Phase-based tuning for better utilized performance-asymmetric multicores

by

Tyler Sondag

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Hridesh Rajan, Major Professor
Morris Chang
Jack Lutz

Iowa State University

Ames, Iowa

2009

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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, I would like to thank my adviser Dr. Hridesh Rajan for being a great mentor over the past few years. His advice and guidance have helped me grow tremendously as a researcher. He continues to be a great source of inspiration and keeps me on the right track. I would also like to thank Dr. Jim Feher and Dr. Kian Pokorny who inspired me to go to graduate school, to pursue a career in research, and gave great advice throughout my undergraduate career.

I thank Patrick Carlson and Paul Murphy for their help developing the analysis and instrumentation framework used for evaluating the ideas in this thesis. Also, thanks to the members of the Laboratory for Software Design at ISU who were helpful in developing the ideas in this paper and for their constructive criticism and suggestions during all stages of my work. I would like to thank Dr. Andrew Miner for his comments and suggestions regarding the presentation of these ideas. I would like to thank the anonymous reviewers and attendees of the IWMSE '09 and PLOS '07 workshops for their comments and suggestions. Thanks are due to the National Science Foundation for financially supporting this project.

Finally, I would like to thank my parents Brenda and Dale Sondag 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 Sondag for sparking my interest in mathematics at an early age and for being a great role model. My brother Trent Sondag letting me practice (and improve) my teaching skills on him. My girlfriend Kasey Williams for being supportive before deadlines and pushing me to finish and submit all of my paperwork for forming my POS committee and graduating. All of my APO brothers and friends who gave me much needed breaks from research and work.

ABSTRACT

The latest trend towards performance asymmetry among cores on a single chip of a multicore processor is posing new software engineering challenges for developers. A key challenge is that for effective utilization of these performance-asymmetric multicore processors, application threads must be assigned to cores such that the resource needs of a thread closely matches resource availability at the assigned core. Determining this assignment manually is tedious, error prone, and it significantly complicates software development. We contribute a transparent and fully-automatic program analysis, which we call *phase-guided tuning*, to solve this problem. Phase-guided tuning adapts an application to effectively utilize performance-asymmetric cores of a processor. Our technique does not require any changes in the compiler or operating system, thus it is easy to deploy in existing tool chains. It does not require any input from the programmer except the application. Furthermore, it is independent of the characteristics (performance-asymmetry) of the target multicore processor, which has two benefits. First, it avoids the need to create multiple customizations of the binary for each target architecture, and second it relieves the programmer of the burden of anticipating the target architecture. Last but not least, our technique significantly improves performance. Compared to the stock Linux scheduler, our best technique shows 215% improvement in throughput and 36% average process speedup, while maintaining fairness and with negligible overheads.

CHAPTER 1. Introduction

CPUs with multiple cores have become commodity items [53, 18]. CPU vendors are projecting that in the next decade the number of cores in a CPU will increase to as many as hundreds [9]. This makes it important to devise techniques for their effective utilization. Recently both CPU vendors and researchers have advocated the need for a class of multicore processors called single-ISA performance-asymmetric multicores [3, 4, 20, 27, 35, 40]. All cores in a performance-asymmetric multicore processor support the same instruction set, however, they differ in terms of performance characteristics such as clock frequency, cache size, etc [20, 28, 40]. These architectures have been shown to provide an effective trade-off between performance, die area, and power consumption compared to homogeneous multicore processors [20, 27, 35, 40].

Programming performance-asymmetric multicore processors is, however, much harder compared to their homogeneous counterparts. To effectively utilize these processors, application threads must be executed on cores such that the resource requirements of a thread closely match the resources provided by the core [26, 35]. This must be done while maintaining fairness between threads. To match the resource requirements of a thread to the resources provided by the core, both must be known. The programmer can manually tune the application to achieve such a mapping, however, this manual tuning has at least three problems. First, the programmer must be aware of the runtime characteristics of the program code as well as the details of the underlying performance asymmetry, which increases the intellectual burden on the programmer. Second, with multiple target architectures this manual tuning must be carried out for each architecture, which can be costly, tedious, and error prone. Third, as a result of this manual tuning a custom version must be created for each architecture, which decreases reusability and creates a maintenance problem. The performance asymmetry present in the target multicore processor may not be known during development, which further complicates manual tuning.

1.1 Contributions

The main technical contribution of this work is a novel program analysis technique, which we call *phase-guided tuning*, for matching resource requirements of threads to the resources provided by the cores of a performance asymmetric multicore processor. Phase-guided tuning builds on a well-known insight that programs exhibit phase behavior [22, 24, 34, 42, 47, 51, 54]. By phase behavior we mean that a program goes through phases of execution that show similar runtime characteristics compared to other phases [5, 11, 12, 13, 19, 49]. Based on this insight, our approach consists of two parts: a static analysis, which identifies likely *phase-transition points*, and a lightweight dynamic analysis that determines thread-to-core assignment on the fly. We define a phase-transition point as a point in the program where runtime characteristics are likely to change. The static analysis results are used to generate standalone binaries in which each phase-transition point is instrumented with a tiny fragment for dynamic analysis. Phase-guided tuning has the following software engineering benefits:

- **Programmer Oblivious:** The programmer need not be aware of the performance characteristics of their application or the performance asymmetry of the target multicore CPU.
- **Transparent Deployment:** No modification is needed to the operating system or compilers. Thus it can be utilized with minimal disruption in the build and deployment chain.
- **Tune Once, Run Anywhere:** The application tuning is independent of the performance characteristics of the target architecture, thus there is no need to create multiple versions[1].
- **Negligible Overhead:** It incurs less than 4% space overhead and less than 0.2% time overhead, i.e. it is useful for overhead conscious software and it is scalable.
- **Improved Utilization:** It improves utilization of performance-asymmetric multicore processors by increasing throughput by as much as 215% and by reducing average process time by as much as 36% while maintaining fairness between applications.

We have implemented phase-guided tuning as part of our binary static analysis and instrumentation framework, which shows the feasibility of the approach.

[1] As usual, a separate version is needed for different instruction sets.

To evaluate the effectiveness of phase-guided tuning, we applied it 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. For these workloads we observed as much as 215% improvement in the utilization of our performance-asymmetric processor setup while maintaining comparable fairness and negligible overheads. On average processes were as much as 36% faster without causing process starvation.

1.2 Organization

The rest of this thesis is organized as follows. Chapter 2 explains the two components of our approach: static phase transition analysis and marking, and dynamic analysis and optimization. Chapter 3 describes our static analysis and instrumentation framework. Chapter 4 presents our experimental results which evaluate the overheads involved with our technique as well as its overall benefits. Chapter 5 describes related work. Chapter 6 outlines our plans and ideas for future work. Chapter 7 concludes.

CHAPTER 2. Phase-guided Tuning

A program exhibits phase behavior [5, 11, 12, 13, 19, 49] in that it goes through several phases of execution that show similar runtime characteristics compared to other phases of execution. The intuition behind our approach is the following. We classify a program's execution into code sections; 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. By classifying a program's execution into sections and sections into clusters independent of the program's input, we see 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-guided tuning works as follows. A static analysis is performed to identify phase-transition points. This analysis proceeds as follows. First, we divide a program's code into *sections*. Second, we classify these sections into one or more *phase types* thereby clustering them into one or more groups such that each section in the cluster is likely to exhibit similar runtime characteristics. Third, we identify points in the program where the control flows [2] from a section of one phase type to a different phase type. These points are identified as phase-transition points.

Each phase-transition point is statically instrumented to insert a small code fragment which we call 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 performance of a small number of representative sections of each phase type. These analysis results are used to determine a suitable core assignment for

[1]The idea of phase marking is similar to the work by Lau *et al.* [33], however, we do not use a program trace to determine our phase marks and make our selections based on a different criteria.

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 representative sections of a given phase type are used as an approximation of the expected characteristics of all sections of that phase type. This allows our technique to automatically tackle new architectures. The rest of this section describes the individual components of our approach in detail.

2.1 Static Phase Transition Analysis

The aim of our static analysis is to determine points in the control-flow where phase behavior is likely to change. We refer to such points as *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 our 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. We then examine three different analysis to detect and mark phase transitions with *phase marks*. The first is a basic block level analysis. The second builds upon this basic block analysis to analyze intervals [2]. The third also builds upon the basic block analysis to analyze loops inter-procedurally.

Figure 2.1 illustrates this process for our basic block level analysis. Step 1 represents the initial procedure. 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, we reduce the phase transition points by 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, we first discuss the analysis techniques for annotating control-flow graphs (CFGs) with types for both of our techniques. Next, we discuss how to use the annotated control-flow graphs to perform the phase transition marking.

[2] Huang *et al.* [25] show that basing processor adaptation on code sections (positional) rather than time (temporal) improves energy reduction techniques. We also take a positional approach.

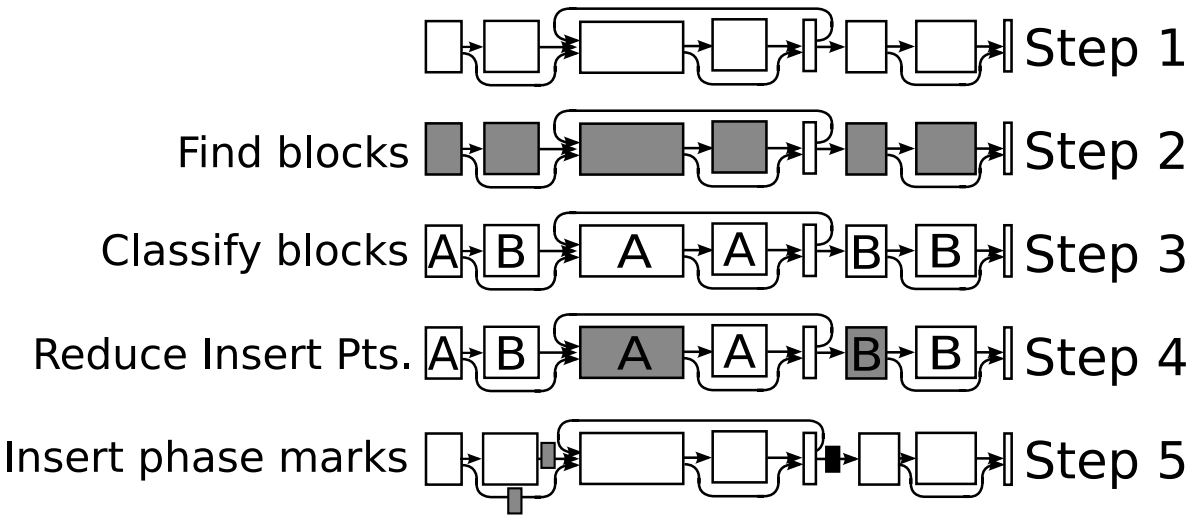

Figure 2.1 Overview of phase transition analysis

2.1.1 Static CFG Annotation

We now discuss the three analysis techniques used to annotate a programs control-flow graph with type information. First, we explain the technique used for our basic block analysis. Then, we expand this technique to include our technique for interval typing. Finally, we describe an inter-procedural loop based technique.

Attributed Control Flow Graph Construction. Our 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 ($\mathcal{B}$) [2]. We use the classic definition of a basic block that it is a section of code that has one entry point and one exit point with no jumps in between [2]. We then classify each basic block into exactly one type ($\pi \in \Pi$) 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 a basic block based on execution traces is given in Section 4.1, however, other methods for basic block classification 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 $\pi \in \Pi$.

Summarizing Intervals. The goal of our intra-procedural interval [2] analysis is to summarize intervals into a single type. To perform our interval analysis, we start with the attributed control-flow graph for each procedure created by the basic block analysis. We then use the basic block types to determine interval types.

For each procedure, we start by partitioning the attributed control-flow graph of the procedure into a unique set of *intervals* ($\mathcal{I}$) using standard algorithms [2]. "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 [2, pp.6]." For each interval, i, we then compute its dominant type 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 for a simple interval in Figure 2.3.

$\rho = \phi$
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 Interval summarization to find dominant type

During a depth-first traversal we maintain 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 $M \oplus \{\pi \mapsto M(\pi) + w_f * \varphi\eta\}$.

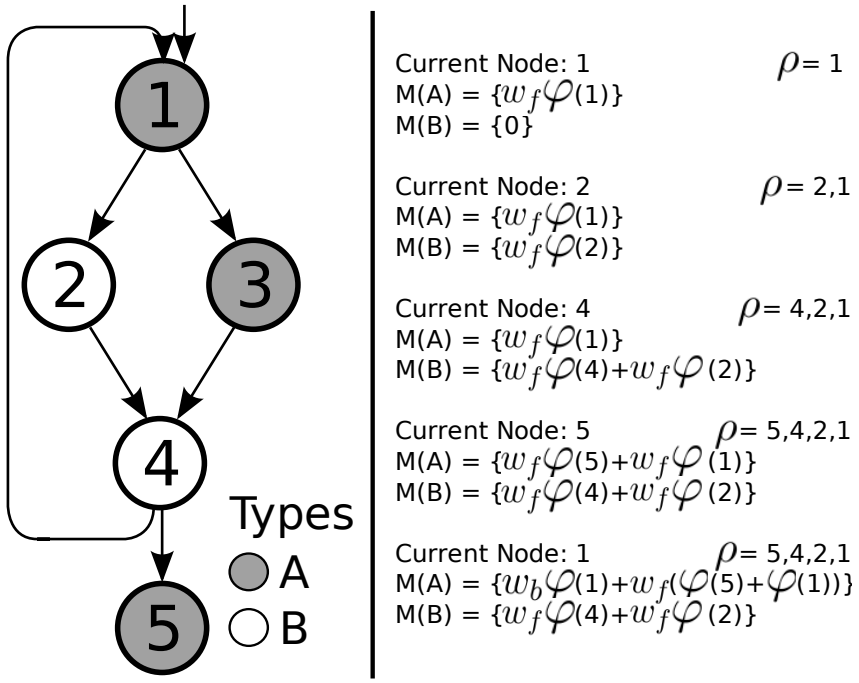

Figure 2.3 Interval summarization illustration

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. We currently use the number of instructions in the node 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, we obtain 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, we refer to them as *attributed interval graphs*. It would be interesting to explore whether summarizing interval graphs again is useful [2], however, in this paper we only consider first-order intervals. Our initial intuition is that the value of applying nth order interval summarization will depend on the average size of procedures.

Summarizing Loops. The goal of our inter-procedural loop analysis is to summarize loops into a single type. To perform our loop analysis, we start with the attributed control-flow graph for each procedure created by the basic block analysis. We then use 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, we randomly choose one procedure to analyze first then analyze all procedures again until a fixpoint is reached.

For each procedure, we start by partitioning the attributed control-flow graph of the procedure into a unique set of *loops* ($\mathcal{L}$) using standard algorithms [41]. For each loop, $l \in \mathcal{L}$, we then compute its dominant type starting with the inner-most loops. We do a breadth-first traversal of the loop starting with the entry node, while ignoring backward control-flow edges. .This summarization algorithm is shown in Figure 2.4 and illustrated in Figure 2.5.

Throughout the traversal, a type map for the loop ($M : \Pi \mapsto \mathbb{R}$) 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 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 contained 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 $w_n(x)$ where x is the nesting level and $w_n : \mathbb{N} \rightarrow \mathbb{R}$ which maps nesting levels to weights.

On completion of the breadth-first traversal, the dominant type of the loop l is π_l, where $\nexists \pi \text{s.t.} M(\pi) > M(\pi_l)$. In case of a tie, a simple heuristic is used. Currently, the type with maximal number of control-flow nodes in the loop is used as a tiebreaker. We also have a strength of the

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. $l' \subset l'' \subset l$ **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
  $T := T \cup \{(l, \pi_l, \sigma_l)\}$
end if

Figure 2.4 Loop summarization to find dominant type. BFS ignores back edges

type, σ which is simply the weight the type π_l over the sum of all other type weights. This strength is used for determining types of nested loops.

Suppose we have a loop l' which contains the current loop l. If both loops have the same type ($\pi_{l'} = \pi_l$), it doesn't make sense to incur the overhead of our analysis and optimization code at each iteration of the outer loop. Instead, we can do this analysis and optimization before the outer loop, and eliminate any work done inside this loop. Thus, after we determine the type for the current loop l, we get the next largest nested loop, l'. If there is no such loop, then we add 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 we add the current loop, l to the type map and remove the nested loop l'. If the types of the two loops differ, we take the type with the higher strength, σ since it is more likely that we have an accurate typing for such a loop.

As a result of this process, we obtain 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, we refer to them as *attributed loop graphs*.

Phase Transitions. Once we have determined types for sections of the program's CFG, we compute the phase transition points. Recall that a phase-transition point is a point in the program where

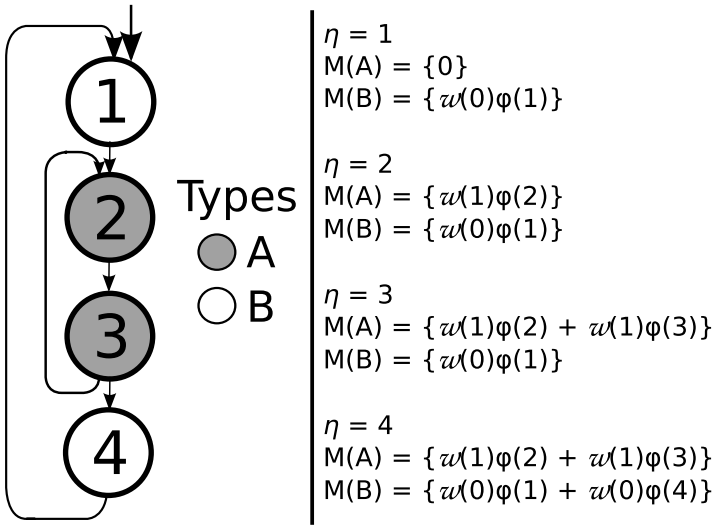

Figure 2.5 Loop summarization illustration

runtime characteristics are likely to change. Since sections of code with the same type should have approximately similar behavior, we assume that program behavior is likely to change when control flows from one type to another. The next section describes our techniques for marking these points in the application.

2.1.2 Phase Transition Marking

Once the phase transitions are determined, we statically insert phase marks 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, we eliminate the need for compiler modifications. Furthermore, by using standard techniques for core switching, we require no OS modification. We have considered several variations of phase transition marking 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 on the phase transitions.

Adding Phase Marks to Attributed CFG. Our 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. 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). Our 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 in a program is small (tens of instructions). Phase marking at this granularity resulted in frequent core switches overshadowing any performance benefit. To avoid this, we use two techniques.

The first technique eliminates small sections of code. In other words, if the section has less than a threshold weight as defined by our 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, we still need to pick better points for phase marks.

The second technique further addresses this problem by only considering a section if at least a fixed percentage of its successors up to a fixed depth have the same type .

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 we look at enough successors, we start seeing the same nodes. Thus, if a loop contains predominately one type of blocks, we 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, we will reduce both the time and space overhead of the technique and hopefully not eliminate much of its benefit.

Adding Phase Marks to Attributed Interval Graphs. Our second class of methods consider a section to be an interval 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 1^{st} order interval graphs, the intervals frequently capture small loops. This is clearly advantageous for adding phase marks since we do not want to have a core switch within a small loop because this would most likely result in far

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

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.

Adding Phase Marks to Attributed Loop Graphs. Our 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 eliminating phase-marks within loop iterations. This an even more coarse view of

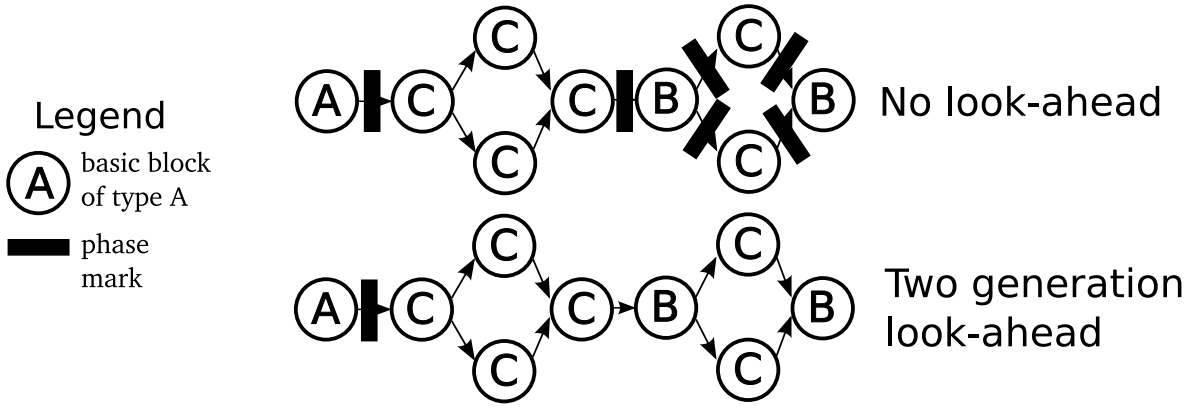

Figure 2.7 Lookahead based reduction of phase marks

the program than the interval based technique. Furthermore, since it is in inter-procedural analysis, transitions across function calls are handled. Just like interval typing, loop typing introduces some imprecision in the type information.

2.2 Dynamic Characteristics Analysis and Tuning

After phase transition marking is complete, we have a modified binary with 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 thread-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 for a phase mark serves two purposes: First, during a transition between different phase types, a core switch is initiated. The target for this switch is the core previously determined to be an optimal fit for this phase type. Second, if an optimal fit for a given phase type has not been determined previously, the current section is monitored to analyze its performance characteristics. The decision about the optimal 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, we do 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 our 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 our technique.

For analyzing the performance characteristics of a section, we measure instructions per cycle (IPC) (similar to [52, 7]). IPC directly correlates to throughput and utilization of performance-asymmetric multicore processors. For example, a core with a high clock frequency can efficiently process arithmetic instructions. However, if the core has a cache miss, it will waste cycles waiting for this data. A core with a lower frequency will waste fewer cycles waiting for data to be retrieved. If a section is being analyzed, its IPC is monitored using hardware performance counters prevalent in modern processors. The optimal core assignment is determined by comparing the observed IPC for each core type.

Our technique for determining optimal core assignment is shown in Figure 2.8. Underlying 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 does 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 Optimal core assignment for n cores

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

core. The intuition is that when the difference is above the threshold, we will save enough cycles to justify taking the space on the more efficient core. By performing the performance analysis at runtime, this algorithm for computing optimal core assignment *does not require knowledge of the program or underlying architecture* thus easing the burden on the programmer. It also *eliminates the need for an optimized version of the program for each target architecture* thus avoiding maintenance problems.

CHAPTER 3. Phase-Guided Tuning Framework

We developed a static analysis and instrumentation framework for phase detection and marking. Compared to similar static instrumentation tools such as Intel ATOM [1], binaries instrumented with our tool execute in less than one tenth of the time taken by binaries generated with ATOM[1]. The low overhead of instrumentation with our framework further reduces the overhead of our phase-marks. By instrumenting binaries rather than source code, we do not require any modifications to compilers. Our framework is based on the GNU Binutils. An overview of this framework and a breakdown of its components is shown in Figure 3.1. To perform core switches, we use the standard process affinity API available for Linux kernels (ver. $\geq$ 2.5). To dynamically monitor the performance of code sections, we use the Performance Application Programming Interface (PAPI) [14]. PAPI provides an interface to control and access information gathered by the processor hardware performance counters.

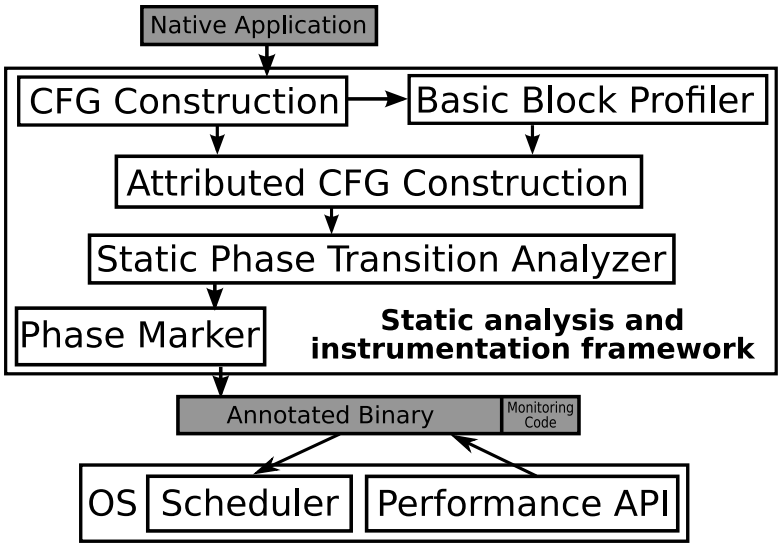

Figure 3.1 Framework components

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

To monitor the current section's performance we use two events made available by PAPI: instructions retired and cycles. These two events allow us to calculate the IPC (instructions retired / cycles) which we use to determine the actual runtime characteristics of representative phases. Unfortunately, many CPUs only make two performance counters available. With only two performance counters, if one program is monitoring its performance using PAPI, other programs that wish to monitor performance must wait. However, our approach requires very little dynamic monitoring. Additionally, performance is only monitored for a very small section of code. Therefore, processes seldom have to wait for the availability of the counters. In the event that a process must wait, the wait time is negligible since the amount of code needed to monitor is small. Because of this, performance is not impacted significantly by the need to wait for the counters to be available.

To determine phase types for the basic blocks to seed our static analysis, we use a profile of the basic block on any input. We then use the observed IPC to assign types to basic blocks. The *difference* in IPC between the core types is compared to an *IPC threshold* value to determine the clustering for code sections. There is room for improvement with respect to this technique, however, as Chapter 4 shows, our tool works quite well in practice.

CHAPTER 4. Evaluation

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

First, we show that phase-guided tuning has low overhead, second, that it 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, we compare the techniques and show how different variations are applicable for various scheduling goals.

4.1 Experimental Setup

This section describes our experimental setup including both hardware and software platforms.

4.1.1 Hardware Setup

Our setup consists of a performance-asymmetric multicore processor containing 4 cores. This setup is emulated using an Intel Core 2 Quad processor with a base clock frequency of 2.4GHz and two cores under-clocked to 1.6GHz. We use the Fedora distribution of Linux with an unmodified kernel and standard compilers. Thus, we demonstrate the transparent deployment benefit of our approach. We use the perfmon2 monitoring interface [16] to measure the throughput of entire workloads using pfmon.

There are two main benefits of using a physical system instead of a simulated system. First, porting our implementation to another system is trivial since we do not require any modifications to the standard Linux kernel. Second, we analyze our approach in a realistic setting. Others have argued that results gathered through simulation may be inaccurate if not carried out on a full system simulator [38]. 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, we believe this platform is sufficient to show the utility of our approach.

4.1.2 Workload Construction

Many systems receive a nearly constant feed of jobs to run [8]. 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. Our approach is targeting these systems, with maximizing throughput as its key objective.

Workloads range in size from 18 to 84 benchmarks in the SPEC CPU 2000 and 2006 benchmark suites. Since we want to improve throughput for systems that have a nearly constant feed of processes/requests (e.g. a server), we maintain a constant number of jobs in the workload. To achieve this, upon completion of a benchmark, another benchmark is immediately started. We determine the next process to start from a queue which is maintained for each workload slot. These queues are determined randomly. When comparing two techniques, the same queue was used for each experiment. This en-

sures that we more accurately capture the behavior of an actual system. If we were to simply restart the same benchmark upon completion, we may see the same benchmarks continuously completing if our technique favors a single type of benchmark.

4.2 Overhead

During our static analysis, we insert phase marks in the program to prepare it for phase-guided tuning. A phase mark consists of data and code. 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. Furthermore, a phase mark's execution time is added to the execution of the original program. If such execution time is high, it is likely to overshadow the gains achieved by our technique. Thus, a low time overhead is also desired. Therefore, the time overhead is described in terms of increase in execution time over the uninstrumented version.

4.2.1 Space Overhead

To measure the space overhead, we compared the size of the original binary and modified binary for several variations of our technique. 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. In this figure, the box represents the range of 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 and fairly clear from the figure. As the minimum size increases the space overhead decreases. Similarly, as the lookahead depth increases the 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 our 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

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.

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 our best technique (loop technique with minimum size of 45), we have less than 4% space overhead. For the same technique we have an average of 20.24 phase marks per benchmark where each phase mark is at most 78 bytes.

4.2.2 Time Overhead

To measure the time overhead (inserted phase marks and core switches), instead of switching to a specific core, we switch to "all cores" allowing the stock Linux scheduler to handle scheduling. Thus, the difference in runtime between the unmodified binary and this instrumented binary shows the cost of running our phase marks at the predetermined points in the program. Table 4.2 shows these costs for variable workload sizes. Figure 4.2 shows results for workloads of size 84.

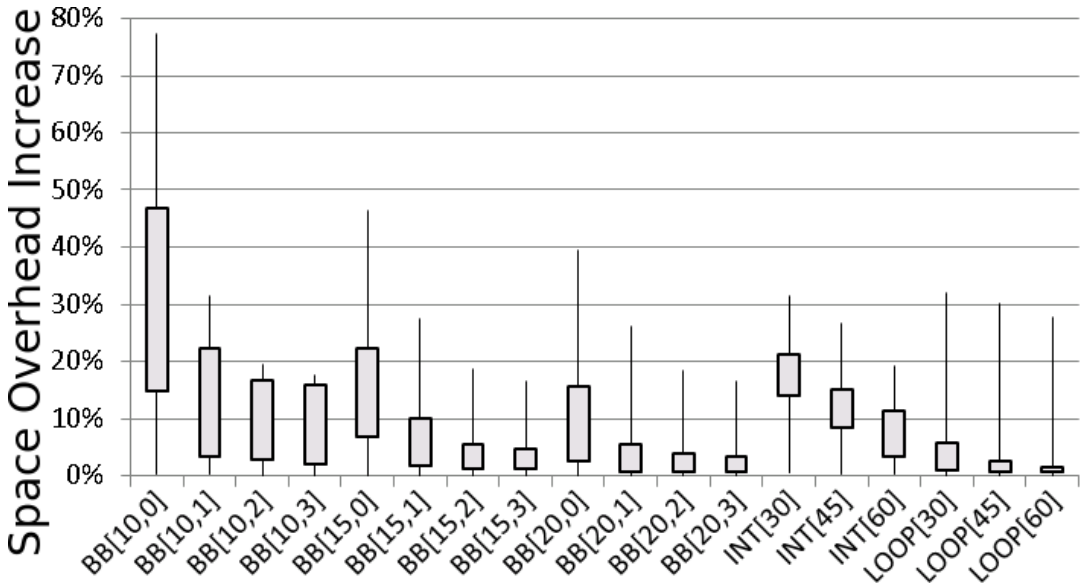

Figure 4.1 Space overhead

The trends shown are expected and are similar to those for space overhead. In some cases overhead was as little as 0.14%. More optimized instrumentation and core switching techniques are likely to decrease this overhead even further. Furthermore 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 our technique has a small overhead both in terms of space and time, which shows the *scalability* of our approach. For long running processes, the overheads are likely to decrease further. Since we only require a small number of blocks to be monitored at run-time, long running benchmarks will most likely have more time to take advantage of the thread-to-core assignment determined by our technique. 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.

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

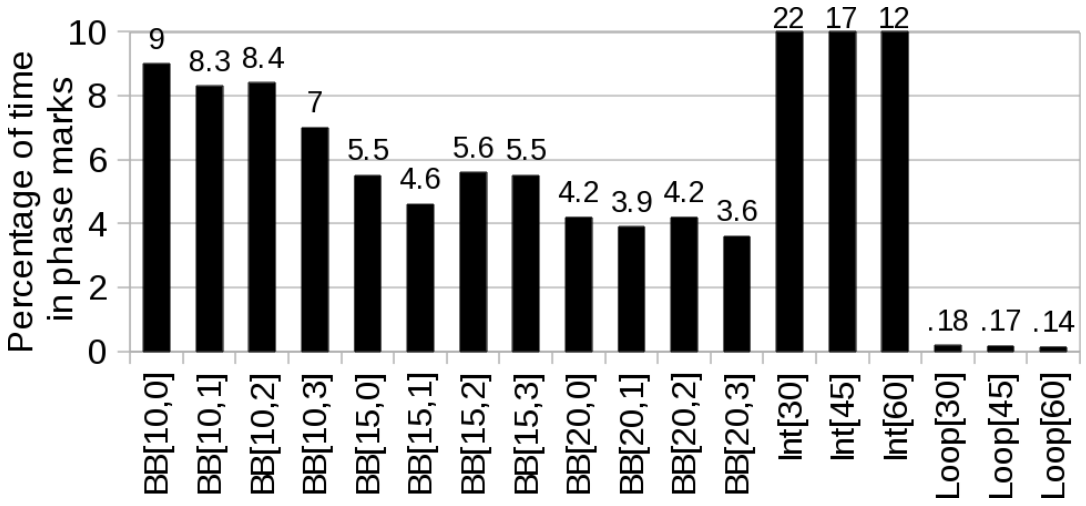

Figure 4.2 Time overhead: workload size 84

4.3 Throughput

To test our hypothesis that *"phase-guided tuning will significantly increase throughput"*, we compared our technique and 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). We measure how variations of our technique and variables in our algorithms affect throughput. 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.

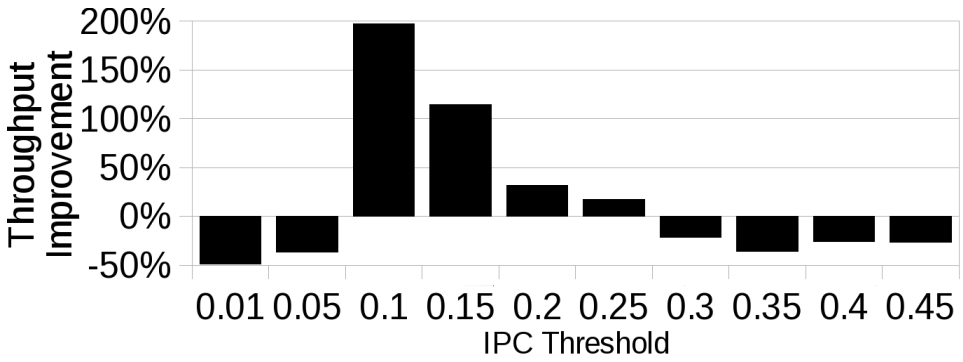

Figure 4.3 Throughput improvement: Basic block strategy, min. block size: 15, lookahead depth: 0, variable IPC threshold

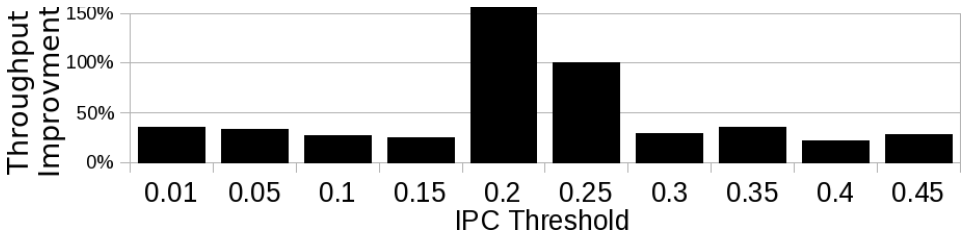

Figure 4.4 Throughput improvement: Interval strategy, minimum interval size: 45, variable IPC threshold

IPC threshold. First, we want to see how the IPC threshold affects throughput. As mentioned in Section 3, IPC threshold is used to determine the thread-to-core assignment. Figure 4.3 and 4.4 show how different threshold values affect throughput when all other variables are fixed (technique, min. size, lookahead, etc).

These results are expected. Extreme thresholds may show a degrade in throughput because the entire workload eventually migrates away from one core type. Between these extremes lies an optimal value. Near optimal thresholds result in a balanced assignment that assigns only well-suited code to the more efficient cores.

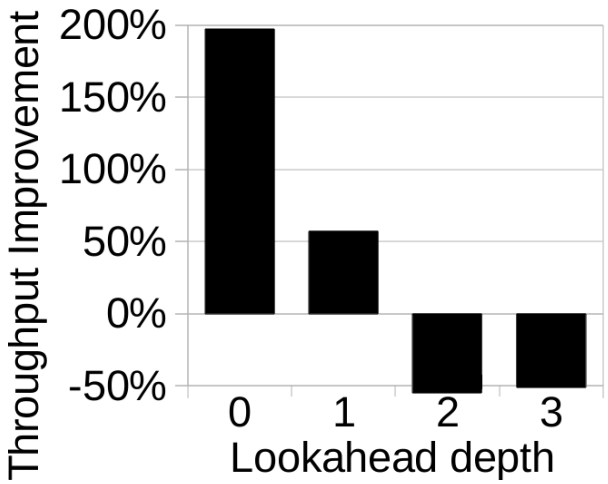

Figure 4.5 Throughput improvement: Basic block strategy, IPC threshold: 0.1, min. block size: 15, Variable lookahead

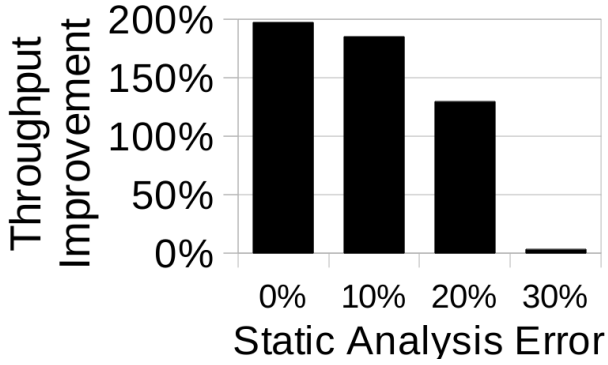

Figure 4.6 Throughput improvement: Basic block strategy, IPC threshold: 0.1, min. block size: 15, lookahead: 0, Variable error

Lookahead depth. Next, we examine the lookahead depth for the basic block technique. Figure 4.3 shows how lookahead affects throughput when other variables are fixed.

This data shows that lower lookaheads generally result in higher throughput. This is because when making decisions at a more fine granularity, the program code is closely matched to ideal cores through-

out more of the programs execution. However, this improvement is at the cost of increased overhead which was discussed in Section 4.2. Furthermore, for large lookahead depths, we can see a decrease in throughput. This is because when we begin to ignore large sections of execution; these ignored sections are frequently assigned in an inefficient way.

Clustering error. Since a static technique for determining similarity is likely to be inaccurate, Figure 4.3 shows how our technique performs with approximate phase information. Note that even when considering no static analysis error, our behavior information is not perfect since it only considers program behavior in isolation. We tested the same variables as Figure 4.3 but with error levels ranging from 0% to 30%. To introduce this error, after determining the clustering of blocks, a percentage of blocks were randomly selected and placed into the opposite cluster.

These results show that our technique is still quite effective even when presented with approximate block clustering. With a 10% error we see almost no loss in performance and with 20% error we still see a significant performance increase. At 30% error we see almost no performance improvement.

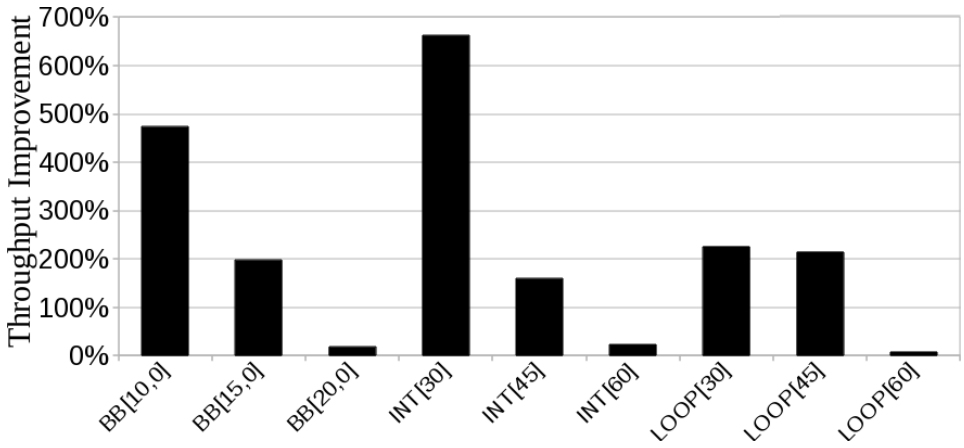

Figure 4.7 Throughput improvement: variable technique and minimum size

Minimum instruction size. Now, we examine how minimum instruction size affects throughput for all three techniques. Figure 4.7 shows this comparison. The results for minimum instruction size are similar to those for lookahead. Considering smaller blocks and intervals generally results in higher throughput. This is for the same reasons as lookahead depths, however, with larger minimum instruction size we may ignore small loops that are executed frequently. As mentioned previously, this improvement must be balanced with overhead costs which were discussed in Section 4.2.

Cache behavior. To help assess where these throughput benefits are coming from, we also measured the cache hit percentage for different variations of our technique. Variations showing an improvement in throughput did not show a statistically significant difference in cache-hit rate.

4.4 Fairness

Improved throughput is clearly advantageous. However, in many systems we desire fairness. Therefore, in this section we show the fairness for variations of our technique. First, the fairness metrics are described followed by the measurements and a brief discussion.

We use three measurements to determine 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 [8]. We now briefly define max-flow and max-stretch. For each process, we have the following data:

- 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

$$\max_j F_j$$

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. We consider this because we want processes to speed up, but not at the expense of others slowing down significantly. These measurements for our techniques are shown in Table 4.3.

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 4.3 Fairness comparison to standard Linux assignment: Improvements are shaded.

When trying to increase both throughput and fairness, our 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 a 800 second time interval using the loop based technique with minimum size of 45 and IPC threshold of 0.15. For these same variables, we see 215% improvement in throughput (as measured in Section 4.3).

4.5 Analysis of Trade-offs

We have shown that our technique 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 we discuss these trade-offs and how different variations of our technique perform with specific scheduling needs.

Speedup vs. Fairness. First, we examine the trade-off between speed and fairness. Speedup refers to the average process run-time. Max-stretch is used for fairness. We expect a positive correlation between the two with some exceptions. These exceptions occur when many processes finish quickly while few starve. Figure 4.8 shows this trade-off for different variations of our technique.

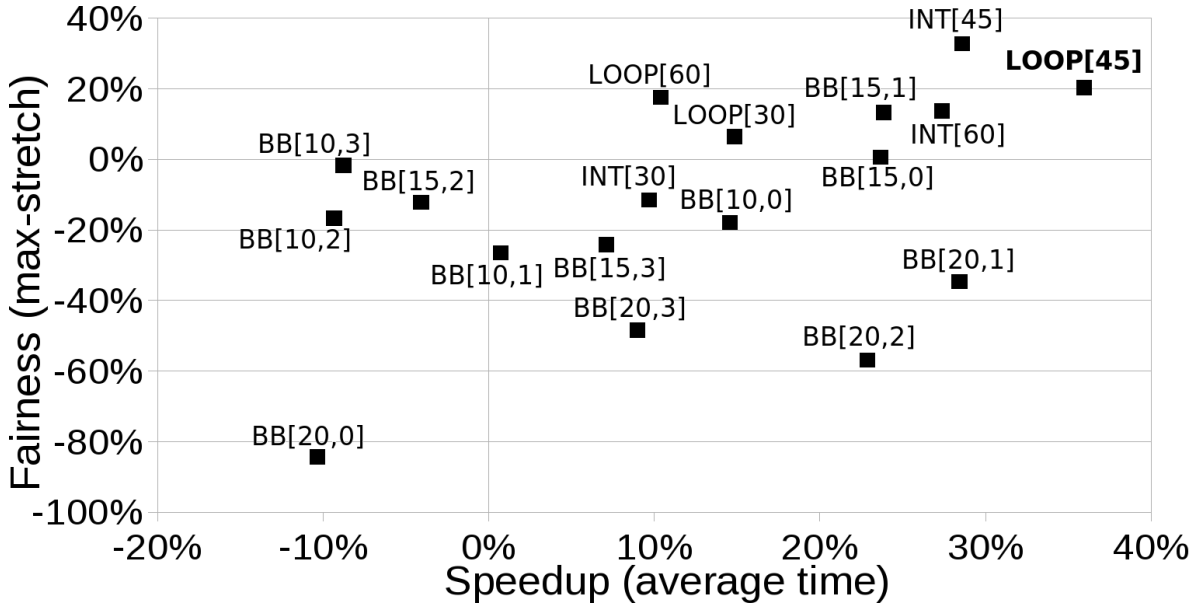

Figure 4.8 Speedup vs fairness: average time vs. max stretch

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

Throughput vs. Fairness. Next, we examine the trade-off between throughput and fairness.

This is important to consider since frequently the goal is to either maximize one of these or to find a reasonable balance. Intuitively, we expect a somewhat negative correlation between the two. This is because very high improvements in throughput are likely to be at the cost of some process starvation. Figure 4.9 shows this comparison.

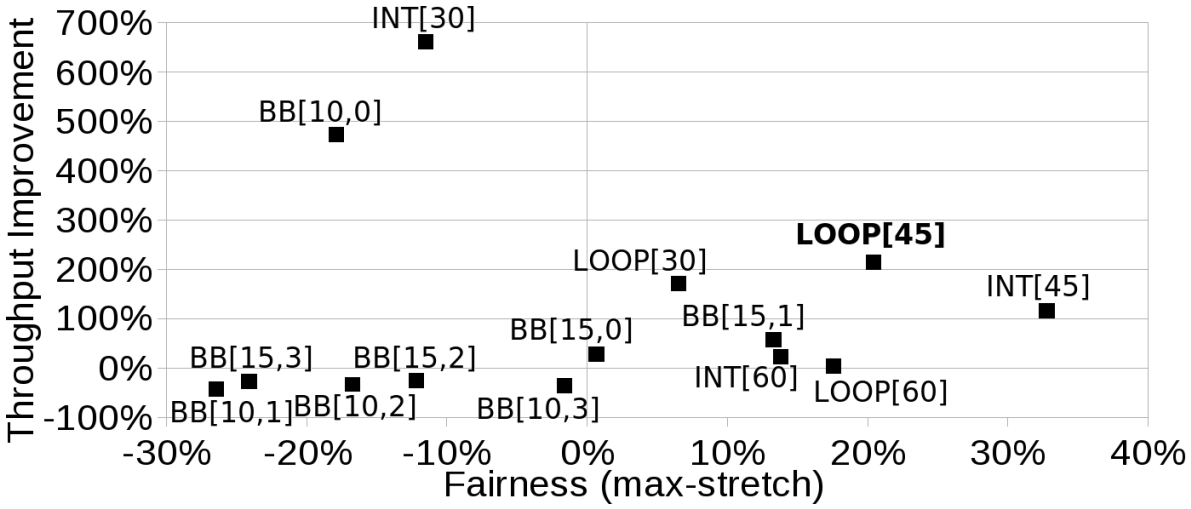

Figure 4.9 Throughput vs fairness (max-stretch)

These results show that we see throughput increase by as much as 662%, but at a significant cost in fairness. Variations of our technique exist that balance the two. For example our loop based technique improves throughput by 215% while improving fairness.

Speedup vs. Throughput. Now, we examine the trade-off between speedup and throughput. Since speedup is somewhat correlated with fairness, the trends are somewhat similar to the previous comparison of throughput and fairness. These two are different in that speedup is degraded by long starving processes while throughput is not affected by these. Figure 4.10 examines this trade-off.

From this figure we can make observations similar to the last comparison. Throughput can be improved significantly, but at the cost of average speedup. For example, fine grained techniques perform quite well at improving throughput, but degrade fairness significantly. Our interval and loop techniques again give a nice trade-off between the two by improving both throughput and speedup.

Summary of Results. In closing, our results show that phase-guided tuning significantly outperforms the stock Linux scheduler in terms of the throughput obtained on a performance-asymmetric

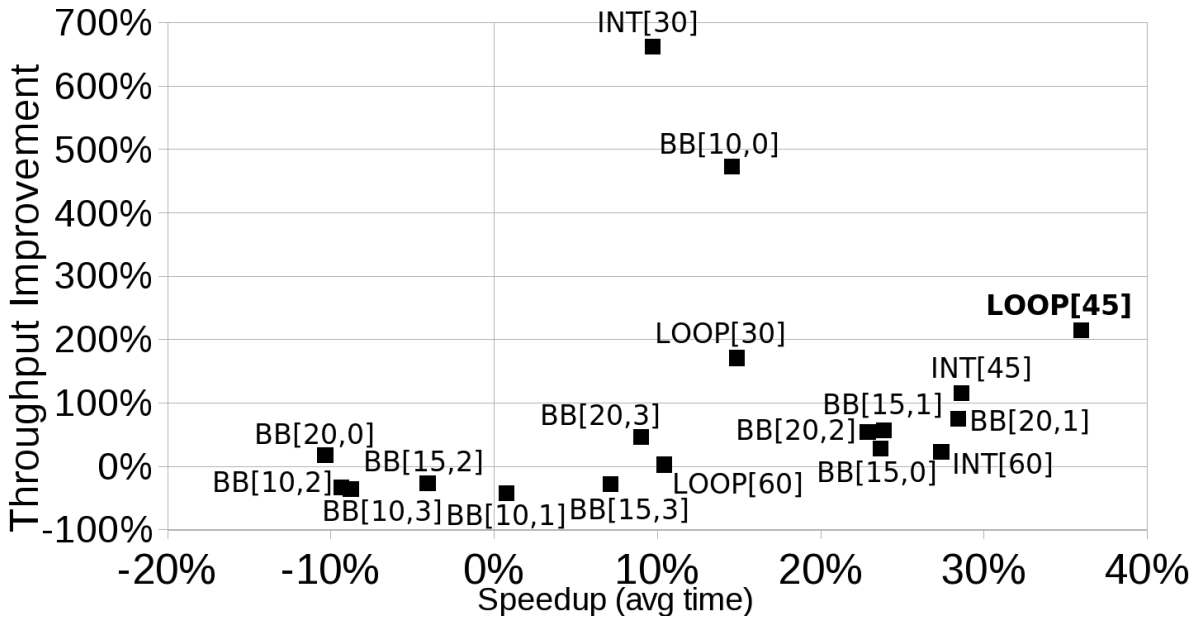

Figure 4.10 Speedup (avg. time) vs Throughput

multicore processor, while maintaining fairness and with a negligible overhead in most cases. In particular, our interval based technique balances throughput and fairness significantly well achieving improvements in throughput as high as 215% and an average process speedup of up to 36%. Our approach thus shows its potential in improving the utilization of performance-asymmetric multicore processors. With recent thrust towards research and development of these processors, the advances in thread-to-core assignment that we propose are timely and important.

CHAPTER 5. Related Work

A preliminary version of this work was presented in our IWMSE workshop paper [50]. This work only considered the static analysis at the basic block and intra-procedural first order interval techniques. We have since significantly enhanced our approach to be inter-procedural, to consider loops, and to handle loop nesting, Furthermore, we have considered the balance of fairness, speedup, and throughput as an important factor. This realization has shaped much of the analysis in this paper. Somewhat contrary to our preliminary results, after rigorous experiments with much larger data set it turned out that the interval-level techniques were quite superior to the basic block techniques. This further led to the development of the loop-level inter-procedural static analysis.

Besides our previous work, we know of no previous work which has all of the benefits of our technique. There does exist previous work that are related to ours in spirit. This work may be divided into three 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. Third, those related to determining phase behavior and those which use phase behavior for dynamic optimizations.

Becchi *et al.* [7] proposed a dynamic assignment technique that uses the IPC of a program segments. However, this work focuses largely on ensuring load balance across cores whereas our technique aims to maximize throughput. Another technique which focuses on load balancing in the scheduler was proposed by Fedorova *et al.* [17]. They make the case that a core assignment must be balanced. Shelepov and Fedorova [46] propose a technique which does not require dynamic monitoring. They estimate cache misses in order to estimate the run-time difference between core types in their architecture. This technique does not consider the changes in behavior a program will make throughout its execution. Li *et al.* [35] also 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 will need some knowledge of the underlying architecture. Our work differs from these in the following way. First, we are not directly concerned with load balancing. Second, we focus on properly scheduling the different phases of a programs behavior. Also similar is the work by Tam *et al.* [52] which determines 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 [28]. After certain time intervals, a sampling phase is triggered. After the sampling phase, the system makes a decision regarding the assignment of all currently executing processes. This procedure is carried out throughout the entire programs execution. To reduce the dynamic overhead, we do not require monitoring once assignment decisions have been made.

A wide range of research has been done on algorithms for optimal distribution of computation over heterogeneous networks of computers. 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 [6, 30, 45] , and techniques that allow programmers to specify the program's characteristics and use this input to obtain an optimal distribution [29, 31]. Instead of working with a heterogeneous network of computers, we focus 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. Our technique eliminates this burden.

There is a large body of work on determining phase behavior [49, 15, 51], using phase behavior to reduce simulation time [5, 12, 48, 19, 51, 47, 34], guide optimizations [24, 22, 42, 39, 43, 44, 54], etc. Most of these techniques determine phase information with a previously generated dynamic profile. As mentioned previously, collecting a this profile requires end-users to develop representative sets of test cases for the program. Techniques that determine the phase information dynamically do not require this input, however, they are likely to incur dynamic overheads. We conduct much of our analysis statically followed by limited analysis dynamically.

CHAPTER 6. Future Work

Future work involves extending phase-guided tuning and its underlying analysis techniques in several directions. This includes three major directions: static approximate phase analysis, improved phase transition marking, and improved dynamic analysis and tuning. There are a few minor enhancements to explore as well. Also, future work involves investigating other optimizations which may benefit from a similar analysis framework.

6.1 Phase-Guided Tuning

There are several directions in which phase-guided tuning may be improved. Here, a few are highlighted that are the most interesting and are concrete plans for immediate future work.

6.1.1 Static Approximate Phase Analysis

The first and most interesting direction of future work is to explore and evaluate techniques for determining approximate similarity between behavior of code segments. Since our technique relies on such an analysis to function well, the development and accuracy of a similarity analysis technique is crucial.

As mentioned in Chapter 5 there is a large body of work on determining program behavior [49, 15, 51]. Again, these techniques either determine phase information using previously generated dynamic profile or determine the phase information at run-time. While accurate, techniques based on profiles require representative sets of inputs for the program and may not be accurate for other inputs. Techniques that determine the phase information dynamically do not require this input, however, they have a runtime overhead and require segments to be executed at least once before we can make decisions regarding its behavior.

Recall that our goal is to predict similarity between behavior regardless of the underlying architecture. The reason for this is to support the idea of "tune once run anywhere" discussed in Chapter 2. To recall, "tune once run anywhere" means that we can use the same version of the optimized binary on any system regardless of its asymmetry. Hoste *et al.* [23] developed a set of metrics called MICA to characterize the behavior of a workload independently of the underlying architecture. The result is a predicted similarity between programs that will hold on any machine. The limitation is that this technique requires dynamic execution traces to determine the values for the metrics.

Thus, the goal for future work is to determine a similar set of metrics which we can approximate statically. Then, using these metrics we can predict similarity between program segments. Potential metrics include but are not limited to[1]:

- Instruction type: the mix of instruction types present in a code segment. For example, floating point operations versus integer operations or percentage of slow instructions (ex: division).
- Register usage: Even though general purpose registers may be used for a wide range of uses, registers typically have a specific use (ex: counters, return values, etc).
- Cache behavior: Memory accesses are significantly slower than cache accesses. Thus, predicting cache performance is crucial. Furthermore, with many processors, memory access becomes a bottleneck [10, 37]. If we can predict the number of memory (or higher-level cache) accesses for a code block based on a variety of cache sizes, we can then group blocks with similar behavior for different cache sizes.
- Branch prediction: Branch miss-prediction can be very expensive (more than 100 cycles on average for our Quad core machine). Thus, the accuracy of hardware branch predictors will have a large impact on performance. Therefore, it is worthwhile to consider how easily the branch predictor can make decisions regarding branch prediction.
- Instruction level parallelism: The degree of instruction level parallelism a block may exhibit is another factor which will impact performance. To predict this, we need to know dependencies

[1] Many similar metrics are used in MICA [23]

between instructions which may be difficult or impossible to determine statically. For example, the memory locations created by memory allocation may not be known until run-time.

- Heap shape transformations: Another hypothesis is that the data access and update patterns of a code segment will give insights into behavior. For example, a search through a linked list may have similar behavior to an insert at a specific location.

Like the results of MICA show, combining all of these metrics is likely to give us some indication of run-time behavior.

6.1.2 Phase Transition Marking

For a thread, switching cores is highly expensive. Thus, the performance of phase-guided tuning will only be effective if we switch cores infrequently. Therefore, we have to very carefully consider the improvements achieved by switching cores at a given point and the cost of the core switch. Too frequent core switching will result in performance loss whereas too infrequent core switching will result in poorly assigned code segments.

In this thesis, several techniques for determining phase transitions were presented. All of which tried to achieve the same goal: only switch to an optimal core for large code sections such that the benefits outweigh the costs of switching. The problem is that these techniques only looked at the control flow graph. Consider the loop analysis. Using our current techniques, a loop that executes twice is considered the same as a loop that will execute thousands of times. It may be worthwhile to switch cores if we are going to have thousands of iterations, but it is intuitive that two small loop iterations are not enough to justify a core switch. Our current technique only considers loops that have more than some number of instructions. This works in some cases, but suppose we have a 20 instruction loop that executes one thousand times and a 100 instruction loop that executes twice. The smaller loop will run 20,000 instructions whereas the bigger one will run 200 instructions. Clearly we are more likely to see benefits from tuning the smaller loop that iterates more times.

To address this problem, the current analysis techniques may be combined with ideas from worst case execution time (WCET) [55, 36] and symbolic bound computation [21]. These techniques will allow us to either predict bounds on the number of iterations a loop may have or in some cases the

exact number of iterations. Using this technique, we can relax our previous restrictions on size and focus more on the time spent in the loop.

6.1.3 Dynamic Tuning

Recall that our key insight for reducing the dynamic overhead is to only require monitoring a small number of code segments and then using these results to schedule all future blocks. This works well for many cases, however, it is well known that within a program there are "warm-up" phases [49]. If our program makes its decision while in such a phase, the assignment may not be valid for other phases. Typically this is not an issue since these phases also involve different code segments being executed. For example, we may take some input, then process it. Sometimes, however, these warm-up phases happen when we take different paths through a loop. For example, the first iteration through a loop may involve allocating memory. Thus, it is likely worthwhile to consider such cases. Furthermore, the characteristics of a core are likely to change based on the current workload [32]. For example, a thread may be executing with few cache misses. Then, another thread may enter the same core that uses much of the cache space. Then, the first thread will probably begin to have cache misses. This results in the same code having different behavior at different times. Therefore, for a long running program, an assignment may become poor as other programs change their resource usage. Clearly these cases need to be considered in practice.

To capture such cases, we plan to implement a feedback mechanism which periodically measures the effectiveness of an assignment strategy to determine if it is still a good assignment. This mechanism will allow is to alter the assignment as the workload characteristics change. Also, it will help avoid poor assignments which may arise due to warm-up stages.

Furthermore, the current scheduling technique makes almost purely local decisions. Another plan is to extend the thread-to-core assignment technique to consider global (system) optimization and potential thread interference.

6.1.4 Minor Enhancements

Minor improvements to phase-guided tuning include the following. Tighter integration with the schedule to reduce the cost of core switching. Streamlining usage of the performance counters for performance monitoring.

6.2 Other Optimizations

Finally, the ideas presented in this paper are a single use case for phase-guided tuning. Future work involves generalizing this framework for a wide range of optimizations. Furthermore, several such optimizations will be explored and implemented to show the effectiveness of a general framework for phase-guided tuning.

For example, cache optimizations may perform differently in the presence of different cache hierarchies. The problem then is to choose a subset of representative versions of the optimized code. Each of these versions is put into the optimized binary. Then, at runtime, we can determine which version is best for the system, the specific core, or the current workload state.

CHAPTER 7. Conclusion

Performance-asymmetric multicore architectures 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 [20, 27, 40]. Devising techniques for their effective utilization is an important software engineering problem that influences the eventual uptake of this class of processors [35, 40]. The need to be aware of, and optimize based on, the applications' characteristics and the nature of multicore processor significantly increases the intellectual burden on the software developer. Furthermore, the need to create separate versions for each target architecture decreases reusability and creates a maintenance burden. In this work, we have shown that phase-guided tuning solves all of these problems by utilizing the phase behavior that is common in programs. Phase-guided 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 software engineering benefits, phase-guided tuning also has several performance advantages. Our experiments show that improvements in throughput by up to 215% with a 36% reduction in the average process time when compared to the stock Linux scheduler. This is all done while incurring negligible overheads (less than 0.2% time overhead) and maintaining fairness.

BIBLIOGRAPHY

[1] *Intel Analysis Tools for Object Modification (Intel Atom): Release Notes: Release 1.0 Beta.*

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF*, pages 29–40, 2006.

[8] 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.

[9] S. Y. Borkar. Platform 2015: Intel processor and platform evolution for the next decade. *Tech. Report - Intel*, 2005.

[10] L. Chai, Q. Gao, and D. K. Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 471–478, Washington, DC, USA, 2007. IEEE Computer Society.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] S. Eranian. permon2: a flexible performance monitoring interface for linux. In *Ottawa Linux Symposium (OLS)*, 2006.

[17] A. Fedorova, D. Vengerov, and D. Doucette. Operating system scheduling on heterogeneous core systems. In *First Workshop on Operating System Support for Heterogeneous Multicore Architectures*, 2007.

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

[19] 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.

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

[21] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL '09: Conference record of the 36th symposium on Principles of programming languages*, pages 127–139, 2009.

[22] 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.

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

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

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

[26] 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.

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

[28] 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.

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

[30] A. Lastovetsky and R. Reddy. Data partitioning with a realistic performance model of networks of heterogeneous computers. April 2004.

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

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

[33] 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.

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

[35] 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.

[36] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim. An accurate worst case timing analysis for risc processors. *IEEE Trans. Softw. Eng.*, 21(7):593–604, 1995.

[37] L. Liu, Z. Li, and A. H. Sameh. Analyzing memory access intensity in parallel programs on multicore. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 359–367, New York, NY, USA, 2008. ACM.

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

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] H. Renard, Y. Robert, and F. Vivien. Static load-balancing techniques for iterative computations on heterogeneous clusters. *The 9th International European Conference on Parallel and Distributed Computing (Euro-Par 2009)*, pages 148–159, 2003.

[46] D. Shelepov and A. Fedorova. Scheduling on heterogeneous multicore processors using architectural signatures. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA-35*, 2008.

[47] 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.

[48] 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.

[49] 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.

[50] T. Sondag and H. Rajan. Phase-guided thread-to-core assignment for improved utilization of performance- asymmetric multi-core processors. In *2nd International Workshop on Multicore Software Engineering*, May 2009.

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

[52] 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.

[53] W. Tichy. The multicore software challenge. In *Proceedings of the 31st International Conference on Software Engineering, ICSE*, 2009.

[54] 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.

[55] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.