Phase-based Tuning for Better Utilization of Performance-Asymmetric Multicore Processors

Tyler Sondag and Hridesh Rajan
Dept. of Computer Science
Iowa State University
Ames, IA 50011
{sondag,hridesh}@iastate.edu

***Abstract*—The latest trend towards performance asymmetry among cores on a single chip of a multicore processor is posing new challenges. For effective utilization of these performance-asymmetric multicore processors, code sections of a program must be assigned to cores such that the resource needs of code sections closely matches resource availability at the assigned core. Determining this assignment manually is tedious, error prone, and significantly complicates software development. To solve this problem, we contribute a transparent and fully-automatic process that we call *phase-based tuning* which adapts an application to effectively utilize performance-asymmetric multicores. Compared to the stock Linux scheduler we see a 36% average process speedup, while maintaining fairness and with negligible overheads.**

I. Introduction

Recently both CPU vendors and researchers have advocated the need for a class of multicore processors called single-ISA performance-asymmetric multicore processor (AMP) [1]–[4]. All cores in an AMP support the same instruction set, however, they differ in terms of performance characteristics such as clock frequency, cache size, etc [1], [4], [5]. AMPs have been shown to provide an effective trade-off between performance, die area, and power consumption compared to symmetric multicore processors [1]–[4].

A. The Problems and their Importance

Programming AMPs is much harder compared to their symmetric counterparts. In general, for effective utilization of AMPs, code sections of a program must be executed on cores such that the resource requirements of a section closely match the resources provided by the core [3], [6]. To match the resource requirements of a code section to the resources provided by the core, both must be known. The programmer may manually perform such a mapping, however, this manual tuning has at least three problems.

1) First, the programmer must know the runtime characteristics of their code as well as the details of the underlying asymmetry. This increases the burden on the programmer. Further, both the resource requirements of processes and the resources provided by cores cannot be determined statically (code behavior may change with inputs, core characteristics may change between different platforms or if the workload on the system changes [7], [8]). This is troubling since utilization is heavily influenced by the accuracy of this knowledge [3], [6], [7].
2) Second, with multiple target AMPs this manual tuning must be carried out for each AMP, which can be costly, tedious, and error prone.
3) Third, as a result of this manual tuning a custom version must be created for each target AMP, which decreases re-usability and creates a maintenance problem. Further, the performance asymmetry present in the target AMP may not be known during development.

Effective utilization of AMPs is a major challenge. Finding techniques for automatic tuning is critical to address this challenge and to realize the full potential of AMPs [7].

B. Contributions to the State-of-the-art

The main technical contribution of this work is a novel program analysis technique, which we call *phase-based tuning*, for matching resource requirements of code sections to the resources provided by the cores of an AMP. Phase-based tuning builds on a well-known insight that programs exhibit phase behavior [9], [10]. That is, programs goes through phases of execution that show similar runtime characteristics compared to other phases [11]. Based on this insight, our approach has two parts: a static analysis which identifies likely *phase-transition points* (where runtime characteristics are likely to change) between code sections, and a lightweight dynamic analysis that determines section-to-core assignment by exploiting a program's phase behavior.

Phase-based tuning has the following benefits:

- **Fully Automatic:** Since phase-based tuning determines core assignments automatically at runtime, the programmer need not be aware of the performance characteristics of the target platform or their application.
- **Transparent Deployment:** Programs are modified to contain their own analysis and core switching code. Thus, no operating system or compiler modification is needed. Therefore, phase-based tuning can be utilized with minimal disruption in the build and deployment chain.
- **Tune Once, Run Anywhere:** The analysis and instrumentation makes no assumptions about the underlying AMP. Thus there is no need to create multiple versions for each target AMP. Also, by making no assumptions about

the target AMP, interactions between multiple threads and processes are automatically handled.
- **Negligible Overhead:** It incurs less than 4% space overhead and less than 0.2% time overhead, i.e. it is useful for overhead conscious software and it is scalable.
- **Improved Utilization:** Phase-based tuning improves utilization of AMPs by reducing average process time by as much as 36% while maintaining fairness.

To evaluate the effectiveness of phase-based tuning, we applied it to workloads constructed from the SPEC CPU 2000 and 2006 benchmark suites which are standard for evaluating processors, memory and compilers. These workloads consist of a fixed number of benchmarks running simultaneously. For these workloads we observed as much as a 36% reduction in average process time while maintaining fairness and incurring negligible overheads.

II. Phase-based Tuning

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

Based on these intuitions, phase-based tuning works as follows. A static analysis is performed to identify phase-transition points. This analysis first divides a program's code into *sections* then classifies these sections into one or more *phase types*. The idea is that two sections with the same phase type are likely to exhibit similar runtime characteristics. Third, we identify points in the program where the control flows [12] from a section of one phase type to a different phase type. These points are the phase-transition points.

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

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

A. Static Phase Transition Analysis

The aim of our static analysis is to determine points in the control-flow where behavior is likely to change, that is *phase-transition points*. The precision and the granularity of identifying such points is likely to determine the performance gains observed at runtime. To that end, the first step in our analysis is to detect similarity among basic blocks in the program and classify them into one or more phase types that are likely to exhibit similar runtime behavior. We then examine three analysis techniques to detect and mark phase transitions with *phase marks*. The first is a basic block level analysis. The second builds upon this basic block analysis to analyze intervals [12]. The third also builds upon the basic block analysis to analyze loops inter-procedurally.

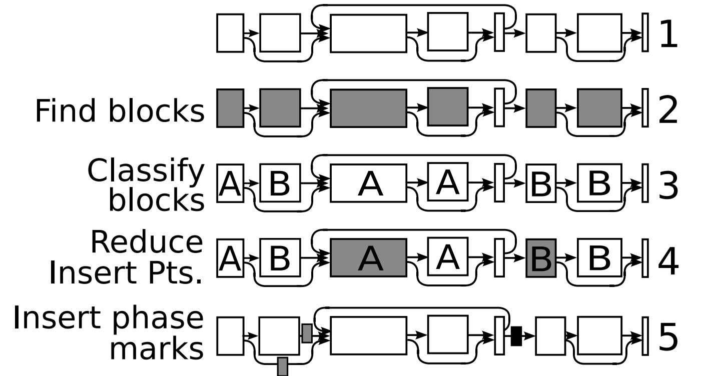

Fig. 1: Overview of Phase Transition Analysis

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 lookahead, this is illustrated in step 4. Finally, step 5 shows the new control-flow graph for the procedure which now includes the phase transition marks.

In this section, we first discuss the analysis techniques for annotating control-flow graphs (CFGs) with types for both of our techniques. Next, we discuss how to use the annotated control-flow graphs to perform the phase transition marking.

1) Static CFG Annotation:

a) Attributed CFG 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 ($\mathcal{B}$) [12]. 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 [12]. We then assign a type ($\pi \in \Pi$) to each basic block to construct the set of attributed basic blocks ($\bar{\mathcal{B}} \subseteq \mathcal{B} \times \Pi$). The notion of type here is different

[1]The idea of phase marking is similar to the work by Lau *et al.* [13], however, we do not use a program trace to determine our phase marks and make our selections based on a different criteria.

[2]Huang *et al.* [14] show that basing processor adaptation on code sections (positional) rather than time (temporal) improves energy reduction techniques. We also take a positional approach.

from types in a program and does not necessarily reflect the concrete runtime behavior of the basic block. Rather it suggests similarity between expected behaviors of basic blocks that are given the same type. A strategy for assigning types to basic blocks statically is given in Section II-A3, however, other methods for classifying basic blocks can also be used.

CFG construction from a binary representation may be incomplete. We currently ignore typing unknown targets, which may introduce some imprecision in the analysis but is safe. If more precision is desired, a more sophisticated analysis may be used or information may be gathered from the source code.

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 $\pi \in \Pi$.

b) Summarizing Intervals: The goal of our intra-procedural interval technique is to summarize intervals into a single type. To that end, 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 [12]. "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 [12, 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. Throughout this traversal, a value is computed for each type. Each node has a weight associated with it (those within cycles are given a higher weight) which is used for this computation. A detailed algorithm may be found in our technical report [15].

c) Summarizing Loops: The goal of our inter-procedural loop analysis is to summarize loops into a single type. We start with the attributed CFG for each procedure created by the basic block analysis. We then use the basic block types to determine loop types. A bottom-up typing is performed with respect to the call graph. In the case of indirect recursion, we randomly choose one procedure to analyze first then analyze all procedures again until a fixpoint is reached.

For each procedure, we start by partitioning the attributed CFG of the procedure into a unique set of *loops* ($\mathcal{L}$) using standard algorithms [16]. For each loop, $l \in \mathcal{L}$, we then compute its dominant type starting with the inner-most loops. We do a breadth-first traversal of the loop starting with the entry node, while ignoring backward edges. This algorithm is shown in Algorithm 1 and illustrated in Figure 2.

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

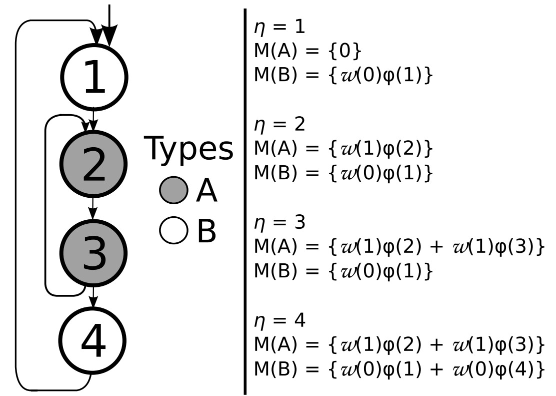

Fig. 2: Loop Summarization Illustration

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

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

Suppose we have a loop l' which contains the current loop l. If both loops have the same type ($\pi_{l'} = \pi_l$), it is not beneficial to incur our analysis and optimization code's overhead at each iteration of the outer loop. Instead, we perform the analysis and optimization before the outer loop, and eliminate any work done inside this loop. Thus, after we determine the type for the current loop l, we find the type for the next largest nested loop, l'. If there is no such loop, then we add the current type information to the loop type map T. If the type of the nested loop (l') is the same as the current loop (l), then we add the current loop, l to the type map and remove the nested loop l'. If the types of the two loops differ, we take the type with the higher strength, σ, since it is more likely that we have an accurate typing for such a loop. Finally, we have a special condition (**else if**) to handle nesting where two disjoint loops, l' and l''', are nested inside a loop, l. In this case, we type the loop l only if the two disjoint loops, l' and l''', have the same type which is also the same type as the outer loop l.

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

d) Phase Transitions: Once we have determined types for sections (blocks, intervals, or loops) of the program's CFG, we compute the phase transition points. Recall that a phase-transition point is a point in the program where runtime characteristics are likely to change. Since sections of code with

Algorithm 1 : Loop Summarization to Find Dominant Type. BFS ignores back edges

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

the same type should have approximately similar behavior, we assume that program behavior is likely to change when control flows from one type to another.

2) *Phase Transition Marking:* Once the phase transitions are determined, we statically insert phase marks in the binary to produce a standalone binary with phase information and dynamic analysis code fragments. These code fragments also handle the core switching. By instrumenting binaries, we eliminate the need for compiler modifications. Furthermore, by using standard techniques for core switching, we require no OS modification. We have considered several variations of phase transition marking that are classified into three kinds based on whether it operates on the attributed control-flow graphs, the attributed interval graphs, or the attributed loop graphs. In all cases, phase marks are placed at the phase transitions.

a) 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 (and the same number of times). This means that the phase type for the section is likely to be accurate and the same as the corresponding basic block type $\pi \in \Pi$, where $\bar{\beta}$ is (β,π). 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 is small (tens of instructions in the SPEC benchmarks). Phase marking at this granularity could result in frequent core switches overshadowing any performance benefit. To avoid this, we use two techniques.

Our first technique is to skip basic blocks with size below a configurable threshold. Our second technique (*lookahead based phase marking*) is to insert a phase mark only if majority of the successor of a code section up to a fixed depth have the same type. The intuition is that if majority of the successors have the same type, a core switch cost will more likely be offset by the gains in improving core utilization.

b) Adding Phase Marks to Attributed Interval Graphs: Our second class of methods consider sections to be intervals in the attributed interval graph. Using intervals for phase marking enables us to look at the program at a more coarse granularity than basic blocks. Even with 1st order interval graphs, the intervals frequently 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.

c) Adding Phase Marks to Attributed Loop Graphs: Our third class of methods consider a section to be loops in the attributed loop graph. Using loops for phase marking has even more advantages than using intervals. Not only does it allow inserting outside of loops, it also allows better handling of nested loops by frequently eliminating phase-marks within loop iterations. This is an even more coarse view of the program than the interval based technique. Furthermore, since it is an inter-procedural analysis, transitions across function calls are handled. Just like interval typing, loop typing introduces some imprecision in the type information.

3) *Determining block types:* In this work, we choose to focus more on developing and evaluating (1) the various techniques and granularity for determining and marking phase transitions and (2) the dynamic analysis and optimization techniques. However, as a proof of concept, we have developed a simple static analysis for determining types of basic blocks.

This analysis involves looking at a combination of instruction types as well as a rough estimate of cache behavior (computation based on reuse distances [17]). Information describing these two components are used to place blocks in a two dimensional space. The blocks are then grouped using the k-means clustering algorithm [18].

We have evaluated the accuracy of this approach in combination with our loop based clustering technique as follows. Blocks are classified into groups by using this simple analysis. Next, the dominant type of the loops are determined using the algorithm from Section II-A1. This loop typing is compared with the actual observed behavior of the loops.

In summary, experimentation shows that this technique miss-classifies only about 15% of loops. As our results show in Section IV, this is accurate enough for our purposes and thus no further improvements are sought at this time. However, a more precise analysis could simply be substituted in future to improve overall performance.

B. Dynamic Analysis and Tuning

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

The code in the phase mark either makes use of previous analysis to make its core choice or observes the behavior of the code section. A variety of analysis policies could be used and any desirable metric for determining performance could be used as well. In this paper, we present a simple analysis and similarity metric used for our experiments.

For this case, the code for a phase mark serves two purposes: First, during a transition between different phase types, a core switch is initiated. The target core is the core previously determined to be an optimal fit for this phase type. Second, if an optimal fit for the current phase type has not been determined, the current section is monitored to analyze its performance characteristics. The decision about the optimal core for that phase type is made by monitoring representative sections from the cluster of sections that have the same phase type. By performing this analysis at runtime, we do not require the programmer to have any knowledge of the target architecture. Furthermore, the asymmetry is determined at runtime removing the need for multiple program versions customized for each target architecture. Since our static technique ensures that sections in the same cluster are likely to exhibit similar runtime behavior, the assignment determined by just monitoring few representative sections will be valid for most sections in the same cluster. Thus, monitoring all sections will not be necessary. This helps to reduce the dynamic overhead of our technique.

For analyzing the performance of a section, we measure instructions per cycle (IPC) (similar to [19], [20]). IPC correlates to throughput and utilization of AMPs. For example, cores with a higher clock frequency can efficiently process arithmetic instructions whereas cores with a lower frequency will waste fewer cycles during stalls (e.g. cache miss). IPC is monitored using hardware performance counters prevalent in modern processors. The optimal core assignment is determined by comparing the observed IPC for each core type.

Our technique for determining optimal core assignment is shown in Algorithm 2. The underlying intuition is that cores which execute code most efficiently will waste fewer clock cycles resulting in higher observed IPC. Since such cores are more efficient, they will be in higher contention. Thus, the algorithm picks a core that improves efficiency but does not overload the efficient cores.

This algorithm first sorts the observed behavior on each core and sets the optimal core to the first in the list. Then, it steps though the sorted list of observed behaviors. If the difference between the current and previous core's behavior is above some threshold, the optimal core is set to the current core. The intuition is that when the difference is above the threshold, we will save enough cycles to justify taking the space on the more efficient core. By performing the performance analysis at runtime, this algorithm for computing optimal core assignment *does not require knowledge of the program or underlying architecture* thus easing the burden on the programmer. It also *eliminates the need for an optimized version of the program for each target architecture* thus avoiding maintenance problems.

Algorithm 2 Optimal Core Assignment for n Cores

select(π,δ):*best core for phase type π, with threshold δ*
  $C := \{c_0, c_1, \ldots, c_n\}$ (set of cores)
  Sort C s.t. $i > j \Rightarrow f(c_i, \pi) > f(c_j, \pi)$.
  $f(c_i, \pi)$ - the actual measured IPC of block type π on core c_i.
  $d \leftarrow c_0$
  for all $c_i \in C \setminus \{c_n\}$ **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

III. Analysis and Instrumentation Framework

We developed a static analysis and instrumentation framework (based on the GNU Binutils) for phase detection and marking. By instrumenting binaries rather than source code, we avoid compiler modification. Low overhead of instrumentation is crucial to reduce the overhead of our phase-marks. Compared to a similar static instrumentation tool, ATOM [21], binaries instrumented with our tool execute 10 times faster[3]. This is because our binary instrumentation strategy is finely tuned for this task compared to that of a general strategy used by ATOM. To summarize, we use code specialization, live register analysis, and instruction motion so that we only incur the cost for an unconditional jump and a relatively small number of pushes. We also do not change the target of any indirect branch.

Core switches are done using the standard process affinity API available for Linux (kernel ver. $\geq$ 2.5). To monitor the performance of code sections, we use PAPI [22] which provides an interface to control and access the processor hardware performance counters.

To monitor a section's IPC with PAPI we use instructions retired and cycles (IPC = instructions retired / cycles). IPC is used to determine the runtime characteristics of phases[4]. To deal with limitations that may be imposed by the number of counters or APIs, we require programs to wait for access to the counters. Since our approach requires very little dynamic monitoring, processes seldom have to wait for counter availability. If a process must wait, the wait time is negligible since the amount of code that must be monitored is small. Because

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

[4]Using IPC as a performance metric may cause problems in presence of some features (e.g. floating point emulation). For a discussion regarding dealing with such problems, we refer to our technical report [15].

of this, performance is not impacted significantly by the need to wait for the counters to be available.

IV. Evaluation

The aim of this section is to evaluate our five claims made in Section I. First, we claim that phase-based tuning requires no knowledge of program behavior or performance asymmetry. Our technique is completely automatic and requires no input from the programmer. In our experiments, workloads are generated randomly and without any knowledge of behavior of the benchmarks. Second, we claim the technique allows for transparent deployment. Since our analysis and instrumentation framework operates on binaries, no modification to compilers is necessary. Furthermore, since we use standard techniques for switching cores, no OS modifications are necessary. Third, we claim that with our technique you can "tune once and run anywhere". Our static analysis makes no assumptions about the underlying asymmetry. Since our performance analysis and section-to-core assignment are done dynamically, the same instrumented applications may be run on varying asymmetric systems. Our final two claims are those related to performance: negligible overhead and improved utilization. In the rest of this section, we use experimental results to evaluate these two claims.

A. Experimental Setup

1) System Setup: Our setup consists of an AMP with 4 cores. This setup uses an Intel Core 2 Quad processor with a clock frequency of 2.4GHz and two cores under-clocked to 1.6GHz. There are two L2 caches shared by two cores each. The cores running at the same frequency share an L2 cache. We use an unmodified Linux 2.6.22 kernel (which uses the $O(1)$ scheduler) and standard compilers. Thus, we demonstrate the transparent deployment benefit of our approach.

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 [23]. 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.

We use the perfmon2 monitoring interface [24] to measure the throughput of workloads. For evaluation purposes, to determine basic block types for our static analysis with little to no error, we use an execution profile from each core. Using the observed IPC, we assign types to basic blocks. The *difference* in IPC between the core types is compared to an *IPC threshold* to determine the typing for basic blocks.

2) Workload Construction: Similar to Kumar *et al.* [5] and Becchi *et al.* [20] our workloads range in size from 18 to 84 randomly selected benchmarks from the SPEC CPU 2000 and 2006 benchmark suites. For example, if we are testing a workload of size 18 there are 18 benchmarks running simultaneously. We refer to such a workload as having 18 slots for benchmarks. Like Kumar *et al.* [5] we want the system to receive jobs periodically, except rather than jobs arriving randomly, our workloads maintain a constant number of running jobs. To achieve this constant workload size, upon completion of a benchmark, another benchmark is immediately started. If 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. Thus, we maintain a job queue for each workload slot. That is, if we have a workload of size 18 then there are 18 queues (one for each slot in the workload). These 18 queues are each created individually from randomly selected benchmarks from the benchmark suites. When a workload is started, the first benchmark in each queue is run. Upon completion of any process in a queue, the next job in the queue is immediately started. When comparing two techniques, the same queues were used for each experiment. This ensures that we more accurately capture the behavior of an actual system.

B. Space and Time Overhead

Statically, we insert phase marks (consisting of data and code) in the program to enable phase-based tuning. Since insertion of large chunks of code may destroy locality in the instruction cache, low space overhead is desired. Also, a phase mark's execution time is added to the execution of the original program. Thus, we must ensure that the overhead does not overshadow the gains achieved by our technique.

1) Space Overhead: To measure space overhead, we compared the sizes of the original and modified binaries for variations of our technique. Figure 3 shows a box plot for the measurements taken from the benchmarks in the SPEC CPU 2000 and 2006 benchmark suites. The box represents the two inner quartiles and the line extends to the minimum and maximum points. The trends are expected. As the minimum size increases, space overhead decreases. Similarly, as lookahead depth increases, space overhead generally decreases. For individual programs this is not always the case because by adding another depth of lookahead, the percentage of blocks belonging to the same type may be pushed over the threshold causing another insertion point.

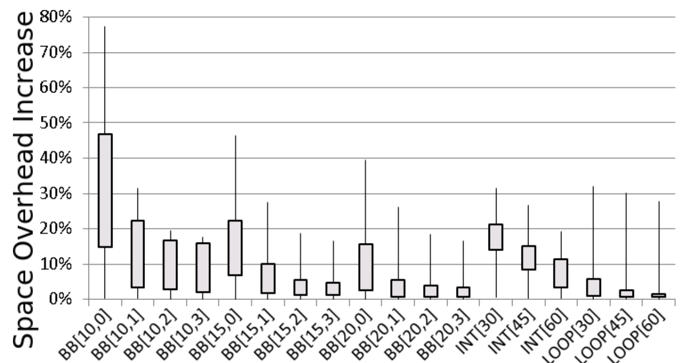

Fig. 3: Space overhead

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

2) Time Overhead: To measure the time overhead of phase marks and core switches instead of switching to a specific core, we switch to "all cores". Switching to "all cores" means that the same API calls are made that optimized programs make, however, instead of defining a specific core, we give all cores in the system. Thus, the difference in runtime between the unmodified binary and this instrumented binary shows the cost of running our phase marks at the predetermined program points. Figure 4 shows results for workloads of size 84.

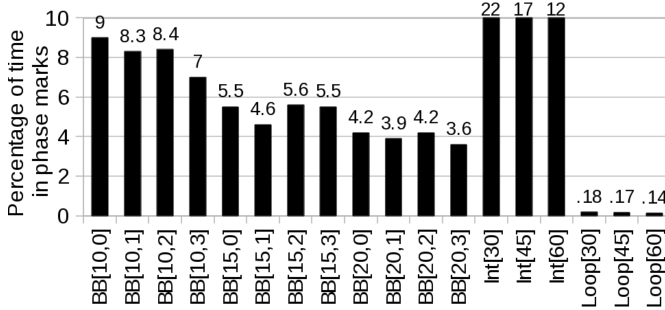

Fig. 4: Time overhead: workload size 84

The trends shown are mostly expected and are similar to those for space overhead. What makes these results interesting is that in some cases overhead was as little as 0.14%. At first, it is quite surprising that the loop based technique reduced overhead as much as it did. There are several reasons for this improvement. First, compared to the interval and basic block techniques, only loops are considered whereas the other techniques considers many intervals and groups of blocks which are not loops. On top of this, it considers nesting of loops. Clearly, removing an insertion point inside of a nested loop will greatly reduce the number of total executions of phase marks. Not only does the technique consider nesting, but it also considers function calls in order to eliminate phase marks in functions that are called inside of loops thereby eliminating more phase marks from loops.

3) Core Switches: Here we analyze the frequency and cost of core switches for each benchmark. First, experiments were done to estimate the cost of each core switch. This was done by writing a program that alternates between cores and then counting the cycles of execution for this program. Using this technique, we have determined that a core switch takes approximately 1000 cycles. More precise measurement could be done, but this is sufficient to gain insight into the necessary cycles require to amortize the cost of a core switch. Next, we consider core switches for each benchmark in detail.

In Table 1 we show the number of core switches and runtime (in isolation) for each benchmark. This table shows that most programs change phase types occasionally throughout execution. Some programs differ in that they have few or only one phase according to our analysis. These benchmarks, aside from choosing the best core to execute on, mostly stay on the same core. Finally, two benchmarks (459 and 473), do not have any phases at all. These benchmarks will simply execute on any core the OS deems appropriate.

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

Table 1: Switches per benchmark (Loop[45], 0.2 threshold)

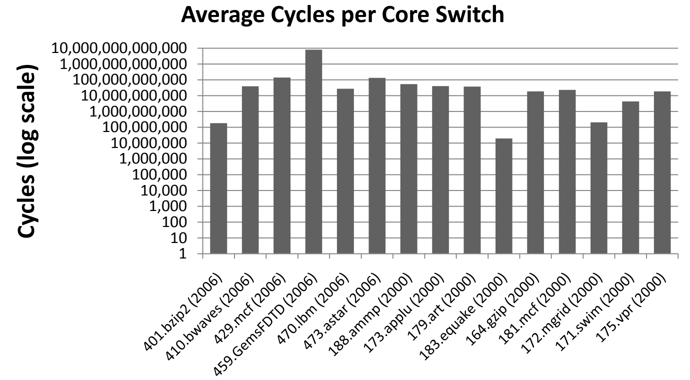

Fig. 5: Cycles per Core Switch

Figure 5 presents the average number of cycles per core switch (log scale). Most benchmarks fall in the range of tens of billions of cycles per core switch which is clearly enough amortize the switching cost.

C. Throughput

To test our hypothesis that *"phase-based tuning will significantly increase throughput"*, we compared our technique and the stock Linux scheduler (for the same workloads run under the same conditions). Throughput was measured in terms of instructions committed over a time interval (0% representing no improvement). We measure how variations of our technique and variables in our algorithms affect throughput. For all figures presented in this section, the data is taken from the first 400 seconds of the workload execution.

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

Thus, throughput still gives insight into how variables in the technique impact performance.

1) IPC threshold: First, we want to see how the IPC threshold affects throughput. As mentioned in Section III, IPC threshold is used to determine the section-to-core assignment. Figure 6 shows how different threshold values affect throughput when all other variables are fixed (technique, min. size, lookahead, etc).

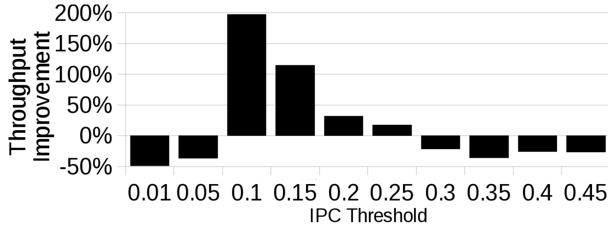

Fig. 6: Throughput: Basic block strategy, min. block size: 15, lookahead depth: 0, variable IPC threshold

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

2) Lookahead depth: We have examined the impact of lookahead depth for the basic block technique. The trends are expected in that less lookahead gives higher throughput but at a significant cost in fairness.

3) Clustering error: Statically predicting similarity will have some inaccuracy. Thus, in Figure 7 we show 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%. For our tests, since we have two core types, our perfect assignment (0% error) consists of two clusters, one for each core type. To introduce this error, after determining the clustering of blocks, a percentage of blocks were randomly selected and placed into the opposite cluster. The result is that blocks we expect to perform better on a "fast" core are run a "slow" core and vice versa.

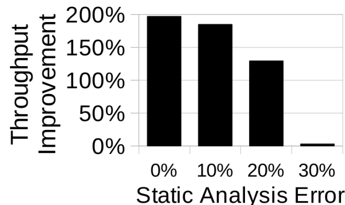

Fig. 7: Throughput improvement: Basic block strategy, min. block size: 15, lookahead depth: 0, variable error

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

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

D. Fairness

Improved throughput is advantageous, however, in many systems we also desire fairness. Therefore, we analyze the fairness of our technique. We use three metrics to analyze fairness: max-flow, max-stretch, and average process time. Max-flow and max-stretch were developed by Bender *et al.* for determining fairness for continuous job streams [25].

For each process, we have the following data: a_i: arrival time of process i, C_i: completion time of process i, and t_i: processing time of process i (in isolation). First, *max-flow* is defined as $\max_j F_j$, where $F_j = C_j - a_j$. This is basically the longest measured execution time. So, if even one process is starving, this number will increase significantly. Second, *max-stretch* is defined as: $\max_j F_j/t_j$. This can be thought of as the largest slowdown of a job. We consider this because we want processes to speed up, but not at the expense of others slowing down significantly. These measurements for our techniques are shown in Table 2.

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

Table 2: Fairness Comparison to standard Linux assignment: Improvements are shaded.

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

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

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

E. 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. Due to space limitations we only present the most interesting results and leave the rest for our technical report [15].

Here we examine the trade-off between speedup and fairness. Speedup refers to the decrease in average process runtime. Max-stretch is used for fairness. Figure 8 shows this trade-off for different variations of our technique.

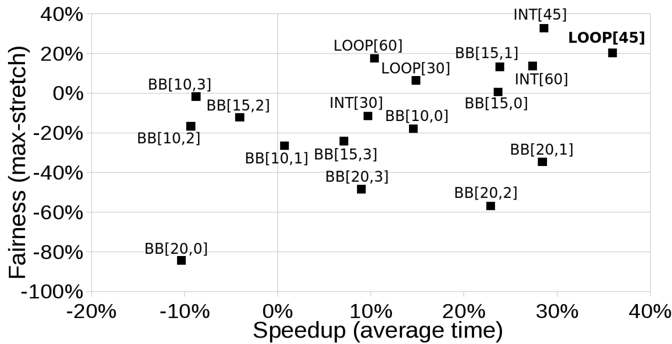

Fig. 8: Speedup vs Fairness: average time vs. max stretch

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

V. Related Work

Our previous work on phase-based tuning considered the static analysis at the basic block and interval levels [26], [27]. This work enhances our technique to be inter-procedural and loop based. We also consider the balance of fairness, speedup, and throughput as an important factor and give more details regarding core switches. Somewhat contrary to our preliminary results, after rigorous experiments with much larger data set it turned out that the interval-level techniques were quite superior to the basic block techniques. This further led to the development of the loop-level inter-procedural static analysis. Also new is a preliminary technique for static block typing.

Becchi *et al.* [20] propose a dynamic assignment technique making use of the IPC of program segments. However, this work focuses largely on the load balance across cores. Shelepov *et al.* [28] propose a technique which does not require dynamic monitoring (uses static performance estimates). However, this technique does not consider behavior changes during execution. Li *et al.* [3] and Koufaty *et al.* [29] focus on load balancing in the OS scheduler. They modify the OS scheduler based on the asymmetry of the cores. While this produces an efficient system, the scheduler needs knowledge of the underlying architecture. 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.

Tam *et al.* [19] determine thread-to-core assignment based on increasing cache sharing. They use cycles per instruction (CPI) as a metric to improve sharing for symmetric multicore processors. Kumar *et al.* propose a temporal dynamic approach [5]. After time intervals, a sampling phase is triggered after which the system makes assignment decisions for 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.

Mars and Hundt [30] use a similar hybrid technique to make use of a range of static optimizations and choose the correct one at runtime using monitoring. Their approach differs in that their optimizations are multiple versions of the code. This requires optimization to be done during compilation to avoid problems such as indirect branches in the binary. Thus they are tied to a specific compiler (we are not). Similarly, Dubach *et al.* [31] use machine learning to dynamically predict desired hardware configurations for program phases. We do not determine the best configuration for phases, instead, we choose the best from a set of choices.

There is a large body of work on determining phase behavior [11], [32], using phase behavior to reduce simulation time [10], [11], [33], guide optimizations [9], [34]–[38], etc. Many of these techniques determine phase information with a previously generated dynamic profile [11]. Other techniques determine phase behavior dynamically [9], these techniques do not require representative input, however, they are likely to incur dynamic overheads.

VI. Discussion

A. Applicability to Multi-threaded Programs

The simplicity of our approach allows it to immediately work on multi-threaded applications. Recall that binaries are modified by inserting code. When an application spawns multiple threads, it is essentially running one or more copies of the same code which was present in the original application. The framework will have analyzed this code and modified it as needed. Thus, each thread will contain the necessary code switching and monitoring code present in the phase marks. Further discussion of how code and data sharing across cores is likely to impact performance is in our technical report [15].

B. Changing Application and Core Behavior

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

actually look at the physical hardware characteristics, we handle the changes in each cores behavior that occurs based on other processes in the system. For changing program behavior, simple feedback mechanisms can be added [15].

C. Scalability for many-cores

While our current implementation performs well for multi-core processor, there is a potential scalability issue for many-core processors since each core in the system would need to be tested for each cluster. However, previous work suggests that as few as two cores types are sufficient [2], [39]. Thus, if we can group cores into types (either manually or automatically using carefully constructed analysis) we can largely reduce the problem to one similar to a multi-core processor.

VII. Conclusion and Future Work

AMPs are an important class of processors that have been shown to provide nice trade-off between the die size, number of cores on a die, performance, and power [1], [2], [4]. Devising techniques for their effective utilization is an important problem that influences the eventual uptake of these processors [3], [4]. The need to be aware of, and optimize based on, the applications' characteristics and the nature of the AMP significantly increases the burden on developers. In this work we have shown that phase-based tuning improves the utilization of AMPs while not increasing the burden on the programmer. Our experiments show a 36% reduction in the average process time compared to the stock Linux scheduler. This is done while incurring negligible overheads (less than 0.2% time overhead) and maintaining fairness.

Future work involves extending phase-based tuning in several directions. First, we would like to study a wider variety of AMPs. We have already tried an additional setup consisting of 3 cores (2 fast, 1 slow). Performance results for our technique are similar for this setup (e.g. 32% speedup). However, it would be sensible to test additional configurations as they become widely available. Other future directions include improving our dynamic characteristics analysis and tuning to include feedback-based adaptation.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and suggestions. The authors were supported in part by the NSF under grant CCF-08-46059.

References

[1] M. Gillespie, "Preparing for the second stage of multi-core hardware: Asymmetric cores," *Tech. Report - Intel*, 2008.
[2] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, 2005.
[3] T. Li *et al.*, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *SC*, 2007.
[4] J. C. Mogul *et al.*, "Using asymmetric single-ISA CMPs to save energy on operating systems," *IEEE Micro*, 2008.
[5] R. Kumar *et al.*, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *ISCA*, 2004, p. 64.
[6] ——, "Core architecture optimization for heterogeneous chip multiprocessors," in *PACT*, 2006.
[7] A. L. Lastovetsky and J. J. Dongarra, *High Performance Heterogeneous Computing*. John Wiley & Sons, Inc., 2009.
[8] C. Boneti *et al.*, "A dynamic scheduler for balancing hpc applications," in *SC '08*, 2008.
[9] P. Nagpurkar, C. Krintz, M. Hind, P. F. Sweeney, and V. T. Rajan, "Online phase detection algorithms," in *CGO*, 2006.
[10] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," in *ASPLOS-XI*, 2004.
[11] T. Sherwood *et al.*, "Automatically characterizing large scale program behavior," in *ASPLOS-X*, 2002.
[12] F. E. Allen, "Control flow analysis," in *Symposium on Compiler optimization*, 1970, pp. 1–19.
[13] J. Lau, E. Perelman, and B. Calder, "Selecting software phase markers with code structure analysis," in *CGO*, 2006.
[14] M. C. Huang *et al.*, "Positional adaptation of processors: application to energy reduction," *Comp. Arch. News*, 2003.
[15] T. Sondag and H. Rajan, "Phase-based tuning for better utilized multi-cores," Iowa State University, Department of Computer Science, Tech. Rep., August 2010.
[16] S. S. Muchnick, *Advanced Compiler Design & Implementation*, D. E. Penrose, Ed. Academic Press, 1997.
[17] K. Beyls and E. H. D'Hollander, "Reuse distance as a metric for cache behavior," in *IASTED*, 2001.
[18] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *5th Berkeley Symposium on Mathematical Statistics and Probability*, 1967.
[19] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," *Operating Systems Review*, 2007.
[20] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *CF*, 2006.
[21] A. Srivastava and A. Eustace, "Atom: a system for building customized program analysis tools," in *PLDI '94*, 1994.
[22] J. Dongarra *et al.*, "Experiences and lessons learned with a portable interface to hardware performance counters," in *PADTAD Workshop*, 2003.
[23] W. Mathur and J. Cook, "Towards accurate performance evaluation using hardware counters," in *WSMR*, 2003.
[24] S. Eranian, "permon2: a flexible performance monitoring interface for linux," in *Ottawa Linux Symposium (OLS)*, 2006.
[25] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan, "Flow and stretch metrics for scheduling continuous job streams," in *Annual symposium on Discrete algorithms*, 1998.
[26] T. Sondag and H. Rajan, "Phase-guided thread-to-core assignment for improved utilization of performance- asymmetric multi-core processors," in *IWMSE*, May 2009.
[27] T. Sondag, V. Krishnamurthy, and H. Rajan, "Predictive thread-to-core assignment on a heterogeneous multi-core processor," in *PLOS*, Oct. 2007.
[28] D. Shelepov *et al.*, "Hass: a scheduler for heterogeneous multicore systems," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 66–75, 2009.
[29] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *EuroSys*, 2010.
[30] J. Mars and R. Hundt, "Scenario based optimization: A framework for statically enabling online optimizations," in *CGO*, 2009.
[31] C. Dubach, T. M. Jones, E. V. Bonilla, and M. F. O'Boyle, "A predictive model for dynamic microarchitectural adaptivity control," 2010.
[32] E. Duesterwald *et al.*, "Characterizing and predicting program behavior and its variability," in *PACT*, 2003.
[33] R. Balasubramonian *et al.*, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *MICRO*, 2000.
[34] S. Hu, "Efficient adaptation of multiple microprocessor resources for energy reduction using dynamic optimization," Ph.D. dissertation, The Univ. of Texas-Austin, 2005.
[35] N. Peleg and B. Mendelson, "Detecting change in program behavior for adaptive optimization," in *PACT*, 2007.
[36] F. Vandeputte, L. Eeckhout, and K. D. Bosschere, "Exploiting program phase behavior for energy reduction on multi-configuration processors," *J. Sys. Archit.*, 2007.
[37] G. Fursin *et al.*, "A practical method for quickly evaluating program optimizations," in *HiPEAC 2005*, 2005.
[38] V.J. Jiménez *et al.*, "Predictive runtime code scheduling for heterogeneous architectures," in *HiPEAC '09*, 2009.
[39] E. Grochowski *et al.*, "Best of both latency and throughput," in *ICCD '04*, Oct. 2004, pp. 236–243.