

Phase-guided Auto-Tuning for Improved Utilization of Performance-Asymmetric Multicore Processors

Tyler Sondag and Hridesh Rajan

TR #08-14a

Initial Submission: March 16, 2009.

Keywords: static program analysis, heterogeneous multicore processors, thread-to-core assignment, phase behavior, performance asymmetry

CR Categories:

D.3.4 [*Programming Languages*] Processors - Optimization D.3.3 [*Programming Languages*] Language Constructs and Features - Control structures D.4.1 [*Operating Systems*] Process Management - Multiprocessing/multiprogramming/multitasking, Scheduling, Threads D.2.7 [*Software Engineering*] Enhancement D.2.11 [*Software Engineering*] Information Hiding

Submitted.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Phase-guided Auto-Tuning for Improved Utilization of Performance-Asymmetric Multicore Processors

Tyler Sondag

Dept. of Computer Science, Iowa State University
sondag@cs.iastate.edu

Hridesh Rajan

Dept. of Computer Science, Iowa State University
hridesh@cs.iastate.edu

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 analysis technique, which we call *phase-guided auto-tuning*, to solve this problem. Phase-guided auto-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 116% improvement in throughput and 29% average process speedup, while maintaining fairness and with negligible overheads.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors - Optimization; D.3.3 [Programming Languages]: Language Constructs and Features - Control structures; D.4.1 [Operating Systems]: Process Management - Multiprocessing/multiprogramming/multitasking, Scheduling, Threads; D.2.7 [Software Engineering]: Enhancement; D.2.11 [Software Engineering]: Information Hiding

General Terms Algorithms, Experimentation, Performance

Keywords static program analysis, heterogeneous multicore processors, thread-to-core assignment, phase behavior, performance asymmetry

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]... \$5.00.

1. Introduction

CPUs with multiple cores have become commodity items [14]. CPU vendors are projecting that in the next decade the number of cores in a CPU will increase to as many as hundreds [8]. 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 [4, 5, 15, 18, 21, 24]. 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 [15, 18, 32]. These architectures have been shown to provide an effective trade-off between performance, die area, and power consumption compared to homogeneous multicore processors [15, 18, 21, 24].

To effectively utilize performance-asymmetric multicore processors, application threads must be executed on cores such that the resource requirements of a thread closely match the resources provided by the core [20, 24]. 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 creates a maintenance problem. The performance asymmetry present in the target multicore processor may not be known during development, which further complicates manual tuning.

Contributions: The main technical contribution of this work is a novel technique, which we call *phase-guided auto-tuning*, for matching the resource requirements of a thread to the resources provided by the cores of a performance asymmetric multicore processor¹. Phase-guided auto-tuning builds on a well-known insight that programs exhibit phase behavior [16, 17, 23, 28, 34, 37, 40]. By phase behavior we mean that a program goes through phases of execution that show similar runtime characteristics compared to other phases [2, 9–11, 31, 38]. Based on this insight, our approach consists of two parts. A static program analysis, which identifies likely *phase-transition points* in a program, 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

¹ A preliminary version of this idea was proposed in our IWMSE workshop paper [36]. Here, we fully develop and evaluate the idea.

phase-transition point is instrumented with a tiny fragment for dynamic analysis. Phase-guided auto-tuning has following 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.
- **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 versions².
- **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.
- **Improved Utilization:** It significantly improves the utilization of performance-asymmetric multicore processors while maintaining fairness between applications.
- **Negligible Overhead:** It incurs low space and time overheads, i.e. it is also useful for overhead conscious software.

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

To evaluate the effectiveness of phase-guided auto-tuning, we applied it to workloads constructed from the SPEC CPU2000 suite, a standard benchmark 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 116% improvement in the utilization of our performance-asymmetric processor setup while maintaining comparable fairness and negligible overheads. Additionally, on an average processes were as much as 29% faster without causing process starvation.

2. Phase-guided Auto-tuning

A program exhibits phase behavior [2, 9–11, 31, 38] 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 auto-tuning works as follows. First, 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 [3] 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*³. A phase mark contains information about the phase type for the current section, code for dynamic performance analysis, and code for making core

² As usual, a separate version is needed for different instruction sets.

³ The idea of phase marking is similar to the work by Lau *et al.* [22], however, we do not use a program trace to determine our phase marks and make our selections based on a different criteria.

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 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. 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. The rest of this section describes 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 entire program and to classify them into one or more phase types that are likely to exhibit similar runtime behavior. We then do one of two different analysis to detect and mark phase transitions with *phase marks*. The first analysis is a basic block level analysis. The second builds upon this basic block analysis to analyze intervals [3].

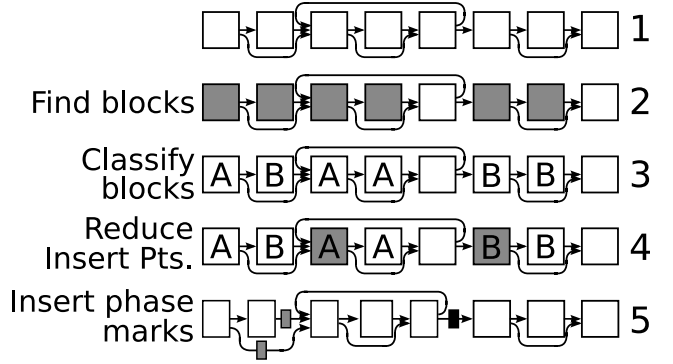


Figure 1. Overview of Phase Transition Analysis

Figure 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 look-ahead, 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 with types for both of our techniques. Next, we discuss how to use the annotated control-flow graphs to perform the phase transition marking.

2.1.1 Static CFG Annotation

We now discuss the two 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.

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}) [3]. 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 [3]. 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 [3] 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 [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 , 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 Algorithm 1 and illustrated in Figure 2.

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 $M \oplus \{\pi \mapsto M(\pi) + w_f * \varphi(\eta)\}$. 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

Algorithm 1 : Interval Summarization to Find Dominant Type

```

 $\rho = \{\}$ 
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  $\max(\text{dom}(M))$ 
end for

```

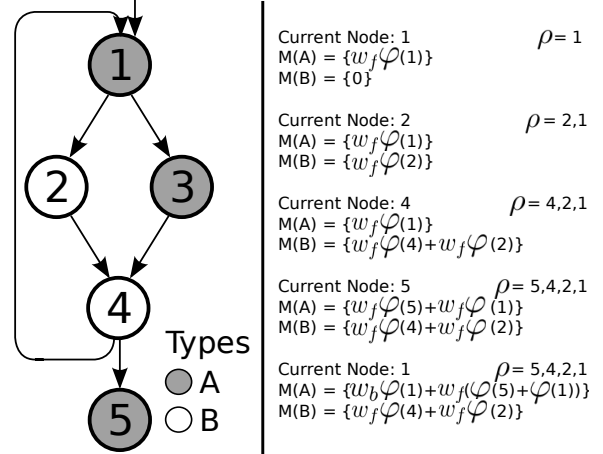


Figure 2. Interval Summarization Illustration

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 $\# \pi'. M(\pi') > M(\pi)$. 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 [3], however, in this paper we only consider first-order intervals. Our initial intuition is that the value of applying n^{th} order interval summarization will depend on the average size of procedures.

The special nodes (\mathcal{S}) in the CFG deserve some discussion. In our formulation, there are two types of special nodes: system calls and procedure calls. The type for system calls is dependent on the target operating environment determined by the combination of the performance-asymmetric multicore processor and the operating system. This is determined once, and the analysis framework parameterized with that data. There could be two types of procedure calls: to the procedures in the shared library and other procedures in the program. The procedures in the shared library are treated similarly to the system calls.

To tackle procedures in the program, a bottom-up approach is applied (lowest layer procedures first). In case of mutually recursive procedures, the cycle in the analysis is broken by randomly assigning a type for one procedure and analyzing the rest until a fixed-point is reached.

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. We have considered several variations of phase transition marking that can be broadly classified into two kinds based on whether it operates on the attributed control-flow graphs or the attributed interval graphs. In both cases, phase marks are placed at the beginning of a section.

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 β 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 .

Look-ahead based Phase Marking This technique is presented in Algorithm 2 and illustrated in Figure 3. 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.

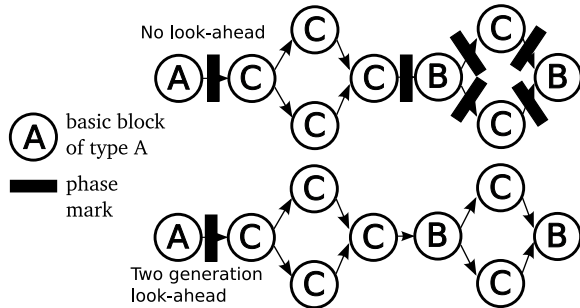


Figure 3. look-ahead based reduction of phase marks

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 easily look at the program at a more coarse granularity than basic blocks. It is also important to notice that even with 1st 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 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.

Algorithm 2 : Look-ahead based Phase Marking

```

Processed Nodes,  $\mathcal{D}$ 
Get Successors to Depth,  $\mathcal{S} : (\eta, \mathbb{N}) \rightarrow \{\bar{\mathcal{B}}\}$ 
Look-ahead depth:  $d$ , Successor threshold:  $e$ 
Same type count:  $c$ , Total count:  $t$ 
Grouping  $\mathcal{U} = \{\pi \mapsto N \mid \forall \pi \in \Pi\}$ 
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

```

2.2 Performance Analysis and Scheduling

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 that was 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. 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.

For analyzing the performance characteristics of a section, we use instructions per cycle (IPC) as a metric (similar to [6, 39]). IPC directly correlates to throughput and improved utilization of performance-asymmetric multicore processors. For example, a core with a high clock frequency can efficiently process arithmetic instructions. However, if the core must load a large amount of data from memory not already in cache, it will waste many cycles waiting for this data. Whereas a core with a lower clock frequency will waste fewer cycles when 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 algorithm for computing optimal core assignment does not require knowledge of the underlying architecture. The intuition behind this algorithm is that cores which execute code most efficiently will waste fewer clock cycles resulting in higher observed IPC. Therefore, these cores will be in highest contention. So, if the dif-

ference in observed IPC between two cores is above the threshold, we assume that we will save a large enough number of cycles to make it worth executing on the more efficient core. This algorithm is shown in detail in Algorithm 3.

Algorithm 3 Optimal Core Assignment for n Cores

Procedure SelectCore(π, δ): best available core for phase of type π with the threshold set to δ .

```

 $C := \{c_0, c_1, \dots, c_n\}$  (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  $\pi$  on core  $c_i$ .
 $d \leftarrow c_0$ 
for all  $c_i \in C \setminus \{c_n\}$  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  $d$ 

```

3. Auto-Tuning Framework

We developed a static analysis and instrumentation framework for phase detection and marking. This framework is based on the GNU Binutils. An overview of this framework and a breakdown of its components is shown in Figure 4. To perform core switches, we use the standard process affinity API available for Linux kernels (ver. ≥ 2.5). To dynamically monitor the performance of code sections, we use the Performance Application Programming Interface (PAPI) [19]. PAPI provides an interface to control and access information gathered by the processor hardware performance counters.

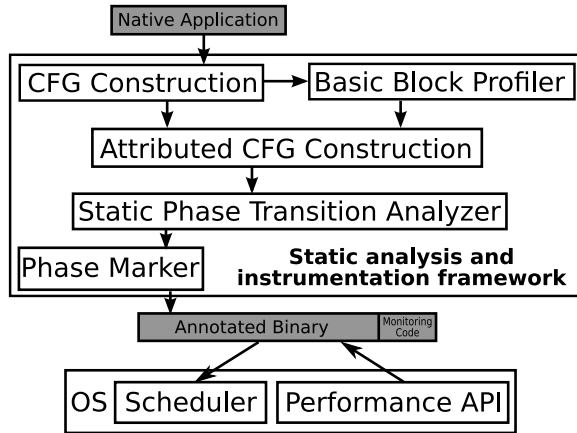


Figure 4. Framework Components

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. 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 Section 4 shows, our tool works quite well in practice.

4. Evaluation

The aim of this section is to evaluate our three claims. First, that phase-guided auto-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.

Hardware. 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. We use the perfmon2 monitoring interface [13] 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 [27]. 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.

Workload Construction Many systems receive a nearly constant feed of jobs to run [7]. 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 CPU2000 benchmark suite. 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 constant workload size, 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 and the same queues were used for all tests. This ensures 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.

For example, for a workload of size 18, we generate 18 random queues. Each queue contains a random list of benchmarks. At the beginning of a workload, the first benchmark in each queue is started. Once a process is completed, the next process in the queue is started. We ensure that the queues are long enough such that no queue is ever completed.

4.2 Space and Time Overhead

To measure the overhead of our approach, we consider both the binary size of instrumented applications and the extra runtime our inserted code introduces.

During our static analysis, we insert phase marks in the original binary to prepare it for phase-guided auto-tuning. A phase mark consists of data as well as code. Since insertion of large chunks of code may destroy locality in the instruction cache, a low space overhead is desired. This section first describes the overhead in terms of the increase in binary size that is 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 undesirably 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.

Space Overhead To measure the space overhead, we compared the size of the original binary and modified binary was performed for several variations of our technique. Table 1 shows summary statistics for the measurements taken from the benchmarks in the SPEC2000 benchmark suite. These results are presented in terms of what percentage of the instrumented application is made up of phase marks. All variations of our technique are not shown due to space constraints, however, the trends are expected and fairly clear from the table. As the minimum size of basic blocks considered increases the space overhead decreases. Similarly, as the look-ahead depth increases the space overhead usually decreases. This is not always the case because by adding another depth of look-ahead, the percentage of blocks belonging to the same type may be pushed over the threshold causing another insertion point.

Technique	Space overhead of phase marks (in %)			
	Average	Minimum	Maximum	Std. Dev
BB[10,0]	8.94	2.42	49.61	19.06
BB[10,1]	6.00	1.54	45.91	17.33
BB[10,2]	5.15	1.33	44.67	16.98
BB[10,3]	5.02	1.45	40.51	15.12
BB[15,0]	3.72	0.63	29.10	13.04
BB[15,1]	2.68	0.56	25.31	10.54
BB[15,2]	2.48	0.42	23.54	10.27
BB[15,3]	2.53	0.49	23.54	9.78
BB[20,0]	1.93	0.40	23.54	10.10
BB[20,1]	1.37	0.40	20.39	8.11
BB[20,2]	1.24	0.28	19.06	7.61
BB[20,3]	1.30	0.29	19.06	7.51
Int[30]	28.33	4.08	32.75	8.16
Int[45]	19.73	2.01	24.51	6.04
Int[60]	14.84	1.18	20.46	4.82

Table 1. Space overhead of phase marks: BB[n,m]: basic block technique with min. block size: n , look-ahead: m . Int[n]: interval technique with min. interval size: n .

These results confirmed our intuition that less phase marks will be inserted for larger basic block sizes and look-ahead 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.

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

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

Table 2. Time spent in phase marks

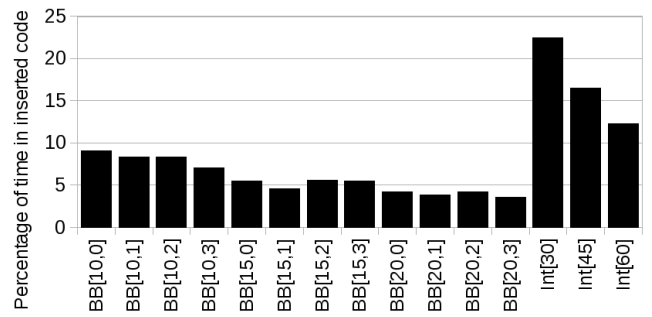


Figure 5. Time overhead: workload size 84

our phase marks at the predetermined points in the program. Table 2 shows these costs for variable workload sizes. Figure 5 shows results for workloads of size 84.

The trends shown are expected and are similar to those for space overhead. In some cases overhead was as little as 3.56%. 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.

4.3 Throughput

To test our hypothesis that “*phase-guided auto-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 want to see how different 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

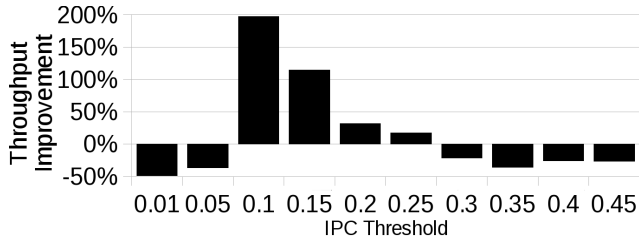


Figure 6. Throughput improvement: Basic block strategy, min. block size: 15, look-ahead depth: 0, variable IPC threshold

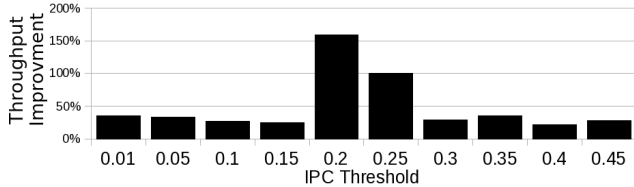


Figure 7. Throughput improvement: Interval strategy, minimum interval size: 45, variable IPC threshold

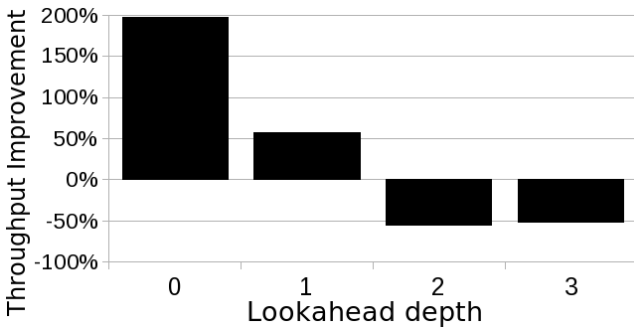


Figure 8. Throughput improvement: Basic block strategy, IPC threshold: 0.1, min. block size: 15, variable look-ahead depth

section, the data is taken from the first 400 seconds of the workload execution.

IPC threshold First, we want to see how the IPC threshold affects throughput. As mentioned previously in Section 3, IPC threshold is used to determine the thread-to-core assignment. Figures 6 and 7 show how different threshold values affect throughput when all other variables are fixed (technique, block size, look-ahead, 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.

Look-ahead depth Next, we examine the look-ahead depth for the basic block technique. Figure 8 shows how look-ahead affects throughput when other variables are fixed.

This data shows that lower look-aheads 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 throughout 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.

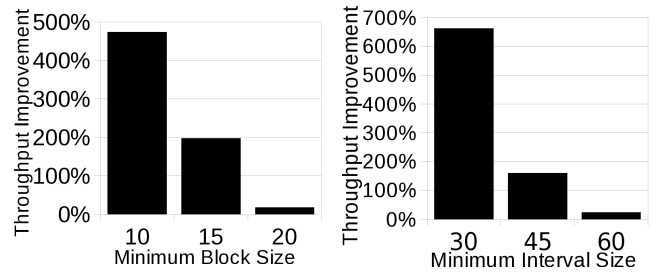


Figure 9. Throughput improvement: variable minimum instruction size - Left: Basic Block, Right: Interval

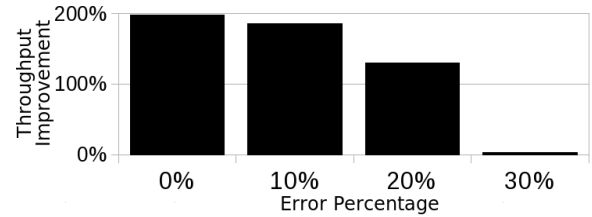


Figure 10. Throughput improvement: Basic block strategy, IPC threshold: 0.1, min. block size: 15, look-ahead depth: 0, variable error

Minimum instruction size Now, we examine how minimum instruction size affects throughput for both block and interval techniques. Figure 9 shows this comparison. The left side illustrates the results for the basic block technique and the right side shows the results for our interval technique.

The results for minimum instruction size are similar to those for look-ahead. Considering smaller blocks and intervals generally results in higher throughput. This is for the same reasons as look-ahead 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.

Clustering error Since a static technique for determining similarity is likely to be inaccurate, Figure 10 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 6 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. In some cases, the throughput actually increases over the technique with no error. Since our technique ignores parts of each programs code, this imprecision can sometimes improve upon our assignment by handling these ignored sections in a better way. With extreme thresholds, the error moves processes away from overloaded cores and improves throughput.

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.

Basic Block vs. Interval Techniques Finally, it is important to consider how our basic block based and interval based techniques compare. Figures 6 and 7 suggest that the two techniques may show comparable improvement. However, as shown in Section 4.2, for

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

Table 3. Comparison to standard Linux assignment: Improvements are shaded.

these same variables, the interval technique has significantly higher overhead. Furthermore, Figure 9 shows that when increasing the minimum interval size to decrease this overhead, we quickly lose much of our benefit. These performance differences are largely because of the inaccuracy in determining interval types. This inaccuracy is due to that fact that we do not know statically which path through the interval will be taken at runtime.

4.4 Fairness

Improved throughput is clearly advantageous. However, in many systems we desire some amount of 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.

Bender *et al.* proposed two metrics for continuous job streams [7]. Both of these metrics were designed to overcome the fact that makespan and average completion time are not suitable for jobs arriving continuously [7]. A brief summary of these metrics follows. For each process, we have the following data: a_i : arrival time of process i , C_i : completion time of process i , 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. So, in order to measure the fairness of our technique, we measure the following:

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

We believe that this additional point, average process time, is also important to consider. For example, we may have a technique that reduces the average completion time. However, if it significantly increases max-flow, the technique is highly unfair. Furthermore, we may have a technique which increases both average time and max-flow. This means that the overheads are most likely overshadowing the benefits. These results are shown in Table 3.

When trying to increase both throughput and fairness, our technique shows the following benefits over the stock Linux scheduler:

- 39.15% decrease in max-flow,
- 32.78% decrease in max-stretch,
- 28.6% average decrease in average process completion time.

These results were gathered over a 800 second time interval for the following variation of our technique: interval based technique, minimum size of 45, IPC threshold of 0.20. For these same variables, we see 116% 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.

Speed 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 11 shows this trade-off for different variations of our technique.

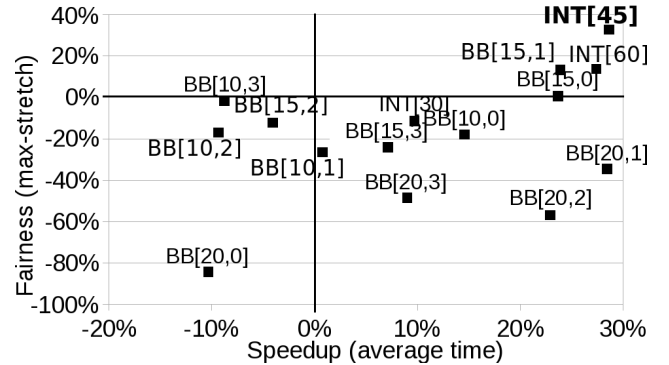


Figure 11. Speedup vs Fairness: average time vs. max stretch

These results are expected and show that a nice balance between the two exists. Our interval techniques appear to perform quite well at balancing these two metrics. Furthermore, 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 12 shows this comparison.

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 interval based technique improves throughput by 116% 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

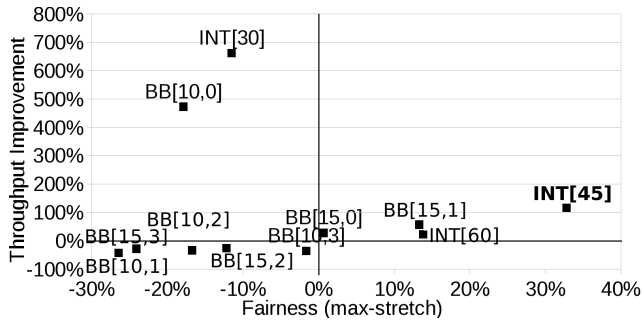


Figure 12. Throughput vs Fairness (max-stretch)

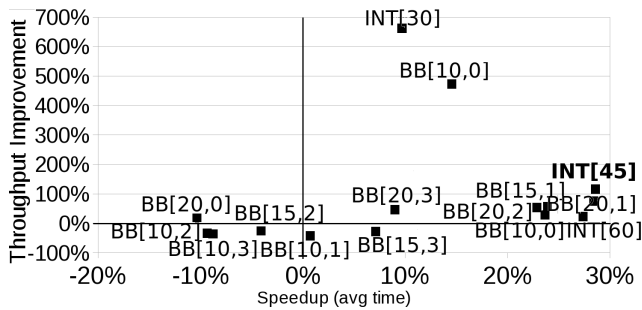


Figure 13. Speedup (avg. time) vs Throughput

that speedup is degraded by long starving processes while throughput is not affected by these. Figure 12 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 technique again gives a nice trade-off between the two since it improves upon both throughput and speedup.

Summary of Results In closing, our results show that phase-guided auto-tuning significantly outperforms the stock Linux scheduler in terms of the throughput obtained on a performance-asymmetric 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 116% and an average process speedup of up to 29%. 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.

5. Related Work

Huang *et al.* [25] show that basing processor adaptation on code sections (positional) rather than intervals in time (temporal) results in up to 50-80% improvement in energy reduction. Similarly, we take a positional approach, however, we do not use subroutines for our code sections. They use knowledge of previous executions of a subroutine to guide future decisions. We also use our idea of similarity to further reduce the dynamic overhead.

Becchi *et al.* [6] 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.* [1]. They make the case that a core assignment must be

balanced. Shelepov and Fedorova [33] 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.* [24] 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.* [39] 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 [32]. 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.

Lau *et al.* [22] define the idea of phase markers and propose a technique to determine these markers around procedures and loop boundaries. Our technique is similar, but uses different points for our phase markers and determines them differently.

There is a large body of work on determining phase behavior [12, 37, 38], using phase behavior to reduce simulation time [2, 10, 23, 31, 34, 35, 37], guide optimizations [16, 17, 26, 28–30, 40], 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.

6. Conclusion and Future Work

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 [4, 5, 15, 18, 21]. Devising techniques for their effective utilization is an important problem that influences the eventual uptake of this class of processors [18, 24]. Besides phase-guided auto-tuning, we know of two other approaches for improving the utilization of performance-asymmetric multicore processors: modifying the OS scheduler to account for asymmetry [24, 32] and load balancing to account for performance asymmetry [1, 6, 24]. These techniques require extensions to the operating system whereas phase-guided auto-tuning transparently improves the throughput for performance-asymmetric multicore processors. A predicted phase behavior and the exhibited execution characteristics of a small set of representative phases are exploited at runtime to determine likely profitable thread-to-core assignments for later phases of the program. By monitoring the execution of only this small representative set instead of monitoring the entire application, our approach reduces the monitoring overhead. Our results show as much as 116% improvement in throughput with a 29% reduction in the average process time when compared to the stock Linux scheduler while incurring negligible overheads and maintaining fairness.

Future work involves extending phase-guided auto-tuning in several directions. Are there advantages to be gained by tightly integrating our technique into the Linux scheduler (similar to Li *et al.* [24]). In particular, is the globally-optimal decision made by the

scheduler based on our technique better compared to locally optimal decisions made in the context of individual programs? Other extensions to our approach could include improve static phase transition analysis, dynamic feedback-based adaptation of thread-to-core assignments, and improved load-balancing techniques.

References

- [1] A. Fedorova *et al.* Operating system scheduling on heterogeneous core systems. In *OSHMA*, 2007.
- [2] A. Georges *et al.* Method-level phase behavior in java workloads. In *OOPSLA*, 2004.
- [3] Frances 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*, June 2005.
- [5] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA*, June 2005.
- [6] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF*, 2006.
- [7] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Annual symposium on Discrete algorithms*, 1998.
- [8] S. Y. Borkar. Platform 2015: Intel processor and platform evolution for the next decade. *Tech. Report - Intel*, 2005.
- [9] Jeanine Cook, Richard L. Oliver, and Eric E. Johnson. Toward reducing processor simulation time via dynamic reduction of microarchitecture complexity. *Perf. Eval. Rev.*, 2002.
- [10] Ashutosh Dhodapkar and James Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA*, 2002.
- [11] Ashutosh S. Dhodapkar and James E. Smith. Comparing program phase detection techniques. In *MICRO 36*, 2003.
- [12] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. In *PACT*, 2003.
- [13] Stephane Eranian. permon2: a flexible performance monitoring interface for linux. In *Ottawa Linux Symposium (OLS)*, 2006.
- [14] David Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.
- [15] Matt Gillespie. Preparing for the second stage of multi-core hardware: Asymmetric cores. *Tech. Report - Intel*, 2008.
- [16] Martin Hock, Karthik Jayaraman, Brian Pellin, and Vivek Shrivastava. Phase capture and prediction with applications. *Technical Report - Comp. Sci. Dept. - Univ. of Wisconsin-Madison*, 2005.
- [17] Shiwen Hu. *Efficient Adaptation of Multiple Microprocessor Resources for Energy Reduction Using Dynamic Optimization*. PhD thesis, The Univ. of Texas-Austin, 2005.
- [18] J. C. Mogul *et al.* Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro*, 2008.
- [19] J. Dongarra *et al.* Experiences and lessons learned with a portable interface to hardware performance counters. In *PADTAD Workshop*, 2003.
- [20] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT*, pages 23–32, 2006.
- [21] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 2005.
- [22] Jeremy Lau, Erez Perelman, and Brad Calder. Selecting software phase markers with code structure analysis. In *CGO*, 2006.
- [23] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Transition phase classification and prediction. In *HPCA*, 2005.
- [24] T. Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC*, 2007.
- [25] M. C. Huang *et al.* Positional adaptation of processors: application to energy reduction. *Comp. Arch. News*, 2003.
- [26] M. Merten *et al.* A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *ISCA*, pages 136–147, 1999.
- [27] Wiplove Mathur and Jeanine Cook. Towards accurate performance evaluation using hardware counters. In *WSMR*, 2003.
- [28] Priya Nagpurkar, Chandra Krintz, Michael Hind, Peter F. Sweeney, and V. T. Rajan. Online phase detection algorithms. In *CGO*, 2006.
- [29] Nitzan Peleg and Bilha Mendelson. Detecting change in program behavior for adaptive optimization. In *PACT*, 2007.
- [30] Cristiano Pereira and Rajesh Gupta. Using program phases as meta-data for runtime energy optimization. Technical report, Dept. of Comp. Sci. & Eng., UC San Diego, 2004.
- [31] R. Balasubramonian *et al.* Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, 2000.
- [32] R. Kumar *et al.* Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA*, page 64, 2004.
- [33] Daniel Shelepov and Alexandra Fedorova. Scheduling on heterogeneous multicore processors using architectural signatures. In *WIOSCA*, 2008.
- [34] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. In *ASPLOS-XI*, 2004.
- [35] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT*, pages 3–14, 2001.
- [36] Tyler Sondag and Hridesh Rajan. Phase-guided thread-to-core assignment for improved utilization of performance- asymmetric multi-core processors. In *IWMSE*, May 2009.
- [37] R. Srinivasan, J. Cook, and S. Cooper. Fast, accurate microarchitecture simulation using statistical phase detection. In *ISPASS*, page 147, 2005.
- [38] T. Sherwood *et al.* Automatically characterizing large scale program behavior. In *ASPLOS-X*, 2002.
- [39] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. *Operating Systems Review*, 2007.
- [40] Frederik Vandeputte, Lieven Eeckhout, and Koen De Bosschere. Exploiting program phase behavior for energy reduction on multi-configuration processors. *J. Sys. Archit.*, 2007.