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

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

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

## Abstract

*CPU vendors are starting to explore trade offs between die size, number of cores on a die, and power consumption leading to performance asymmetry among cores on a single chip. For efficient utilization of these performance-asymmetric multi-core processors, application threads must be assigned to cores such that the resource needs of a thread closely matches resource availability at the assigned core. This significantly complicates the task of an average programmer. The contribution of this work is a technique for automatically determining the mapping between threads and performance-asymmetric cores of a processor. Our approach, which we call* phase-guided thread-to-core assignment*, builds on a well-known insight that programs exhibit phase behavior. We first take code sections and group them into clusters such that each section in a cluster is likely to exhibit similar runtime characteristics. The key idea is that with this clustering, characteristics of a small number of representative sections in a cluster give insight into the behavior of the entire cluster. Thus the exhibited characteristics of the representative sections on different types of cores can be used for automating thread-to-core assignment at a lower runtime cost. Variations of our technique show up to an average 150% improvement in throughput over the stock Linux scheduler for systems with a constant feed of jobs, while maintaining comparable fairness and efficiency.*

## 1. Introduction

CPUs with multiple cores have become commodity items [12]. CPU vendors are projecting that in the next decade the number of cores in a CPU will increase to as many as hundreds [29]. This makes it important to devise techniques for their effective utilization. Recently both CPU vendors and researchers have advocated the need for a class of multi-core processors called performance-asymmetric or heterogeneous multi-cores [3, 4, 13, 27, 35, 17]. All cores in a performance-asymmetric multi-core processor support the same instruction set, however, they differ in terms of performance characteristics such as clock frequency, cache size, etc [13, 26, 17]. These architectures have been shown to provide an effective trade-off between performance, die area, and power consumption compared to homogeneous multi-core processors [13, 27, 35, 17].

To effectively utilize performance-asymmetric multi-core processors, application threads must be executed on cores such that the resource requirements of a thread closely matches the resources provided by the core. This must be done while maintaining fairness between threads. For example, Kumar *et al.* [28] have shown that when workload characteristics are matched well to heterogeneous cores, performance gains of up to 40% are observed (similar results have been shown by Li *et al.* [35]).

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

The main contribution of this work is a technique, which we call *phase-guided thread-to-core assignment*, for matching the resource requirements of a thread to the resources provided by the cores of performance asymmetric multi-core processors. Our technique builds on a well-known insight that programs exhibit phase behavior [14, 15, 20, 22, 30, 34, 36]. By phase behavior we mean that a program goes through phases of execution that show similar runtime characteristics compared to other phases [25, 6, 8, 9, 1, 32]. Based on this insight, our approach consists of two parts. An offline program analysis, which identifies likely *phase-*

*transition points* in a program, and a lightweight dynamic analysis that determines thread-to-core mapping on the fly. We define a phase-transition point as a point in the program where runtime characteristics are likely to change. We use the offline analysis results to generate standalone binaries in which each phase-transition point is instrumented with a tiny fragment for dynamic analysis. This technique does not require any modifications to the operating system. It also does not make any assumptions about the performance characteristics of the target architecture. Thus we avoid the need for multiple versions for each target platform.

We evaluated our approach using workloads constructed from the SPEC CPU200 benchmark suite. Both overheads and improvement in throughput are measured. For these workloads, we observe significant improvement in overall throughput when compared to the assignment strategy of the stock Linux scheduler with only minor overheads.

## 2 Phase-guided Thread-to-core Assignment

A program exhibits phase behavior [25, 6, 8, 9, 1, 32] in that it goes through several phases of execution that show similar runtime characteristics compared to other phases of execution. If we can classify a program's execution into code sections; group these sections into clusters such that all sections in the same cluster are likely to exhibit similar runtime characteristics; the actual runtime characteristics of a small number of representative sections in the cluster are likely to manifest the behavior of the entire cluster.

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

Based on these intuitions, phase-guided thread-to-core assignment works as follows. First, an offline analysis is performed to identify phase-transition points. This analysis proceeds as follows. First, we divide a program's code into *sections*. Second, we classify these sections into one or more *phase types* thereby clustering them into one or more groups such that each section in the cluster is likely to exhibit similar runtime characteristics. Third, we identify points in the program where the control flows [2] from a section of one phase type to a different phase type. These points are identified as phase-transition points.

Each phase-transition point is statically instrumented to insert a small code fragment, *phase mark*. The idea of phase marking is similar to the work by Lau *et al.* [19], however, we do not use a program trace to determine our phase marks and make our selections based on a different criteria. A phase mark contains information about the phase type for the current section, performs dynamic performance analysis, and makes core switching decisions. At runtime the phase marks analyze the performance of a small number of representative sections of each phase type. These analysis results are used to determine a suitable core mapping for the phase type such that the resources provided by the core matches the expected resources for sections of that phase type. On determining a satisfactory mapping for a phase type, all future phase marks for that phase type reduce to simply making appropriate core switching decisions. Thus, the actual characteristics of few representative sections of a given type are used as an approximation of the expected characteristics of all sections of that phase type. The rest of this section describes components of our approach in detail.

### 2.1. Offline Phase Transition Analysis

The aim of our offline phase transition analysis is to determine points in the control flow where its phase behavior is likely to change (*phase-transition points*). The precision and the granularity of identifying such points is likely to determine the performance gains observed at runtime. To that end, the first step in our analysis is to detect similarity among basic blocks in the entire program and to classify them into one or more types that are likely to exhibit similar runtime behavior. We then do an intra-procedural analysis that uses the results of the basic block analysis to summarize intervals [2] into a single type. The result of the basic block analysis and summarization is used to construct an inter-procedural control flow graph, which is used to detect and mark phase transitions with *phase marks*.

**Attributed Control Flow Graph Construction** Our offline analysis first divides a program into procedures ($\mathcal{P}$) and each procedure $p \in \mathcal{P}$ into basic blocks to construct the set of basic blocks ($\mathcal{B}$) [2]. We use the classic definition of a basic block that it is a section of code that has one entry point and one exit point with no jumps in between [2]. We then classify each basic block into exactly one type ($\pi \in \Pi$) to construct the set of attributed basic blocks ($\bar{\mathcal{B}} \subseteq \mathcal{B} \times \Pi$). The notion of type here is different from types in a program and does not necessarily reflect the concrete runtime behavior of the basic block. Rather it suggests similarity between expected behaviors of basic blocks that are given the same type. A strategy for classification of basic block based on execution traces is given in Section 3, however, other methods for basic block classification can easily be used.

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

**Summarizing Intervals** The attributed control-flow graph of a procedure is then partitioned into a unique set of *intervals* ($\mathcal{I}$) using standard algorithms [2]. "An *interval* ($i(\eta) \in \mathcal{I}$) corresponding to a node $\eta \in \mathcal{N}$ is the maximal, single entry subgraph for which $\eta$ is the entry node and in which all closed paths contain $\eta$ [2, pp.6]." For each $i$, we then compute its dominant type by doing a depth-first traversal of the interval starting with the entry node, while ignoring backward control flow edges (marked with $b$) unless traversal gets stuck at a non-leaf node. The exit nodes of the interval represent the leaf nodes. A sample run of this summarization algorithm is illustrated in Figure 1.

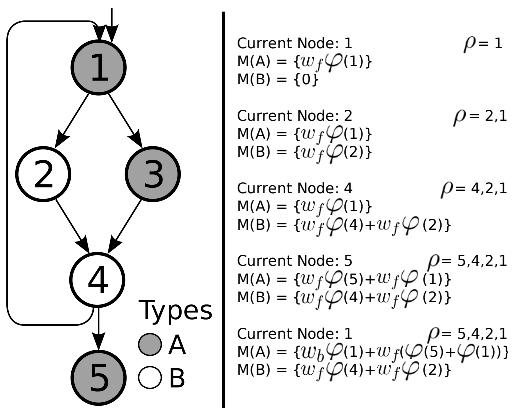


**Figure 1. Interval Summarization Illustration**

During a depth-first traversal we maintain a stack of control flow nodes encountered thus far ($\rho = \eta + \rho'$) with the entry node of the interval at the bottom of this stack and the currently visited node at the top of the stack. A type map for the interval ($M : \Pi \mapsto \mathbb{R}$) is maintained. On visiting a control flow node $\eta$ 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, $\pi$ is the type of the control flow node, $w_f$ is the forward edge weight, $\varphi$ maps nodes to node weights, and $\oplus$ is the overriding operator for finite functions.

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

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

As a result of this process, we obtain another control flow graph of the procedure where nodes are tuples of intervals and their types. To distinguish these from control flow graphs of basic blocks, we refer to them as *attributed interval graphs*. It would be interesting to explore whether summarizing interval graphs again is useful [2], however, in this paper we only consider first-order intervals. Our initial intuition is that the value of applying n$^{th}$ order interval summarization will depend on the average size of procedures.

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

## 2.2. Phase Transition Marking

Once the phase transition analysis is complete, we statically insert phase marks in the binary to produce a standalone binary with phase information and dynamic analysis code fragments. We have considered several variations of phase transition marking that can be broadly classified into two kinds based on whether it operates on the attributed control flow graphs or the attributed interval graphs. In both cases, phase marks are placed at the beginning of a section.

**Adding Phase Marks to Attributed CFG** Our first class of methods all consider a section to be a basic block ($\bar{\beta}$) 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 ($\beta$,$\pi$). Our naïve phase marking technique marks all edges in the attribute CFG where the source and the target sections have different phase types. As is evident, this technique has a problem. The average basic block size in a program is small (tens of instructions). Phase marking at this granularity resulted in frequent core switches overshadowing any performance benefit. To avoid this, we use two techniques.

The first technique only considers sections for marking that are longer than a fixed number of instruction. More generally, if the section has more than a threshold weight as defined by our node weight function, $\varphi : \mathcal{N} \mapsto \mathbb{R}$. This eliminates core switching for very small blocks of code.

The second technique further addresses this problem by only considers a section if at least a fixed percentage of its successors up to a fixed depth have the same type (illustrated in Figure 2). The intuition behind this is the following. If the successors of a section have the same type, it is more likely that a core switch will be worth its cost.

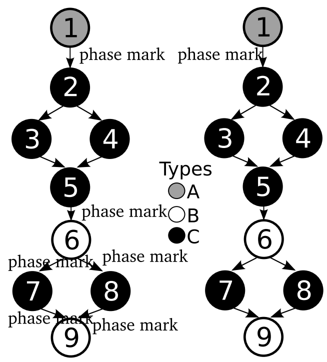


**Figure 2. look-ahead for fewer phase marks**

**Adding Phase Marks to Attributed Interval Graphs** Our second class of methods consider a section to be an interval in the attributed interval graph. Using intervals for phase marking enables us to easily look at the program at a more coarse granularity than basic blocks. For example, even with $1^{st}$ order intervals, the intervals frequently capture small loops. Since we do not want to insert a core switch within a small loop, this is a clear advantage. 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.

### 2.3. Performance Analysis and Scheduling

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

The code for a phase mark serves two purposes: First, during a transition between different phase types, a core switch is initiated. The target for this switch is the core that is previously determined to be an optimal fit for this phase type. Second, if an optimal fit for a given phase type has not been determined previously the current section is monitored to analyze its performance characteristics. The decision about the optimal core for that phase type is made by monitoring representative sections from the cluster of sections that have the same phase type. If our intuition that “all sections that have the same phase type are likely to exhibit similar runtime behavior” holds, the decision about optimal core made by just monitoring few representative sections will be valid for all sections of the same phase type. Thus, monitoring all sections will not be necessary.

For analyzing the performance characteristics of a section, we use instructions per cycle (IPC) as a metric (similar to [7, 5, 33]). IPC directly correlates to throughput and improved utilization of performance-asymmetric multicore processors. The optimal core assignment is determined by comparing the observed IPC for each core type.

Our algorithm for computing optimal core assignment does not require knowledge of the underlying architecture. The intuition behind this algorithm is that cores which execute code most efficiently will waste fewer clock cycles resulting in higher observed IPC. Therefore, these cores will be in highest contention. So, if the difference in observed IPC between two cores is above the threshold, we assume that we will save a large enough number of cycles to make it worth executing on the more efficient core.

## 3. Experimental Setup

Our experimental setup consisted of a performance-asymmetric multi-core processor setup containing 4 cores. We obtain this setup by using an Intel Core 2 Quad processor with a base clock frequency of 2.4GHz and two cores under-clocked to 1.6GHz. This setup is limited in hardware configurations to test. However, this platform shows the utility of our approach. Also, porting our implementation to another system is trivial since we do not require any modifications to the standard Linux kernel. To perform core switches, we used the standard process affinity API available for Linux kernels (ver. $\geq 2.5$).

We developed a static analysis and instrumentation framework for phase detection and marking. This framework is based on the GNU Binutils. To dynamically monitor the performance of code sections, we used the Performance Application Programming Interface (PAPI) [18]. PAPI provides an interface to control and access information gathered by the processor hardware performance counters. We used the perfmon2 monitoring interface [11] to measure the throughput of entire workloads using pfmon.

We are not presenting a static phase approximation technique at this time. Therefore, our experiments use previously determined knowledge of program performance on all core types in the system. This is determined by running each of the programs entirely on each core type and measuring the average IPC of each section of code. We then use look at the *difference* in IPC between the core types and use an *IPC threshold* value to determine the clustering for code sections. For example, suppose we are grouping code sections into two clusters. For a section, if the IPC difference is above the threshold, the section is placed in the first cluster. Otherwise, it is placed in the other cluster.

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

# 4. Experimental Results

Many systems receive a nearly constant feed of jobs to run. Improving the overall throughput of such a system will increase the amount of jobs the machine can complete in an interval of time. This increase will in turn will enable the system to handle larger workload sizes. Our approach is targeting these systems, with maximizing throughput as its key objective. Our hypothesis is that *our technique will improve the throughput of such a system while incurring a small time and space overhead*. The results in this section validate this hypothesis. First, we briefly analyze the time and space overhead of our approach. We then investigate the throughput observed for workloads with phase-guided thread-to-core assignment and compare it with the throughput observed while running the stock Linux scheduler.

## 4.1. Space and Time Overhead

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

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

To measure the space overhead, a comparison between the size of the original binary and modified binary was performed for several variations of our technique. Table 1 shows some of these measurements for a subset of benchmarks in the SPEC2000 benchmark suite. All benchmarks are not shown due to space constraints. These results are expected in that they confirm our intuition that less phase marks will be inserted for larger basic block sizes and look-ahead depths. The results for interval graph-based phase marking are interesting in that they show significantly large increase in binary size. This is primarily because interval summarization results in the grouping of smaller basic blocks into intervals creating more sections above the instruction size threshold.

| Technique | Space overhead of phase marks (in %) | | | | | |
|---|---|---|---|---|---|---|
| | ammp | art | crafty | gzip | mcf | vpr |
| BB[10, 0] | 0.09 | 25.60 | 0.01 | 2.59 | 31.02 | 5.07 |
| BB[10, 1] | 0.09 | 21.68 | 0.01 | 1.94 | 26.79 | 3.41 |
| BB[10, 2] | 0.09 | 21.68 | 0.01 | 1.94 | 23.77 | 2.25 |
| BB[20, 0] | 0.09 | 14.93 | 0.01 | 0.50 | 4.71 | 0.38 |
| BB[20, 1] | 0.09 | 12.50 | 0.01 | 0.42 | 3.83 | 0.27 |
| BB[20, 2] | 0.09 | 12.50 | 0.01 | 0.42 | 4.71 | 0.17 |
| Int[10] | 93.47 | 120.20 | 17.22 | 21.58 | 81.75 | 77.80 |
| Int[25] | 40.41 | 42.21 | 7.61 | 6.56 | 29.74 | 34.21 |
| Int[55] | 16.48 | 19.64 | 3.07 | 1.55 | 4.64 | 12.44 |

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

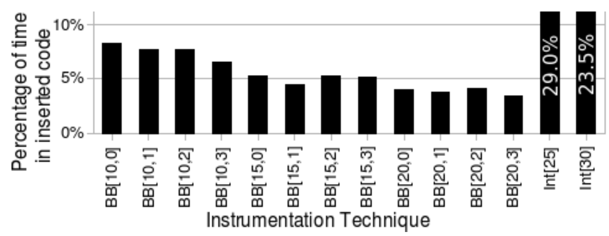


**Figure 3. Time overhead: workload size 84**

To measure the time overhead (inserted phase marks and core switches), instead of switching to a specific core, we switch to "all cores" allowing the stock Linux scheduler to handle scheduling. Thus, the difference in runtime between the unmodified binary and this instrumented binary shows the cost of running our phase marks and core switching at the predetermined points in the program. Figure 3 shows results for workloads of size 84. The trends shown are expected and are similar to those for space overhead. More optimized instrumentation and core switching techniques are likely to decrease this overhead even further.

## 4.2 Throughput

To test our hypothesis that "phase-guided thread-to-core assignment 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 time intervals of execution. Again, we want to improve performance for systems that have a nearly constant feed of processes/requests (e.g. a server). Thus, we maintained a constant number of jobs in the workload in the system for both cases. To achieve this, when a job is completed, another job is immediately given to the system.

Figure 4 shows the observed improvement in throughput

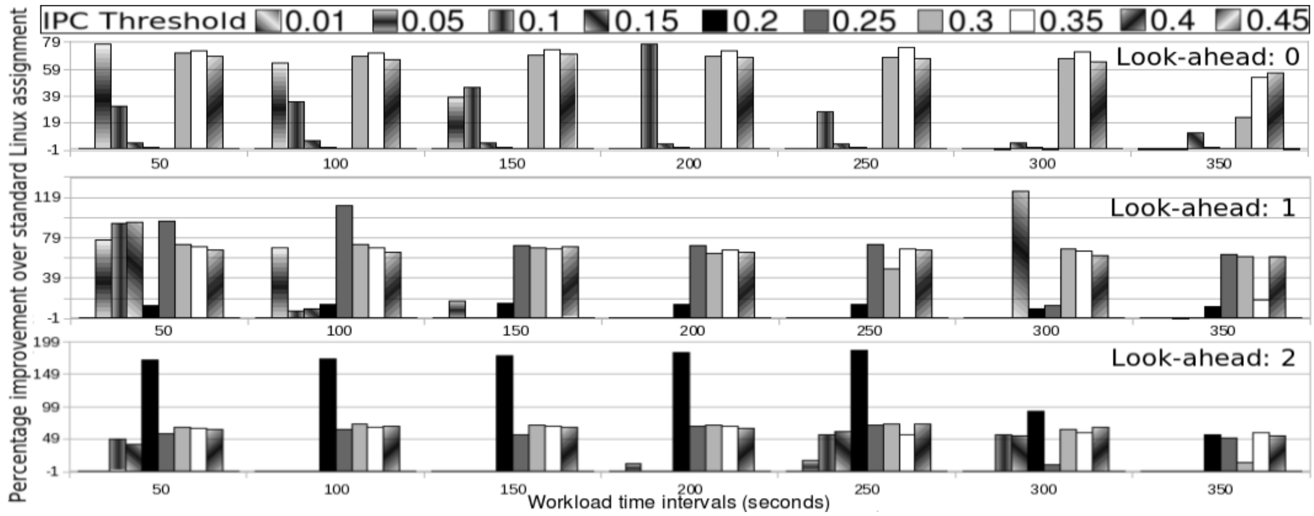


**Figure 4. Throughput improvement: Basic block strategy, min. block size: 10, variable look-ahead**

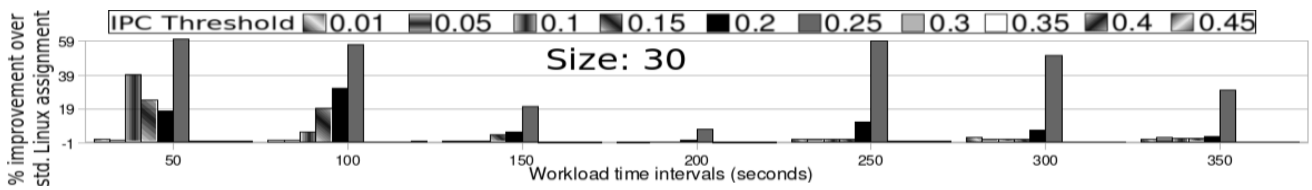


**Figure 5. Throughput improvement: Interval strategy, first order intervals, min. interval size: 30**

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

Figure 5 shows the observed improvement in throughput our technique gives when using the interval strategy for first order intervals. As we observed with basic blocks using look-aheads, for most thresholds, we see less improvement than the techniques which map at a more fine grained level. We also see significantly less improvement than all look-ahead depths which is largely because of the inaccuracy in determining interval types. However, in some cases, we still notice exceptionally high improvement. Again, there is a trade-off with overhead that we previously discussed.

Since a static technique for determining similarity is likely to be innacurate, Figure 6 shows how our technique performs with approximate phase information. We tested the same variables as Figure 4 with a look-ahead of 0 but with error levels of 20% and 30%. To introduce this error, after determining the clustering of blocks, a percentage of blocks were randomly selected and placed into the opposite cluster. These results show that our technique is still quite effective even when presented with approximate block clustering. In some cases, the throughput actually increases over the technique with no error. Since our technique ignores parts of each programs code, this imprecision can improve upon our mapping by better assigning these ignored sections. With extreme thresholds, the error moves processes away from overloaded cores and improves throughput.

Next, to gain insight about the fairness of our scheduling technique, we observe the completion times of benchmarks in the workload. We present a small portion of this data in Figure 7 which was taken from a test using the basic block strategy with a minimum block size of 10 and a look-ahead depth of 2. Results are shown for thresholds of 0.10 and 0.20 (which gave the highest throughput). We can observe that within this time interval roughly the same number of benchmarks are completed. Also, for the 0.20 threshold, we have benchmarks completing in roughly the same fashion.

**Summary** In closing, our results show that phase-guided assignment can significantly outperform the stock Linux scheduler in terms of the throughput obtained on a performance-asymmetric multi-core processor, while main-

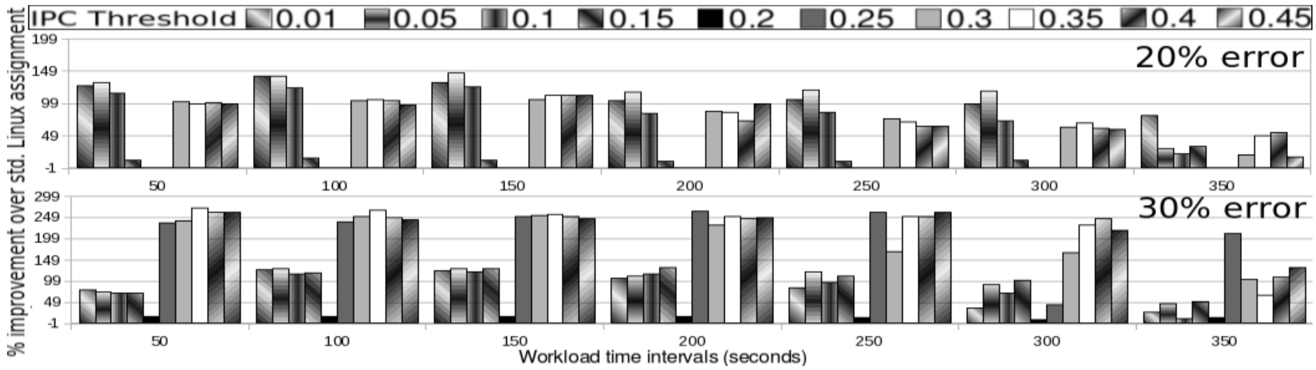


**Figure 6. Improvement: Basic block, min. size: 10, look-ahead: 0, variable phase behavior accuracy**

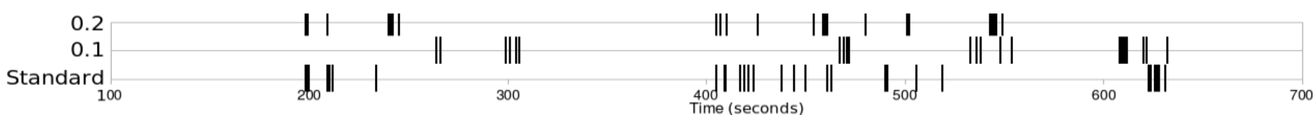


**Figure 7. Process completion times: Basic block, min. size: 10, look-ahead: 2, thresholds: 0.1, 0.2**

taining fairness and with a negligible overhead in most cases. With recent thrust towards research and development of these processors, the advances in thread-to-core assignment that we propose are timely and important.

## 5. Related Work

Our previous work [33], focuses on a static analysis technique to predict phase behavior and identify phase transition points in the program. We now focus on dynamic assignment instead of static analysis. Furthermore, in the previous work, no evaluation was presented. In this paper we show the benefits of our approach via a rigorous evaluation.

Huang *et al.* [16] show that basing processor adaptation on code sections (positional) rather than intervals in time (temporal) results in up to 50-80% improvement in energy reduction. This is similar to our work in that we take a positional approach, however, we do not use subroutines for our code sections. They use knowledge of previous executions of a subroutine to guide future decisions. We also use our idea of similarity to further reduce the dynamic overhead.

Becchi *et al.* [5] proposed a dynamic mapping technique that uses the IPC of a program segments. However, this work focuses largely on ensuring load balance across cores whereas our technique aims to maximize throughput. Also similar is the work by Tam *et al.* [7] which determines thread-to-core mapping based on increasing cache sharing. They use cycles per instruction (CPI) as a metric to improve sharing for symmetric multi-core processors. Kumar *et al.* propose a temporal dynamic approach [26]. After certain time intervals, a sampling phase is triggered. After the sampling phase, the system makes a decision regarding the mapping of all currently executing processes. This procedure is carried out throughout the entire programs execution. To reduce the dynamic overhead, we do not require monitoring once mapping decisions have been made.

Lau *et al.* [19] define the idea of phase markers and propose a technique to determine these markers around procedures and loop boundaries. Our technique is similar, but uses different points for our phase markers. We also determine our phase markers without ever running the program.

There is a large body of work on determining phase behavior [32, 10, 34], using phase behavior to reduce simulation time [25, 8, 31, 1, 34, 30, 20], guide optimizations [15, 14, 22, 21, 23, 24, 36], etc. Most of these techniques determine phase information with a previously generated dynamic profile. As mentioned previously, collecting a this profile requires end-users to develop representative sets of test cases for the program. Techniques that determine the phase information dynamically do not require this input, however, they are likely to incur dynamic overheads. We conduct much of our analysis statically followed by limited analysis dynamically.

## 6. Conclusion

Performance-asymmetric multi-core architectures are an important class of processors that have been shown to provide nice tradeoff between the die size, number of cores on a die, performance, and power [3, 4, 13, 27, 17]. Devising techniques for their effective utilization is an important problem that influences the eventual uptake of this class of processors [35, 17]. Besides phase-guided thread-to-core assignment, we know of two other approaches for improving the utilization of performance-asymmetric multi-core processors: modifying the OS scheduler to account for asymmetry [35, 26] and load balancing to account for performance asymmetry [5]. These techniques require extensions to the operating system whereas phase-guided thread-to-core assignment transparently improves the throughput for performance-asymmetric multi-core processors. A predicted phase behavior and the exhibited execution characteristics of a small set of representative phases is exploited at runtime to determine likely profitable thread-to-core assignments for later phases of the program. By monitoring the execution of only this small representative set instead of monitoring the entire application, our approach reduces the monitoring overhead. Our evaluation shows up to 150% improvement in average throughput compared to the stock Linux scheduler while incurring negligible overheads.

## References


[1] A. Georges *et al.* Method-level phase behavior in java workloads. In *OOPSLA*, 2004.

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

[3] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *ISCA*, June 2005.

[4] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures pp 506. In *ISCA*, June 2005.

[5] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Conference on Computing frontiers (CF)*, pages 29–40, 2006.

[6] J. Cook, R. L. Oliver, and E. E. Johnson. Toward reducing processor simulation time via dynamic reduction of microarchitecture complexity. *Perform. Eval. Rev.*, 30(1), 2002.

[7] D. Tam *et al.* Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *EuroSys*, 2007.

[8] A. Dhodapkar and J. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA*, 2002.

[9] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *MICRO*, page 217, 2003.

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

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

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

[13] M. Gillespie. Preparing for the second stage of multi-core hardware: Asymmetric (heterogeneous) cores. Technical report, Intel Corporation, July 2008.

[14] M. Hock, K. Jayaraman, B. Pellin, and V. Shrivastava. Phase capture and prediction with applications. *Technical Report - Com. Sci. Dept. - University of Wisconsin-Madison*, 2005.

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

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

[17] J. C. Mogul *et al.* Using asymmetric single-isa cmps to save energy on operating systems. *IEEE Micro*, 2008.

[18] J. Dongarra *et al.* Experiences and lessons learned with a portable interface to hardware performance counters. In *PADTAD*, 2003.

[19] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. In *CGO*, 2006.

[20] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *HPCA*, 2005.

[21] M. Merten *et al.* A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *ISCA*, 1999.

[22] P. Nagpurkar, C. Krintz, M. Hind, P. F. Sweeney, and V. T. Rajan. Online phase detection algorithms. In *CGO*, 2006.

[23] N. Peleg and B. Mendelson. Detecting change in program behavior for adaptive optimization. In *PACT*, 2007.

[24] C. Pereira and R. Gupta. Using program phases as meta-data for runtime energy optimization. Technical report, Dept. of Computer Sc. & Eng., UC San Diego, 2004.

[25] R. Balasubramonian *et al.* Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, 2000.

[26] R. Kumar *et al.* Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA*, page 64, 2004.

[27] R. Kumar *et al.* Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[28] R. Kumar *et al.* Core architecture optimization for heterogeneous chip multiprocessors. In *PACT*, 2006.

[29] S. Y. Borkar et al. Platform 2015: Intel processor and platform evolution for the next decade. Technical Report White Paper, Intel Corporation, 2005.

[30] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS-XI*, 2004.

[31] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT*, 2001.

[32] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X*, 2002.

[33] T. Sondag, V. Krishnamurthy, and H. Rajan. Predictive thread-to-core assignment on a heterogeneous multi-core processor. In *PLOS*, Oct. 2007.

[34] R. Srinivasan, J. Cook, and S. Cooper. Fast, accurate microarchitecture simulation using statistical phase detection pp 147. In *ISPASS*, 2005.

[35] T. Li *et al.* Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Conference on Supercomputing*, 2007.

[36] F. Vandeputte, L. Eeckhout, and K. D. Bosschere. Exploiting program phase behavior for energy reduction on multi-configuration processors. *J. Syst. Archit.*, 53(8), 2007.