Phase-guided Tuning for Better Utilized Multicores

Tyler Sondag and Hridesh Rajan

TR #08-14b
Initial Submission: January 23, 2009.

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

CR Categories:
D.3.4 [*Programming Languages*] Processors - Optimization D.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

The work described in this article is the revised and extended version of a paper presented at the International Workshop on Multicore Software Engineering 2009 in Vancouver, Canada. This work has been supported in part by the US National Science Foundation under grants CNS-06-27354, CNS-07-09217, and CCF-08-46059.
Authors' address: T. Sondag, sondag@cs.iastate.edu, Computer Science, Iowa State University, Ames, IA 50010. H. Rajan, hridesh@cs.iastate.edu, Computer Science, Iowa State University, Ames, IA 50010.

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

Tyler Sondag and Hridesh Rajan
Iowa State University

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

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.13 [**Software Engineering**]: Reusable Software

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: static program analysis, heterogeneous multicore processors, thread-to-core assignment, phase behavior, performance asymmetry

1. INTRODUCTION

CPUs with multiple cores have become commodity items [Tichy 2009]. CPU vendors are projecting that in the next decade the number of cores in a CPU will increase to as many as hundreds [Borkar 2005]. This makes it important to devise techniques for their effective utilization. Recently both CPU vendors and researchers have advocated the need for a class of multicore processors called single-ISA performance-asymmetric multicores [Gillespie 2008; Kumar et al. 2005; Li et al. 2007; Mogul et al. 2008]. All cores in a performance-asymmetric multicore processor support the same instruction set, however, they differ in terms of performance characteristics such as clock frequency, cache size, etc [Gillespie 2008; Kumar et al. 2004; Mogul et al. 2008]. These architectures have been shown to provide an effective trade-off between performance, die area, and power consumption compared to homogeneous multicore processors [Gillespie 2008; Kumar et al. 2005; Li et al. 2007; Mogul et al. 2008]. This class of multicore processors is also claimed to better suit high-performance computing compared to symmetric multicores [Williams et al. 2006].

1.1 The Problems and their Importance

Programming performance-asymmetric multicore processors is, however, much harder compared to their symmetric counterparts. To illustrate consider a simple function shown in Figure 1. This function takes a two dimmensional array as a parameter. It first calculates various statistical properties of each row in the array (lines 5–15). The resulting statistical

data is output on lines 16–17. It then transposes the array in place (lines 18–20). The program then computes various statistical properties of each row in the transposed array (lines 21–27). The resulting statistical data is output on line 28. Finally, the sum of all elements in the array is computed (lines 30–32) and output (line 33). The runtime behavior of this simple program can be partitioned into four code sections: a compute intensive section on lines 5–15, a memory intensive section on lines 18–20, a compute intensive section on lines 21–27, and finally another compute intensive section on lines 30–32.

```
1 void fun(int array[][N]){
2  int i, j, mmLoc = -1;
3  int max[N], min[N];
4  float maxMean = MIN_FLOAT;
5  float median[N], mean[N], sd[N];
6  i = 0;
7  while(i < N){
8    median[i] = calcMed(array[i],N);
9    mean[i] = calcMean(array[i],N);
10   max[i] = calcMax(array[i],N);
11   min[i] = calcMin(array[i],N);
12   sd[i] = calcSd(array[i],N);
13   if(mean[i] > maxMean){
14     maxMean = mean[i];
15     mmLoc = i;
16   }
17   i++;
18 }
19 output(median,mean,max,min,
20         sd,maxMean,mmLoc);
21 i = 0;
22 while(i < N-1){
23  j = i + 1;
24  while(j < N){
25   swap(array[j][i], array[i][j]);
26   j++;
27  }
28  i++;
29 }
30 i = 0;
31 while(i < N){
32   median[i] = calcMed(array[i],N);
33   mean[i] = calcMean(array[i],N);
34   max[i] = calcMax(array[i],N);
35   min[i] = calcMin(array[i],N);
36   sd[i] = calcSd(array[i],N);
37   i++;
38 }
39 output(median,mean,max,min,sd);
40 int sum = 0;
41 i = 0;
42 while(i < N){
43   sum += (int)(mean[i] * N);
44   i++;
45 }
46 output(sum);
47 }
```

Fig. 1. An example function where behavior changes from compute intensive (dark gray) to memory intensive (light gray) and back to compute intensive.

Let us assume that this program is run on a performance-asymmetric processor with two cores (c_0,c_1), where c_0 has a higher clock frequency compared to c_1. If this program is run entirely on c_0, this core may be under-utilized during the execution of memory intensive parts , lines 18–20. On the other hand, if this program is run entirely on c_1, compute intensive parts (lines 5–15, 21–27, 30–32) would run at a sub-optimal speed. Ideally we would like the memory intensive parts to run on c_1 and the compute intensive part to run on c_0, which would result in an effective tradeoff between program execution speed and core utilization.

In general, for effective utilization of performance-asymmetric processors code sections of a program must be executed on cores such that the resource requirement of a section closely match the resources provided by the core [Kumar et al. 2006; Li et al. 2007]. To match the resource requirements of a code section to the resources provided by the core,

both must be known. The programmer can manually tune the application to achieve such a mapping, however, this manual tuning has at least three problems.

(1) First, the programmer must be aware of the runtime characteristics of the program code as well as the details of the underlying performance asymmetry of cores in a processor, which increases the intellectual burden on the programmer. Thus, distributing work among the cores is heavily influenced by, the knowledge of the relative speeds of the cores and the characteristics of applications [Kumar et al. 2006; Li et al. 2007]. The overall performance/speedup of the application will be affected by the accuracy of the estimation of relative speeds and the applications' characteristics [Lastovetsky and Dongarra 2009]. Unfortunately, both the resource requirements of processes and the resources provided by cores can not be correctly estimated statically because computation's characteristics may change with inputs and performance characteristics of the performance-asymmetric multicore may change if the application is deployed on a different processor or if the existing workload on the processor changes [Lastovetsky and Dongarra 2009; Boneti et al. 2008]. For example different input sizes may substantially alter the memory access patterns of a computation.
(2) Second, with multiple target architectures this manual tuning must be carried out for each architecture, 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 architecture, which decreases re-usability and creates a maintenance problem. The performance asymmetry present in the target multicore processor may not be known during development, which further complicates manual tuning.

Effective utilization of performance-asymmetric multicore processors is a major challenge facing the high-performance computing community. Finding techniques for automatically tuning HPC applications is critical to address this challenge and to realize the full potential of performance-asymmetric multicores [Lastovetsky and Dongarra 2009; Boneti et al. 2008].

1.2 Overcoming Problems

Because we are unable to determine program and core behavior statically, we must turn to a dynamic technique. Not only will a dynamic technique overcome the first problem mentioned in Section 1.1 it also enables the binary to be used on a variety of system not requiring customized versions based on the underlying asymmetry. We could take each program and instrument it with a *feedback-driven control system* [Lee and Markus 1967] capable of monitoring itself and automatically tuning itself to any executing heterogeneous platform. A problem with such instrumentation needs is that it incurs overheads (sometimes as much as 300% [Luk et al. 2005; Wallace and Hazelwood 2007]) that often outweighs the benefits obtained by an optimal distribution. Since a dynamic technique is desirable but potentially expensive, we aim to reduce the overhead of such dynamic techniques by exploiting a program's *phase behavior*.

1.3 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 a performance-asymmetric multicore processor. Phase-based tuning builds on a well-known insight that programs exhibit phase behavior [Lau

et al. 2005; Nagpurkar et al. 2006; Shen et al. 2004]. By phase behavior we mean that a program goes through phases of execution that show similar runtime characteristics compared to other phases [Sherwood et al. 2002]. Based on this insight, our approach consists of two parts: a static analysis, which identifies likely *phase-transition points* between code sections, and a lightweight dynamic analysis that determines section-to-core assignment on the fly by exploiting a program's phase behavior. We define a phase-transition point as a point in the program where runtime characteristics are likely to change. The static analysis results are used to generate standalone binaries in which each phase-transition point is instrumented with a tiny fragment for dynamic analysis. These fragments contain analysis code as well as phase information. This phase information reduces dynamic analysis costs by using the runtime behavior of previously executed sections to make future assignments.

Phase-based tuning has the following software engineering benefits:

—**Programmer Oblivious:** Since phase-based tuning determines core assignments automatically based on runtime information, the programmer need not be aware of the performance characteristics of the target multicore CPU or their application.

—**Transparent Deployment:** With our technique, the programs themselves are modified to contain analysis and core switching code. Thus, no modification is needed to the operating system or compilers. 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 regarding the target platform resulting in application tuning which is independent of the performance characteristics of the target architecture. Further, there is no need to create multiple versions[1]. Also, by making no assumptions about the target hardware, interactions between multiple threads and processes on the target system are handled without complications.

—**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 performance-asymmetric multicore processors by reducing average process time by as much as 36% while maintaining fairness between applications.

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

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.

The rest of this paper is organized as follows. Section 2 explains the two components of our approach: static phase transition analysis, and dynamic analysis, Section 3 describes our evaluation framework, Section 4 presents our experimental results, Section 5 describes related work. Section 6 gives a brief discussion of relevant issues, and Section 7 concludes.

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

2. PHASE-BASED TUNING

In this section, we describe our phase-based approach for matching resource requirements of code sections to the resources provided by the cores of a performance-asymmetric multicore processor. As described previously, the key challenge is to determine this match efficiently at runtime. 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 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 [Allen 1970] from a section of one phase type to a different phase type. These points are identified as phase-transition points.

Each phase-transition point is statically instrumented to insert a small code fragment which we call a *phase mark*[2]. 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[3]. Thus, the actual characteristics of few representative sections of a given phase type are used as an approximation of the expected characteristics of all sections of that phase type. This allows our technique to significantly reduce runtime overhead and automatically tackle new architectures. The rest of this section describes components of our approach in detail.

2.1 Static Phase Transition Analysis

The aim of our static analysis is to determine points in the control-flow where phase behavior is likely to change. We refer to such points as *phase-transition points*. The precision and the granularity of identifying such points is likely to determine the performance gains observed at runtime. To that end, the first step in our analysis is to detect similarity among basic blocks in the program and classify them into one or more phase types that are likely to exhibit similar runtime behavior. We then examine three different analysis to detect and mark phase transitions with *phase marks*. The first is a basic block level analysis. The second builds upon this basic block analysis to analyze intervals [Allen 1970]. The third

[2]The idea of phase marking is similar to the work by Lau *et al.* [Lau et al. 2006], however, we do not use a program trace to determine our phase marks and make our selections based on a different criteria.

[3]Huang *et al.* [Huang et al. 2003] show that basing processor adaptation on code sections (positional) rather than time (temporal) improves energy reduction techniques. We also take a positional approach.

also builds upon the basic block analysis to analyze loops inter-procedurally.

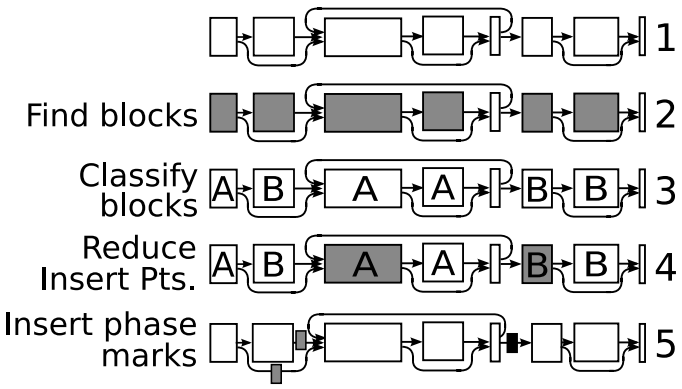

Fig. 2. Overview of Phase Transition Analysis

Figure 2 illustrates this process for our basic block level analysis. Step 1 represents the initial procedure. Step 2 finds the blocks which are larger than the threshold size (shaded). Next, step 3 finds the type for each block considered in the previous step. Then, we reduce the phase transition points by using a lookahead, this is illustrated in step 4. Finally, step 5 shows the new control-flow graph for the procedure which now includes the phase transition marks.

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

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

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

2.1.1.1 *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 ($\mathcal{B}$) [Allen 1970]. 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 [Allen 1970]. We then classify each basic block into exactly one type ($\pi \in \Pi$) to construct the set of attributed basic blocks ($\bar{\mathcal{B}} \subseteq \mathcal{B} \times \Pi$). The notion of type here is different from types in a program and does not necessarily reflect the concrete runtime behavior of the basic block. Rather it suggests similarity between expected behaviors of basic blocks that are given the same type. A strategy for assigning types to a basic block based on execution traces is given in Section 4.1, however, other methods for basic block classification can also be used.

Using the attributed basic blocks, attributed intra-procedural control-flow graphs for procedures are created. An attributed intra-procedural control-flow graph $\mathcal{CFG}$ is $\langle \mathcal{N}, \mathcal{E}, \eta_0 \rangle$. Here, $\mathcal{N}$, the set of control-flow graph nodes is $\bar{\mathcal{B}} \cup \mathcal{S}$, where $\mathcal{S}$ ranges over special nodes representing system calls and procedure invocations. The set of directed edges in the control-flow is defined as $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N} \times \{b, f\}$, where b, f represent backward and forward control-flow edges. $\eta_0 \equiv (\beta, \pi)$ is a special block representing the entry point of the procedure, where $\beta \in \mathcal{B}$ and $\pi \in \Pi$.

2.1.1.2 *Summarizing Intervals.* The goal of our intra-procedural interval [Allen 1970] analysis is to summarize intervals into a single type. To perform our interval analysis, we start with the attributed control-flow graph for each procedure created by the basic block analysis. We then use the basic block types to determine interval types.

For each procedure, we start by partitioning the attributed control-flow graph of the procedure into a unique set of *intervals* ($\mathcal{I}$) using standard algorithms [Allen 1970]. "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 [Allen 1970, pp.6]." For each i, we then compute its dominant type (as defined in Algorithm 1) by doing a depth-first traversal of the interval starting with the entry node, while ignoring backward control-flow edges (marked with b) unless traversal gets stuck at a non-leaf node. The exit nodes of the interval represent the leaf nodes. This summarization algorithm is shown in Algorithm 1 and illustrated in Figure 3.

Algorithm 1 : Interval Summarization to Find Dominant Type

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

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

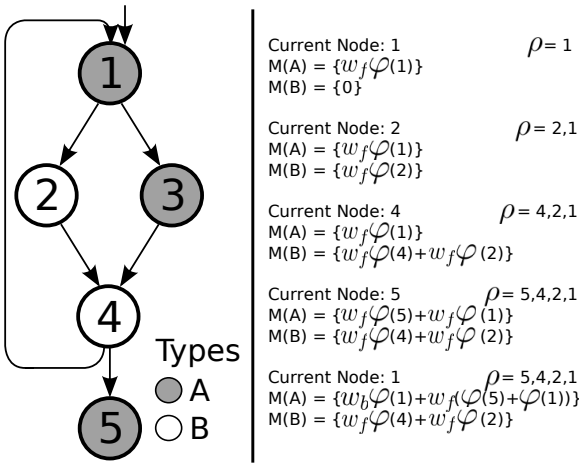

Fig. 3. Interval Summarization Illustration

maintained. On visiting a control-flow node η in the interval, the type map M is changed to M' where M' is $M \oplus \{\pi \mapsto M(\pi) + w_f * \varphi(\eta)\}$. Here, π is the type of the control-flow node, w_f is the forward edge weight, φ maps nodes to node weights, and $\oplus$ is the overriding operator for finite functions.

On reaching a control-flow node with an outgoing backward edge, if the backward edge has not previously been traversed, the target control-flow node (η') of the backward edge is computed. For each control-flow node η'' from η' to η on the stack ρ, the type map M is changed to M' where M' is $M \oplus \{\pi \mapsto M(\pi) + w_b * \varphi\eta\}$ and w_b is the backward edge weight. The values for w_f and w_b are heuristically decided, but intuitively it makes sense to have w_b greater than w_f (to give more weight to nodes in loops). The node weight function, $\varphi : \mathcal{N} \mapsto \mathbb{R}$, maps nodes to values based on a heuristic measure of the expected execution time of the block. We currently use the number of instructions in the node as this measure.

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

As a result of this process, we obtain another control flow graph of the procedure where nodes are tuples of intervals and their types. To distinguish these from control-flow graphs of basic blocks, we refer to them as *attributed interval graphs*. It would be interesting to explore whether summarizing interval graphs again is useful [Allen 1970], however, in this paper we only consider first-order intervals. Our initial intuition is that the value of applying nth order interval summarization will depend on the average size of procedures.

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

For each procedure, we start by partitioning the attributed control-flow graph of the procedure into a unique set of *loops* ($\mathcal{L}$) using standard algorithms [Muchnick 1997]. For each loop, $l \in \mathcal{L}$, we then compute its dominant type starting with the inner-most loops. We do a breadth-first traversal of the loop starting with the entry node, while ignoring backward control-flow edges. .This summarization algorithm is shown in Algorithm 2 and illustrated in Figure 4.

Algorithm 2 : Loop Summarization to Find Dominant Type. BFS ignores back edges

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

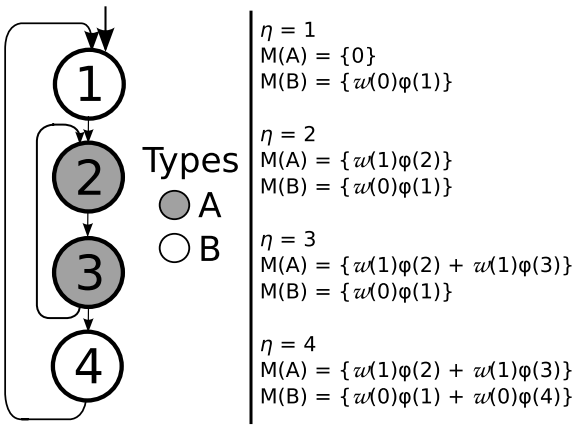

Fig. 4. Loop Summarization Illustration

Throughout the traversal, a type map for the loop ($M : \Pi \mapsto \mathbb{R}$) is maintained which maps types to weights. On visiting a control-flow node in the loop, $\eta \in l$, the type map M

is changed to $M' = M \oplus \{\pi \mapsto M(\pi) + w_n(\lambda) * \varphi(\eta)\}$. Here, π is the type of the control-flow node η, w_n maps nodes to nesting level weights, φ maps nodes to node weights, and $\oplus$ is the overriding operator for finite functions. Since loops are usually executed multiple times, nodes in contained loops should have more impact on the type of the overall loop. Thus, nodes which belong to inner loops are given a higher weight via the function $w_n(x)$ where x is the nesting level and $w_n : \mathbb{N} \to \mathbb{R}$ 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. Currently, the type with maximal number of control-flow nodes in the loop is used as a tiebreaker. We also have a strength of the type, σ which is simply the weight the type π_l over the sum of all other type weights. This strength is used for determining types of nested loops.

Suppose we have a loop l' which contains the current loop l. If both loops have the same type ($\pi_{l'} = \pi_l$), it is not beneficial to incur the overhead of our analysis and optimization code at each iteration of the outer loop. Instead, we perform this analysis and optimization before the outer loop, and eliminate any work done inside this loop. Thus, after we determine the type for the current loop l, we find 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 (the **else if** in the algorithm) to handle nesting where we have two disjoint loops, l' and l''', which 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*.

2.1.1.4 *Phase Transitions.* Once we have determined types for sections of the program's CFG, we compute the phase transition points. Recall that a phase-transition point is a point in the program where runtime characteristics are likely to change. Since sections of code with the same type should have approximately similar behavior, we assume that program behavior is likely to change when control flows from one type to another. The next section describes our techniques for marking these points in the application.

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

2.1.2.1 *Adding Phase Marks to Attributed CFG.* Our first class of methods all consider a section to be a basic block ($\bar{\beta}$) in the attributed CFG ($\mathcal{CFG}$). The advantage of using basic blocks is that execution of a single instruction in a block implies that all instructions in the

block will execute. This means that the phase type for the section is likely to be accurate and the same as the corresponding basic block type $\pi \in \Pi$, where $\bar{\beta}$ is (β,π). Our naïve phase marking technique marks all edges in the attribute CFG where the source and the target sections have different phase types. As is evident, this technique has a problem. The average basic block size in a program is small (tens of instructions). Phase marking at this granularity resulted in frequent core switches overshadowing any performance benefit. To avoid this, we use two techniques.

The first technique eliminates small sections of code. In other words, if the section has less than a threshold weight as defined by our node weight function, $\varphi : \mathcal{N} \mapsto \mathbb{R}$. This eliminates core switching for very small blocks of code. For example, a basic block may consist of a single instruction. Clearly it would not be cost effective to initiate a core switch so that a single instruction can execute more efficiently. Basic blocks are usually in the tens of instruction and often smaller. Even at this size the benefit of switching cores probably does not outweigh the cost of switching cores. So, we still need to pick better points for phase marks.

The second technique further addresses this problem by only considering a section if at least a fixed percentage of its successors up to a fixed depth have the same type (illustrated in Figure 5).

2.1.2.2 *Lookahead based Phase Marking.* This technique is presented in Algorithm 3 and illustrated in Figure 5. The intuition is the following. If the successors of a section have the same type, it is more likely that a core switch will be worth its cost. For small loops, when we look at enough successors, we start seeing the same nodes. Thus, if a loop contains predominately one type of blocks, we can simply make a core switch before the loop begins. Furthermore, this technique serves to reduce the number of phase marks in a program. Since adding each phase mark translates to adding a small number of instructions to the footprint of the binary and the control-flow path, we will reduce both the time and space overhead of the technique and hopefully not eliminate much of its benefit.

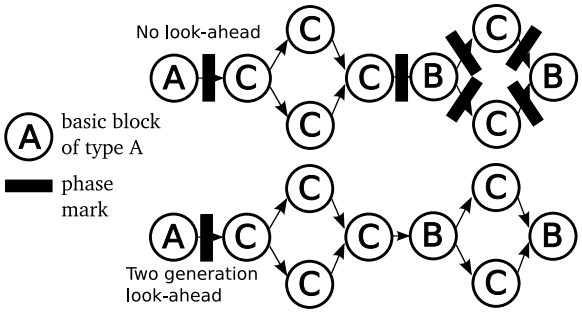

Fig. 5. lookahead based reduction of phase marks

2.1.2.3 *Adding Phase Marks to Attributed Interval Graphs.* Our second class of methods consider a section to be an interval in the attributed interval graph. Using intervals for phase marking enables us to look at the program at a more coarse granularity than basic blocks. Even with 1^{st} order interval graphs, the intervals frequently capture small loops. This is clearly advantageous for adding phase marks since we do not want to have a core

Algorithm 3 : Lookahead based Phase Marking

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

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

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

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

For analyzing the performance characteristics of a section, we measure instructions per cycle (IPC) (similar to [Tam et al. 2007; Becchi and Crowley 2006]). IPC directly correlates to throughput and utilization of performance-asymmetric multicore processors. For example, a core with a high clock frequency can efficiently process arithmetic instructions. However, if the core has a cache miss, it will waste cycles waiting for this data. A core with a lower frequency will waste fewer cycles waiting for data to be retrieved. If a section is being analyzed, its IPC is monitored using hardware performance counters prevalent in modern processors. The optimal core assignment is determined by comparing the observed IPC for each core type.

Our technique for determining optimal core assignment is shown in Algorithm 4. 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.

Algorithm 4 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 $c_i \in C \backslash \{c_n\}$ **do**
  $\theta = f(c_{i+1}, \pi) - f(c_i, \pi)$
  if $\theta > \delta \wedge f(c_{i+1}, \pi) > f(d, \pi)$ **then**
    $d \leftarrow c_{i+1}$
  end if
end for
return d

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.

3. AUTO-TUNING FRAMEWORK

We developed a static analysis and instrumentation framework for phase detection and marking. Compared to similar static instrumentation tools such as Intel ATOM [Srivastava and Eustace 1994], binaries instrumented with our tool execute in less than one tenth of the time taken by binaries generated with ATOM[4]. The low overhead of instrumentation with our framework further reduces the overhead of our phase-marks. By instrumenting binaries rather than source code, we do not require any modifications to compilers. Our framework is based on the GNU Binutils. An overview of this framework and a breakdown of its components is shown in Figure 6. To perform core switches, we use the standard process affinity API available for Linux kernels (ver. $\geq$ 2.5). To dynamically monitor the performance of code sections, we use the Performance Application Programming Interface (PAPI) [Dongarra et al. 2003]. PAPI provides an interface to control and access information gathered by the processor hardware performance counters.

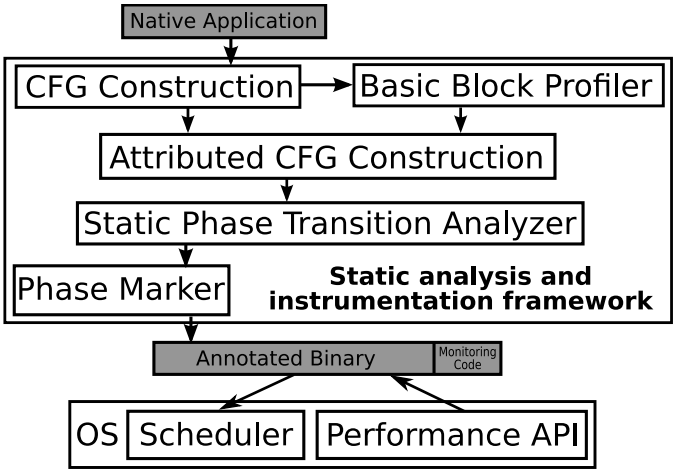

Fig. 6. Framework Components

To monitor the current section's performance we use two events made available by PAPI: instructions retired and cycles. These two events allow us to calculate the IPC (instructions retired / cycles) which we use to determine the actual runtime characteristics of representative phases. Unfortunately, many CPUs only make two performance counters available. With only two performance counters, if one program is monitoring its performance using PAPI, other programs that wish to monitor performance must wait. However, our approach requires very little dynamic monitoring. Additionally, performance is only monitored for a very small section of code. Therefore, processes seldom have to wait for the availability

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

of the counters. In the event that a process must wait, the wait time is negligible since the amount of code needed to monitor is small. Because of this, performance is not impacted significantly by the need to wait for the counters to be available.

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

4. EVALUATION

The aim of this section is to evaluate our five claims made in Section 1. First, we claimed that phase-based tuning is programmer oblivious (it requires no knowledge of program behavior or performance asymmetry). Our technique is completely automatic and requires no input from the programmer. In our experiments, workloads are generates randomly and without any knowledge of behavior of the benchmarks. Second, we claimed the technique allowed for transparent deployment. Since our analysis and instrumentation framework operates on binaries, no modification to compilers is necessary. Furthermore, since we use standard techniques for switching cores, no OS modifications are necessary. For example, in our experiments, we use the standard build scripts and compilers for the SPEC CPU benchmarks and an unmodified Linux OS. Third, we claim that with our technique you can "tune once and run anywhere". Our static analysis makes no assumptions about the underlying asymmetry. Since our performance analysis and 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.

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

4.1 Experimental Setup

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

4.1.0.5 *System Setup* .. Our setup consists of a performance-asymmetric multicore processor containing 4 cores. This setup is emulated using an Intel Core 2 Quad processor with a base clock frequency of 2.4GHz and two cores under-clocked to 1.6GHz. In this processor, there are two L2 caches shared by two cores each. So, the two cores running at 2.4GHz share an L2 cache and the cores running at 1.6GHz share L2 cache. We use the Fedora distribution of Linux with an unmodified kernel (version 2.6.22) and standard compilers. Thus, we demonstrate the transparent deployment benefit of our approach. We use the perfmon2 monitoring interface [Eranian 2006] to measure the throughput of entire workloads using pfmon.

There are two main benefits of using a physical system instead of a simulated system. First, porting our implementation to another system is trivial since we do not require any

modifications to the standard Linux kernel. Second, we analyze our approach in a realistic setting. Others have argued that results gathered through simulation may be inaccurate if not carried out on a full system simulator [Mathur and Cook 2003]. This is because all aspects of the system are not considered. Therefore, a full system simulator is desired. This setup is limited in hardware configurations to test. However, we believe this platform is sufficient to show the utility of our approach.

4.1.0.6 *Workload Construction* .. Similar to Kumar *et al.*[Kumar et al. 2004] and Becchi *et al.*[Becchi and Crowley 2006] our workloads range in size from 18 to 84 randomly selected benchmarks from the SPEC CPU 2000 and 2006 benchmark suites. The entire SPEC CPU 2000 and 2006 benchmark suite is used for random selection. 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.*[Kumar et al. 2004] 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 queue was used for each experiment. This ensures that we more accurately capture the behavior of an actual system.

4.2 Space and Time Overhead

During our static analysis, we insert phase marks in the program to prepare it for phase-based tuning. A phase mark consists of data and code. Since insertion of large chunks of code may destroy locality in the instruction cache, low space overhead is desired. This section first describes the overhead in terms of the increase in binary size caused by insertion of phase marks. Furthermore, a phase mark's execution time is added to the execution of the original program. If such execution time is high, it is likely to overshadow the gains achieved by our technique. Thus, a low time overhead is also desired. Therefore, the time overhead is described in terms of increase in execution time over the uninstrumented version.

4.2.0.7 *Space Overhead* .. To measure the space overhead, we compared the size of the original binary and modified binary for several variations of our technique. Table I shows summary statistics and Figure 7 shows a box plot for the measurements taken from the benchmarks in the SPEC CPU 2000 and 2006 benchmark suites. In this figure, the box represents the range of the two inner quartiles and the line extends to the minimum and maximum points. These results are presented in terms of what percentage of the instrumented application is made up of phase marks. The trends are expected and fairly clear from the figure. As the minimum size increases the space overhead decreases. Similarly, as the lookahead depth increases the space overhead generally decreases. For individual programs this is not always the case because by adding another depth of lookahead, the percentage of blocks belonging to the same type may be pushed over the threshold causing

another insertion point.

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

Table I. Space overhead of phase marks: BB[n,m]: basic block technique with min. block size: n, lookahead: m. Int[n]: interval technique with min. interval size: n.

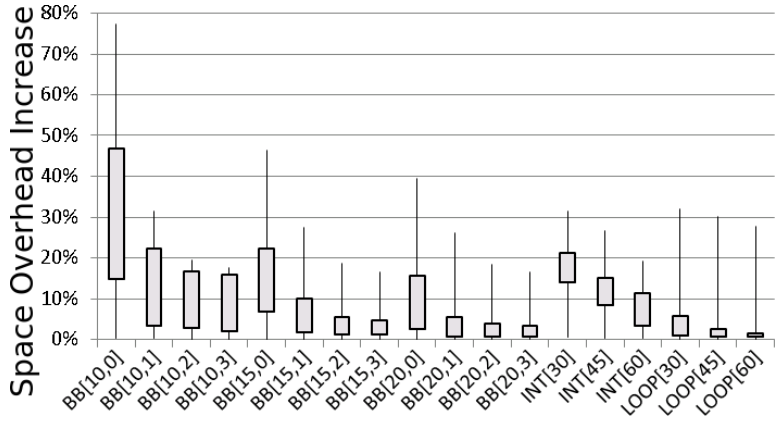

Fig. 7. Space overhead

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

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

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

Table II. Time spent in phase marks

4.2.0.8 *Time Overhead* .. To measure the time overhead (inserted phase marks and core switches), instead of switching to a specific core, we switch to "all cores" allowing the stock Linux scheduler to handle scheduling. Switching to "all cores" means that we still make the standard API calls that our instrumented programs make (`sched_setaffinity`), however, for the parameter defining which cores the process can run on, 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 points in the program. Table II shows these costs for variable workload sizes. Figure 8 shows results for workloads of size 84.

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

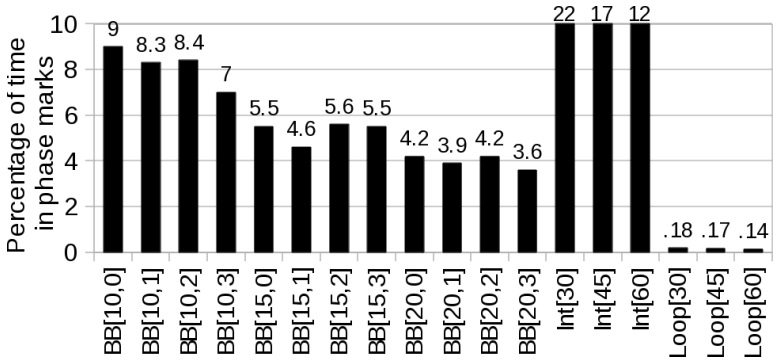

Fig. 8. Time overhead: workload size 84

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

While our current implementation performs well for multi-core systems, there is a potential scalability issue for many-core machines. For example, if we have 100 cores with unknown types, we would have to monitor for each core in the system for each cluster. To address this issue we propose the following. First, we expect that for asymmetric many-core systems, the number of core types is still likely to be small [5]. Thus, if we know a grouping of the cores into types we can largely reduce the problem to one similar to a multi-core system thus avoiding this scalability issue. There are several options for producing such a grouping. One option is to manually determine it based on the processor specification. Another option is to automate the process using carefully written test programs which are run before jobs are sent to the system. These programs will monitor their performance on several cores and determine a grouping of the cores. Care must be taken to ensure that the test programs expose all potential differences which may impact performance.

4.3 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. As mentioned previously in Section 4.1, workloads consist of a fixed number of processes running simultaneously. For all figures presented in this section, the data is taken from the first 400 seconds of the

[5]Previous work seems to suggest situations where as few as two cores types is sufficient [Kumar et al. 2005; Grochowski et al. 2004].

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 somewhat. 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.

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

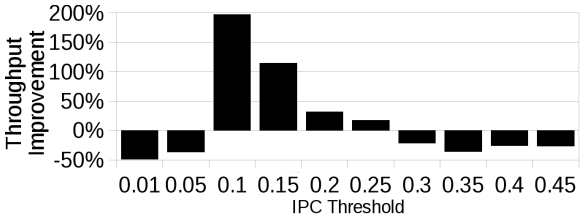

Fig. 9. Throughput improvement: Basic block strategy, min. block size: 15, lookahead depth: 0, variable IPC threshold

4.3.0.9 *IPC threshold* .. First, we want to see how the IPC threshold affects throughput. As mentioned in Section 3, IPC threshold is used to determine the section-to-core assignment. Figure 9 shows how different threshold values affect throughput when all other variables are fixed (technique, min. size, lookahead, etc).

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

4.3.0.10 *Lookahead depth* .. Next, we examine the lookahead depth for the basic block technique. Figure 10 shows how lookahead affects throughput when other variables are fixed.

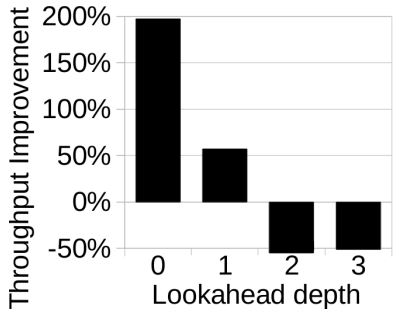

Fig. 10. Throughput improvement: Basic block strategy, min. block size: 15, variable lookahead

This data shows that lower lookaheads generally result in higher throughput. This is because when making decisions at a more fine granularity, the program code is closely matched to ideal cores throughout more of the programs execution. However, this improvement is at the cost of increased overhead which was discussed in Section 4.2. Furthermore, for large lookahead depths, we can see a decrease in throughput. This is because when we begin to ignore large sections of execution; these ignored sections are frequently assigned in an inefficient way.

4.3.0.11 *Clustering error* .. Since a static technique for determining similarity is likely to be inaccurate, Figure 11 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 9 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.

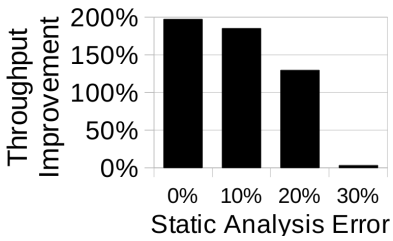

Fig. 11. Throughput improvement: Basic block strategy, min. block size: 15, lookahead depth: 0, variable error

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 almost no performance improvement.

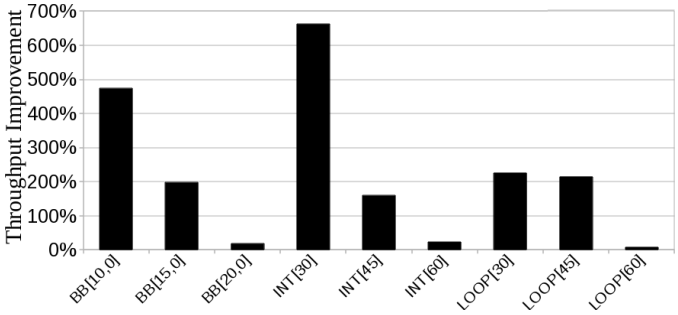

Fig. 12. Throughput improvement: variable technique and minimum size

4.3.0.12 *Minimum instruction size* .. Now, we examine how minimum instruction size affects throughput for all three techniques. Figure 12 shows this comparison. The results for minimum instruction size are similar to those for lookahead. Considering smaller blocks and intervals generally results in higher throughput. This is for the same reasons as lookahead depths, however, with larger minimum instruction size we may ignore small loops that are executed frequently. As mentioned previously, this improvement must be balanced with overhead costs which were discussed in Section 4.2.

4.4 Fairness

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

—max-flow,

—max-stretch, and

—average process time.

Max-flow and max-stretch were developed by Bender *et al.* for determining fairness for continuous job streams [Bender et al. 1998]. We now briefly define max-flow and max-stretch. For each process, we have the following data:

—a_i: arrival time of process i,

—C_i: completion time of process i, and

—t_i: processing time of process i (in isolation).

First, *max-flow* is defined as

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

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

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

This can be thought of as the largest slowdown of a job. 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 III.

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

—12.04% decrease in max-flow,

—20.41% decrease in max-stretch, and

—30.95% average decrease in process completion time.

These results were gathered over a 800 second time interval using the loop based technique with minimum size of 45 and IPC threshold of 0.15. For these same variables, we see 215% improvement in throughput (as measured in Section 4.3).

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 III. Fairness Comparison to standard Linux assignment: Improvements are shaded.

4.5 Analysis of Trade-offs

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

4.5.0.13 *Speed vs. Fairness* .. First, we examine the trade-off between speed and fairness. Speedup refers to the average process run-time. Max-stretch is used for fairness. We expect a positive correlation between the two with some exceptions. These exceptions occur when many processes finish quickly while few starve. Figure 13 shows this trade-off for different variations of our technique.

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

4.5.0.14 *Throughput vs. Fairness* .. Next, we examine the trade-off between throughput and fairness. This is important to consider since frequently the goal is to either maximize one of these or to find a reasonable balance. Intuitively, we expect a somewhat negative correlation between the two. This is because very high improvements in throughput are likely to be at the cost of some process starvation. Figure 14 shows this comparison.

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

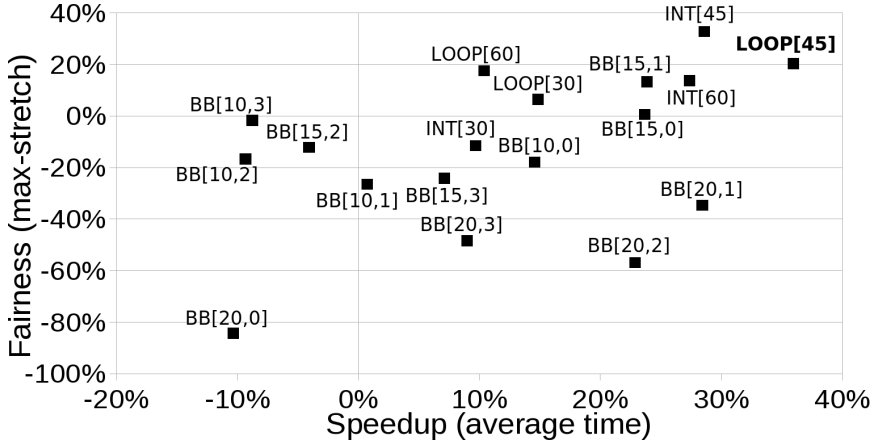

Fig. 13. Speedup vs Fairness: average time vs. max stretch

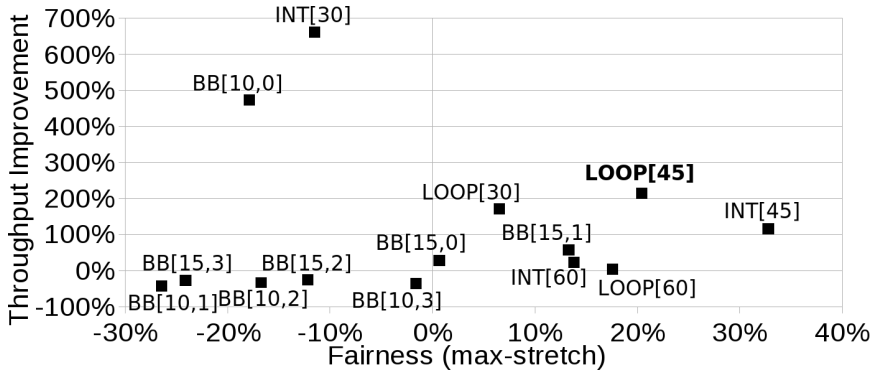

Fig. 14. Throughput vs Fairness (max-stretch)

4.5.0.15 *Speedup vs. Throughput* .. Now, we examine the trade-off between speedup and throughput. Since speedup is somewhat correlated with fairness, the trends are somewhat similar to the previous comparison of throughput and fairness. These two are different in that speedup is degraded by long starving processes while throughput is not affected by these. Figure 15 examines this trade-off.

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

4.5.0.16 *Summary of Results* .. In closing, our results show that phase-based tuning significantly outperforms the stock Linux scheduler in terms of the throughput obtained on a performance-asymmetric multicore processor, while maintaining fairness and with a negligible overhead in most cases. In particular, our loop based technique balances throughput and fairness significantly well achieving an average process speedup of up to 36%. Our

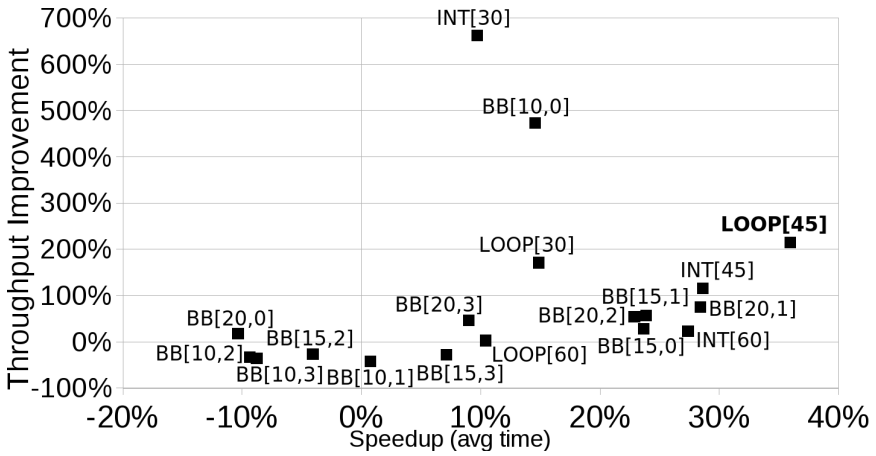

Fig. 15. Speedup (avg. time) vs Throughput

approach thus shows its potential in improving the utilization of performance-asymmetric multicore processors. With recent thrust towards research and development of these processors, the advances in automatic tuning of applications for these processors that we propose are timely and important.

5. RELATED WORK

A preliminary version of this work was presented in our IWMSE workshop paper [Sondag and Rajan 2009]. This work only considered the static analysis at the basic block and intra-procedural first order interval techniques. We have since significantly enhanced our approach to be inter-procedural, to consider loops, and to handle loop nesting, Furthermore, we have considered the balance of fairness, speedup, and throughput as an important factor. This realization has shaped much of the analysis in this paper. Somewhat contrary to our preliminary results, after rigorous experiments with much larger data set it turned out that the interval-level techniques were quite superior to the basic block techniques. This further led to the development of the loop-level inter-procedural static analysis.

Besides our previous work, we know of no previous work which has all of the benefits of our technique. There does exist previous work that are related to ours in spirit. This work may be divided into three categories. First, those which aim to improve performance for heterogeneous multicore processors. Second, those which aim to develop algorithms and programming environments for heterogeneous systems. Third, those related to determining phase behavior and those which use phase behavior for dynamic optimizations.

Becchi *et al.* [Becchi and Crowley 2006] proposed a dynamic assignment technique that uses the IPC of a program segments. However, this work focuses largely on ensuring load balance across cores whereas our technique aims to maximize throughput. Another technique which focuses on load balancing in the scheduler was proposed by Fedorova *et al.* [Fedorova et al. 2007]. They make the case that a core assignment must be balanced. Shelepov and Fedorova [Shelepov and Fedorova 2008] propose a technique which does not require dynamic monitoring. They estimate cache misses in order to estimate the run-time difference between core types in their architecture. This technique does not consider the changes in behavior a program will make throughout its execution. Li *et al.* [Li et al. 2007]

also focus on load balancing in the OS scheduler. They modify the OS scheduler based on the asymmetry of the cores. While this produces an efficient system, the scheduler will need some knowledge of the underlying architecture. Our work differs from these in the following way. First, we are not directly concerned with load balancing. Second, we focus on properly scheduling the different phases of a programs behavior. Also similar is the work by Tam *et al.* [Tam et al. 2007] which determines thread-to-core assignment based on increasing cache sharing. They use cycles per instruction (CPI) as a metric to improve sharing for symmetric multicore processors. Kumar *et al.* propose a temporal dynamic approach [Kumar et al. 2004]. After certain time intervals, a sampling phase is triggered. After the sampling phase, the system makes a decision regarding the assignment of all currently executing processes. This procedure is carried out throughout the entire programs execution. To reduce the dynamic overhead, we do not require monitoring once assignment decisions have been made. Jiménez *et al.* developed a scheduler aimed at making use of heterogeneity in terms of CPU and GPU [Jiménez et al. 2009]. This work is targeted at a specific type if heterogeneous multicores with different ISAs whereas we focus on single ISA asymmetric multicores. Since they focus on cores with different ISAs they require the programmer to note which functions can be executed on both core types or provide implementations for both core types.

A wide range of research has been done on algorithms for optimal distribution of computation over heterogeneous networks of computers. This work can be divided into two categories: approaches for finding an optimal distribution that assume that the processor characteristics and the program characteristics are known [Lastovetsky and Reddy 2004; Renard et al. 2003] , and techniques that allow programmers to specify the program's characteristics and use this input to obtain an optimal distribution [Lastovetsky and Reddy 2006]. Instead of working with a heterogeneous network of computers, we focus on heterogeneous multicores. The main issue with these two classes of ideas is that currently the level of specifications expected from the programmer is high, which significantly increases the intellectual burden of the task, which in turn impairs its practicality. Our technique eliminates this burden.

There is a large body of work on determining phase behavior [Sherwood et al. 2002; Duesterwald et al. 2003], using phase behavior to reduce simulation time [Balasubramonian et al. 2000; Sherwood et al. 2001; Shen et al. 2004; Lau et al. 2005], guide optimizations [Hu 2005; Nagpurkar et al. 2006; Peleg and Mendelson 2007; Vandeputte et al. 2007], etc. Many of these techniques determine phase information with a previously generated dynamic profile [Sherwood et al. 2002]. As mentioned previously, collecting a this profile requires end-users to develop representative sets of test cases for the program. Other techniques determine phase behavior dynamically [Nagpurkar et al. 2006] Techniques that determine the phase information dynamically do not require this input, however, they are likely to incur dynamic overheads. We conduct much of our analysis statically followed by limited analysis dynamically.

6. DISCUSSION

This section discusses some issues that although not central to our work, are relevant.

6.1 Applicability to Multi-threaded Programs

A potential threat to this experimental setup is the absence of multi-threaded benchmarks. However, the simplicity of our approach allows it to immediately work on multi-threaded

applications. First, recall that the framework modifies binaries 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 applications. The framework will have analyzed this code and modified it as needed. Thus, each thread will contain the necessary code switching and monitoring code present in the phase marks.

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

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

6.2 Changing Application and Core Behavior

As we have pointed out previously, the workload on a system may change the perceived characteristics of the individual cores. Furthermore, program behavior may change itself periodically (ex: warm-up phase). While not addressed explicitly in this paper, solutions for these problems only require minor modifications to the techniques presented in this paper. We now give a brief discussion of how to handle such problems.

Dealing with changing program behavior is trivial. For example, a warm-up phase is usually caused by one of two things. First, different code may account for the warm-up phase such as an I/O section of code before the computational portion as in Figure 1. This is the easier case and is already handled by our technique since these two phases are in separate code sections. The second cause is best illustrated by considering compulsory (or cold) cache misses. For example, the first time some code is executed, the data it operates on is not in cache. However, upon each successive execution, the data is likely in the cache. This case is easily handled with our technique by simply ignoring the first execution of each block.

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

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

6.3 Floating Point Emulation and IPC

Using IPC as a metric for performance poses a few problems. For example, a core without a floating point unit may do floating point emulation which will result in many potentially

fast instructions being executed in place of one slow floating point instruction. As a result, the IPC of a floating point intense section of code running on a core with floating point emulation may give a high IPC. However, it is clear that a core with a floating point unit is more desirable. To address this issue, calculation of IPC can be done as follows. Cycles for a section can still be determined using performance counters. Then to determine instruction count, a combination of static analysis and code instrumentation can be used to embed the number of instructions (determined by counting actual floating point instructions not emulated instructions) into the code for calculating IPC. In this case, care must be taken to avoid excessive overhead of the instrumentation code.

7. CONCLUSION AND FUTURE WORK

Performance-asymmetric multicore architectures are an important class of processors that have been shown to provide nice trade-off between the die size, number of cores on a die, performance, and power [Gillespie 2008; Kumar et al. 2005; Mogul et al. 2008]. Devising techniques for their effective utilization is an important software engineering problem that influences the eventual uptake of this class of processors [Li et al. 2007; Mogul et al. 2008]. The need to be aware of, and optimize based on, the applications' characteristics and the nature of multicore processor significantly increases the intellectual burden on the software developer. Furthermore, the need to create separate versions for each target architecture decreases reusability and creates a maintenance burden. In this work, we have shown that phase-based tuning solves all of these problems by utilizing the phase behavior that is common in programs. Phase-based tuning is fully automatic, can be deployed in existing tool chains, and produces asymmetry-independent binaries. This significantly reduces the expertise necessary for programming performance-asymmetric multicores. Apart from these software engineering benefits, phase-based tuning also has several performance advantages. Our experiments show a 36% reduction in the average process time when compared to the stock Linux scheduler. This is all done while incurring negligible overheads (less than 0.2% time overhead) and maintaining fairness.

Future work involves extending phase-based tuning in several directions. First, we would like to study different configurations of performance asymmetric multicores. 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 of section-to-core assignments.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and attendees of the IWMSE '09 workshop for their comments and suggestions. Authors were supported in part by the US National Science Foundation under grants CNS-06-27354, CNS-08-08913 and CCF-08-46059.

REFERENCES

ALLEN, F. E. 1970. Control flow analysis. In *Symposium on Compiler optimization*. 1–19.

BALASUBRAMONIAN, R., ALBONESI, D., BUYUKTOSUNOGLU, A., AND DWARKADAS, S. 2000. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *International symposium on Microarchitecture*. 245–257.

BECCHI, M. AND CROWLEY, P. 2006. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF*. 29–40.

BENDER, M. A., CHAKRABARTI, S., AND MUTHUKRISHNAN, S. 1998. Flow and stretch metrics for scheduling continuous job streams. In *Annual symposium on Discrete algorithms*. 270–279.

BONETI, C., GIOIOSA, R., CAZORLA, F., CORBALAN, J., LABARTA, J., AND VALERO, M. 2008. Balancing hpc applications through smart allocation of resources in mt processors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. 1 –12.

BONETI, C., GIOIOSA, R., CAZORLA, F. J., AND VALERO, M. 2008. A dynamic scheduler for balancing hpc applications. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, Piscataway, NJ, USA, 1–12.

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

DONGARRA, J., LONDON, K., MOORE, S., MUCCI, P., TERPSTRA, D., YOU, H., AND ZHOU, M. 2003. Experiences and lessons learned with a portable interface to hardware performance counters. In *PADTAD Workshop*.

DUESTERWALD, E., CASCAVAL, C., AND DWARKADAS, S. 2003. Characterizing and predicting program behavior and its variability. In *PACT '03: Parallel Architectures and Compilation Techniques*.

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

FEDOROVA, A., VENGEROV, D., AND DOUCETTE, D. 2007. Operating system scheduling on heterogeneous core systems. In *First Workshop on Operating System Support for Heterogeneous Multicore Architectures*.

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

GROCHOWSKI, E., RONEN, R., SHEN, J., AND WANG, H. 2004. Best of both latency and throughput. In *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*. 236–243.

HU, S. 2005. Efficient adaptation of multiple microprocessor resources for energy reduction using dynamic optimization. Ph.D. thesis, The University of Texas at Austin.

HUANG, M. C., RENAU, J., AND TORRELLAS, J. 2003. Positional adaptation of processors: application to energy reduction. *Computer Architecture News 31,* 2, 157–168.

JIMÉNEZ, V. J., VILANOVA, L., GELADO, I., GIL, M., FURSIN, G., AND NAVARRO, N. 2009. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*. Springer-Verlag, Berlin, Heidelberg, 19–33.

KUMAR, R., TULLSEN, D. M., AND JOUPPI, N. P. 2006. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. 23–32.

KUMAR, R., TULLSEN, D. M., JOUPPI, N. P., AND RANGANATHAN, P. 2005. Heterogeneous chip multiprocessors. *Computer 38,* 11, 32–38.

KUMAR, R., TULLSEN, D. M., RANGANATHAN, P., JOUPPI, N. P., AND FARKAS, K. I. 2004. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04: 31st annual international symposium on Computer architecture*. 64.

LASTOVETSKY, A. AND REDDY, R. 2004. Data partitioning with a realistic performance model of networks of heterogeneous computers. In *International Parallel and Distributed Processing Symposium*.

LASTOVETSKY, A. AND REDDY, R. 2006. HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. *J. Parallel Distrib. Comput. 66*, 197–220.

LASTOVETSKY, A. L. AND DONGARRA, J. J. 2009. *High Performance Heterogeneous Computing*. John Wiley & Sons, Inc.

LAU, J., PERELMAN, E., AND CALDER, B. 2006. Selecting software phase markers with code structure analysis. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*.

LAU, J., SCHOENMACKERS, S., AND CALDER, B. 2005. Transition phase classification and prediction. In *11th International Symposium on High-Performance Computer Architecture (HPCA)*. 278–289.

LEE, E. AND MARKUS, L. 1967. *Foundations of Optimal Control Theory*. MINNEAPOLIS CENTER FOR CONTROL SCIENCES.

LI, T., BAUMBERGER, D., KOUFATY, D. A., AND HAHN, S. 2007. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC '07: conference on Supercomputing*. 1–11.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the conference on Programming language design and implementation*. 190–200.

MATHUR, W. AND COOK, J. 2003. Towards accurate performance evaluation using hardware counters. In *ITEA Modeling and Simulation Workshop*.

MOGUL, J. C., MUDIGONDA, J., BINKERT, N., RANGANATHAN, P., AND TALWAR, V. 2008. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro 28,* 3, 26–41.

MUCHNICK, S. S. 1997. *Advanced Compiler Design & Implementation*. Academic Press.

NAGPURKAR, P., KRINTZ, C., HIND, M., SWEENEY, P. F., AND RAJAN, V. T. 2006. Online phase detection algorithms. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*. 111–123.

PELEG, N. AND MENDELSON, B. 2007. Detecting change in program behavior for adaptive optimization. In *PACT '07: 16th international conference on Parallel architectures and compilation techniques*. 150–162.

RENARD, H., ROBERT, Y., AND VIVIEN, F. 2003. Static load-balancing techniques for iterative computations on heterogeneous clusters. *The 9th International European Conference on Parallel and Distributed Computing (Euro-Par 2009)*, 148–159.

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

SHEN, X., ZHONG, Y., AND DING, C. 2004. Locality phase prediction. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. 165–176.

SHERWOOD, T., PERELMAN, E., AND CALDER, B. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: 10th international conference on Parallel architectures and compilation techniques*. 3–14.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. 45–57.

SONDAG, T. AND RAJAN, H. 2009. Phase-guided thread-to-core assignment for improved utilization of performance- asymmetric multi-core processors. In *IWMSE*.

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

TAM, D., AZIMI, R., AND STUMM, M. 2007. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. *Operating Systems Review 41,* 3, 47–58.

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

VANDEPUTTE, F., EECKHOUT, L., AND BOSSCHERE, K. D. 2007. Exploiting program phase behavior for energy reduction on multi-configuration processors. *Journal of Systems Architecture 53,* 8.

WALLACE, S. AND HAZELWOOD, K. 2007. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*. 209–220.

WILLIAMS, S., SHALF, J., OLIKER, L., KAMIL, S., HUSBANDS, P., AND YELICK, K. 2006. The potential of the cell processor for scientific computing. In *Conference on Computing frontiers (CF)*. 9–20.