouping rather than doing so immediately after metrics are computed at compile time.

Lazy grouping has several benefits. First, it is able to tackle the problem of computing program segment similarity for a wide variety of target machines since grouping is delayed until install time. Thus problems caused by an unknown target machine are avoided. Second, typical use of the technique (program install time) is very efficient since most of the overhead occurs at compile time and OS/machine install time. Third, phase-based tuning is more efficient (than when a purely compile time approach is used) for three reasons. First, the optimal number of output groups (e.g. number of core types) may be specified while accuracy is maintained. Second, group distribution may be tuned, through neural network training, to fit the target AMPs core distribution. Third, phase-based tuning no longer needs to perform runtime monitoring and analysis since lazy groups have knowledge of the approximate behavior of the code segments in each group.

To evaluate lazy grouping, its accuracy with respect to single core program behavior as well as its use for AMPs is considered. This evaluation shows that lazy grouping is significantly more accurate than compile time grouping and requires fewer groups. Further, results show that lazy grouping is more than 90% accurate for nearly all target machines and AMPs tested. Finally, an evaluation of lazy grouping’s impact on phase-based tuning is done. In this case, there is a 17% higher average process speedup when lazy grouping is used.

1.3 Benefits of Phase-based Tuning

Phase-based tuning has the following benefits:

- **Fully Automatic:** Since phase-based tuning determines core assignments automatically at runtime, the programmer need not be aware of the performance characteristics of the target platform or their application.

- **Transparent Deployment:** Programs are modified to contain their own analysis and core switching code. Thus, no operating system or compiler modification is needed. Therefore, phase-based tuning can be utilized with minimal disruption in the build and deployment chain.

- **Tune Once, Run Anywhere:** The analysis and instrumentation makes no assumptions about the underlying AMP. Thus there is no need to create multiple versions for each target AMP. Also, by making no assumptions about the target AMP, interactions between multiple threads and processes are automatically handled.

- **Negligible Overhead:** It incurs less than 3% space overhead and less than 0.1% time overhead, i.e. it is useful for overhead conscious software and it is scalable.

- **Improved Utilization:** Phase-based tuning improves utilization of AMPs by reducing average process time by as much as 36% while maintaining fairness.

To evaluate the effectiveness of phase-based tuning, it was implemented as part of a custom binary static analysis and instrumentation framework and applied to workloads constructed from the SPEC CPU 2000 and 2006 benchmark suites which are standard for evaluating processors, memory and compilers. These workloads consist of a fixed number of benchmarks running simultaneously. These workloads, running on a simple AMP where clock frequencies differ, see as much as a 36% reduction in average process time while maintaining fairness and incurring negligible overheads.

1.4 Organization

The rest of this thesis is organized as follows. Chapter 2 describes the hybrid analysis and optimization technique called phase-based tuning for effectively utilizing AMPs. Next, Chapter 3 describes the static binary analysis and instrumentation framework used for the evaluation of phase-based tuning. Chapter 4 presents the experimental setup for evaluating phase-based

tuning and the results. Chapter 5 gives a brief discussion of relevant issues facing practical and future use of phase-based tuning. In Chapter 6, lazy grouping, a novel static behavior similarity grouping technique, is discussed. Next, Chapter 7 describes the experimental setup for evaluating the accuracy and effectiveness of lazy grouping and presents the evaluation results. Then, Chapter 8 describes related work. Finally, Chapter 9 discusses potential future work and Chapter 10 concludes.

CHAPTER 2. Phase-based Tuning with Compile Time Grouping

In this chapter, phase-based tuning is described. The goal of phase-based tuning is to match resource requirements of code sections to the resources provided by the cores of a performance-asymmetric multicore processor.

The intuition behind phase-based tuning is the following. If static analysis can classify a program's execution into code sections and group these sections into clusters such that all sections in the same cluster are likely to exhibit similar runtime characteristics; then the actual runtime characteristics of a small number of representative sections in the cluster are likely to manifest the behavior of the entire cluster. Thus, the exhibited runtime characteristics of the representative sections can be used to determine the match between code sections in the cluster and cores without analyzing each section in the cluster.

Classifying a program's execution into sections and sections into clusters independent of the program's input, has several benefits. Most importantly, no development efforts for representative inputs are needed; and thread-to-core assignments for unanticipated use cases and varying architectures are automatically tackled.

Based on these intuitions, phase-based tuning works as follows. A static analysis is performed to identify phase-transition points. This analysis first divides a program's code into *sections* then classifies these sections into one or more *phase types*. The idea is that two sections with the same phase type are likely to exhibit similar runtime behavior characteristics. Third, the analysis identifies points in the program where the control flows [3] from a section of one phase type to a section with a different phase type. These points are called the phase-transition points.

Each phase-transition point is statically instrumented to insert a small code fragment which is called a *phase mark*[1]. A phase mark contains information about the phase type for the current section, code for dynamic performance analysis, and code for making core switching decisions.

At runtime, the dynamic analysis code in the phase marks analyzes the actual characteristics of a small number of representative sections of each phase type. These analysis results are used to determine a suitable core assignment for the phase type such that the resources provided by the core matches the expected resources for sections of that phase type. On determining a satisfactory assignment for a phase type, all future phase marks for that phase type reduce to simply making appropriate core switching decisions[2]. Thus, the actual characteristics of few sections of a given phase type are used as an approximation of the expected characteristics of all sections of that phase type. This allows phase-based tuning to significantly reduce runtime overhead and automatically tackle new architectures. The rest of this chapter describes components of phase-based tuning in detail.

2.1 Static Phase Transition Analysis

The aim of phase-based tunings static analysis is to determine points in the control-flow where behavior is likely to change, that is *phase-transition points*. The precision and the granularity of identifying such points is likely to determine the performance gains observed at runtime. To that end, the first step in this analysis is to detect similarity among basic blocks in the program and classify them into one or more phase types that are likely to exhibit similar runtime behavior. Then, three analysis techniques are examined for detecting and marking phase transitions with *phase marks*. The first is a basic block level analysis. The second builds upon this basic block analysis to analyze intervals [3]. The third also builds upon the basic block analysis to analyze loops inter-procedurally.

[1]The idea of phase marking is similar to the work by Lau *et al.* [50], however, this technique does not use a program trace to determine phase marks and makes its selections based on a different criteria.

[2] Huang *et al.* [38] show that basing processor adaptation on code sections (positional) rather than time (temporal) improves energy reduction techniques. Phase-based tuning also takes a positional approach.

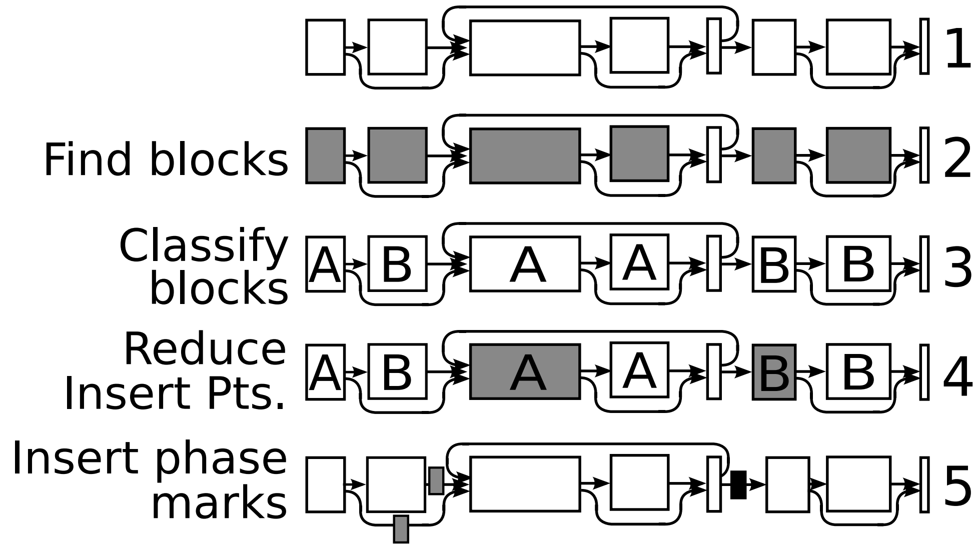

Figure 2.1 Overview of phase transition analysis

Figure 2.1 illustrates this process for a basic block level analysis. Step 1 represents the initial procedure. Blocks in the figure represent basic blocks and edges between them represent paths that may be taken at runtime. Step 2 finds the blocks which are larger than the threshold size (shaded). Next, step 3 finds the type for each block considered in the previous step. Then, the phase transition points are reduced using a lookahead, this is illustrated in step 4. Finally, step 5 shows the new control-flow graph for the procedure which now includes the phase transition marks.

In this section, first the analysis techniques for annotating control-flow graphs (CFGs) with types for all of these techniques are discussed. Next, how to use the annotated control-flow graphs to perform the phase transition marking is discussed.

2.1.1 Static CFG Annotation

The three analysis techniques used to annotate a programs control-flow graph with type information are now discussed. First, the techniques used for basic block analysis are described. Then, this technique is expanded to include a technique for interval typing. Finally, an inter-procedural loop based technique is described.

This static analysis is performed using a custom framework for binary analysis and instrumentation that is described in Chapter 3. While there are limitations to constructing CFGs from a binary representation, since phase-based tuning does not require sound results (incorrect assumptions just result in code that may be less efficient), this technique can make many assumptions safely. For example, suppose part of a code segment has a branch or call with an unknown target. In the current implementation, code segments are typed while ignoring the missing target code's impact on the type. Another option is to skip typing this code segment. If the branch or call has several known potential targets, one could take the following approach. For each target, determine the type for the segment assuming this specific target is executed. If this type is the same for all potential targets, the segment receives this type. If different types are determined as the result of analyzing each different target, then look to see if a majority of the potential targets result in the same type. If they do, this most common type could be chosen, otherwise, the segment may be left untyped. If more precision is desired, a more sophisticated analysis may be used or information may be gathered from the source code.

2.1.1.1 Attributed CFG Construction

The static analysis first divides a program into procedures ($\mathcal{P}$) and each procedure $p \in \mathcal{P}$ into basic blocks to construct the set of basic blocks ($\mathcal{B}$) [3]. The classic definition of a basic block is used (that it is a section of code that has one entry point and one exit point with no jumps in between [3]). The analysis then assigns a type ($\pi \in \Pi$) to each basic block to construct the set of attributed basic blocks ($\bar{\mathcal{B}} \subseteq \mathcal{B} \times \Pi$). The notion of type here is different from types in a program and does not necessarily reflect the concrete runtime behavior of the basic block. Rather it suggests similarity between expected behaviors of basic blocks that are given the same type. A strategy for assigning types to basic blocks statically is given in Section 2.1.3, however, other methods for classifying basic blocks can also be used.

Using the attributed basic blocks, attributed intra-procedural control-flow graphs for procedures are created. An attributed intra-procedural control-flow graph $\mathcal{CFG}$ is $\langle \mathcal{N}, \mathcal{E}, \eta_0 \rangle$. Here,

$\mathcal{N}$, the set of control-flow graph nodes is $\bar{\mathcal{B}} \cup \mathcal{S}$, where $\mathcal{S}$ ranges over special nodes representing system calls and procedure invocations. The set of directed edges in the control-flow is defined as $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N} \times \{b, f\}$, where b, f represent backward and forward control-flow edges. $\eta_0 \equiv (\beta, \pi)$ is a special block representing the entry point of the procedure, where $\beta \in \mathcal{B}$ and $\pi \in \Pi$.

2.1.1.2 Summarizing Intervals

The goal of the intra-procedural interval [3] analysis is to summarize intervals into a single type. To perform the interval analysis, start with the attributed control-flow graph for each procedure created by the basic block analysis. Then use the basic block types to determine interval types.

For each procedure, start by partitioning the attributed control-flow graph of the procedure into a unique set of *intervals* ($\mathcal{I}$) using standard algorithms [3]. "An *interval* ($i(\eta) \in \mathcal{I}$) corresponding to a node $\eta \in \mathcal{N}$ is the maximal, single entry subgraph for which η is the entry node and in which all closed paths contain η [3, pp.6]." For each i, the analysis computes its dominant type (as defined in Figure 2.2) by doing a depth-first traversal of the interval starting with the entry node, while ignoring backward control-flow edges (marked with b) unless traversal gets stuck at a non-leaf node. The exit nodes of the interval represent the leaf nodes. This summarization algorithm is shown in Figure 2.2 and illustrated in Figure 2.1.1.2.

$\rho = \phi$
for all DFS(I) **do**
  if $\eta \in \rho$ **then**
    $M \oplus \{\pi \mapsto M(\pi) + w_b * \varphi(\eta)\}$
  else
    $M \oplus \{\pi \mapsto M(\pi) + w_f * \varphi(\eta)\}$
  end if
  $\rho = \eta + \rho$
  return *max*(*dom*(M))
end for

Figure 2.2 Algorithm for interval summarization to find dominant type

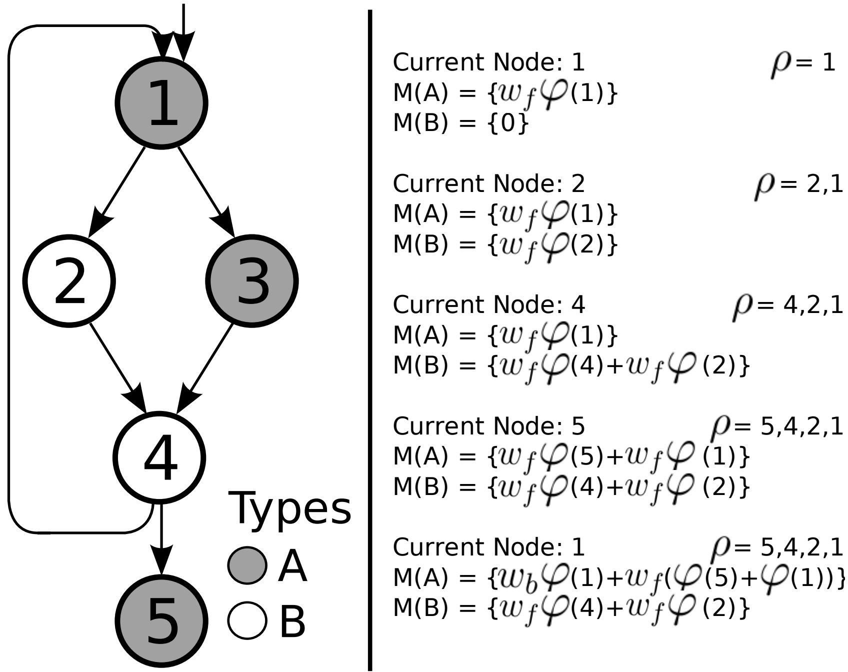

Figure 2.3 Interval summarization illustration

During a depth-first traversal the analysis maintains a stack of control-flow nodes encountered thus far ($\rho = \eta + \rho'$) with the entry node of the interval at the bottom of this stack and the currently visited node at the top of the stack. A type map for the interval ($M : \Pi \mapsto \mathbb{R}$) is maintained. On visiting a control-flow node η in the interval, the type map M is changed to M' where M' is $M \oplus \{\pi \mapsto M(\pi) + w_f * \varphi(\eta)\}$. Here, π is the type of the control-flow node, w_f is the forward edge weight, φ maps nodes to node weights, and $\oplus$ is the overriding operator for finite functions.

On reaching a control-flow node with an outgoing backward edge, if the backward edge has not previously been traversed, the target control-flow node (η') of the backward edge is computed. For each control-flow node η'' from η' to η on the stack ρ, the type map M is changed to M' where M' is $M \oplus \{\pi \mapsto M(\pi) + w_b * \varphi\eta\}$ and w_b is the backward edge weight. The values for w_f and w_b are heuristically decided, but intuitively it makes sense to have w_b greater than w_f (to give more weight to nodes in loops). The node weight function,

$\varphi : \mathcal{N} \mapsto \mathbb{R}$, maps nodes to values based on a heuristic measure of the expected execution time of the block. Currently the number of instructions in the node is used as this measure.

On completion of the depth-first traversal, the dominant type of the interval is π, where $\nexists \pi'.M(\pi') > M(\pi)$. In case of a tie, a simple heuristic is used. Currently, the type with maximal number of control-flow nodes in the interval is used as a tiebreaker.

As a result of this process, the analysis obtains another control flow graph of the procedure where nodes are tuples of intervals and their types. To distinguish these from control-flow graphs of basic blocks, they are referred to as *attributed interval graphs*. It would be interesting to explore whether summarizing interval graphs again is useful [3], however, in this work only first-order intervals are considered. The initial intuition is that the value of applying nth order interval summarization will depend on the average size of procedures.

2.1.1.3 Summarizing Loops

The goal of the inter-procedural loop analysis is to summarize loops into a single type. The analysis starts with the attributed CFG for each procedure created by the basic block analysis. The analysis then uses the basic block types to determine loop types. A bottom-up typing is performed with respect to the call graph. In the case of indirect recursion, the analysis randomly chooses one procedure to analyze first then analyze all procedures again until a fixpoint is reached.

For each procedure, the analysis starts by partitioning the attributed CFG of the procedure into a unique set of *loops* ($\mathcal{L}$) using standard algorithms [63]. For each loop, $l \in \mathcal{L}$, the analysis computes its dominant type starting with the inner-most loops. The analysis does a breadth-first traversal of the loop starting with the entry node, while ignoring backward edges. This algorithm is shown in Figure 2.5 and illustrated in Figure 2.4.

Throughout the loop traversal, a type map ($M : \Pi \mapsto \mathbb{R}$) is maintained which maps types to weights. On visiting a control-flow node in the loop, $\eta \in l$, the type map M is changed to $M' = M \oplus \{\pi \mapsto M(\pi) + w_n(\lambda) * \varphi(\eta)\}$. Here, π is the type of the control-flow node η, w_n

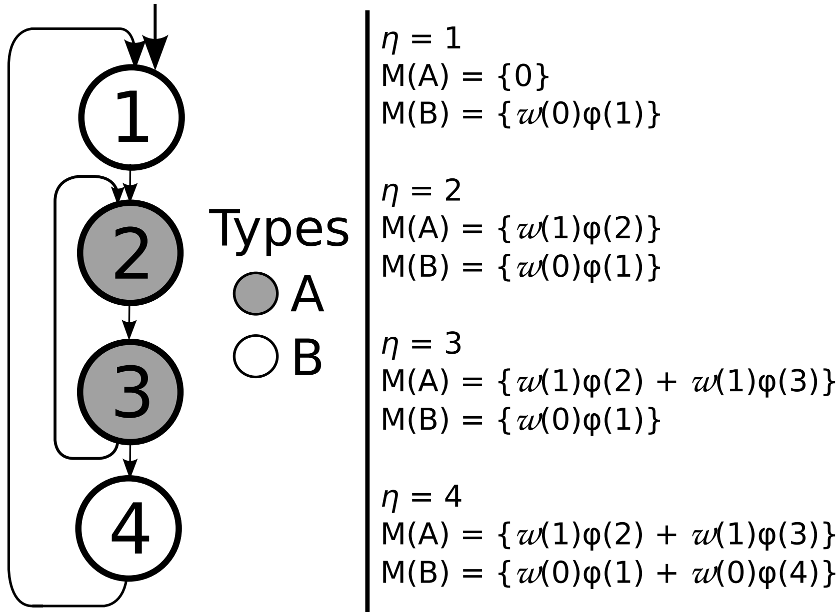

Figure 2.4 Loop summarization illustration

maps nodes to nesting level weights, φ maps nodes to node weights, and $\oplus$ is the overriding operator for finite functions. Since loops are usually executed multiple times, nodes in nested loops should have more impact on the type of the overall loop. Thus, nodes which belong to inner loops are given a higher weight via the function $w_n : \mathbb{N} \rightarrow \mathbb{R}$ which maps nesting levels to weights.

On completion of the breadth-first traversal, the dominant type of the loop l is π_l, where $\nexists\pi$ s.t. $M(\pi) > M(\pi_l)$. In case of a tie, a simple heuristic is used (e.g. number of control-flow nodes). The analysis also has a *type strength*, σ which is simply the weight the type π_l over the sum of all other type weights $(M(\pi_l)/\sum_{\pi\in dom(M)} M(\pi))$. This strength is used for typing nested loops.

Suppose there is a loop l' which contains the current loop l. If both loops have the same type $(\pi_{l'} = \pi_l)$, it is not beneficial to incur the analysis and optimization code's overhead at each iteration of the outer loop. Instead, runtime analysis and optimization is performed before the outer loop, and eliminate any work done inside this loop. Thus, after the type for

the current loop, l, is determined, the analysis finds the type for the next largest nested loop, l'. If there is no such loop, then the analysis adds the current type information to the loop type map T. If the type of the nested loop (l') is the same as the current loop (l), then the analysis adds the current loop, l to the type map and remove the nested loop l'. If the types of the two loops differ, the analysis takes the type with the higher strength, σ, since it is more likely that the typing for such a loop is more accurate. Finally, there is a special condition (**else if**) to handle nesting where two disjoint loops, l' and l''', are nested inside a loop, l. In this case, the analysis types the loop l only if the two disjoint loops, l' and l''', have the same type which is also the same type as the outer loop l.

for all $\eta \in \text{BFS}(l \in \mathcal{L})$ **do**
  $\lambda := \left|\{l' \in \mathcal{L} | l' \subset l \wedge \eta \in l'\}\right|$
  $M \oplus \{\pi \mapsto M(\pi) + w_n(\lambda) * \varphi(\eta)\}$
end for
$M(\pi_l) = \max_{\pi \in dom(M)}(M(\pi))$
$\sigma_l := M(\pi_l) / \sum_{\pi \in dom(M)} M(\pi)$
if $\exists l'$ s.t. $l' \subset l \wedge \nexists l''$ s.t. $l' \subset l'' \subset l \wedge (\nexists l'''$ s.t. $l''' \subset l \wedge \nexists l''$ s.t. $l''' \subset l'' \subset l)$ **then**
  if $(l', \pi_{l'}, \sigma_{l'}) \in T \wedge (\pi_{l'} = \pi_l \vee \sigma_{l'} < \sigma_l)$ **then**
    $T := T \cup \{(l, \pi_l, \sigma_l)\}$
    $T := T \setminus \{(l', \pi_{l'}, \sigma_{l'})\}$
  end if
else if $\exists l'$ s.t. $l' \subset l \wedge \nexists l''$ s.t. $l' \subset l'' \subset l \wedge (\exists l'''$ s.t. $l''' \subset l \wedge \nexists l''$ s.t. $l''' \subset l'' \subset l)$ **then**
  if $(l', \pi_{l'}, \sigma_{l'}) \in T \wedge (l''', \pi_{l'''}, \sigma_{l'''}) \in T \wedge \pi_{l'} = \pi_{l'''} \wedge \pi_{l'} = \pi_l$ **then**
    $T := T \cup \{(l, \pi_l, \sigma_l)\}$
    $T := T \setminus \{(l', \pi_{l'}, \sigma_{l'})\}$
    $T := T \setminus \{(l''', \pi_{l'''}, \sigma_{l'''})\}$
  end if
else
  $T := T \cup \{(l, \pi_l, \sigma_l)\}$
end if

Figure 2.5 Loop summarization to find dominant type. BFS ignores back edges

As a result of this process, the analysis obtains another control flow graph of the procedure where nodes are tuples of loops and their types. To distinguish these from control-flow graphs of basic blocks, they are referred to as *attributed loop graphs*.

2.1.1.4 Phase Transitions

Once the analysis has determined types for sections (blocks, intervals, or loops) of the program's CFG, the analysis computes the phase transition points. Recall that a phase-transition point is a point in the program where runtime characteristics are likely to change. Since sections of code with the same type should have approximately similar behavior, phase-based tuning assumes that program behavior is likely to change when control flows from one type to another. The next section describes the techniques for marking these points in the application.

2.1.2 Phase Transition Marking

Once the phase transitions are determined, phase marks are statically inserted in the binary to produce a standalone binary with phase information and dynamic analysis code fragments. These code fragments also handle the core switching. By instrumenting binaries, the need for compiler modifications is eliminated. Furthermore, by using standard techniques for core switching, no OS modification is required. Several variations of phase transition marking have been considered that are classified into three kinds based on whether it operates on the attributed control-flow graphs, the attributed interval graphs, or the attributed loop graphs. In all cases, phase marks are placed at the phase transitions.

2.1.2.1 Adding Phase Marks to Attributed CFG

The first class of methods all consider a section to be a basic block ($\bar{\beta}$) in the attributed CFG ($\mathcal{CFG}$). The advantage of using basic blocks is that execution of a single instruction in a block implies that all instructions in the block will execute (and the same number of times). This means that the phase type for the section is likely to be accurate and the same as the corresponding basic block type $\pi \in \Pi$, where $\bar{\beta}$ is (β,π). This naïve phase marking technique marks all edges in the attribute CFG where the source and the target sections have different phase types. As is evident, this technique has a problem. The average basic block size is small (tens of instructions in the SPEC benchmarks). Phase marking at this granularity could

result in frequent core switches overshadowing any performance benefit. To avoid this, two techniques are used.

The first technique eliminates small sections of code. In other words, if the section has less than a threshold weight as defined by a node weight function, $\varphi : \mathcal{N} \mapsto \mathbb{R}$. This eliminates core switching for very small blocks of code. For example, a basic block may consist of a single instruction. Clearly it would not be cost effective to initiate a core switch so that a single instruction can execute more efficiently. Basic blocks are usually in the tens of instruction and often smaller. Even at this size the benefit of switching cores probably does not outweigh the cost of switching cores. So, better points for phase marks still must be chosen. The second technique addresses this problem by only marking a section if at least a fixed percentage of its successors up to a fixed depth have the same type (illustrated in Figure 2.7).

2.1.2.2 Lookahead based Phase Marking

This technique is presented in Figure 2.6 and illustrated in Figure 2.7. The intuition is the following. If the successors of a section have the same type, it is more likely that a core switch will be worth its cost. For small loops, when enough successors are considered, nodes begin to repeat. Thus, if a loop contains predominately one type of blocks, the technique can simply make a core switch before the loop begins. Furthermore, this technique serves to reduce the number of phase marks in a program. Since adding each phase mark translates to adding a small number of instructions to the footprint of the binary and the control-flow path, both the time overhead and space overhead of the technique will be reduced and will hopefully not eliminate much of its benefit.

2.1.2.3 Adding Phase Marks to Attributed Interval Graphs

The second class of methods consider sections to be intervals in the attributed interval graph. Using intervals for phase marking enables us to look at the program at a more coarse granularity than basic blocks. Even with 1st order interval graphs, the intervals frequently

Processed Nodes, $\mathcal{D}$
Get Successors to Depth, $\mathcal{S} : (\eta, \mathbb{N}) \rightarrow \{\bar{\mathcal{B}}\}$
Lookahead depth: d, Successor threshold: e
Same type count: c, Total count: t
Grouping $\mathcal{U} = \{\pi \mapsto N | \forall \pi \in \Pi\}$
Node list $N = \{\nu\}$.
for all $p \in \mathcal{P}$ **do**
  $\mathcal{D} = \phi$
  for all $(\eta, \pi) \in (\bar{\mathcal{B}} \setminus \mathcal{D})$ **do**
    $c \leftarrow 0$, $t \leftarrow 0$, $S = \mathcal{S}(\eta, d)$
    for all $(\eta', \pi') \in S$ **do**
      if $\pi' = \pi$ **then**
        $c \leftarrow c + 1$
      end if
      $t \leftarrow t + 1$
    end for
    if $c/t \geq e$ **then**
      $\mathcal{U} \oplus \{\pi \mapsto \mathcal{U}(\pi) \cup \{\eta\}$
      $\mathcal{D} = \mathcal{D} \cup \{\eta\} \cup S$
    end if
  end for
end for

Figure 2.6 Lookahead based phase marking

capture small loops. This is clearly advantageous for adding phase marks since it is not desired to have a core switch within a small loop because this would most likely result in far too frequent core switches. The disadvantage is that interval summarization to obtain dominant types introduces imprecision in the phase type information. As a result, statically computed dominant type may not to be actual exhibited type for the interval based on which instructions in the interval are executed and how many times they are executed.

2.1.2.4 Adding Phase Marks to Attributed Loop Graphs

The third class of methods consider a section to be loops in the attributed loop graph. Using loops for phase marking has even more advantages than using intervals. Not only does it allow inserting outside of loops, it also allows better handling of nested loops by frequently

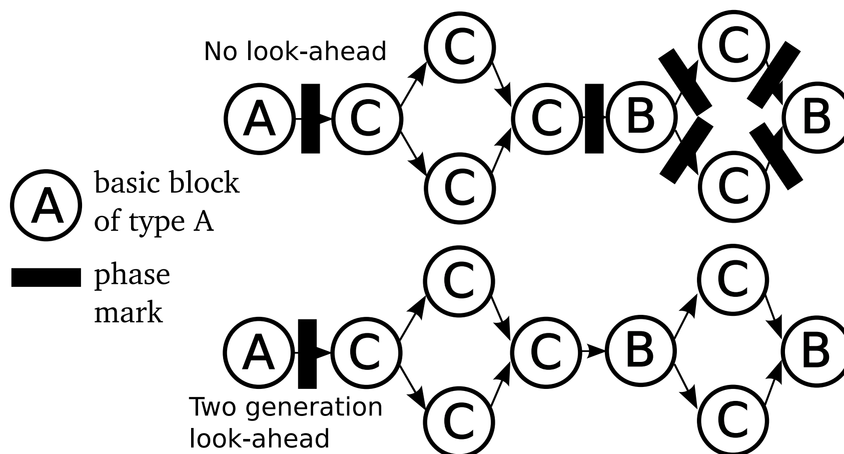

Figure 2.7 Lookahead based reduction of phase marks

eliminating phase-marks within loop iterations. This is an even more coarse view of the program than the interval based technique. Furthermore, since it is an inter-procedural analysis, transitions across function calls are handled. Just like interval typing, loop typing introduces some imprecision in the type information.

2.1.3 Determining block types

This chapter and its corresponding evaluation focuses more on developing and evaluating (1) the various techniques and granularity for determining and marking phase transitions and (2) the dynamic analysis and optimization techniques. As a proof-of-concept, a simple static analysis for determining types of basic blocks is used.

This analysis involves looking at a combination of instruction types as well as a rough estimate of cache behavior (computation based on reuse distances [11]). Information describing these two components are used to place blocks in a two dimensional space. The blocks are then grouped using the k-means clustering algorithm [55].

The accuracy of this approach has been evaluated in combination with the loop based clustering technique as follows. Blocks are classified into groups by using this simple analysis. Next, the dominant type of the loops are determined using the algorithm from Section 2.1.1. This loop typing is compared with the actual observed behavior of the loops.

In summary, experimentation shows that this technique miss-classifies only about 15% of loops. As the results show in Chapter 4, this is accurate enough for phase-based tuning. A more precise analysis (such as lazy grouping which is described in Chapter 6) may simply be substituted to improve overall performance.

2.2 Dynamic Analysis and Tuning

After phase transition marking is complete, the program binary contains phase marks at appropriate points in the control flow. These phase marks contain an executable part and the phase type for the current section. The executable part contains code for dynamic performance analysis and section-to-core assignment. During the static analysis, this dynamic analysis code is customized according to the phase type of the section to reduce overhead.

The code in the phase mark either makes use of previous analysis to make its choice of core types or it observes the behavior of the code section. A variety of analysis policies could be used and any desirable metric for determining performance could be used as well. In this section, the lightweight analysis and similarity metric used for evaluating phase-based tuning is described.

For this case, the code for a phase mark serves two purposes: First, during a transition between different phase types, a core switch is initiated. The target core is the core previously determined to be a good fit for this phase type. Second, if a good fit for the current phase type has not been determined, the current section is monitored to analyze its performance characteristics. The decision about the preferred core for that phase type is made by monitoring representative sections from the cluster of sections that have the same phase type. By performing this analysis at runtime, phase-based tuning does not require the programmer to have any knowledge of the target architecture. Furthermore, the asymmetry is determined at runtime removing the need for multiple program versions customized for each target architecture. Since the static technique ensures that sections in the same cluster are likely to exhibit similar

runtime behavior, the assignment determined by just monitoring few representative sections will be valid for most sections in the same cluster. Thus, monitoring all sections will not be necessary. This helps to reduce the dynamic overhead of phase-based tuning.

For analyzing the performance of a section, instructions per cycle (IPC) is measured (similar to [81, 9]). IPC correlates to throughput and utilization of AMPs. For example, cores with a higher clock frequency can efficiently process arithmetic instructions whereas cores with a lower frequency will waste fewer cycles during stalls (e.g. cache miss). IPC is monitored using hardware performance counters prevalent in modern processors. The preferred core assignment is determined by comparing the observed IPC for each core type.

The technique for determining core assignment is shown in Figure 2.8. The intuition is that cores which execute code most efficiently will waste fewer clock cycles resulting in higher observed IPC. Since such cores are more efficient, they will be in higher contention. Thus, the algorithm picks a core that improves efficiency but aims to not overload the efficient cores.

select(π,δ): *best core for phase type π, with threshold δ*
  $C := \{c_0, c_1, \ldots, c_n\}$ (set of cores)
  Sort C s.t. $i > j \Rightarrow f(c_i, \pi) > f(c_j, \pi)$.
  $f(c_i, \pi)$ - the actual measured IPC of block type π on core c_i.
  $d \leftarrow c_0$
  for all $c_i \in C \backslash \{c_n\}$ **do**
    $\theta = f(c_{i+1}, \pi) - f(c_i, \pi)$
    if $\theta > \delta \wedge f(c_{i+1}, \pi) > f(d, \pi)$ **then**
      $d \leftarrow c_{i+1}$
    end if
  end for
  return d

Figure 2.8 Algorithm for expected optimal core assignment for n cores

This algorithm first sorts the observed behavior on each core and sets the preferred core to the first in the list. Then, the algorithm steps though the sorted list of observed behaviors. If the difference between the current and previous core's behavior is above some threshold, the preferred core is set to the current core. The intuition is that when the difference is above

the threshold, executing on the efficient core will save enough cycles to justify taking the space on the more efficient core. By doing the performance analysis at runtime, this algorithm for computing the preferred core assignment *does not require knowledge of the program or underlying architecture*.

CHAPTER 3. Analysis and Instrumentation Framework

To evaluate the ideas in this thesis, a custom static binary analysis and instrumentation framework was developed. In this chapter, a brief overview of this framework is given.

3.1 Disassembly

At the lowest level, the tool uses code and libraries from GNU BinUtils to translate programs from binary to machine opcodes and operands. Next, a custom analysis converts this assembly like representation to an object oriented representation of the program.

First, addresses must be converted to labels. That is, instead of knowing that an instruction may jump to a specific address, the tool must know what instruction this address refers to. Consider the small example in Figure 3.1. On the left side of this figure, we have the unprocessed assembly representation of the program. On line 4, there is a conditional jump to the address corresponding to the instruction on line 2. The same code after translation is shown on the right side of the figure. The figure shows that now, the conditional jump operand is a new label instead of an address. Also, the target instruction now has a label associated with it.

```
1 ...
2 add $0x1, %eax
 ;at address 0x804896ab
3 ...
4 jne 0x804896ab
5 ...
```

```
1 ...
2 newLabel1: add $0x1, %eax
3 ...
4 jne $newLabel1
5 ...
```

Figure 3.1 Conversion from address to labels. The left side shows a few instructions before conversion. The right shows the same instructions after conversion.

The tool must also correctly lay out the data sections (e.g. `.rodata`, `.data`, and `.bss`) including translating address values which occur in these data sections (`.rodata` and `.data`) and special symbols which also occur in these sections (e.g. `stderr`, `stdout`, `stdin`).

This representation is capable of being output as assembly code which is able to be assembled (i.e. converted back to a binary) by a standard assembler. However, the main reason behind this representation is to enable analysis and instrumentation of the program.

3.2 Analysis

The largest and most complex components of this framework are the various static analysis techniques. An overview of each of these components is now described.

3.2.1 Control Flow Analysis

The tool includes a range of control flow analysis techniques including intra-procedural analyses for finding basic blocks and intervals. Further the tool can identify various control structures (e.g. while loops, do-while loops, if-then, and if-then-else) and, if desired, graphically present these structures [77]. The targets of calls are also determined in order to perform inter-procedural loop analysis as is used for phase-based tuning (and other static analysis described later).

3.2.2 Abstract Interpretation

Abstract interpretation is a sound static program analysis framework for analyzing programs with respect to all possible program paths [19].

For example, the analysis may choose to analyze a property such as the sign of values contained by variables. Consider the example in Figure 3.2. The analysis starts by analyzing the first block, A, resulting in the state S_2 that contains the knowledge that `x` is positive. Next, the analysis reaches the branch (in this case an if-then-else structure). Since the analysis knows

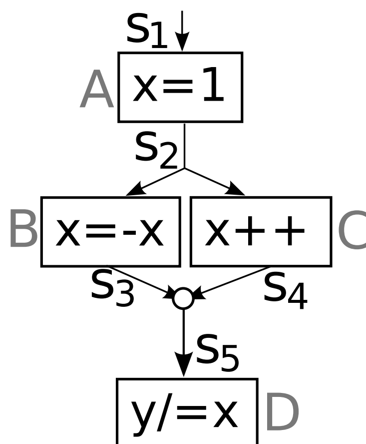

Figure 3.2 Example of simple abstract interpretation based analysis for variable sign.

that the sign of the variable x is positive each branch is analyzed with this information. For the left path (block B) the sign is flipped (x becomes negative). So, state S_3 knows that x will be negative at this point. For the right path (block C), x is incremented. Since addition of two positive numbers (1 and x) results in another positive number, x remains positive for this path. So, state S_4 knows that x will be positive at this point. Next, the analysis needs to analyze the next block, D. To analyze this block, an input state is needed. To create this input state, the states S_3 and S_4 (the results for each potential path) must be merged or "joined". One case says that x is negative and the other says it is positive. Thus the resulting joined state S_5 says that x may be either positive or negative (or "top" may be used, which means unknown).

This tool contains an abstract interpretation framework for analyzing both sequential and parallel programs. To implement such an analysis, one simply creates their analysis class inheriting from a base "Abstraction" class. Then, the user must only implement a few simple functions, "step" which defines how an individual instruction is analyzed, "join" which defines how two combine to states, and "equals", "copy", and "destruct" methods which have the standard meaning. For analyzing parallel programs, a "parallelCombine" method must also be defined which handles merging the results from two threads.

On top of this framework, several specific analyses are provided. This includes analysis for multi-level cache behavior and cache coherency behavior. Analysis for ILP and cache

behavior based on reuse distance [11] (described later) are also provided which are similar to the abstract interpretation based analyses, but do not guarantee soundness.

3.3 Instrumentation

The tool also has the ability to instrument (or modify) the input program. That is, at a point on the control flow path (assuming enough information is available) user supplied code may be added.

The main goal of this feature is to output programs that execute efficiently. This is a major goal since low overhead of instrumentation is crucial to reduce the overhead of phase-marks for phase-based tuning. Compared to a similar static instrumentation tool, ATOM [80], binaries instrumented with this tool execute 10 times faster[1]. This is because this tool uses a binary instrumentation strategy that is finely tuned for specific optimizions compared to that of a general strategy used by ATOM. Further, the tool ignores instrumentation for cases where instrumentation is especially difficult due to insufficient static information.

Instrumentation occurs as follows. The user writes their own (or uses an existing) tool that chooses points for insertion (similar to ATOM [80] or Pin [54]). Consider the input program on the left side of Figure 3.3. Suppose the user wants to insert some code before the second instruction.

For each such insertion point, the user specifies a function name to insert. This function must exist in a user supplied binary file. Though each piece of code to be inserted is contained in a function the entire function is not inserted into the resulting binary. The tool inserts the body of this function along with the necessary context saving and restoring before and after the inserted code. On the right side of the figure (the output program) the code to be inserted has been added (toward the bottom) along with the necessary context saving and restoring code around it.

[1]These experiments were done by inserting code before every basic block for SPEC CPU2000 benchmarks.

```
1 0x80488800: mov %eax,%edx
2 0x80488802: mov 0xef1a2fb3,%eax
3 0x80488807: lea (%edx,%eax,1),%eax
```

```
1 0x80488800: mov %eax,%edx
2 0x80488802: jmp $newLabel1
3 0x80488806: nop
4 0x80488807: lea (%edx,%eax,1),%eax
5 ...
6 newLabel1:
7 ; code to save context
8 ; insert function
9 ; code to restore context
10 mov 0xef1a2fb3,%eax
11 jmp 0x80488807
```

Figure 3.3 Example of binary instrumentation (idea based on fast break-points [41]). Left: input program. Desired insertion point is before the second instruction. Right: output program. Instructions from the input program are shaded. The second instruction was replaced with padding and a jump to the inserted code plus the replaced instruction.

Now, control flow must be modified to reach this new code. Also, in doing so, the tool must not "break" indirect branches which occur in the program being modified. The technique used for solving this problem is similar to the idea of fast break-points [41]. Recall that the user wanted to insert the code before the second instruction. Notice in the figure that this instruction has been relocated (original instructions are shaded). In its original place, a jump instruction is inserted as well as padding to make sure that the modified code takes up the same space (so that indirect branches still target the correct instruction). This jump target goes to the context save code which occurs before the inserted function code. Additionally, after the context restore code (which is after the inserted function code), the relocated instruction and another jump are inserted. This jump returns back to the point after the relocated instruction. If desired, call/return could be used instead of jumps and code duplication to potentially improve cache behavior and space overhead. However, this has the disadvantage of being unable to optimize the inserted code based on the "caller".

By analyzing and instrumenting binaries rather than source code, compiler modification is not required and the analysis techniques are applicable to any binary rather than a single language or set of languages.

CHAPTER 4. Evaluation of Phase-based Tuning

The aim of this chapter is to evaluate the five claims made in Chapter 1 regarding phase-based tuning. First, it was claimed that phase-based tuning requires no knowledge of program behavior or performance asymmetry. Phase-based tuning is completely automatic and requires no input from the programmer. In these experiments, workloads are generated randomly and without any knowledge of behavior of the benchmarks. Second, it was claimed that the technique allows for transparent deployment. Since the analysis and instrumentation framework operates on binaries, no modification to compilers is necessary. Furthermore, since standard techniques are used for switching cores, no OS modifications are necessary. For example, in these experiments, the standard build scripts and compilers for the SPEC CPU benchmarks are used with an unmodified Linux OS. Third, it was claimed that with phase-based tuning one can "tune once and run anywhere". The static analysis makes no assumptions about the underlying asymmetry. Since performance analysis and section-to-core assignment are done dynamically, the same instrumented applications may be run on varying asymmetric systems. The final two claims are those related to performance: negligible overhead and improved utilization. In the rest of this chapter, experimental results are used to evaluate these two claims.

First, it is shown that phase-based tuning has low overhead, second, that phase-based tuning significantly improves the throughput of processes compared to standard Linux scheduler, and third that it maintains fairness among processes compared to the standard Linux scheduler. Finally, techniques are compared to show how different variations are applicable for various scheduling goals.

4.1 Experimental Setup

This section describes the experimental setup including both hardware and software platforms. It is also discussed how workloads were constructed for these experiments.

4.1.1 System Setup

The system consists of an AMP with 4 cores. This setup uses an Intel Core 2 Quad processor with a clock frequency of 2.4GHz. To create an asymmetric system, two cores underclocked to 1.6GHz. There are two L2 caches shared by two cores each. The cores running at the same frequency share an L2 cache. An unmodified Linux 2.6.22 kernel (which uses the $O(1)$ scheduler) is used with standard compilers. Thus, the transparent deployment benefit of phase-based tuning is demonstrated.

There are two main benefits of using a physical system instead of a simulated system. First, porting the implementation to another system is trivial since it does not require any modifications to the standard Linux kernel. Second, phase-based tuning is analyzed in a realistic setting. Others have argued that results gathered through simulation may be inaccurate [57]. This is because all aspects of the system are not considered. Therefore, a full system simulator is desired. This setup is limited in hardware configurations to test. However, this platform is sufficient to show the utility of phase-based tuning.

The perfmon2 monitoring interface [25] is used to measure the throughput of workloads. For evaluation purposes, to determine basic block types for the static analysis with little to no error, an execution profile from each core is used. Using the observed IPC, basic blocks are assigned types. The *difference* in IPC between the core types is compared to an *IPC threshold* to determine the typing for basic blocks.

4.1.2 Workload Construction

Many systems receive a nearly constant feed of jobs to run [10]. Improving the overall throughput of such a system will increase the amount of jobs the machine can complete in an interval of time. This increase will in turn enable the system to handle larger workload sizes. Phase-based tuning is targeting these systems, with maximizing throughput as its key objective. Similar to Kumar *et al.*[45] and Becchi *et al.*[9] the workloads range in size from 18 to 84 randomly selected benchmarks from the SPEC CPU 2000 and 2006 benchmark suites. For example, when testing a workload of size 18 there are 18 benchmarks running simultaneously. Such a workload is referred to as having 18 slots for benchmarks. Like Kumar *et al.*[45] the system receives jobs periodically, except rather than jobs arriving randomly, workloads maintain a constant number of running jobs. To achieve this constant workload size, upon completion of a benchmark, another benchmark is immediately started. If one were to simply restart the same benchmark upon completion, the same benchmarks may continuously complete if the technique favors a single type of benchmark. Thus, a job queue is maintained for each workload slot. That is, for a workload of size 18 then there are 18 queues (one for each slot in the workload). These 18 queues are each created individually from randomly selected benchmarks from the benchmark suites. When a workload is started, the first benchmark in each queue is run. Upon completion of any process in a queue, the next job in the queue is immediately started. When comparing two techniques, the same queues were used for each experiment. This ensures more accurate capture of the behavior of an actual system.

4.2 Space and Time Overhead

Statically, phase marks (consisting of data and code) are inserted in the program to enable phase-based tuning. Since insertion of large chunks of code may destroy locality in the instruction cache, low space overhead is desired. This section first describes the overhead in terms of the increase in binary size caused by insertion of phase marks. Also, a phase mark's

execution time is added to the execution of the original program. Thus, it must be ensured that the overhead does not overshadow the gains achieved by phase-based tuning. Therefore, the time overhead is described in terms of increase in execution time over the uninstrumented version. Finally, the average number of cycles per core switch is observed for the benchmarks (this can be thought of as the average cycles between core switches rather than the cost of core switch in cycles).

4.2.1 Space Overhead

To measure space overhead, the sizes of the original and modified binaries are compared for variations of phase-based tuning. Table 4.1 shows summary statistics and Figure 4.1 shows a box plot for the measurements taken from the benchmarks in the SPEC CPU 2000 and 2006 benchmark suites. The box represents the two inner quartiles and the line extends to the minimum and maximum points. These results are presented in terms of what percentage of the instrumented application is made up of phase marks. The trends are expected. As the minimum size increases, space overhead decreases. Similarly, as lookahead depth increases, space overhead generally decreases. For individual programs this is not always the case because by adding another depth of lookahead, the percentage of blocks belonging to the same type may be pushed over the threshold causing another insertion point.

These results confirmed the intuition that less phase marks will be inserted for larger minimum sizes and lookahead depths. The results for interval graph-based phase marking are interesting in that they show significantly large increase in binary size. This is primarily because interval summarization results in the grouping of smaller basic blocks into intervals creating more sections above the instruction size threshold. The trends in space overhead offer insight into trends in time overhead.

For the best technique (loop technique with minimum size of 45), there is less than 3% space overhead. For the same technique there is an average of 20.24 phase marks per benchmark where each phase mark is at most 78 bytes.

Technique	Space overhead of phase marks (in %)			
	Average	Minimum	Maximum	Std. Dev
BB[10,0]	35.58	0.26	77.29	0.26
BB[10,1]	19.39	0.05	46.54	0.15
BB[10,2]	12.31	0.05	39.46	0.12
BB[10,3]	13.61	0.26	31.51	0.11
BB[15,0]	8.62	0.05	27.43	0.08
BB[15,1]	5.32	0.05	26.18	0.07
BB[15,2]	9.23	0.12	19.58	0.08
BB[15,3]	4.93	0.05	18.74	0.05
BB[20,0]	4.00	0.05	18.35	0.05
BB[20,1]	8.71	0.05	17.75	0.07
BB[20,2]	5.16	0.05	16.70	0.05
BB[20,3]	4.13	0.05	16.70	0.05
Int[30]	18.92	0.48	36.35	0.11
Int[45]	12.15	0.39	26.70	0.08
Int[60]	8.08	0.26	19.33	0.06
Loop[30]	5.69	0.05	32.13	0.09
Loop[45]	3.98	0.05	30.28	0.08
Loop[60]	3.48	0.05	27.71	0.08

Table 4.1 Space overhead of phase marks. BB[n,m]: basic block technique with min. block size: n, lookahead: m. Int[n]: interval technique with min. interval size: n. As expected, fine grain techniques tend to have higher space overhead.

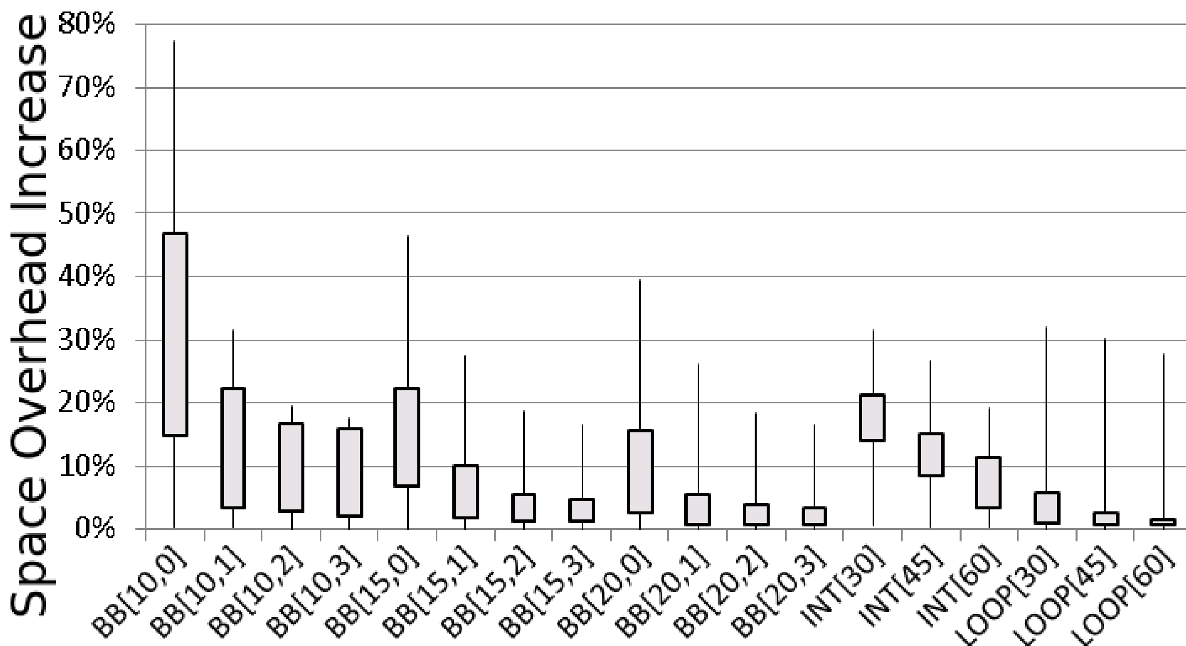

Figure 4.1 Space overhead of phase marks. As expected, fine grained techniques have higher space overhead. Loop based techniques have the lowest overhead (e.g. 3%).

4.2.2 Time Overhead

To measure the time overhead of phase marks and core switches the following is done. Instead of the programs switching to a specific core, programs switch to "all cores". Switching to "all cores" means that the same API calls are made that optimized programs make, however, instead of defining a specific core, all cores in the system are given. Thus, the difference in runtime between the unmodified binary and this instrumented binary shows the cost of running the phase marks at the predetermined program points. Table 4.2 shows these costs for variable workload sizes. Figure 4.2 shows results for workloads of size 84.

The trends shown are mostly expected and are similar to those for space overhead. What makes these results interesting is that in some cases overhead was as little as 0.14%. At first, it is quite surprising that the loop based technique reduced overhead as much as it did. There are several reasons for this improvement. First, compared to the interval and basic block techniques, only loops are considered whereas the other techniques considers many intervals and groups of blocks which are not loops. On top of this, it considers nesting of loops. Clearly,

Technique	% time spent in phase marks			
	36	52	68	84
BB[10,0]	12.46	8.80	8.98	9.04
BB[10,1]	9.46	7.12	8.75	8.30
BB[10,2]	6.45	7.42	8.75	8.36
BB[10,3]	8.31	7.52	7.47	7.01
BB[15,0]	7.31	5.44	6.57	5.53
BB[15,1]	6.16	4.06	5.66	4.61
BB[15,2]	7.31	3.46	5.06	5.59
BB[15,3]	5.16	6.33	5.81	5.47
BB[20,0]	6.30	4.75	5.21	4.18
BB[20,1]	5.30	5.54	5.96	3.87
BB[20,2]	5.59	5.04	6.26	4.24
BB[20,3]	7.16	5.04	6.26	3.56
Int[30]	29.03	18.83	19.42	22.44
Int[45]	19.54	16.55	15.41	16.47
Int[60]	15.13	10.97	10.55	12.33
Loop[30]	0.81	0.69	0.32	0.18
Loop[45]	0.60	0.61	0.64	0.17
Loop[60]	0.29	0.26	0.23	0.14

Table 4.2 Time spent in phase marks. As expected, fine grain techniques tend to have higher time overhead. However, interval techniques have very high overhead. Loop based techniques have less than 0.2% time overhead.

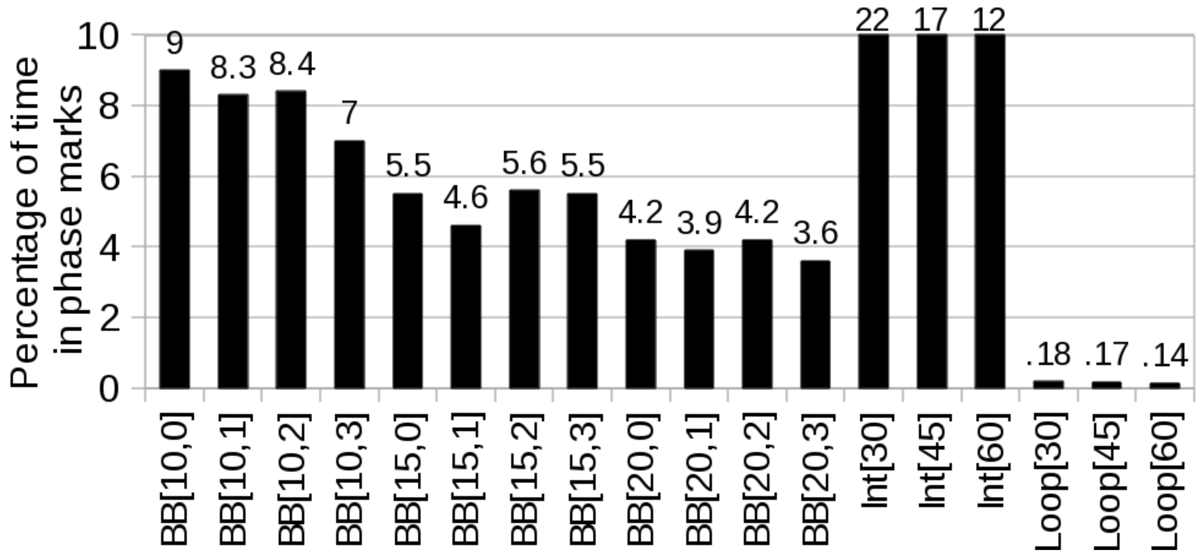

Figure 4.2 Time overhead of phase marks. Gathered using a workload size of 84. As expected, fine grained techniques tend to have higher time overhead. However, interval techniques have very high overhead. Loop based techniques have less than 0.2% time overhead.

removing an insertion point inside of a nested loop will greatly reduce the number of total executions of phase marks. Not only does the technique consider nesting, but it also considers function calls in order to eliminate phase marks in functions that are called inside of loops thereby eliminating more phase marks from loops. Further optimized instrumentation and core switching techniques are likely to decrease this overhead even more. Also, tighter integration with the system scheduler is likely to decrease this overhead as well, but at the expense of requiring OS modification.

These results show that phase-based tuning has a small overhead both in terms of space and time, which shows the *scalability* of phase-based tuning. For long running processes, the overheads are likely to decrease further. Since only a small number of blocks need to be monitored at run-time, long running benchmarks will most likely have more time to take advantage of the section-to-core assignment determined by phase-based tuning. This is especially the case for many server applications such as daemons. For example, a web server will determine its assignment quickly, then be able to make use of this assignment throughout its entire up-time.

4.2.3 Core Switches

Here the frequency and cost of core switches for each benchmark is analyzed. First, experiments were done to estimate the cost of each core switch. This was done by writing a program that alternates between cores and then counting the cycles of execution for this program. Using this technique, it has been determined that a core switch takes approximately 1000 cycles (not including the potential for additional cache misses caused by the loss of cached instructions and data). More precise measurement could be done, but this is sufficient to gain insight into the necessary cycles require to amortize the cost of a core switch. Next, consider core switches for each benchmark in detail.

Benchmark	Switches	Runtime (s)
401.bzip2 (2006)	4837	364
410.bwaves (2006)	205	33636
429.mcf (2006)	15	872
459.GemsFDTD (2006)	0	3327
470.lbm (2006)	99	1123
473.astar (2006)	0	55
188.ammp (2000)	3	67
173.applu (2000)	205	3414
179.art (2000)	3	46
183.equake (2000)	7715	62
164.gzip (2000)	3	23
181.mcf (2000)	6	58
172.mgrid (2000)	2005	172
171.swim (2000)	3204	5720
175.vpr (2000)	6	46

Table 4.3 Switches per benchmark (Loop[45], 0.2 threshold)

In Table 4.3 the number of core switches and runtime (in isolation) for each benchmark is shown. This table shows that most programs change phase types occasionally throughout execution. Some programs differ in that they have few or only one phase according to the static analysis. These benchmarks, aside from choosing the best core to execute on, mostly stay on

the same core. Finally, two benchmarks (459 and 473), do not have any phases at all. These benchmarks will simply execute on any core the OS deems appropriate.

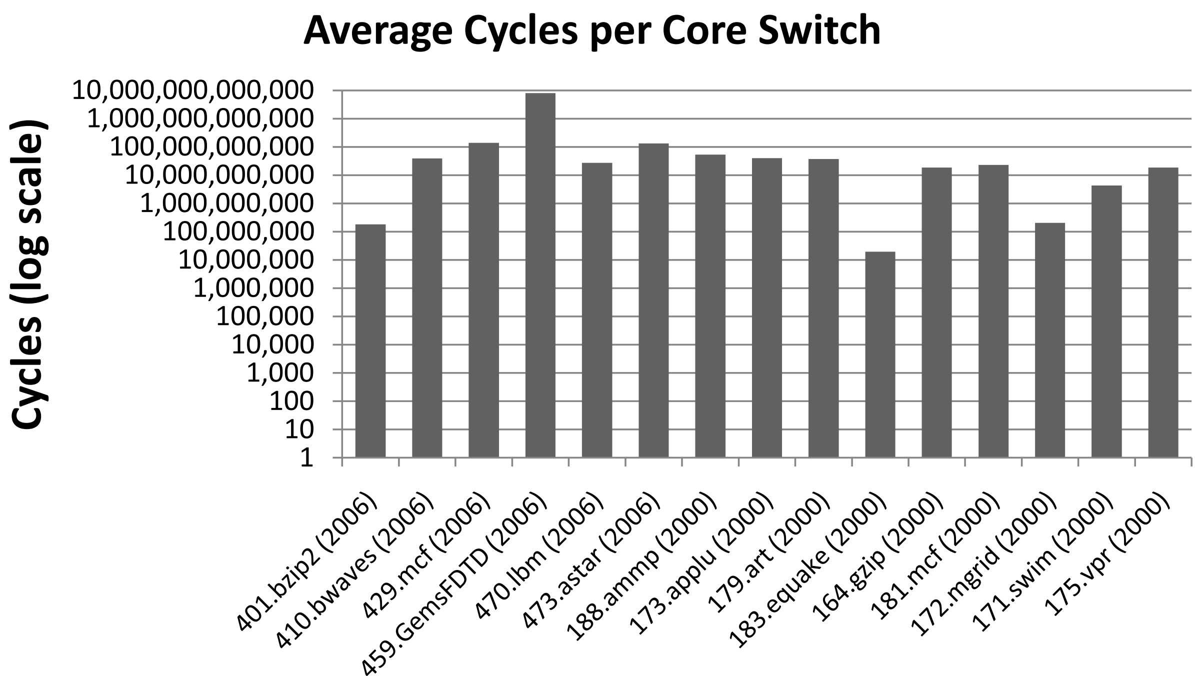

Figure 4.3 Average cycles between core switches

Figure 4.3 presents the average number of cycles per core switch (log scale). Most benchmarks fall in the range of tens of billions of cycles per core switch which is clearly enough amortize the switching cost.

4.3 Throughput

To test the hypothesis that *"phase-based tuning will significantly increase throughput"*, phase-based tuning is compared to the stock Linux scheduler (for the same workloads run under the same conditions). Throughput was measured in terms of instructions committed over a time interval (0% representing no improvement). Variations of phase-based tuning and its algorithm's variables are compared. As mentioned previously in Section 4.1, workloads consist of a fixed number of processes running simultaneously. For all figures presented in this section, the data is taken from the first 400 seconds of the workload execution.

It is important to note that the measurements for throughput include the instructions inserted as part of the phase marks. This code is efficient and is likely to skew the measurements. Nevertheless, throughput is considered in order to measure the impact of several variables in phase-based tuning. For example, with the same technique, variations in the threshold used to determine core assignment will result in a minor impacts to the throughput by the extra instructions in phase marks. Thus, throughput still gives insight into how variables in the technique impact performance.

A better picture of how this technique improves performance is given by the average process time. Average process time also incorporates some level of fairness, these results are given in the next section.

4.3.1 IPC threshold

First, consider how the IPC threshold affects throughput. As mentioned in Section 3, IPC threshold is used to determine the section-to-core assignment. Figures 4.4 and 4.5 show how different threshold values affect throughput when all other variables are fixed (technique, min. size, lookahead, etc).

These results are as expected. Extreme thresholds may show a degradation in throughput because the entire workload eventually migrates away from one core type. Between these

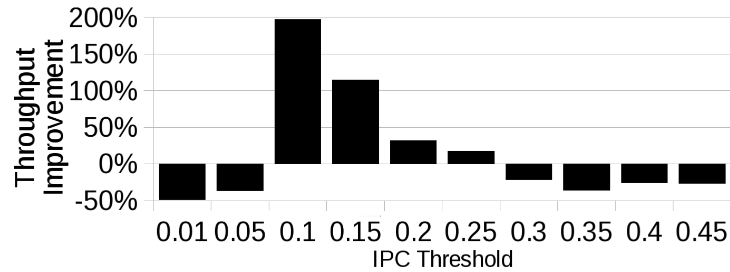

Figure 4.4 Throughput improvement: Basic block strategy, min. block size: 15, lookahead depth: 0, variable IPC threshold

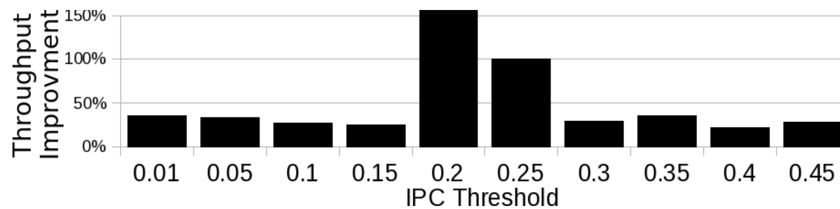

Figure 4.5 Throughput improvement: Interval strategy, minimum interval size: 45, variable IPC threshold

extremes lies an near optimal value. Near optimal thresholds result in a balanced assignment that assigns only well-suited code to the more efficient cores.

4.3.2 Clustering error

Statically predicting similarity will have some inaccuracy. Thus, Figure 4.6 shows how phase-based tuning performs with approximate similarity information. The same variables as Figure 4.4 are tested but with error levels ranging from 0% to 30%. For these tests, since there are two core types, a perfect assignment (0% error) consists of two clusters, one for each core type. To introduce this error, after determining the clustering of blocks, a percentage of blocks were randomly selected and placed into the opposite cluster. The result is that blocks expected to perform better on a “fast” core are run a “slow” core and vice versa.

These results show that phase-based tuning is still effective with approximate block clustering. With a 10% error there is almost no loss in performance and with 20% error there is still a significant performance increase. At 30% error there is little performance improvement.

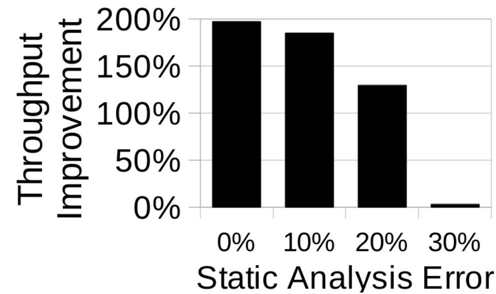

Figure 4.6 Throughput improvement: Basic block strategy, min. block size: 15, lookahead depth: 0, variable error

4.3.3 Minimum instruction size

Now, it is shown how minimum instruction size affects throughput for the three techniques. Figure 4.7 shows this comparison. The results are expected and similar to those for lookahead. Considering smaller blocks, intervals, and loops generally results in higher throughput. This is for the same reasons as lookahead depths, however, with larger minimum instruction size small loops may be ignored that are executed frequently. As mentioned previously, this improvement must be balanced with overhead costs which were discussed in Section 4.2. Also, recall that throughput includes instructions inserted in phase-marks, thus the throughput results are skewed.

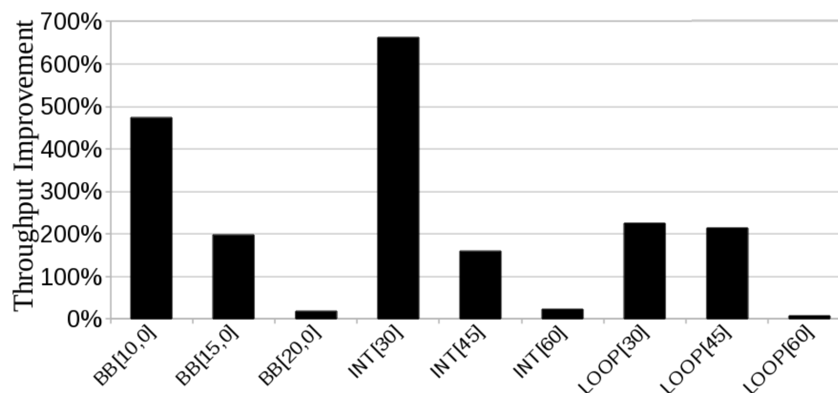

Figure 4.7 Throughput improvement: variable technique and minimum size

4.4 Fairness

Improved throughput is advantageous, however, in many systems fairness is also desired. Therefore, the fairness of phase-based tuning is analyzed. First, the fairness metrics are described followed by the measurements and a brief discussion.

Three metrics are used to analyze fairness:

- max-flow,
- max-stretch, and
- average process time.

Max-flow and max-stretch were developed by Bender *et al.* for determining fairness for continuous job streams [10]. Now, max-flow and max-stretch are briefly defined.

For each process, the following data is gathered:

- a_i: arrival time of process i,
- C_i: completion time of process i, and
- t_i: processing time of process i (in isolation).

First, *max-flow* is defined as

$$\max_j F_j,$$

$$\text{where } F_j = C_j - a_j$$

This is basically the longest measured execution time. So, if even one process is starving, this number will increase significantly. Second, *max-stretch* is defined as:

$$\max_j \frac{F_j}{t_j}$$

This can be thought of as the largest slowdown of a job. This is considered because it is desired that processes speed up, but not at the expense of others slowing down significantly. These measurements for variations of phase-based tuning are shown in Table 4.4.

Technique	% decrease over standard Linux		
	Max-Flow	Max-Stretch	Avg. Time
BB[10,0]	-10.75	-17.87	14.57
BB[10,1]	-28.89	-26.44	0.74
BB[10,2]	-51.21	-16.73	-9.34
BB[10,3]	-43.19	-1.63	-8.78
BB[15,0]	17.01	0.65	23.65
BB[15,1]	18.33	13.29	25.73
BB[15,2]	-27.81	-12.19	-4.08
BB[15,3]	-36.51	-24.13	7.11
BB[20,0]	-39.55	-84.33	-10.35
BB[20,1]	-17.27	-34.65	28.42
BB[20,2]	-41.54	-56.90	22.88
BB[20,3]	-56.41	-48.46	9.00
Int[30]	3.86	-11.50	9.69
Int[45]	39.15	32.78	28.60
Int[60]	-27.36	13.80	27.38
Loop[30]	3.24	6.54	14.86
Loop[45]	**12.04**	**20.41**	**35.95**
Loop[60]	-16.10	17.57	10.40

Table 4.4 Fairness Comparison to standard Linux assignment: Improvements are shaded.

The best technique shows the following benefits over the stock Linux scheduler.

- 12.04% decrease in max-flow,
- 20.41% decrease in max-stretch, and
- 30.95% average decrease in process completion time.

These results were gathered over an 800 second time interval using the loop based technique with minimum size of 45 and IPC threshold of 0.15.

4.5 Analysis of Trade-offs

The previous results have shown that phase-based tuning has clear advantages over the stock Linux scheduler while maintaining fairness. However, the goal of a scheduler varies based on how the system is used. Some systems desire high levels of fairness while others are only concerned with throughput. It may also be the case that a balance is desired instead. Therefore, it is important to analyze the trade-off between fairness, average speedup, and throughput. In this section these trade-offs and how different variations of phase-based tuning perform with specific scheduling needs are discussed.

Here, the trade-off between speed and fairness is examined. Speedup refers to the decrease in average process run-time. Max-stretch is used for fairness. Figure 4.8 shows this trade-off for different variations of phase-based tuning.

These results show that a balance between the two exists. The interval and loop techniques perform quite well at balancing these two metrics. Many variations show significant increases in speedup, but at a loss of fairness.

4.6 Summary of Results

In closing, these results show that phase-based tuning significantly outperforms the stock Linux scheduler in terms of the throughput obtained on an AMP, while maintaining fairness

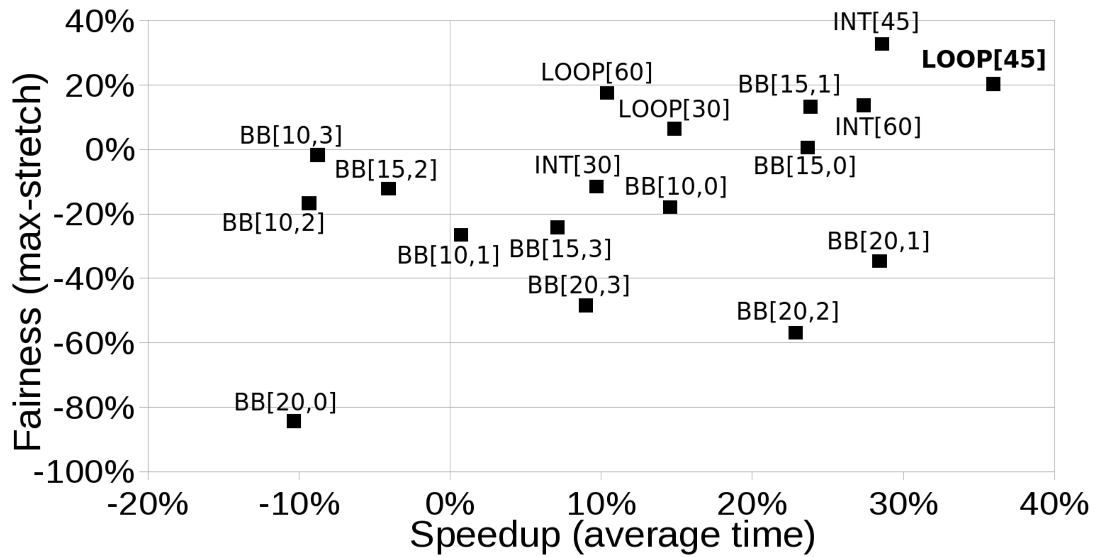

Figure 4.8 Speedup vs fairness: average time vs. max stretch

and with a negligible overhead in most cases. In particular, the loop based technique balances throughput and fairness significantly well achieving an average process speedup of up to 36%. Phase-based tuning thus shows its potential in improving the utilization of AMPs.

CHAPTER 5. Practical and Future Use of Phase-based Tuning

This chapter discusses some issues that although not central to phase-based tuning, are relevant for its practical or future applications.

5.1 Applicability to Multi-threaded Programs

The simplicity of phase-based tuning allows it to immediately work on multi-threaded applications. Recall that binaries are modified by inserting code. When an application spawns multiple threads, it is essentially running one or more copies of the same code which was present in the original application. The framework will have analyzed this code and modified it as needed. Thus, each thread will contain the necessary code switching and monitoring code present in the phase marks.

Furthermore, the cache performance impact of code sharing across threads will be handled as well. For example, suppose two identical threads are running. One of the two will pick an assignment based on its observed behavior. Since the code is identical, the other thread will then have the same assignment decision. Thus, its preferred core type is the same as those with shared code.

If the threads share some data rather than code, it is expected that a similar situation will happen. For example, the first thread will pick its desired core type. Next, the other thread, while picking its core type, will likely see improved performance when executing on a core that shares cache with the first thread due to increased cache hits. Thus, it is more likely to chose core types which result in better cache behavior.

5.2 Changing Application and Core Behavior

Recall that the workload on a system may change the perceived characteristics of the individual cores. Furthermore, program behavior may change itself periodically (e.g. warm-up phase). While not addressed explicitly in this thesis, solutions for these problems only require minor modifications to the techniques presented here.

Dealing with changing program behavior is trivial. For example, a warm-up phase is usually caused by one of two things. First, different code may account for the warm-up phase such as an I/O section of code before the computational portion. This is the easier case and is already handled by phase-based tuning since these two phases are in separate code sections. The second cause is best illustrated by considering compulsory (or cold) cache misses. For example, the first time some code is executed, the data it operates on is not in cache. However, upon each successive execution, the data is likely in the cache. This case is easily handled with phase-based tuning by simply ignoring the first execution of each block.

Handling changes in the system behavior due to workload changes is slightly more complex. For short running programs, this is not likely to be an issue since the program may finish (or mostly finish) before behavior changes. Furthermore, assignment is re-computed for each program run so a poor assignment will be short lived. However, for long running programs such as a web-server, a poor assignment would last far too long. To address this issue, a simple feedback mechanism is needed to re-assess core mappings periodically. For example, after some amount of time, the dynamic analysis code would begin monitoring blocks again to assess and possibly modify the current assignment strategy. More complex schemes could be implemented as well which look at processes entering and leaving the system as well as phase transitions in other processes.

In summary, by not actually looking at the hardware characteristics, phase-based tuning handles the changes in each cores behavior that occurs based on other processes in the system.

5.3 Floating Point Emulation and IPC

Using IPC as a metric for performance poses a few problems. For example, a core without a floating point unit may do floating point emulation which will result in many potentially fast instructions being executed in place of one slow floating point instruction. As a result, the IPC of a floating point intense section of code running on a core with floating point emulation may give a high IPC. However, it is clear that a core with a floating point unit is more desirable. To address this issue, calculation of IPC can be done as follows. Cycles for a section can still be determined using performance counters. Then to determine instruction count, a combination of static analysis and code instrumentation can be used to embed the number of instructions (determined by counting actual floating point instructions not emulated instructions) into the code for calculating IPC. In this case, care must be taken to avoid excessive overhead of the instrumentation code.

5.4 Scalability for many-cores

While the current implementation performs well for multi-core systems, there is a potential scalability issue for many-core machines since each core in the system would need to be tested for each cluster. To address this issue the following is proposed. First, it is expected that for asymmetric many-core systems, the number of core types is still likely to be small [1]. Thus, if a grouping of the cores into types is known the problem may largely be reduced to one similar to a multi-core system thus avoiding this scalability issue. There are several options for producing such a grouping. One option is to manually determine it based on the processor specification. Another option is to automate the process using carefully written test programs which are run before jobs are sent to the system. These programs will monitor their performance on several cores and determine a grouping of the cores. Care must be taken to ensure that the test programs expose all potential differences which may impact performance.

[1] Previous work seems to suggest situations where as few as two cores types is sufficient [44, 33].

CHAPTER 6. Lazy Grouping

An overview of lazy grouping is now given. Lazy grouping consists of three major steps including both compile and install time analysis. After the overview, each step will be described in detail.

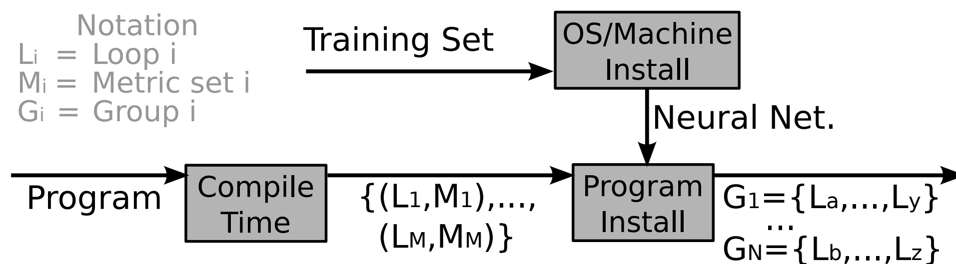

Figure 6.1 Lazy Grouping. At compile time, static analysis computes behavior metrics for each loop. At OS install time, a neural network is trained for the current machine. At program install time, the metrics (computed at compile time) are sent to the neural network which computes the groups.

Compile Time . At compile time, static analysis is used to approximate metrics of program behavior (e.g. cache behavior, instruction type, etc.). As many metrics as are desired may be analyzed since this analysis only happens once per program and the *results are used for all target machines*. The bottom left of Figure 6.1 shows that the compile time portion takes as input a program and outputs a set of pairs each consisting of a loop and its corresponding metric sets.

OS/Machine Install Time . At OS or machine install time, a neural network is trained for the current machine. The training is done, only once per machine, using a given training

set. The top of Figure 6.1 shows that the OS/machine install time component takes as input the training set and output a neural network. Like the output of the compile time step, the training set consists of pairs of loops and their metrics. To determine the desired network outputs, each loop in the training set is run on the current machine and its behavior observed.

Program Install Time . At program install time, the metrics for the program (computed at compile time) are submitted to the neural network (trained at OS install time) to compute the grouping. This grouping is called "lazy" since it waits till the target machine is known instead of grouping immediately after computing the metrics. The bottom right of Figure 6.1 shows that loops and their metrics (from compile time analysis) along with the neural network (generated at OS install time) are input. The output of this step are the groups for the loops of the input program for this machine.

Each of these components is now described in detail.

6.1 Compile Time – Computing Similarity Metrics

To determine behavior similarity between program segments static analysis is used to approximate several behavior metrics. Since this happens only once per program regardless of the target machines, the analysis must analyze enough metrics such that they may approximate the behavior of each segment on any target machine (that uses the same instruction set). Fortunately, the analysis can afford to spend this effort analyzing the program since it must only be done once and it is static. This means that anyone installing the program need not spend any resources analyzing the program and its behavior.

For example, on the left of Figure 6.2 a simple input program is shown containing two loops. The static analysis labels these loops and computes a set of metrics for each. Here, the focus is on analyzing loops which were shown to be the most effective granularity for phase-based tuning. In the case of loop nesting, each nesting level is analyzed separately in order to provide the necessary type information to the phase-based tuning optimization algorithms.

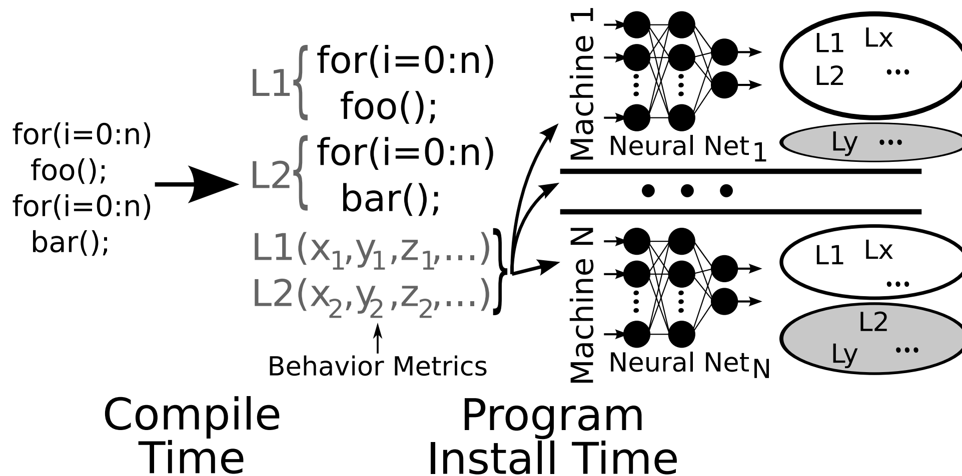

Figure 6.2 Compile and program install time: At compile time, loops and behavior metrics are determined. This information, along with the program, is sent to each machine. At program install time, a neural network computes grouping.

An overview of the behavior metrics statically computed for the evaluation of lazy grouping is now given. These metrics are a reasonable subset of those necessary to approximate behavior in order to demonstrate the utility of the approach. In practice, more metrics could be used based on the users needs. Note that adding more metrics does not increase the number of required groups.

6.1.1 Instruction Latency

The first metric estimates instruction execution times. Since lazy grouping should work for a wide range of target cores all with potentially different implementations of operations (some with in-order and some with out-of-order execution and other issues like memory accesses) it is not possible to compute how long an instruction will take. However, an analysis may estimate that some instructions will take longer than others (e.g. division takes longer than addition). Thus, the goal becomes estimating some measure of execution time which correlates with behavior on at least most target cores.

Several options for this measure were explored including instruction type groups (e.g. arithmetic, data transfer, shift, floating point, etc.) and estimations derived from monitoring

behavior of programs on modern architectures [32]. The most effective technique found was using the cycle counts on 486 CPUs from an instruction reference [76]. These cycle counts also take into account the operands involved. For example, an instruction accessing memory will likely take longer than the same instruction accessing only registers.

6.1.2 Instruction Cache

Cache behavior is an important aspect of similarity detection. As the gap between CPU speeds and memory speeds widen, the result is that accessing main memory can take thousands of cycles on some machines [34]. Clearly this presents a large range of potential costs for executing a memory operation.

Therefore, the next metrics are several rough static estimates of instruction cache behavior. This includes an analysis of both the best case and worst case behavior for simple caches with several degrees of associativity. The predicted “hit rate” (percentage of accesses that will be in cache when requested) for each analysis is used.

The theory behind this analysis is based on previous work by Ferdinand and Wilhelm [27]. The basic idea is to use abstract interpretation [19] (a sound static analysis framework) to model the cache replacement policy (e.g. least recently used (LRU)) in cache sets. The best and worst case proximity to eviction is tracked for each block potentially in cache. The analysis implementation is based on previous work [78] for more precisely analyzing cache hierarchies.

6.1.3 Data Cache

Since instruction cache behavior is only part of the cache behavior of a program, approximate data cache behavior is also analyzed. The idea is to statically analyze reuse distances of data accesses [11].

Reuse distance refers to the number of unique accesses between the current accesses and the previous access to the same location. However, since this is a static analysis, instead of looking at execution profiles, it analyzes instruction operands to compute an approximate best

case reuse distance. This reuse distance is then used to place each access into a bucket of similar data accesses (e.g. all accesses with a distance between 2^2 and 2^3 are put in the same bucket). This is similar to part of the MICA technique for clustering whole benchmarks [36]. Six buckets are used which collect accesses within a specific range – $[0, 2^0), [2^0, 2^1), \ldots, [2^5, \infty)$. The average (per instruction) number of accesses in each bucket is used.

6.1.4 Instruction Level Parallelism (ILP)

Instruction level parallelism has been used for many years now to improve program performance (especially with the smaller increases in CPU clock speeds) [34]. Since ILP has become rather heavily relied on for improving performance by hardware manufacturers [34], it impact on program behavior is likely to vary, especially when comparing simple and complex architectures (e.g. consider the out-of-order Intel i7 and the in-order Intel Atom).

Therefore, rough static approximations of ILP are also included. Similar to the cache reuse distance analysis, the operands for each instruction are analyzed Again, only static approximations are used.

For each register and memory location accessed by each instruction the analysis performs a backwards search for dependencies. This includes register dependencies (e.g. if an instruction reads or writes a register value written by a previous instruction) and memory dependencies (e.g. the same as register dependency, but with memory locations).

A worst case analysis is used, however, it does not look at some forms of dependence such as blocking of functional units. For example, suppose there are two potential paths leading up to an instruction that reads some register. In one path, there are two instructions until the closest dependence. In the other, there are four instructions until the closest dependence. In such a case, the analysis considers the worst-case distance to be two.

Like the data cache analysis, each instruction is placed into one of five buckets depending on the distance (in instructions) to the worst-case previous dependency.

6.1.5 Loop Size

The final two metrics are simple and look at loop size in terms of total instruction count and basic block count. These metrics on their own do not necessarily predict behavior well. However, they provide some assistance to predicting behavior when combined with other metrics (as determined by feature selection).

6.2 OS/Machine Install Time – Network Training

At OS or machine install time, a neural network is trained for the machine. This neural network is then used for grouping the loops for all programs to be optimized for this machine.

A neural network (or artificial neural network) is a model that mimics certain aspects of a biological neural network (i.e. a brain) [70]. This network has layers. The first and last layers of the network are called the *input and output layers* respectively. The layers in between are called *hidden layers*. Data is fed into nodes in the input layer. Data is passed through the network where each node takes inputs from the previous layer, multiplies each by some weight, aggregates the result, and passes the result through an *activation function*. To give desired outputs, the network is *trained* using a *training set* consisting of inputs and desired outputs. This training set is repeatedly sent through the network. Each time, weights are adjusted to bring the actual outputs closer to the desired outputs. This continues until a desired error rate is reached.

For lazy grouping, the size of the input layer is equal to the number of similarity metrics that were analyzed. Experimentation has found that networks with three hidden layers, each roughly the same size as the input layer, appear to work best. Various activation functions have been tested. While periodic functions (e.g. sin, cos) help achieve the desired error rate more quickly, the output network is much less accurate than networks using the sigmoid logistic function or similar approximations. The size of the output layer is equal to the number of desired groups (e.g. approximately the number of core types in an AMP). For example, in

Figure 6.3 the network has two output nodes. This means that loops will be divided into two groups. Each node in the output layer computes a probability that the input belongs to the corresponding group. For example, if there are two output nodes, n_1 and n_2 the input loop belongs to group c_1 if $out(n_1) > out(n_2)$ where out gives the output value from a node.

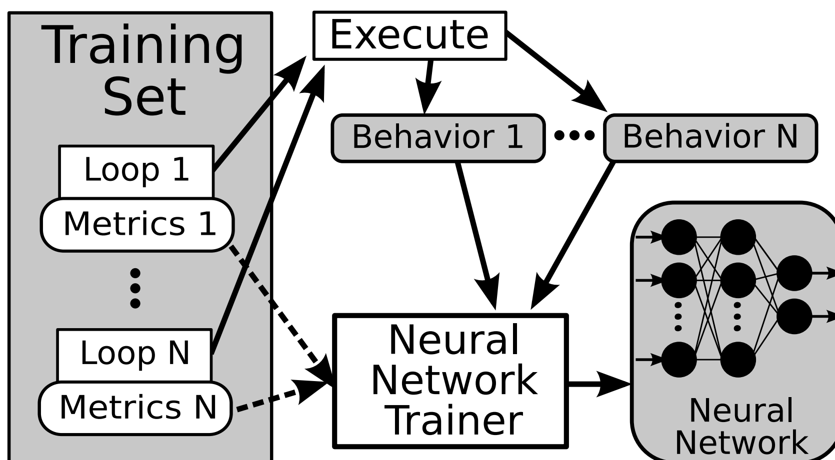

Figure 6.3 OS install time. The training set is supplied to the machine. Loops are executed and their behavior is gathered. Desired outputs, for training, are computed from this observed behavior.

The training set consists of a set of benchmarks and precomputed metrics for the loops in each benchmark (shown on the left of Figure 6.3). Each loop is a single training example in the training set. These metrics for each loop make up the inputs of each member of the training set. Recall that the outputs of the network are probabilities for which group the loop should belong to. Thus, the analysis needs a way to determine, for each loop in the training set, which group is ideal.

To determine the desired groups, execution behavior is monitored for each loop (top right of Figure 6.3). To do so, each loop is instrumented to gather performance data. After execution, the behavior of all executions (e.g. nested loops which executed many times) are averaged. Suppose the performance data gathered was IPC and that two groups are desired. Then, a single threshold IPC is used (if n groups are desired, $n - 1$ thresholds are used). Any loop whose IPC is above this threshold goes in one group. The others, go in the other group. For

application to AMPs, the difference in IPC between core types is used to determine grouping.

To avoid over-fitting, values that lie within some small distance, ϵ, from the threshold(s) are set with equal probability for the groups on either side of the threshold. For example, suppose there is a threshold of 0.5 and that two loops have IPCs of 0.49 and 0.51. It probably makes sense for these loops to be considered similar (and they likely have similar metric values). Informing the neural network otherwise may hurt accuracy. In experiments, ϵ ranges from 0.01 when cores only differ in frequency to 0.03 for cores with significant differences (or for single core type grouping). These values avoid over-fitting loops which lie directly on either side of the threshold while at the same time avoid capturing too many loops as being fit for either group.

Once the desired output for each loop is determined, the network is trained (shown on the bottom right of Figure 6.3). This entire step is lengthy due to gathering the observed behavior of each loop. Fortunately, it only needs to be done one time, when the machine is being set up. Throughout the entire lifetime of the machine, the same network may be used. Additionally, it would be reasonable that users could obtain a network that has been pre-trained based on their machine configuration.

One can influence the group distribution through the training set. This means the distribution of groups can be tuned based on the target optimization. For example, suppose the target optimization is phase-based tuning for an AMP with two core types (fast and power efficient). If the desired behavior is efficient power use, the analysis only puts segments in the "fast" core's group when the benefits of doing so are large. With a k-means clustering based technique, one must rely on what distribution is produced causing potentially undesirable use of the system.

6.3 Program Install Time – Grouping

The final step, grouping, happens at program install time (when the program is placed on the target machine). This step only happens once per benchmark per target machine.

This process is illustrated on the right of Figure 6.2. The neural network used to compute the grouping was already created at OS/machine install time. Thus, the analysis simply sends the metrics computed at compile time to the neural network. A benefit is that groups are computed very quickly, though this is not necessary since it occurs statically.

Unlike a compile time grouping, where the behavior of the loops in each group is unknown, lazy grouping knows approximately the expected behavior of the loops in each group. That is, that they likely behave above or below some threshold(s). This gives a tremendous benefit. Thus lazy grouping no longer requires, or can at least simplify, runtime analysis.

6.4 Summary of Benefits

Lazy grouping has several benefits.

- It can tackle a wide range of target machines since, by delaying grouping until install time, the output is a grouping tuned for the target machine.

- Phase-based tuning is more efficient for three reasons. First, users can specify the number of groups to use without losing accuracy. For example, for an AMP with two core types, two groups can be used. Second, users can influence the relative size of groups. For example, if an AMP has a skewed set of core types, more extreme thresholds can be chosen when training the network resulting in skewed group sizes. Third, no runtime monitoring and analysis is required to make decisions dynamically since lazy grouping knows if segments are likely to behave above or below some threshold.

- Typical use of the approach is efficient since most of the overhead occurs at compile time (computing metrics) and OS/machine install time (machine setup).

CHAPTER 7. Evaluation of Lazy Grouping

The hypothesis is that lazy grouping will produce accurate groupings (>80% accuracy) for the target machines (all that were accessible). Further, it is hypothesized that lazy grouping will produce more accurate groupings than compile time grouping and result in more effective application of phase-based tuning.

To evaluate these hypotheses, benchmarks are run on a wide variety of machines to observe their actual behavior. This actual behavior is then used to separate loops in these benchmarks into groups to determine perfect grouping information. Groupings are computed using lazy grouping and the results are compared with perfect groupings to determine accuracy along four dimensions: raw accuracy, group accuracy, statistical test p-value, and visual analysis.

7.1 Experimental setup

The experimental setup including how behavior is determined for target configurations, hardware and software setups, and techniques used to determine accuracy are now described.

7.1.1 System Setup

All systems used for this experimentation are physical systems running GNU/Linux. Figure 7.1 shows all core types used in this experimentation. Modern processors (e.g. Core i7, Opteron 6168) are included as well as power efficient processors (e.g. Atom, Pentium M), older processors (e.g. Pentium 4), and some in between. For each core type, frequency was varied (if possible) as well, creating a wide range of core types.

Series and Model	**Frequencies** (GHz)	**L1 (i/d)** (KB)	**L2** (KB)	**L3** (MB)	**Cores**
Core i7, 870	1.2,1.6,2.0,2.4[1],2.93	32/32	4x256	8	4/8
Atom, N270	0.8,1.6	32/24	512		1/2[2]
Core 2 Quad, Q6600	1.6,2.4	32/32	2x4096		4
Core 2 Duo, E6300	1.6,1.83	32/32	2048		2
Pentium 4, 2.0	2.0	12K[3]/8	512		1
Pentium M, 725	0.8,1.6	32/32	2048		1
Opteron, 2431	0.8,1.2,1.5,1.9,2.4	64/64	6x512	6	6
Opteron, 6168	0.8,1.3,1.9	64/64	6x512	12	12

Table 7.1 Core types used for evaluation. All L1 caches are private for a single core and split (instruction and data caches are separate). For cores with hyperthreading, the number physical cores and total threads are shown.

The benchmarks used in this evaluation are the benchmarks from the SPEC CPU 2000 benchmark suite[4]. This suite is used since it runs in a reasonable amount of time under fine grained monitoring on all target machines.

The Fast Artificial Neural Network Library (fann) [65] is used for constructing and training the neural networks. In these experiments, a grouping for each individual benchmark is computed. To do so, a technique called leave-one-out cross-validation [70, pp.663] is used. That is, when computing a grouping for a single benchmark, the network is trained using the rest of the benchmarks (i.e. the current benchmark and all its loops are excluded from the training set). Further, also excluded are all runs of the same benchmark under different input where such alternate input sets exist.

7.1.2 Computing Actual Behavior for Evaluation

Behavior is gathered using PAPI [22]. The custom program analysis and instrumentation framework described in Chapter 3 detects loops and inserts the appropriate PAPI calls. When

[1] Actually 2.39GHz, but rounded for clarity.
[2] Uses in-order execution.
[3] Uses a 12K μop trace cache which behaves similarly to an 8-16KB i-cache [71].
[4] excluding perlbmk, gcc, eon, fma3d, and sixtrack which either do not execute or analyze properly

sufficient information is unavailable to safely instrument or detect a loop, it is ignored. For loops executed multiple times, the average of the executions is taken.

Both individual core behavior and difference in core behavior in asymmetric configurations (as used by phase-based tuning) are considered. For asymmetric configurations, difference in behavior is approximated using two core types, sometimes in different systems. While some of these AMPs do not exist and other characteristics of each machine may differ, the difference in behavior is representative of some potential AMP. Asymmetric configurations with two core types are used because previous work suggests that two types are sufficient to realize the benefits of AMPs [44, 33].

7.1.3 Accuracy Metrics

To determine the accuracy of a grouping, several techniques are used. Each technique is now described.

Raw Accuracy . Raw accuracy is the percent of loops which "fit" in their group. For example, suppose there are two groups both containing five loops (10 loops total). Suppose all loops are correctly grouped except one (ideal grouping is determined in the same way training sets are created for the neural network). This group has 90% raw accuracy. A random grouping will give, on average, 50% raw accuracy.

Group Accuracy . Suppose there are two groups, one with eight loops, the other with two. The group with eight loops is completely accurate (100%) whereas the other is entirely incorrect (0%). In this case, the group accuracy is 50% (i.e. each group gets equal weight regardless of its size). This case would be reported as 80% accurate with the raw accuracy metric. This case could occur if one were to choose an extreme threshold and cause the size of one group to become very small. Simply placing all loops in the larger group will give a high raw accuracy. Group accuracy solves this problem by giving a 50% group accuracy. Similarly, a random grouping will on average give a 50% group accuracy.

p-value . The p-value for the statistical test where the null hypothesis is that the two groups have the same mean is also considered. A sufficiently low p-value (e.g., < 0.05) will reject this hypothesis. That is to say that the two groups have a statistically significant difference in their means.

Boxplots . Finally, a visual analysis using boxplots which help illustrate the distribution of data is used. Consider the left of Figure 7.2. The boxes represent the inner quartile range. The lines on either end extend to the minimum and maximum of the data set excluding outliers (shown as circles). The line in the middle of the box represents the median. For lazy grouping, ideally, there would be no overlap between groups. However, a good grouping should at least not have the boxes (i.e. inner quartiles) overlapping.

7.2 Compile Time vs. Lazy Grouping

The evaluation in this section demonstrates that lazy grouping is more accurate and requires significantly fewer groups than compile time grouping. This section also shows that lazy grouping outperforms a compile time grouping tuned for the target machine

For comparing the accuracy of the techniques, each core type is considered as well as asymmetric configurations containing the two most different core types (i7 and Atom). For these asymmetric configurations, the most different frequencies are used (e.g. i7 at 2.93GHz and Atom at 0.80GHz).

7.2.1 Universal Compile Time Grouping

For this compile time grouping, the same metrics as lazy grouping are used except two (the second highest data cache bucket and the second highest ILP bucket). These two metrics were considered irrelevant by feature selection for all configurations. Additionally, all metrics were scaled to be in the range $[0 - 1]$ in order to give each metric equal weight. Finally, k-means clustering is applied to the resulting data set.

Figure 7.1 shows the raw accuracy of the two techniques for several configurations. Bars are labeled with the minimum number of groups needed to achieve this accuracy. Of course, a grouping applicable to all machines would have fixed groups. For these experiments, accuracy is the same as the presented data when increasing groups up to 200.

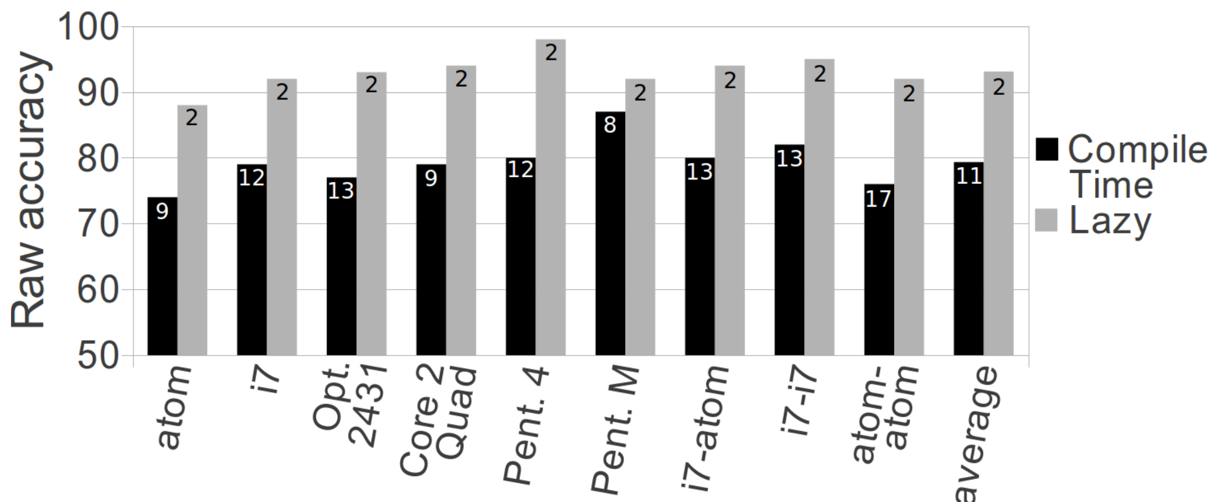

Figure 7.1 Compile time vs lazy grouping. Numbers on bars are the minimum number of groups required to achieve this accuracy. 50% accuracy denotes random grouping. The figure also shows that lazy grouping performs significantly better.

The figure shows that accuracy for lazy grouping is significantly better than compile time grouping. On average, lazy grouping improves raw accuracy by 17%. Therefore, dynamic optimizations receive more accurate behavior knowledge and thus can make more effective decisions. Lazy grouping, on average, needs one sixth of the groups which helps reduce run-time analysis overhead (if required).

7.2.2 Target Machine Specific Compile Time Grouping

One could use a compile time approach and modify the weights on each similarity metric for each target machine. Unfortunately, this either involves significant expertise or exploration of a large parameter space. A major problem is that the *relationships between metrics* may not be straightforward. Consider ILP metrics whose impact depends on several other factors

like cache behavior, branching, etc. Even using automated techniques, searching this huge parameter space for an optimal configuration is not feasible.

Consider this approach for the two most different cores (i7 and Atom). Using feature selection, it was determined that a combination of instruction latency and cache behavior works well. In this case, estimated cache behavior combines instruction and data cache buckets with different weights.

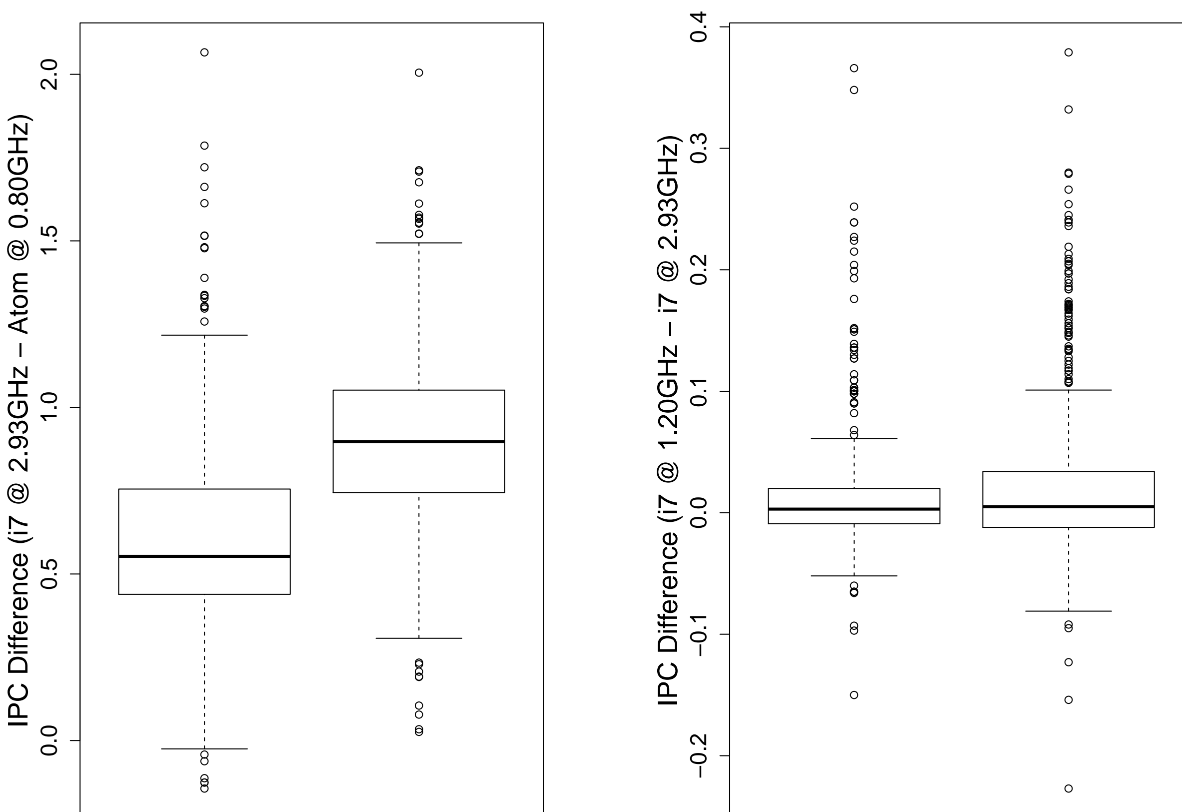

Figure 7.2 Grouping using k-means at compile time. Left: Intel i7 @ 2.93GHz with Intel Atom @ 0.80GHz. Right: Intel i7 @ 2.93GHz with Intel i7 @ 1.20GHz.

The result is a grouping technique which gives 81% raw accuracy with only two groups (group distribution is shown on the left of Figure 7.2) which is better than the previous compile time grouping not tuned or this machine which achieved 80% raw accuracy but required 13 groups. However, lazy grouping still performs better at 94% accuracy for this configuration. Lazy grouping also has the benefit of being purely automatic. Finally, note that using this

technique tuned for this specific machine gives random, or nearly random, groupings for other machines. For example, the right side of Figure 7.2 shows that there is no difference in the groups for an AMP consisting of two i7 cores (p-value is 0.88).

7.3 Accuracy of Lazy Grouping

In this section, the accuracy of lazy grouping is demonstrated for a wide variety of core types and asymmetric configurations.

Asymmetric configurations were chosen for a variety of reasons. Some were chosen because they combine recent high power cores with power saving cores (e.g. i7 with Atom or Pentium M). Others were chosen because they have the similar clock frequency but differ in other (known or unknown) ways (e.g. Core 2 Quad and Opteron). Finally, some configurations were chosen which differ only in frequency.

Figure 7.2 gives the average raw and group accuracy (defined in Section 7.1.3) of lazy grouping for each configuration. For all, there is a p-value of < 0.01. Thus p-value is left out of the table. Unless otherwise noted, measurements are for groupings with two groups. The focus is on two groups since this is a likely use case (e.g. one group for each core type). However, a few experiments are also included which use three groups in order to demonstrate the flexibility of lazy grouping. The table shows that for nearly all configurations, lazy grouping gives greater than 90% raw and group accuracy.

For selected configurations boxplots are presented that show the distribution of the behavior of groups across all benchmarks.

In Figure 7.3, plots are shown for the two most different cores (i7 and Atom) for both two and three groups. Next, asymmetric configurations were chosen which either provide interesting asymmetry and/or test various types of asymmetry.

In Figure 7.4, a plot for the configuration containing the two most varied core types (the i7 and Atom at their most extreme frequencies) is shown. The figure also contains boxplots

Core Type 1		Core Type 2			Group
Model	Freq.	Model	Freq.	Accur.	Accur.
*i7	2.93			92.2%	94.0%
*i7	2.93	three groups		85.0%	89.3%
*Atom	1.60			88.4%	87.3%
*Atom	1.60	three groups		89.8%	88.1%
Core 2 Quad	2.40			94.0%	93.9%
Pentium 4	2.00			98.3%	96.6%
Pentium M	1.60			92.2%	92.2%
Opteron 2431	2.40			92.8%	92.5%
*i7	2.93	i7	1.20	95.0%	91.0%
*i7	2.93	Atom	0.80	94.0%	93.6%
i7	2.40	Core 2 Quad	2.40	94.5%	92.3%
i7	1.60	Core 2 Quad	1.60	94.5%	88.8%
i7	2.93	Core 2 Duo	1.60	98.7%	90.0%
i7	1.60	Core 2 Duo	1.60	95.3%	92.5%
i7	2.93	Pentium 4	2.00	92.7%	92.4%
*i7	2.93	Pentium M	0.80	91.6%	91.4%
i7	2.40	Opteron 2431	2.40	92.6%	89.0%
i7	2.93	Opteron 2431	0.80	91.0%	88.9%
*Atom	1.60	Atom	0.80	92.3%	89.2%
Atom	1.60	Core 2 Quad	1.60	93.1%	93.4%
Atom	1.60	Core 2 Duo	1.60	92.8%	90.2%
Atom	0.80	Core 2 Duo	1.60	93.2%	92.0%
Atom	0.80	Opteron 2431	2.40	92.5%	88.0%
Atom	0.80	Opteron 2431	0.80	96.3%	90.1%
Core 2 Quad	1.60	Core 2 Duo	1.60	94.3%	94.3%
Core 2 Quad	2.40	Pentium 4	2.00	94.4%	92.6%
Core 2 Quad	2.40	Pentium M	0.80	94.3%	93.3%
Core 2 Quad	2.40	Opteron 2431	2.40	96.0%	94.8%
Core 2 Quad	2.40	Opteron 2431	0.80	95.8%	93.8%
Core 2 Duo	1.83	Pentium 4	2.00	95.2%	93.4%
Core 2 Duo	1.83	Pentium M	0.80	94.5%	92.5%
Pentium 4	2.00	Pentium M	0.80	95.2%	95.5%
Pentium 4	2.00	Opteron 2431	2.40	94.0%	91.7%
Pentium 4	2.00	Opteron 2431	0.80	92.8%	90.9%
Pentium M	0.80	Opteron 2431	2.40	90.2%	91.3%
Pentium M	0.80	Opteron 2431	0.80	94.2%	91.1%
Opteron 2431	1.90	Opteron 6168	1.90	97.8%	95.8%
Opteron 2431	0.80	Opteron 6168	0.80	97.6%	94.9%

Table 7.2 Accuracy of lazy grouping per configuration. Those with one core type represent grouping of single core behavior. Configurations marked with * have boxplots shown later.

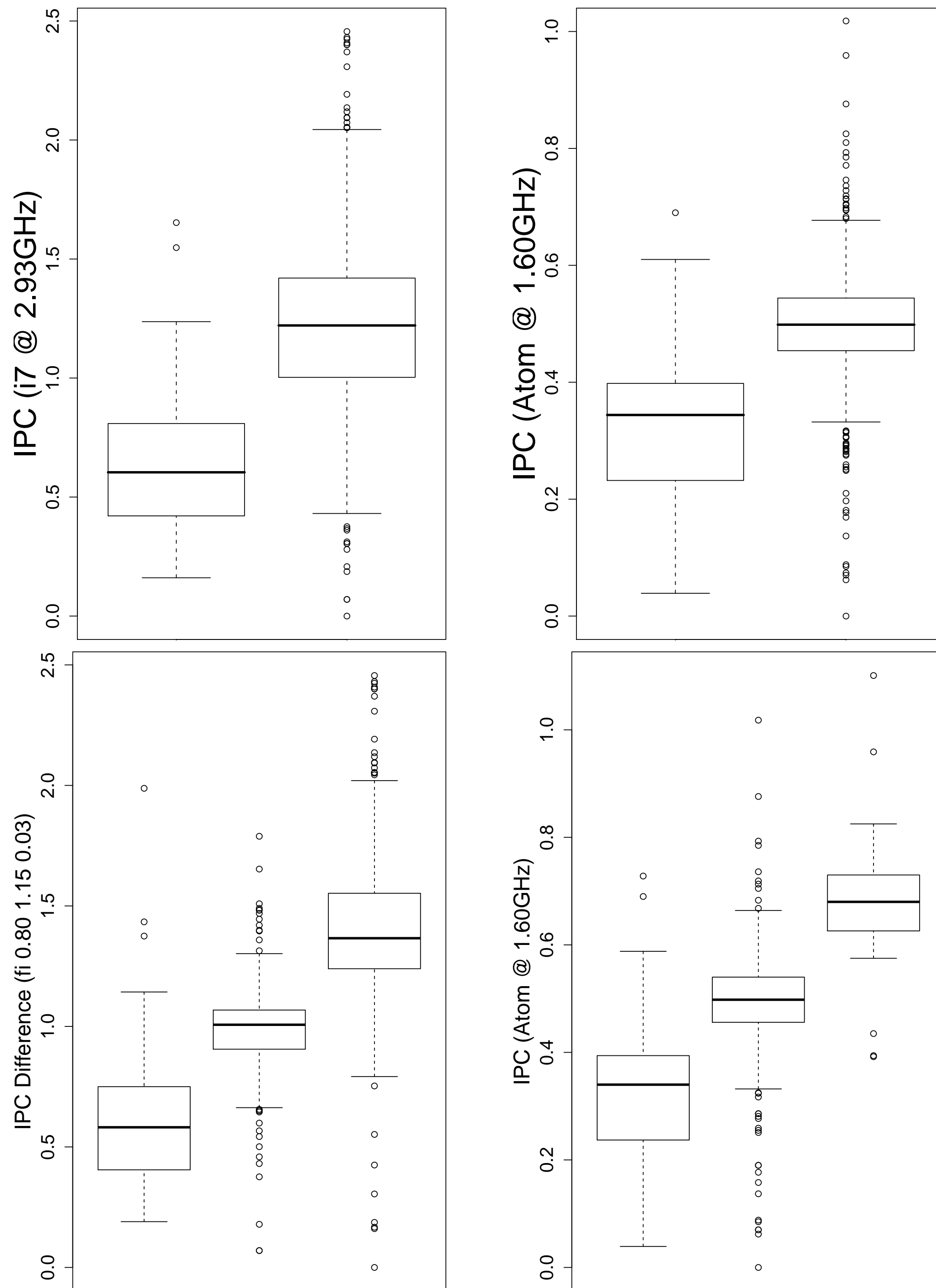

Figure 7.3 Group distribution across all benchmarks for single core types. For all cores, we see a distinct difference in group behavior.

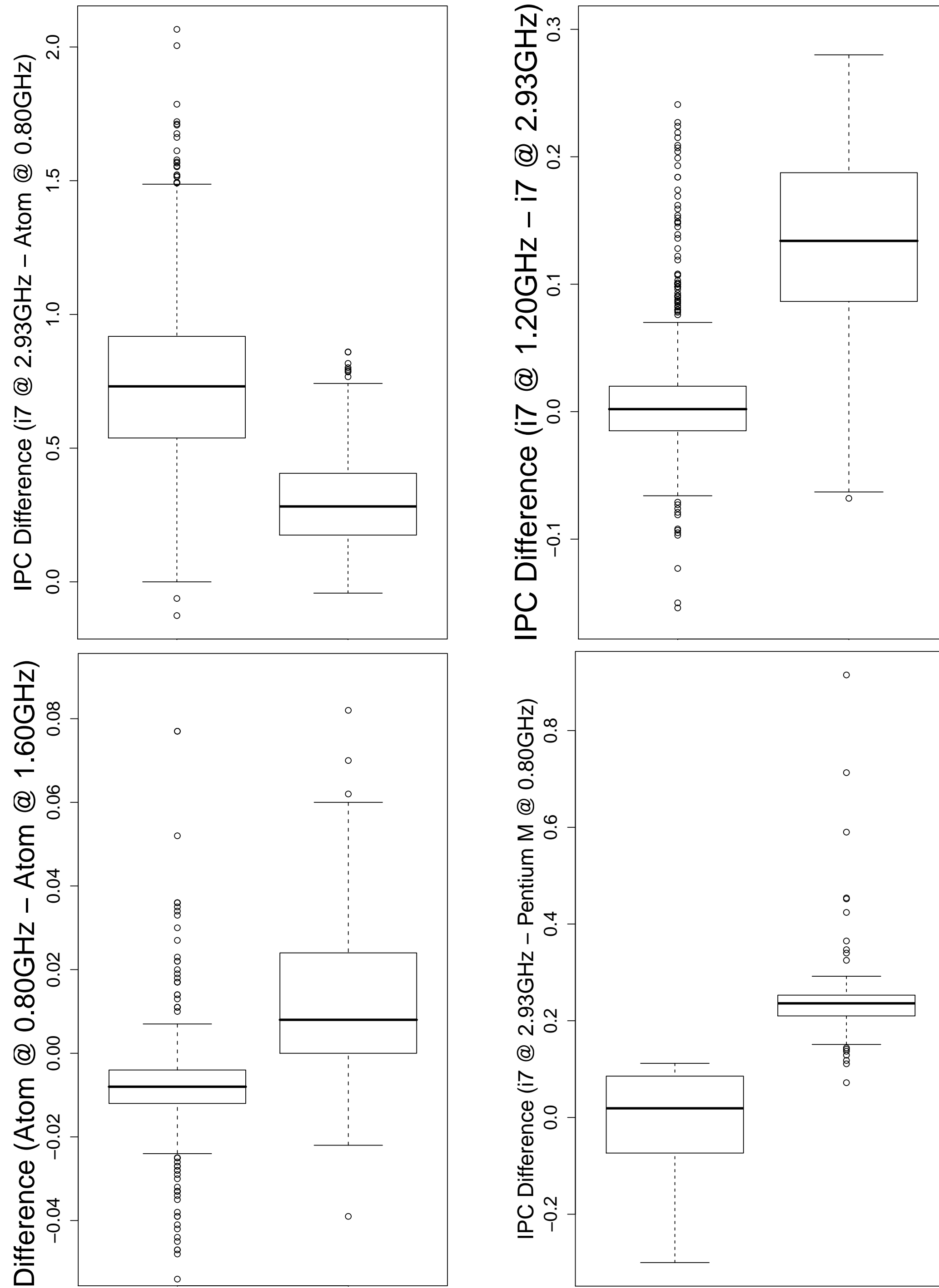

Figure 7.4 Group distribution across all benchmarks for AMPs. For all configurations, we see a distinct difference in group behavior.

for configurations containing the most complex core (i7) and the most simple core (Atom) where only the frequency is varied. Finally, a plot for the combination of the i7 and Pentium M running at their extreme frequencies is also shown. This differs from the configuration containing the i7 and the Atom in that both cores use out-of-order execution (Atom uses in-order execution).

For all configurations, there is a significant difference in the behaviors for each groups. Configurations with smaller differences between core types appear to have more outliers, however, the accuracy is still high in such cases.

Finally, for the most varied asymmetric configuration (containing the i7 at 2.93GHz with the Atom at 0.80GHz) the boxplots for the grouping for each benchmark for each input set is presented in Figure 7.5.

Nearly all benchmark and input combinations show a significant difference in the distribution of behavior for the two groups. Two benchmarks, ammp and gzip, differ in that they only give a single group (in both cases, the second group). Fortunately, as the figure shows, the behavior of all loops in these benchmarks fit nicely into this group. The only benchmark and input combination where the inner quartile ranges slightly overlap is bzip2 with its third input (denoted bzip2.2 in the figure). Fortunately, the overlap is small, and the accuracy for this grouping is still 86%.

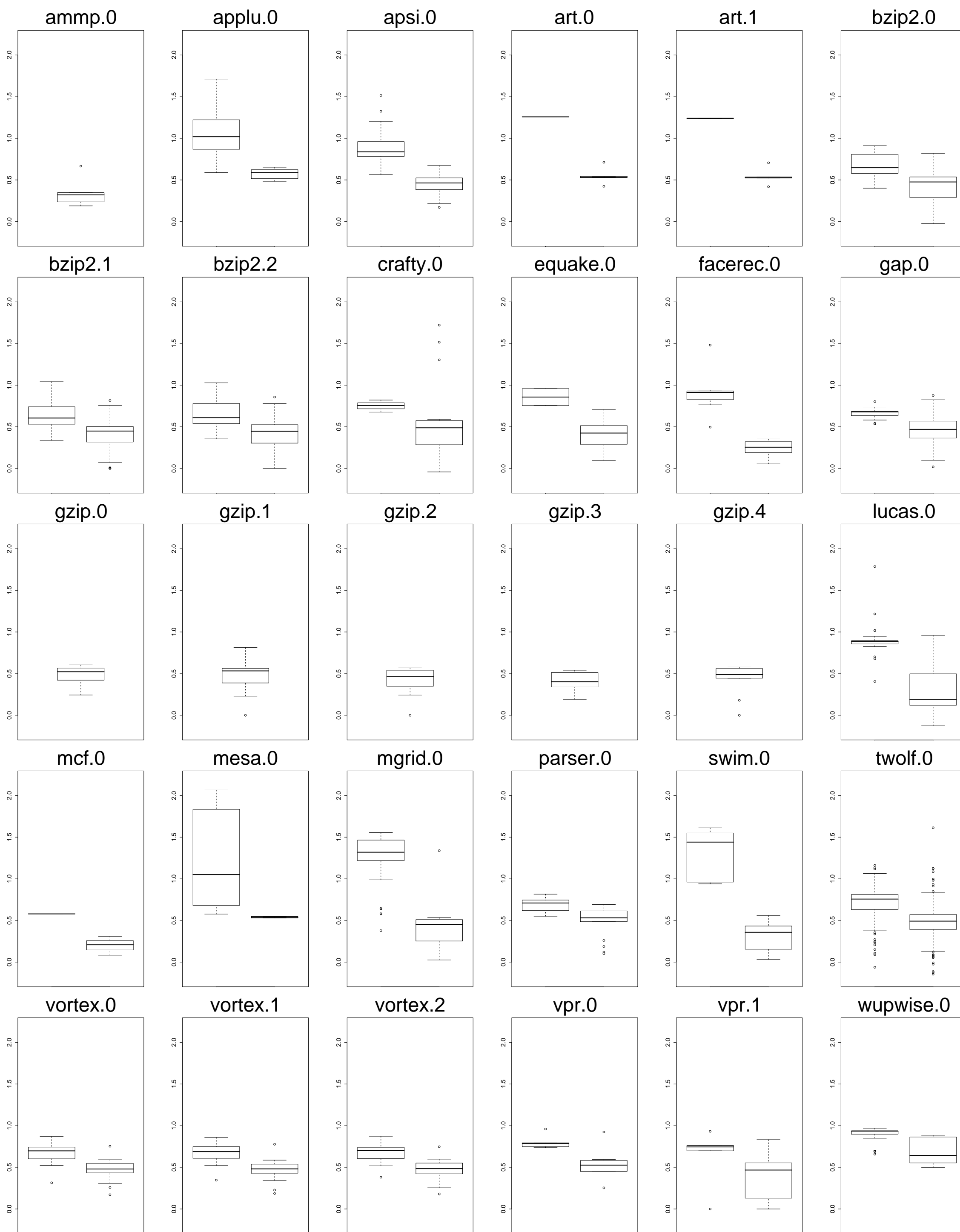

Figure 7.5 Boxplots for each benchmark and input combination for the asymmetric configuration containing two core types: Intel i7 @ 2.93GHz and Intel Atom @ 0.80GHz. Y-axis behavior is the difference between in behavior between the two core types. Numbering after the benchmark name denotes input set.

7.4 Space Overhead

Sending the similarity metrics (computed at compile time) with programs has some overhead in terms of space. Approximately 80bytes per loop is needed in the current implementation, however, this could easily be reduced if necessary. Figure 7.6 shows the required space overhead for each benchmark.

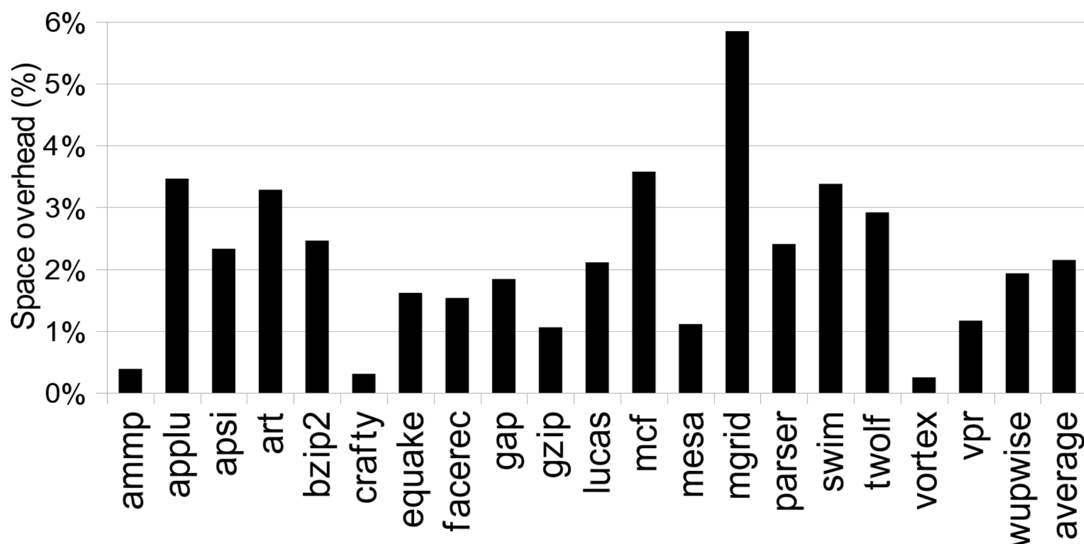

Figure 7.6 Space overhead of behavior metrics.

On average the space overhead is approximately 2%. The best case, vortex, has only 0.25% space overhead. Again, this overhead could be reduced if necessary by simply using more compact data types (e.g., the implementation currently uses, but does not necessarily need, four bytes to store the number of basic blocks in a loop).

7.5 Lazy Grouping in Phase-based Tuning

Lazy grouping has been tested within phase-based tuning on an AMP containing the Opteron 2431 with five cores at 0.80GHz and one at 2.40GHz. This system was chosen over one similar to the previous evaluation since the core types vary more (“fast” core frequency is 3x “slow” core) and the core type ratio is more realistic (more skewed). Because there are fewer “fast”

cores (1/6 instead of 1/2), it is expected that speedups will be smaller than those reported in the previous evaluation.

7.5.1 Impact of Lazy Grouping on Overheads

Since lazy grouping eliminates the need for runtime monitoring and analysis, the code needed to perform phase-based tuning at runtime is simplified significantly.

7.5.1.1 Space Overhead

In terms of space overhead, there is an average decrease of 25.3% over phase-based tuning with compile time grouping. That is, the space overhead of the best technique (loop strategy with minimum size of 45) gives an average 2.95% space overhead instead of 3.98%. This reduction is because the body of the inserted code no longer needs to contain code for monitoring behavior and making core mapping decisions.

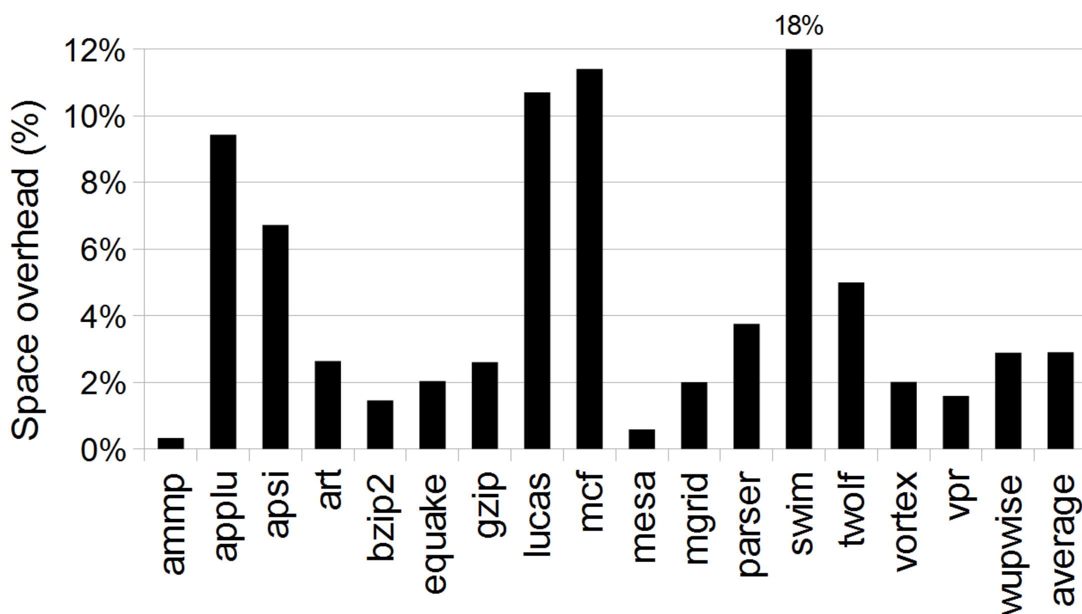

Figure 7.7 Space overhead per benchmark of phase-based tuning when using lazy grouping.

Figure 7.7 shows the space overhead per benchmark of phase-based tuning when lazy grouping is used. The figure shows that nearly all benchmarks have less than 4% space over-

head (most are around 2%). A few benchmarks have a significantly higher space overhead (e.g., swim, mcf, applu, and lucas). Some of these are very small programs (e.g., swim and mcf) whereas the others have many non-nested loops (e.g., applu and lucas).

7.5.1.2 Time Overhead

For time overhead, lazy grouping gives an average decrease of 55.9% time overhead from phase-based tuning with compile time grouping. Again using the best technique lazy grouping reduces time overhead to 0.075% from 0.17% when compile time grouping was used. This reduction is mainly because the inserted code no longer needs to check if a decision regarding core mapping has been made. Additionally, the reduction in space overhead may improve cache behavior in some cases.

The previous measurements considered the time overhead of benchmarks running as part of workloads. Thus, latency of system calls is hidden by executing other processes in the workload while waiting for the system calls to complete. We now consider the overhead of benchmarks running in isolation. Figure 7.8 shows these overheads per benchmark and Figure 7.9 shows a summary boxplot of this data.

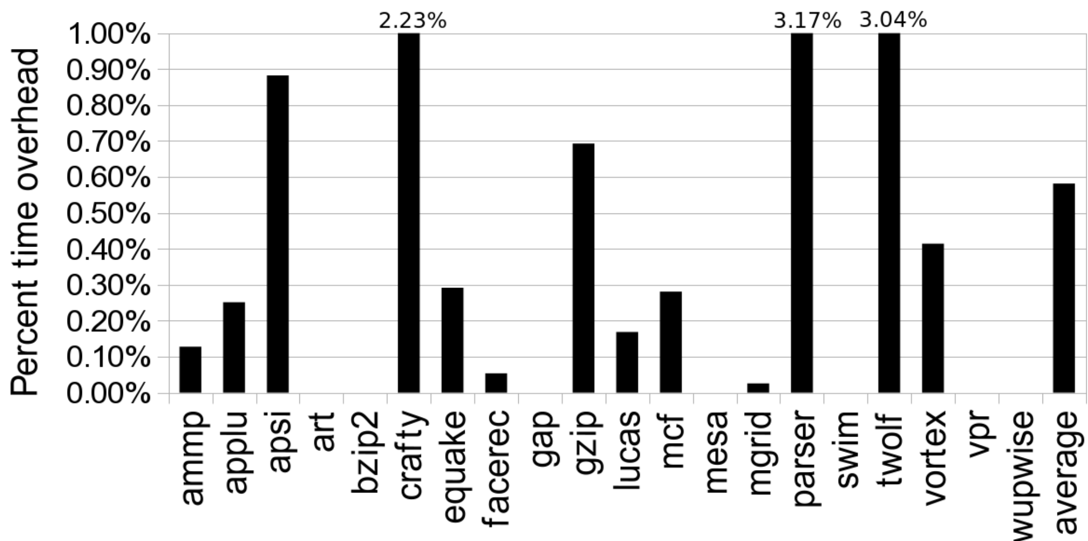

Figure 7.8 Time overhead per benchmark of phase-based tuning when using lazy grouping.

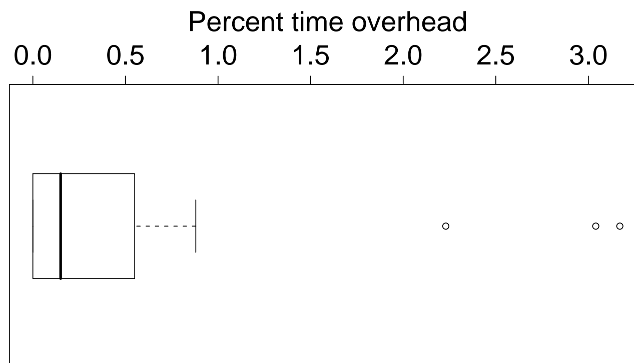

Figure 7.9 Summary of time overhead of phase-based tuning when using lazy grouping.

These figures shows that most benchmarks have extremely low time overheads even when running in isolation (long latency system calls can not be hidden and cache behavior is more likely to be degraded). Most benchmarks have below 0.3% time overhead and several have no noticeable overhead.

7.5.2 Impact of Lazy Grouping on Speedup and Fairness

For this experimental setup, using phase-based tuning with compile time grouping gives approximately 6% average process speedup (for a continuous job feed) over the stock Linux scheduler. With lazy grouping an average speedup of approximately 25% was observed. This improvement over phase-based tuning with compile time grouping is due to three factors. First, the lazy grouping has a distribution more in line with the core type distribution (i.e., more skewed). Second, the group-to-core assignment of the lazy groups is known statically whereas the compile time grouping approach must perform runtime monitoring. Third, lazy grouping is more accurate (Figure 7.1).

In terms of fairness, the use of lazy grouping reduces max-flow by and additional 5% (a 17% decrease in max-flow instead of 12% achieved with compile-time grouping). Max-stretch is further reduced by 8% (28% decrease instead of 20%).

7.6 Summary

The results in this chapter demonstrated that lazy grouping is more accurate than compile time grouping. The results also showed that lazy grouping is accurate for a wide range of target machines and AMPs while requiring few groups. Finally, the results showed that lazy grouping drastically improves the application of phase-based tuning and makes it applicable to a more wide range of target AMPs.

CHAPTER 8. Related Work

This chapter gives an overview of previous work that is related to the major components of this thesis. First, previous techniques related to phase-based tuning and optimization for asymmetric multicore processors is discussed. Then, previous work related to lazy grouping and machine learning are discussed.

8.1 Related Work: Phase-based Tuning

While there is no previous work which has all of the benefits of phase-based tuning, there does exist previous work which aims to effectively utilize asymmetric systems. This work may be divided into two categories. First, those which aim to improve performance for heterogeneous multicore processors. Second, those which aim to develop algorithms and programming environments for heterogeneous systems.

Becchi *et al.* [9] propose a dynamic assignment technique making use of the IPC of program segments. However, this work focuses largely on the load balance across cores whereas lazy grouping aims to maximize throughput. Another technique which focuses on load balancing in the scheduler was proposed by Fedorova *et al.* [72]. They make the case that a core assignment must be balanced. Shelepov *et al.* [72] propose a technique which does not require dynamic monitoring (uses static performance estimates). However, this technique does not consider behavior changes during execution. Li *et al.* [53] and Koufaty *et al.* [42] focus on load balancing in the OS scheduler. They modify the OS scheduler based on the asymmetry of the cores. While this produces an efficient system, the scheduler needs knowledge of the

underlying architecture. Phase-based tuning differs from these in the following way. First, this work is not directly concerned with load balancing. Second, this work focuses on properly scheduling the different phases of a programs behavior.

Tam *et al.* [81] determine thread-to-core assignment based on increasing cache sharing. They use cycles per instruction (CPI) as a metric to improve sharing for symmetric multicore processors. Kumar *et al.* propose a temporal dynamic approach [45]. After pre-defined time intervals, a sampling phase is triggered. After this sampling phase the system makes assignment decisions for all currently executing processes. This assignment is then used for some amount of time. This procedure is carried out throughout the entire programs execution. To reduce the dynamic overhead, phase-based tuning does not require monitoring once assignment decisions have been made.

Jiménez *et al.* developed a scheduler aimed at making use of heterogeneity in terms of CPU and GPU [40]. This work is targeted at a specific type if heterogeneous multicores with different ISAs whereas this work focuses on single ISA asymmetric multicores. Since they focus on cores with different ISAs they require the programmer to note which functions can be executed on both core types or provide implementations for both core types.

Mars and Hundt [56] use a similar hybrid technique to make use of a range of static optimizations and choose the correct one at runtime using monitoring. Their approach differs in that their optimizations are multiple versions of the code. This requires optimization to be done during compilation to avoid problems such as indirect branches in the binary. Thus they are tied to a specific compiler (phase-based tuning is not). Similarly, Dubach *et al.* [23] use machine learning to dynamically predict desired hardware configurations for program phases. Phase-based tuning does not determine the best configuration for phases, instead, it chooses the best from a set of choices.

A wide range of research has been done on algorithms for optimal distribution of computation over heterogeneous networks of computers [47, 48, 69]. This work can be divided into two categories: approaches for finding an optimal distribution that assume that the processor

characteristics and the program characteristics are known [8, 47, 69], and techniques that allow programmers to specify the program's characteristics and use this input to obtain an optimal distribution [46, 48]. Instead of working with a heterogeneous network of computers, this work focuses on heterogeneous multicores. The main issue with these two classes of ideas is that currently the level of specifications expected from the programmer is high, which significantly increases the intellectual burden of the task, which in turn impairs its practicality. Phase-based tuning eliminates this burden.

8.2 Related Work: Lazy Grouping

This section gives an overview of previous work which is related to lazy grouping. This includes techniques for grouping similar programs or program segments and techniques that use machine learning for program optimization.

8.2.1 Behavior Similarity

There is a large body of work on determining phase behavior [75, 24, 79], using phase behavior to reduce simulation time [7, 20, 74, 30, 79, 73, 51], guide optimizations [37, 35, 64, 59, 67, 68, 82, 28, 13], etc. Many of these techniques determine phase information with a previously generated dynamic profile [75]. Other techniques determine phase behavior dynamically [64], these techniques do not require representative input, however, they are likely to incur dynamic overheads.

Among these techniques, the most closely related is that of Sherwood *et al.* [75] which developed a technique for clustering segments of execution time. Runtime profiles are gathered and execution count of basic blocks is used to cluster program segments using k-means [55]. Lazy grouping differs in that it groups structural elements of the program rather than execution time. Further, their clustering is based on profiles of executed basic blocks rather than statically computed behavior metrics like lazy grouping.

Hoste and Eeckhout [36] present the MICA tool which uses execution profiles to cluster benchmarks with similar behavior. Unlike their work, all of the analysis used for lazy grouping is static. Also, lazy grouping considers program segments rather than whole programs.

8.2.2 Machine Learning for Optimization

There has been substantial previous work using machine learning to pick effective optimizations both statically and at runtime [17, 1, 52, 83, 15, 2, 62]. Here, the previous work most related to lazy grouping is discussed.

Choi and Yueng [17] use machine learning to determine efficient thread distributions on SMT systems. Periodically throughout execution (i.e. every n cycles) threads are reassigned to potentially different cores. Lazy grouping differs in that it analyzes behavior based on program structure rather than time. Also, they train their neural network dynamically whereas lazy grouping uses neural networks to determine grouping statically. However, the general ideas from lazy grouping could potentially be used to move some of their dynamic analysis into static analysis.

AbouGhazaleh *et al.* [1] use machine learning to dynamically scale clock frequency in embedded systems. Representative inputs are used to determine efficient power management policies which are used for training. Their compiler uses this information to optimize the program. At runtime by monitoring performance periodically, decisions are made regarding the power management policy. Lazy grouping differs in that its "phases" are determined without ever running the program.

Li *et al.* [52] use neural networks to predict power consumption of algorithms. Their networks take as input user supplied values for time complexity, space complexity, and input scale. Lazy grouping differs in that it is completely automatic. Lazy grouping also focuses on approximating behavior rather than power consumption (e.g. lazy grouping does not focus on how many iterations a loop may take but instead behavior of the iterations).

Wang and O'Boyle [83] use machine learning to help partition stream based programs onto

cores. The goal is to choose a good combination of parameters (e.g. level of loop unrolling, number of threads per loop, when to split and join, etc). Unlike their approach, lazy grouping does not consider modifications to program structure. Further, lazy grouping focuses on behavior of individual segments rather than entire workloads.

Moss *et al.* [62] use machine learning to improve the scheduling of straight line code (i.e. basic blocks). Lazy grouping differs in that it is focused on predicting behavior rather than changing it. Lazy grouping also looks at loops (a larger granularity) rather than basic blocks.

CHAPTER 9. Future Work

Here, future work is discussed. First ideas for future applications of phase-based tuning and lazy grouping are discussed. Then, potential improvements for the existing applications of phase-based tuning and lazy grouping are given.

9.1 Potential Applications of Phase-based Tuning

Here, some potential applications of phase-based tuning beyond optimization for AMPs are discussed. This includes improving JIT optimizers and optimizing for power consumption.

9.1.1 JIT Optimizations

One potential direction for future work is to use phase-based tuning to apply more typical program optimizations at runtime like those found in a JIT optimizing compiler. The advantage of applying phase-based tuning to this style of optimization is that the JIT optimizer could focus on optimizations to code that are more likely to be beneficial and also begin to perform them before they are needed (i.e., before execution reaches these points). For example, suppose the target optimization is loop splitting. If the JIT optimizer determines that loop splitting gave significant benefits for a certain loop, similar loops could be optimized (in parallel to program execution) before they are reached. Phase-based tuning could also be used to avoid attempting to optimize loops in groups determined not worthwhile to split.

The difficulties of applying phase-based tuning to this problem are mostly with respect to the idea of similarity. In phase-based tuning, similarity is defined in terms of response

to the target optimization(s). Since the target optimization(s) are substantially different than previous work (and include code modification), similarity will have a different meaning than that explored in this work. Consider the example of the loop splitting optimization. In this case, loops could be grouped based on their structure and/or various cache metrics. Future work in this direction would be focused on answering the question “What does it mean for two code segments to be similar?” for various JIT optimizations.

9.1.2 Power Consumption

Another application for phase-based tuning is to use frequency scaling to optimize the performance to power ratio for processors. Frequency scaling is commonly used in mobile CPUs for reducing power consumption. The idea is that when the system is under low load and/or running on battery, the clock frequency (and cache associativity in the Intel Atom) is reduced. Unfortunately, there are times when the system is under a heavy load but may spend much of its time waiting for other resources such as memory. Consider an application of phase-based tuning where instead of switching core types, core frequency of the current core is scaled. This has the advantage of increasing the core frequency only when it will greatly improve performance. Further, grouping can be tuned based on the desired aggressiveness of power savings.

A major difficulty with this direction for future work is that processes often do not run by themselves on a core. That is, other processes often share the same core. So, changes in frequency to a core will potentially cause other processes to not execute at their desired frequency. For example, suppose we have two processes running on the same core. One process does not benefit from high-frequency execution whereas the other does. Since most schedulers will continually switch between the two processes, if frequency scaling is done the way previous work performs core switches, one process will execute at a non-ideal frequency. For example, suppose the process wanting low frequency enters this phase first and changes the core frequency to low. Next, the process wanting high frequency enters this phase and

changes the frequency to high. In this case, the process trying to save power by using a low frequency will not save any power.

A potential solution to this problem is to make the scheduler aware of this power saving technique. Then, schedulers could choose to switch frequency during context switches and/or perform load balancing based on each processes desired frequency. For example, suppose we have a system with two cores that is running four processes. Half the processes want to run at low frequency and the other half desire a high frequency. If the operating system groups these processes onto cores by desired frequency (the two "fast" processes on one core and the two "slow" on the other), it can avoid constantly changing core frequencies. Both of these techniques have complex runtime costs and benefits that would need to be explored. For example, costs of changing frequency (in terms of both run time and power) would need to be determined as well as costs of changing cores (run time and power). Knowledge of these costs would be necessary in order to determine whether or not frequency should be switched upon context switches or if processes should be groups (and potentially moved).

9.2 Potential Applications of Lazy Grouping

The basic idea behind lazy grouping is to statically group program segments which will behave similarly. This has many potential applications. Some of these are now discussed.

9.2.1 Guide Optimization

One potential application of lazy grouping is for determining program segments for static optimization. A lazy grouping of segments could be used to find groups of program segments that perform poorly on the target system and thus could use optimization. For example, different systems have different cache hierarchies. Depending on the target system, cache optimizations such as loop splitting may or may not be desirable. Thus, at install time, we can statically optimize the program for the target machine.

To adapt lazy grouping for this use, the construction of the training set would change. Instead of observed execution behavior, the expected outputs in the training set would be response to given optimizations. Thus, each code segment in the training set would need to be run with and without the candidate optimization and benefits between the two determined. Benefits here depends on the goal of optimization (e.g., reducing space, time, etc.). A potential difficulty with this direction is that additional similarity metrics may be needed for accurate grouping (e.g., metrics regarding control structure).

9.2.2 Benchmark Selection

There has been a substantial amount of previous work for predicting similarity in terms of whole benchmarks [36] and throughout program execution [75]. This previous work is frequently used for choosing a small but diverse set of benchmarks or points within benchmarks that represent a wide range of program behavior. These techniques typically select a representative set for all potential target architectures. Lazy grouping could be used to determine such a grouping tailored specifically for the target system rather than all possible systems. This would likely further reduce the number of benchmarks or points within benchmarks needed to represent a wide range of program behavior.

9.3 Small Improvements

There are several options for small improvements to the techniques presented in this thesis. These future directions for both phase-based tuning and lazy grouping are described here.

9.3.1 Phase-based Tuning

Load balancing during phase-based tuning for AMPs could be improved in specific situations. For example, for systems aiming to maximize performance, the scheduler could ensure that the “fast” cores are never idle while “slower” cores are busy. To do this efficiently, the

OS scheduler's load balancing techniques would likely need to be changed. Additionally new "soft" CPU affinity calls would likely need to be added (current affinity calls are "hard" in that the scheduler must respect call arguments for allowable cores). This has the disadvantage of requiring OS modifications. These techniques could be worked into the phase-mark code. However, the overhead easily becomes too large.

9.3.2 Lazy Grouping

Minor improvements to lazy grouping mostly revolve around improving the computation of similarity metrics. Here, some of these potential improvements are described.

9.3.2.1 More similarity metrics

The most obvious source of improvements is adding more similarity metrics. For example, the accuracy of runtime branch prediction could be estimated. Since branch miss-prediction at runtime has a high penalty, estimating this aspect of behavior will likely improve grouping accuracy. Possible techniques for estimating this behavior could involve looking at patterns in the code (like Buse and Weimer [14]) or use of more complex static analysis for branch prediction. Another metric could be register usage rates (like those used in MICA [36]).

9.3.2.2 Precision of Similarity Metrics

Another source of improvement is improving the precision of each underlying similarity metric. For example, the reuse distance analysis used for data cache behavior could look more closely at operands. Currently, the "stepping" of memory locations is not considered. Consider an instruction which uses memory location that is increased by the size of a word at the end of each loop iteration (e.g. array of words). In such a case, it is likely that accessing this memory location will be a cache hit (rough probability of $\frac{n-1}{n}$ where n is the number of words per cache block). Now, suppose the memory location is increased by more than the size of a cache block at each iteration (e.g. array of large structures). In this case, it is likely that accessing

this memory location will be a cache miss (roughly 100% assuming pre-fetching is not done). However, in the current analysis, the best case assumption is made, so cases such as this are handled less accurately.

For another example, consider the static estimation of ILP. One way to improve precision would be to have the analysis consider functional unit hazards instead of just register and memory location dependencies.

CHAPTER 10. Conclusion

AMPs are an important class of processors that have been shown to provide nice trade-off between the die size, number of cores on a die, performance, and power [31, 44, 61]. Devising techniques for their effective utilization is an important problem that influences the eventual uptake of these processors [53, 61]. The need to be aware of, and optimize based on, the applications' characteristics and the nature of the AMP significantly increases the burden on developers. Furthermore, the need to create separate versions for each target AMP decreases reusability and creates a maintenance burden.

This thesis described a novel technique called phase-based tuning which solves all of these problems by utilizing the phase behavior that is common in programs. Phase-based tuning is fully automatic, can be deployed in existing tool chains, and produces asymmetry-independent binaries. This significantly reduces the expertise necessary for programming performance-asymmetric multicores. Apart from these benefits, phase-based tuning also has several performance advantages. Experiments show that, for systems with simple asymmetry, a 36% reduction in the average process time compared to the stock Linux scheduler. This is done while incurring negligible overheads (less than 0.2% time overhead) and maintaining fairness.

To make phase-based tuning applicable to a wide variety of AMPs, lazy grouping was proposed and evaluated. Lazy grouping is a novel static for accurately detecting similar program segments. Results demonstrated that lazy grouping is significantly more accurate than compile time grouping (when the target machine is unknown). Experimental results show that lazy grouping is more than 90% accurate for nearly all target machines. Finally, the benefits of lazy grouping over compile time grouping for phase-based tuning were demonstrated.

BIBLIOGRAPHY

[1] N. AbouGhazaleh, A. Ferreira, C. Rusu, R. Xu, F. Liberato, B. Childers, D. Mosse, and R. Melhem. Integrated CPU and L2 cache voltage scaling using machine learning. In *LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 41–50, New York, NY, USA, 2007. ACM.

[2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, 2006.

[3] F. E. Allen. Control flow analysis. In *Symposium on Compiler optimization*, pages 1–19, 1970.

[4] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttling. In *ISCA '05: Proceedings of the 32st annual international symposium on Computer architecture*, June 2005.

[5] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA '05: Proceedings of the 32st annual international symposium on Computer architecture*, June 2005.

[6] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymme-

try in emerging multicore architectures. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer architecture*, pages 506–517, 2005.

[7] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *International symposium on Microarchitecture*, pages 245–257, 2000.

[8] O. Beaumont, V. Boudet, and A. Petitet. A proposal for a heterogeneous cluster scalapack (dense linear solvers). *IEEE Trans. Comput.*, 50(10):1052–1070, 2001.

[9] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF '06: International Conference on Computing Frontiers*, pages 29–40, 2006.

[10] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Annual symposium on Discrete algorithms*, pages 270–279, 1998.

[11] K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, pages 617–662, 2001.

[12] C. Boneti, R. Gioiosa, F. J. Cazorla, and M. Valero. A dynamic scheduler for balancing hpc applications. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[13] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Java '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, New York, NY, USA, 1999. ACM.

[14] R. P. L. Buse and W. Weimer. The road not taken: Estimating path execution frequency statically. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 144–154, Washington, DC, USA, 2009. IEEE Computer Society.

[15] J. Cavazos. *Automatically Constructing Compiler Optimization Heuristics Using Supervised Learning*. PhD thesis, University of Massachusetts, Amherst, 2004.

[16] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: 20th annual conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, 2005.

[17] S. Choi and D. Yeung. Learning-based smt processor resource distribution via hill-climbing. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 239–251, Washington, DC, USA, 2006. IEEE Computer Society.

[18] J. Cook, R. L. Oliver, and E. E. Johnson. Toward reducing processor simulation time via dynamic reduction of microarchitecture complexity. *ACM SIGMETRICS Performance Evaluation Review*, pages 252–253, 2002.

[19] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[20] A. Dhodapkar and J. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA '02: Proceedings of the 29st annual international symposium on Computer architecture*, pages 233–244, 2002.

[21] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *MICRO 36: Proceedings of the 36th annual International Symposium on Microarchitecture*, page 217, 2003.

[22] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters. In *PADTAD Workshop*, 2003.

[23] C. Dubach, T. M. Jones, E. V. Bonilla, and M. F. O'Boyle. A predictive model for dynamic microarchitectural adaptivity control. In *MICRO 43: 43rd annual International Symposium on Microarchitecture*, 2010.

[24] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *PACT '03: Parallel Architectures and Compilation Techniques*, 2003.

[25] S. Eranian. perfmon2: a flexible performance monitoring interface for Linux. In *OLS 06': Ottawa Linux Symposium*, 2006.

[26] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: a vector extension to the Alpha architecture. *SIGARCH Comput. Archit. News*, 30(2):281–292, 2002.

[27] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-timesystems. *Real-Time Syst.*, 17:131–181, December 1999.

[28] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *HiPEAC '05: Proceedings of the 1st International Conference on High Performance Embedded Architectures & Compilers*, number 3793 in LNCS, pages 29–46. Springer Verlag, November 2005.

[29] D. Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[30] A. Georges, D. Buytaert, L. Eeckhout, and K. D. Bosschere. Method-level phase behavior in Java workloads. In *OOPSLA '04: Proceedings of the 19th annual conference on Object-oriented programming, systems, languages, and applications*, pages 270–287, 2004.

[31] M. Gillespie. Preparing for the second stage of multi-core hardware: Asymmetric cores. *Tech. Report - Intel*, 2008.

[32] T. Granlund. Instruction latencies and throughput for AMD and Intel x86 processors, March 2011. http://gmplib.org/ tege/x86-timing.pdf.

[33] E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of both latency and throughput. In *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pages 236–243, Oct. 2004.

[34] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[35] M. Hock, K. Jayaraman, B. Pellin, and V. Shrivastava. Phase capture and prediction with applications. *Technical Report - Computer Science Department - University of Wisconsin-Madison*, 2005.

[36] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72, 2007.

[37] S. Hu. *Efficient Adaptation of Multiple Microprocessor Resources for Energy Reduction Using Dynamic Optimization*. PhD thesis, The University of Texas at Austin, 2005.

[38] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. *Computer Architecture News*, 31(2):157–168, 2003.

[39] Intel Corp. Intel®64 and IA-32 architectures software developer's manual volume 1: Basic architecture, March 2009. http://www.intel.com/products/processor/manuals/.

[40] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 19–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[41] P. B. Kessler. Fast breakpoints: design and implementation. In *PLDI '10: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 78–84, New York, NY, USA, 1990. ACM.

[42] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 125–138, New York, NY, USA, 2010. ACM.

[43] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 23–32, 2006.

[44] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[45] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04: 31st annual international symposium on Computer architecture*, page 64, 2004.

[46] A. Lastovetsky. Adaptive parallel computing on heterogeneous networks with mpC. *Parallel Comput.*, 28(10):1369–1407, 2002.

[47] A. Lastovetsky and R. Reddy. Data partitioning with a realistic performance model of networks of heterogeneous computers. In *International Parallel and Distributed Processing Symposium*, April 2004.

[48] A. Lastovetsky and R. Reddy. HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. *J. Parallel Distrib. Comput.*, 66:197–220, 2006.

[49] A. L. Lastovetsky and J. J. Dongarra. *High Performance Heterogeneous Computing*. John Wiley & Sons, Inc., 2009.

[50] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[51] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *HPCA '05: 11th International Symposium on High-Performance Computer Architecture*, pages 278–289, 2005.

[52] Q. Li, B. Guo, Y. Shen, J. Wang, Y. Wu, and Y. Liu. An embedded software power model based on algorithm complexity using back-propagation neural networks. In *GREENCOM-CPSCOM '10: Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 454–459, Washington, DC, USA, 2010. IEEE Computer Society.

[53] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC '07: conference on Supercomputing*, pages 1–11, 2007.

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

[55] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *5th Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[56] J. Mars and R. Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *CGO '09: Proceedings of the International Symposium on Code Generation and Optimization*, 2009.

[57] W. Mathur and J. Cook. Towards accurate performance evaluation using hardware counters. In *ITEA Modeling and Simulation Workshop*, 2003.

[58] D. Menascé and V. Almeida. Cost-performance analysis of heterogeneity in supercomputer architectures. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 169–177, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[59] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. mei W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 136–147, 1999.

[60] Z. Miled and J. Fortes. A heterogeneous hierarchical solution to cost-efficient high performance computing. In *8th IEEE Symposium on Parallel and Distributed Processing*, volume -, pages 138–145, Oct 1996.

[61] J. C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro*, 28(3):26–41, 2008.

[62] E. Moss, P. Utgoff, J. Cavazos, D. Precup, D. Stefanovic, C. Brodley, and D. Scheeff. Learning to schedule straight-line code. In *In Proceedings of Neural Information Processing Symposium*, pages 929–935. MIT Press, 1997.

[63] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Academic Press, 1997.

[64] P. Nagpurkar, C. Krintz, M. Hind, P. F. Sweeney, and V. T. Rajan. Online phase detection algorithms. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 111–123, 2006.

[65] S. Nissen. Implementation of a fast artificial neural network library (fann). Technical report, Department of Computer Science University of Copenhagen (DIKU), 2003. http://fann.sf.net.

[66] J. Parkhurst, J. Darringer, and B. Grundmann. From single core to multi-core: preparing for a new exponential. In *ICCAD '06: International conference on Computer-aided design*, pages 67–72, 2006.

[67] N. Peleg and B. Mendelson. Detecting change in program behavior for adaptive optimization. In *PACT '07: 16th international conference on Parallel architectures and compilation techniques*, pages 150–162, 2007.

[68] C. Pereira and R. Gupta. Using program phases as meta-data for runtime energy optimization. Technical report, Department of Computer Science & Engineering, UC San Diego, 2004.

[69] H. Renard, Y. Robert, and F. Vivien. Static load-balancing techniques for iterative computations on heterogeneous clusters. *Euro-Par '09: The 9th International European Conference on Parallel and Distributed Computing*, pages 148–159, 2003.

[70] S. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice Hall series in artificial intelligence. Prentice Hall, 2010.

[71] D. Sager, D. P. Group, and I. Corp. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 1:2001, 2001.

[72] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. Hass: a scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.*, 43(2):66–75, 2009.

[73] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 165–176, 2004.

[74] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: 10th international conference on Parallel architectures and compilation techniques*, pages 3–14, 2001.

[75] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, 2002.

[76] Z. Smith. The intel 8086 / 8088/ 80186 / 80286 / 80386 / 80486 instruction set, September 2011. http://80386.tk/.

[77] T. Sondag, K. L. Pokorny, and H. Rajan. Frances: A tool for understanding code generation. In *SIGCSE '10: The 41st ACM Technical Symposium on Computer Science Education*, March 2010.

[78] T. Sondag and H. Rajan. A more precise abstract domain for multi-level caches for tighter WCET analysis. In *RTSS '10: The 31st IEEE Real Time Systems Symposium*, November 2010.

[79] R. Srinivasan, J. Cook, and S. Cooper. Fast, accurate microarchitecture simulation using statistical phase detection. In *ISPASS '05: Proceedings of the International Symposium on Performance Analysis of Systems and Software*, page 147, 2005.

[80] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM.

[81] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. *Operating Systems Review*, 41(3):47–58, 2007.

[82] F. Vandeputte, L. Eeckhout, and K. D. Bosschere. Exploiting program phase behavior for energy reduction on multi-configuration processors. *Journal of Systems Architecture*, 53(8), 2007.

[83] Z. Wang and M. F. O'Boyle. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 307–318, New York, NY, USA, 2010. ACM.