

Phase-guided Thread-to-core Assignment for Improved Utilization of Performance-Asymmetric Multi-Core Processors

Tyler Sondag and Hridesh Rajan

TR #08-14

Initial Submission: January 31, 2008.

Keywords: static program analysis, heterogeneous multi-core processors, thread-to-core assignment, phase behavior.

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

Submitted.

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

Phase-guided Thread-to-core Assignment for Improved Utilization of Performance-Asymmetric Multi-Core 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

CPU vendors are starting to explore trade offs between die size, number of cores on a die, and power consumption leading to performance asymmetry among cores on a single chip. For efficient utilization of these performance-asymmetric multi-core processors, application threads must be assigned to cores such that the resource needs of a thread closely matches resource availability at the assigned core. This significantly complicates the task of an average programmer. The contribution of this work is a technique for automatically determining the mapping between threads and performance-asymmetric cores of a processor. Our approach, which we call *phase-guided thread-to-core assignment*, builds on a well-known insight that programs exhibit phase behavior. We first take code sections and group them into clusters such that each section in a cluster is likely to exhibit similar runtime characteristics. The key idea is that with this clustering, characteristics of a small number of representative sections in the cluster give insight into the behavior of the entire cluster. Thus the exhibited characteristics of the representative sections on different types of cores can be used for automating thread-to-core assignment at a lower runtime cost. Variations of our technique show up to 150% improvement in throughput over the stock Linux scheduler for systems with a constant feed of jobs, while maintaining comparable fairness and efficiency. No modifications to existing compilers or underlying operating systems is necessary, facilitating transparent deployment. Furthermore, our framework implementing this strategy produces standalone binaries that are independent of the characteristics of the target performance-asymmetric multi-core architecture thus avoiding the need to create multiple customizations of the binary.

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

General Terms Algorithms, Experimentation, Performance

Keywords static program analysis, heterogeneous multi-core processors, thread-to-core assignment, phase behavior

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 [15]. CPU vendors are projecting that in the next decade the number of cores in a CPU will increase to as many as hundreds [36]. 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 multi-core processors called performance-asymmetric or heterogeneous multi-cores [5, 6, 16, 34, 43, 21]. All cores in a performance-asymmetric multi-core processor support the same instruction set, however, they differ in terms of performance characteristics such as clock frequency, cache size, etc [16, 33, 21]. These architectures have been shown to provide an effective trade-off between performance, die area, and power consumption compared to homogeneous multi-core processors [16, 34, 43, 21].

To effectively utilize performance-asymmetric multi-core processors, application threads must be executed on cores such that the resource requirements of a thread closely matches the resources provided by the core. This must be done while maintaining fairness between threads. For example, Kumar *et al.* [35] have shown that when workload characteristics are matched well to heterogeneous cores, performance gains of up to 40% are observed (similar results have been shown by Li *et al.* [43]).

To match the resource requirements of a thread to the resources provided by the core, both must be known. The programmer can do this manually, however, this introduces several problems. First, the programmer must know the runtime characteristics of the program code as well as details about the underlying architecture. Furthermore, with multiple target architectures, this problem is exacerbated since this manual process must be carried out for each architecture. Also, this is a manual process and may be prone to errors. With all of this in mind, we desire an automatic technique to remove these burdens from the programmer.

With an automatic technique, we still need to know both the resource requirements of the code and the resources provided by the cores. To determine the resource requirements of a thread, we can perform an offline analysis of an execution trace for some representative input. This raises some problems. First, the trace for representative inputs may not account for unanticipated use cases, and second, the same thread may exhibit vastly different characteristics on another processor with a different core configuration. It is possible to create variants for each core configuration, however, that is highly likely to lead to the binary-version explosion problem. An online analysis of resource requirements by monitoring the program periodically throughout its execution is likely to be more precise. However, even with hardware counters [22], such analysis is likely to incur runtime overhead. This overhead may partially or completely erode the performance gains of the improved thread-to-core assignment.

The main contribution of this work is a technique, which we call *phase-guided thread-to-core assignment*, for matching the resource requirements of a thread to the resources provided by the cores of performance asymmetric multi-core processors. Our technique builds on a well-known insight that programs exhibit phase behavior [17, 19, 25, 28, 37, 42, 44]. By phase behavior we mean that a program goes through phases of execution that show similar runtime characteristics compared to other phases [32, 10, 12, 13, 2, 39].

Based on this insight, our approach consists of two parts. An offline program analysis, which identifies likely *phase-transition points* in a program, and a lightweight dynamic analysis that determines thread-to-core mapping on the fly. We define a phase-transition point as a point in the program where it is likely to change its runtime characteristics. We use the offline analysis results to generate standalone binaries in which each phase-transition point is instrumented with a tiny fragment for dynamic analysis. This modified binary can be run on stock Linux distributions. In other words, our technique does not require any modifications to the operating system. It also does not make any assumptions about the performance characteristics of the target architecture. Thus we avoid the need for multiple versions for each target platform.

We evaluated our approach on workloads constructed from 36 to 84 benchmarks in the SPEC CPU2000 benchmark suite (benchmarks are run simultaneously). Since the main goal of our technique is to improve *throughput* for systems with a near constant feed of jobs, upon completion of a benchmark another is immediately started. For these workloads, phase-guided thread-to-core assignment has shown significant improvements in overall throughput when compared to the standard assignment strategy of the stock Linux scheduler with only minor overheads.

The rest of this paper is organized as follows. Section 2 describes two components of our approach: offline phase transition analysis, and dynamic analysis, Section 3 describes our experimental setup, Section 4 describes experimental results, Section 5 describes related work, and Section 6 describes our plans for future work and concludes.

2. Phase-guided Thread-to-core Assignment

A program exhibits phase behavior [32, 10, 12, 13, 2, 39] in that it goes through several phases of execution that show similar runtime characteristics compared to other phases of execution. If we can 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; the actual runtime characteristics of a small number of representative sections in the cluster are likely to manifest the behavior of the entire cluster.

If the process of classifying a program’s execution into sections and sections into clusters is largely independent of the program’s input, a phase-guided thread-to-core assignment technique will have several benefits. No development efforts for representative inputs will be needed; and thread-to-core assignments for unanticipated use cases and varying architectures could be automatically tackled.

Based on these intuitions, phase-guided thread-to-core assignment works as follows. First, an offline 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, *phase mark*. The idea of phase marking is similar to the work by Lau *et al.* [24], however, we do not use a program trace to determine our phase marks and make our selections based on a different criteria. A phase mark contains information about the phase type for the current section, performs dynamic performance analysis, and makes core switching decisions. At runtime the phase marks analyze the performance of a small number of representative sections of each phase type. These analysis results are used to determine a suitable core mapping 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 mapping 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 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 Offline Phase Transition Analysis

The aim of our offline phase transition analysis is to determine points in the control flow where its 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 types that are likely to exhibit similar runtime behavior. We then do an intra-procedural analysis that uses the results of the basic block analysis to summarize intervals [3] into a single type. The result of the basic block analysis and summarization is used to construct an inter-procedural control flow graph, which is used to detect and mark phase transitions with *phase marks*.

Attributed Control Flow Graph Construction Our offline 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 classification of basic block based on execution traces is given in Section 3, however, other methods for basic block classification can easily be used.

From these, attributed intra-procedural control-flow graphs for procedures in the program 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 attributed control-flow graph of a procedure is then partitioned 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 inter-

val represent the leaf nodes. A sample run of this summarization algorithm is illustrated in Figure 1. The summarization algorithm is shown in Algorithm 1.

Algorithm 1 : Interval Summarization to Find Dominant Type

```

ρ = {}
for all DFS(I) do
  if η ∈ ρ then
    M ⊕ {π ↦ M(π) + wb * φ(η)}
  else
    M ⊕ {π ↦ M(π) + wf * φ(η)}
  end if
  ρ = η + ρ
  return max(dom(M))
end for

```

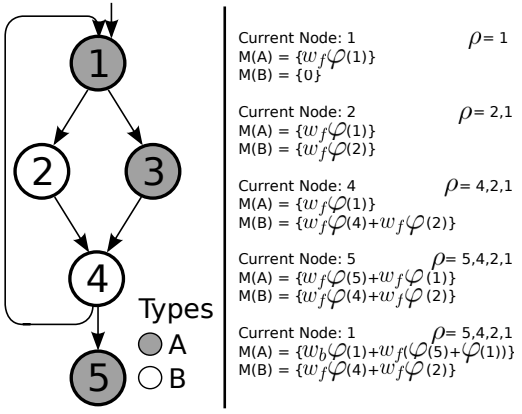


Figure 1. Interval Summarization Illustration

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, compute the target control flow node (η') of the backward edge. For each control flow node η'' from η' on the stack ρ , change the type map M 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 are currently using 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. 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.

2.2 Phase Transition Marking

Once the phase transition analysis is complete, we statically insert phase marks in the binary to produce a standalone binary with phase information and dynamic analysis code fragments. 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 (β) 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 only considers sections for marking that are longer than a fixed number of instruction. More generally, if the section has more 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, it is possible to have a basic block consisting 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. We must make our switching decisions at a coarser granularity. As mentioned previously, 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 (results supporting this are given in Section 4). So, we still need to pick better points for phase marks.

The second technique further addresses this problem by only considers 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 2. The intuition behind this 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. Notice that 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

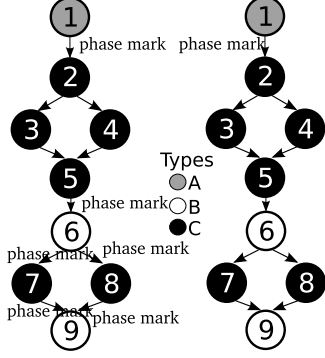


Figure 2. look-ahead for fewer phase marks

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.

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  $s/t \geq e$  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

```

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 be actual exhibited type for the interval based on which instructions in the interval are executed and how many times they are executed.

2.3 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 contains 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 offline analysis, this dynamic analysis code is customized according to the phase type of the section to reduce runtime 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 is 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. If our intuition that “all sections that have the same phase type are likely to exhibit similar runtime behavior” holds, the decision about optimal core made by just monitoring few representative sections will be valid for all sections of the same phase type. 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 [11, 7, 41]). IPC directly correlates to throughput and improved utilization of performance-asymmetric multicore processors. If the execution of a section is not well suited to a certain core it will take more cycles to complete on that core. 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 difference 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.

3. Experimental Setup

Our experimental setup can be broken down into two major categories. First, we will describe the hardware setup and describe why we choose our evaluation platform. Then, we will breakdown the software components used and developed for our approach.

3.1 Hardware setup

Our experimental setup consisted of a performance-asymmetric multi-core processor setup containing 4 cores. Two cores were operating at 2.4GHz and the other two at 1.6GHz. This setup was 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 used the Fedora distribution of Linux with an unmodified kernel. There were two main reasons to use a physical system instead of a simulated system. First, we intended to evaluate our ideas using an implementation that is easier to port to a realistic setting. Second, we wanted to analyze our approach in a setting that is closer to 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. We considered two full system simulators M5 [8] and Simics [30], however, due to architectural limitations in their available CPU models, we could not use them. This setup is limited in hardware configurations to test. However, this platform shows the utility of

our approach. Also, porting our implementation to another system is trivial since we do not require any modifications to the standard Linux kernel. To perform core switches, we used the standard process affinity API available for Linux kernels (ver. ≥ 2.5).

3.2 Software setup

We developed a static analysis and instrumentation framework for phase detection and marking. This framework is based on the GNU Binutils. Initial tests were run using the Intel Analysis Tools for Object Modification (ATOM) [1], however, ATOM has several limitations that preclude it for use in our experimentation. The Intel ATOM project is closed source and no longer active. A full version was never released. Many advanced features that we needed were not implemented, e.g. getting the target of a branch instruction. Our framework addresses these limitations and also executes the instrumented binaries much more efficiently (this will be discussed along with our results). To dynamically monitor the performance of code sections, we used the Performance Application Programming Interface (PAPI) [22]. PAPI provides an interface to control and access information gathered by the processor hardware performance counters. For our method, the events we need to monitor are instructions retired and cycles. These two events allow us to calculate the IPC (instructions retired / cycles) which we use to determine the efficiency of a mapping. Unfortunately, on our system, we only have 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, when performance is monitored, it is only monitored for 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 likely to be impacted significantly by the need to wait for the counters to be available. We used the perfmon2 monitoring interface [14] to measure the throughput of entire workloads using pfmon.

We are not presenting a static phase approximation technique at this time. Therefore, our experiments use previously determined knowledge of program performance on all core types in the system. This is determined by running each of the programs entirely on each core type and measuring the average IPC of each section of code. The IPC threshold values used in the figure are used to determine the phase types for basic blocks.

In order to take care of concerns such as load balancing and locking of the performance counters, we used shared memory for inter-process communication.

Workload Construction. Workloads range in size from 36 to 84 benchmarks in the SPEC CPU2000 benchmark suite. For example, for a workload of size 84, we run 84 benchmarks simultaneously on the system. Upon completion of a benchmark, another is immediately started to maintain a constant workload size.

4. Experimental Results

Many systems receive a nearly constant feed of jobs to run. 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 will enable the system to handle larger workload sizes. Our approach is targeting these systems, with maximizing throughput as its key objective. Our hypothesis is that *our technique will improve the throughput of such a system while incurring a small time and space overhead*. The results in this section validate this hypothesis. First, we briefly analyze the time and space overhead of our approach. We then investigate the throughput observed for workloads with phase-guided thread-to-core assign-

ment and compare it with the throughput observed while running the stock Linux scheduler.

4.1 Space and Time Overhead

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

During our offline analysis, we insert phase marks in the original binary to prepare it for phase-guided assignment. 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 undesirable 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.

Technique	Space overhead of phase marks (in %)					
	ampp	art	crafty	gzip	mcf	vpr
BB[10, 0]	0.09	25.60	0.01	2.59	31.02	5.07
BB[10, 1]	0.09	21.68	0.01	1.94	26.79	3.41
BB[10, 2]	0.09	21.68	0.01	1.94	23.77	2.25
BB[10, 3]	0.09	23.67	0.01	2.01	26.19	2.13
BB[15, 0]	0.09	18.79	0.01	0.64	20.15	0.84
BB[15, 1]	0.09	14.86	0.01	0.51	18.85	0.48
BB[15, 2]	0.09	14.86	0.01	0.57	19.54	0.38
BB[15, 3]	0.09	17.81	0.01	0.64	18.94	0.38
BB[20, 0]	0.09	14.93	0.01	0.50	4.71	0.38
BB[20, 1]	0.09	12.50	0.01	0.42	3.83	0.27
BB[20, 2]	0.09	12.50	0.01	0.42	4.71	0.17
BB[20, 3]	0.09	13.95	0.01	0.50	3.83	0.17
Int[10]	93.47	120.20	17.22	21.58	81.75	77.80
Int[25]	40.41	42.21	7.61	6.56	29.74	34.21
Int[30]	32.10	34.37	6.57	4.31	23.10	29.10
Int[55]	16.48	19.64	3.07	1.55	4.64	12.44

Table 1. Space overhead of phase marks for different variations: BB[n, m] represents basic block technique with minimum block size of n and look-ahead of m and Int[n] represents interval-based technique with minimum block size of n .

To measure the space overhead, a comparison between the size of the original binary and modified binary was performed for several variations of our technique. Table 1 shows some of these measurements for a subset of benchmarks in the SPEC2000 benchmark suite. All benchmarks are not shown due to space constraints, however, the trends are fairly clear from the table. Except for ammp and crafty, where binary sizes were too small for any noticeable difference, as the size of basic block considered increased the space overhead decreased. Similarly, as the look-ahead depth increased the space overhead usually decreased. 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. 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. In this case also we see decrease in space overhead with increase in interval size. The trends in space overhead offers insight into trends in time overhead.

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[25]	27.51	28.09	27.85	28.95
Int[30]	24.93	23.24	24.08	23.54
Int[55]	9.60	6.23	4.68	5.59

Table 2. Time spent in phase marks

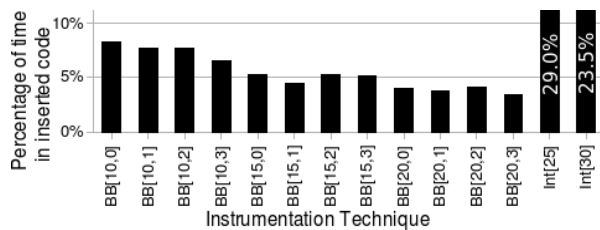


Figure 3. Time overhead: workload size 84

To measure the time overhead of inserted phase marks (including the core switches), we created a version of our workloads that was exactly the same as the code for other tests with one exception. 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 original unmodified binary and this version of the instrumented binary shows the cost of running our phase marks and core switching at the predetermined points in the program. Table 2 shows these costs for variable workload sizes and Figure 3 shows results for workloads of size 84. The trends are similar to space overhead, with time overhead decreasing with larger basic block sizes and increased depth of look-ahead. In some cases overhead was as little as 3.56%. Nevertheless, 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, as well as tighter integration with the system scheduler, but at the expense of requiring OS modification.

4.2 Throughput

To test our hypothesis that “phase-guided thread-to-core assignment will significantly increase throughput”, we compared the throughput achieved by our technique and the stock Linux scheduler for the same set of workloads run under the same conditions. Throughput was measured in terms of instructions committed over several time intervals of execution. Again, we want to improve performance for systems that have a nearly constant feed of processes/requests (e.g. a server). Thus, we maintained a constant number of jobs in the workload in the system for both cases. To achieve this, when a job is completed, our experimentation script immediately gives another job to the system.

Figure 4 shows the observed improvement in throughput for our technique when using the basic block level phase marking with

varying levels of look-ahead. This figure shows us several things. First, as look-ahead increases, throughput decreases. This is because with less look-ahead we are assigning many more blocks to cores that they are well matched for, however, there is a trade-off with overhead since a strategy that switches more often will incur the extra cost of these core switches. These overheads were presented in Figure 3. Second, we can clearly see a optimal threshold level and on either side of it performance decreases as we reach extreme cases for the threshold. As we get to the highest and lowest thresholds show, we even see a throughput decrease. This performance decrease is partially due to the fact that extreme thresholds create load imbalance across the cores.

Figure 5 shows the observed improvement in throughput our technique gives when using the interval strategy for first order intervals. As we observed with basic blocks using look-aheads, for most thresholds, we see less improvement than the techniques which map at a more fine grained level. We also see significantly less improvement than all look-ahead depths which is largely because of the inaccuracy in determining interval types (since we do not know statically which path through the interval will be taken at runtime). However, in some cases, we still notice exceptionally high improvement. Again, there is a trade-off with overhead that we previously discussed.

Since our technique for determining phase behavior may be inaccurate, Figure 6 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 with a look-ahead of 0 but with error levels of 20% and 30%. To introduce this error, after determining the grouping 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 phase behavior data. In some cases, the throughput actually increases over the technique with no error. Since our technique does not consider some portions of the program (these tests do not consider all basic blocks smaller than 10 instructions), we ignore part of each program in the workload. So, in some cases our error can actually improve the throughput by randomly picking a strategy that better assigns these sections of code. In most cases the throughput is less than our technique with no error, however, this difference is very small. With extreme thresholds, the error moves processes away from overloaded cores and improves throughput.

Next, to gain insight about the fairness of our scheduling technique, we observe the completion times of benchmarks in the workload. We present a small portion of this data in Figure 7 which was taken from a test using the basic block strategy with a minimum block size of 10 and a look-ahead depth of 2. Results are shown for thresholds of 0.10 and 0.20 (which gave the highest throughput). We can observe that within this time interval roughly the same number of benchmarks are completed. Also, for the 0.20 threshold, we have benchmarks completing in roughly the same fashion. More extreme thresholds had no processes finish within this time interval.

Summary In closing, our results show that phase-guided assignment can significantly outperform the stock Linux scheduler in terms of the throughput obtained on a performance-asymmetric multi-core processor, while maintaining fairness and with a negligible overhead in most cases. 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

Our previous work [41], focuses on a static analysis technique to predict phase behavior and identify phase transition points in the

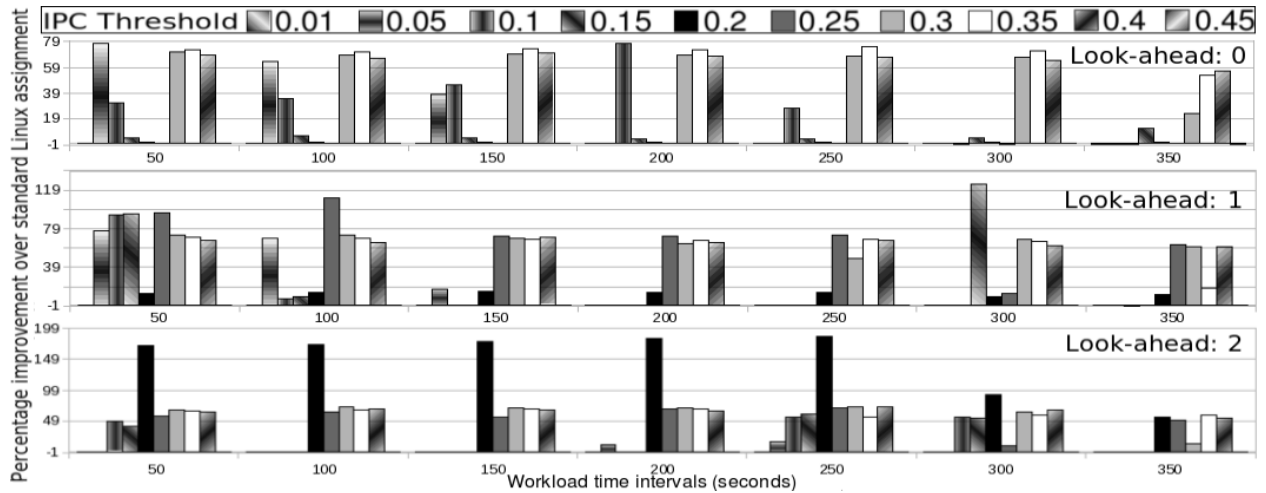


Figure 4. Throughput improvement: Basic block strategy, min. block size: 10, variable look-ahead

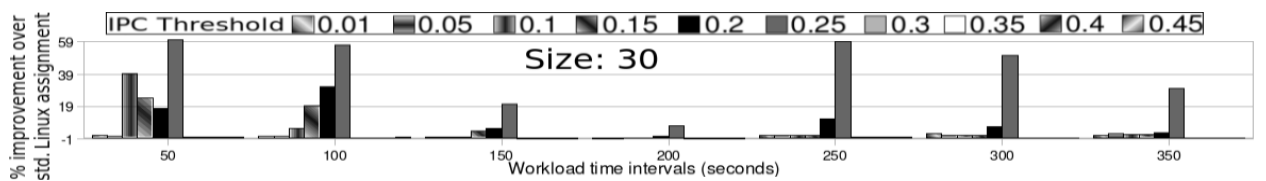


Figure 5. Throughput improvement: Interval strategy, first order intervals, min. interval size: 30

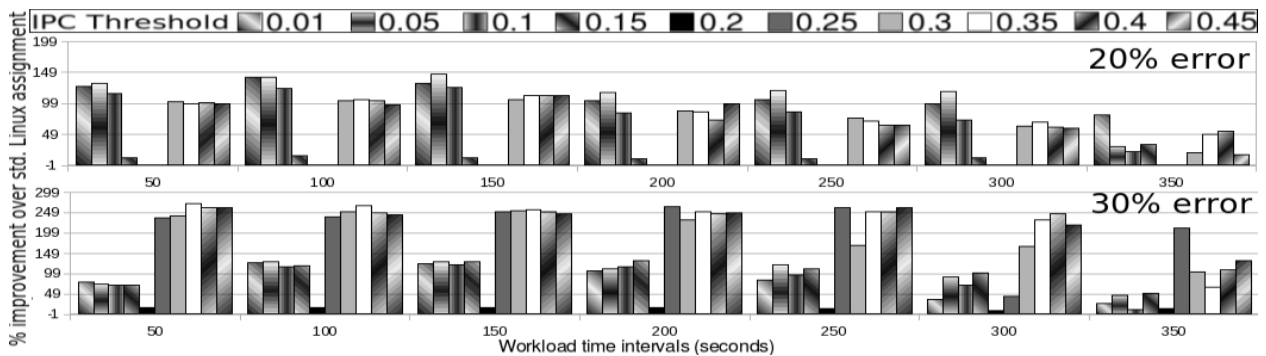


Figure 6. Improvement: Basic block, min. size: 10, look-ahead: 0, variable phase behavior accuracy

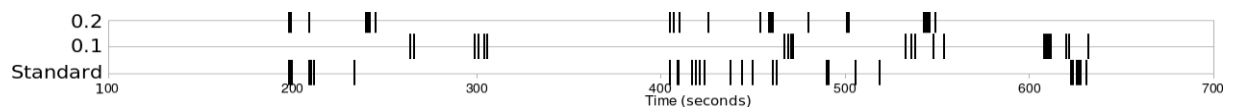


Figure 7. Process completion times: Basic block, min. size: 10, look-ahead: 2, thresholds: 0.1, 0.2

program. We now focus on dynamic assignment instead of static analysis. Furthermore, in the previous work, no evaluation was presented. In this paper we show the benefits of our approach via a rigorous evaluation.

Huang *et al.* [20] 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. This is similar to our work in that 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.* [7] proposed a dynamic mapping 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. Also similar is the work by Tam *et al.* [11] which determines thread-to-core mapping based on increasing cache sharing. They use cycles per instruction (CPI) as a metric to improve sharing for symmetric multi-core processors. Kumar *et al.* propose a temporal dynamic approach [33]. After

certain time intervals, a sampling phase is triggered. After the sampling phase, the system makes a decision regarding the mapping 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 mapping decisions have been made.

Lau *et al.* [24] 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. We also determine our phase markers without ever running the program.

There is a large body of work on using phase-analysis to quickly test the effectiveness of various ideas by reducing the time taken for simulation, e.g. by Balasubramonian *et al.* [32], Dhodapkar and Smith [13, 12], Sherwood *et al.* [38, 39], Georges *et al.* [2], Srinivasan *et al.* [42, 10], Shen *et al.* [37], Lau *et al.* [25]. Sherwood *et al.* [39] developed a technique to determine similar program phases by comparing segments of executed code in order to identify simulation points in a program to reduce architectural simulation time. Sherwood *et al.* [40] also presented a hardware technique for determining and predicting phases based on proportions of code being executed at run-time. Vandeputte *et al.* [44] identify phases through profiling, then determine optimal hardware configuration per phase using offline analysis, Hu [19] use profiles to reduce search space for optimal configuration of reconfigurable microprocessors, Kim *et al.* [23] use phases to predict branch prediction effectiveness, Hock *et al.* [17] use phases to dynamically reconfigure hardware and branch predictors that are based on the current phase, Nagpurkar *et al.* [28] propose a method for dynamic phase detection and optimization based on the phase information, etc.

The techniques described are related to our ideas in spirit, however, most of these techniques gather phase information by analyzing a previously generated dynamic profile. As we have mentioned previously, collecting a dynamic profile requires end-users to develop a representative set of test cases for the program. *The end-users may not have the desired expertise for collecting such input*, furthermore collecting dynamic profile is time consuming. Techniques that determine the phase information completely dynamically do not require end-user input, however, they are likely to incur performance overheads. We take a middle-of-the road approach, where much of the analysis is conducted statically followed by a limited analysis at runtime. On one hand, statically approximating the similarity between sections of program removes the need for upfront dynamic profiling and test input generation. On the other hand, the approximation helps significantly lower runtime monitoring overhead.

There is also a large body of work on using phase analysis for various optimizations. For example, Merten *et al.* [26] use hardware-based profiling to detect phase behavior that in turn drives optimization decision, Peleg and Mendelson citepeleg1 detect changes in the behavior of a program to trigger re-optimization. One category of such optimizations is to reduce power consumption. In this category, notable related ideas include Pereira *et al.*'s approach [31] to use phases to guide operating system power management decisions, Rusu *et al.*'s approach [9] that uses phases to help optimize power consumption, Hu *et al.*'s approach to [18] uses a slightly modified version of Sherwoods phase determination (run a representative phase) to estimate power consumption of entire program, etc.

There is also a body of work on using phase analysis to guide program monitoring. For example, Vaswani [4] turn on and off monitoring based on phases, Nagpurkar *et al.* [29] use phases for performance monitoring, etc.

We argue that most of these techniques are complementary. Key ideas from our approach for static approximate phase analysis guided thread-to-core can also be utilized for power management and optimal monitoring goals.

6. Conclusion and Future Work

Performance-asymmetric multi-core 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 [5, 6, 16, 34, 21]. Devising techniques for their effective utilization is an important problem that influences the eventual uptake of this class of processors [43, 21]. Besides phase-guided thread-to-core assignment, we know of two other approaches for improving the utilization of performance-asymmetric multi-core processors: modifying the OS scheduler to account for asymmetry [43, 33] and load balancing to account for performance asymmetry [7]. These techniques require extensions to the operating system whereas phase-guided thread-to-core assignment transparently improves the throughput for performance-asymmetric multi-core processors. A predicted phase behavior and the exhibited execution characteristics of a small set of representative phases is 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 evaluation shows up to 150% improvement in throughput compared to the stock Linux scheduling policy while incurring negligible overheads.

Future work involves extending phase-guided thread-to-core assignment in several directions. It would be interesting to investigate whether there are advantages to be gained by tightly integrating our technique into the Linux scheduler similar to the work of Li *et al.* [43]. 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] *Intel Analysis Tools for Object Modification (Intel Atom): Release Notes: Release 1.0 Beta.*
- [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] Matthew W. Alsleben and Jeanine Cook. Toward dynamic recognition of workload phases. In *4th Annual Austin CAS Conference*, 2003.
- [5] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *ISCA*, June 2005.
- [6] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures pp 506. In *ISCA*, June 2005.
- [7] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Conference on Computing frontiers (CF)*, pages 29–40, 2006.
- [8] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. In *Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2003.
- [9] C. Rusu *et al.* Integrated cpu and l2 cache frequency/voltage scaling using supervised learning. 2007.
- [10] Jeanine Cook, Richard L. Oliver, and Eric E. Johnson. Toward reducing processor simulation time via dynamic reduction of microarchitecture complexity. *Perform. Eval. Rev.*, 30(1), 2002.

- [11] D. Tam *et al.* Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *EuroSys*, 2007.
- [12] Ashutosh Dhodapkar and James Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA*, 2002.
- [13] Ashutosh S. Dhodapkar and James E. Smith. Comparing program phase detection techniques. In *MICRO*, page 217, 2003.
- [14] Stephane Eranian. permon2: a flexible performance monitoring interface for linux. In *Ottawa Linux Symposium (OLS)*, 2006.
- [15] David Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.
- [16] Matt Gillespie. Preparing for the second stage of multi-core hardware: Asymmetric (heterogeneous) cores. Technical report, Intel Corporation, July 2008.
- [17] Martin Hock, Karthik Jayaraman, Brian Pellin, and Vivek Shrivastava. Phase capture and prediction with applications. *Technical Report - Com. Sci. Dept. - University of Wisconsin-Madison*, 2005.
- [18] Chunling Hu, John McCabe, Daniel A. Jimenez, and Ulrich Kremer. Infrequent basic block-based program phase classification and power behavior characterization. In *Workshop on Interaction between Compilers and Computer Architectures*, 2006.
- [19] Shiwen Hu. *Efficient Adaptation of Multiple Microprocessor Resources for Energy Reduction Using Dynamic Optimization*. PhD thesis, The University of Texas at Austin, 2005.
- [20] Michael C. Huang, Jose Renau, and Josep Torrellas. Positional adaptation of processors: application to energy reduction. *SIGARCH Comput. Archit. News*, 31(2):157–168, 2003.
- [21] J. C. Mogul *et al.* Using asymmetric single-isa cmps to save energy on operating systems. *IEEE Micro*, 2008.
- [22] J. Dongarra *et al.* Experiences and lessons learned with a portable interface to hardware performance counters. In *PADTAD*, 2003.
- [23] Hyesoon Kim, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. 2d-profiling: Detecting input-dependent branches with a single input data set pp 159. In *CGO*, 2006.
- [24] Jeremy Lau, Erez Perelman, and Brad Calder. Selecting software phase markers with code structure analysis. In *CGO*, 2006.
- [25] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Transition phase classification and prediction. In *HPCA*, 2005.
- [26] M. Merten *et al.* A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *ISCA*, 1999.
- [27] Wiplove Mathur and Jeanine Cook. Towards accurate performance evaluation using hardware counters. In *ITEA Modeling and Simulation Workshop*, 2003.
- [28] Priya Nagpurkar, Chandra Krintz, Michael Hind, Peter F. Sweeney, and V. T. Rajan. Online phase detection algorithms. In *CGO*, 2006.
- [29] Priya Nagpurkar, Hussam Mousa, Chandra Krintz, and Timothy Sherwood. Efficient remote profiling for resource-constrained devices. *ACM Trans. Archit. Code Optim.*, 3(1):35–66, 2006.
- [30] P. Magnusson *et al.* Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.
- [31] Cristiano Pereira and Rajesh Gupta. Using program phases as meta-data for runtime energy optimization. Technical report, Dept. of Computer Sc. & Eng., UC San Diego, 2004.
- [32] R. Balasubramonian *et al.* Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, 2000.
- [33] R. Kumar *et al.* Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA*, page 64, 2004.
- [34] R. Kumar *et al.* Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.
- [35] R. Kumar *et al.* Core architecture optimization for heterogeneous chip multiprocessors. In *PACT*, 2006.
- [36] S. Y. Borkar *et al.* Platform 2015: Intel processor and platform evolution for the next decade. Technical Report White Paper, Intel Corporation, 2005.
- [37] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. In *ASPLOS-XI*, 2004.
- [38] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT*, 2001.
- [39] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X*, 2002.
- [40] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *ISCA*, 2003.
- [41] Tyler Sondag, Viswanath Krishnamurthy, and Hridayesh Rajan. Predictive thread-to-core assignment on a heterogeneous multi-core processor. In *PLOS*, Oct. 2007.
- [42] R. Srinivasan, J. Cook, and S. Cooper. Fast, accurate microarchitecture simulation using statistical phase detection pp 147. In *ISPASS*, 2005.
- [43] T. Li *et al.* Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Conference on Supercomputing*, 2007.
- [44] Frederik Vandeputte, Lieven Eeckhout, and Koen De Bosschere. Exploiting program phase behavior for energy reduction on multi-configuration processors. *J. Syst. Archit.*, 53(8), 2007.