Trace Based Phase Prediction For Tightly-Coupled Heterogeneous Cores

Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan, Ann Arbor, MI
{shrupad, lukefahr, reetudas, mahlke}@umich.edu

ABSTRACT

Heterogeneous multicore systems are composed of multiple cores with varying energy and performance characteristics. A controller dynamically detects phase changes in applications and migrates execution onto the most efficient core that meets the performance requirements. In this paper, we show that existing techniques that react to performance changes break down at fine-grain intervals, as performance variations between consecutive intervals are high. We propose a predictive trace-based switching controller that predicts an upcoming phase change in a program and preemptively migrates execution onto a more suitable core. This prediction is based on a phase's individual history and the current program context. Our implementation detects repeatable code sequences to build history, uses these histories to predict an phase change, and preemptively migrates execution to the most appropriate core. We compare our method to phase prediction schemes that track the frequency of code blocks touched during execution as well as traditional reactive controllers, and demonstrate significant increases in prediction accuracy at fine-granularities. For a big-little heterogeneous system that is comprised of a high performing out-of-order core (Big) and an energy-efficient, in-order core (Little), at granularities of 300 instructions, the trace based predictor can spend 28% of execution time on the Little, while targeting a maximum performance degradation of 5%. This translates to an increased energy savings of 15% on average over running only on Big, representing a 60% increase over existing techniques.

Categories and Subject Descriptors

C.1.3 [**Architectures**]: Heterogeneous systems

General Terms

Design, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MICRO'46 , December 7-11, 2013, Davis, CA, USA
2013 Copyright is held by the owner/author(s). Publication rights licensed to ACM
ACM 978-1-4503-2638-4/13/12 ...$15.00.

Keywords

heterogeneous processors, fine-grained phase prediction, energy-efficiency

1. INTRODUCTION

An approach to increasing energy efficiency in modern processors is to combine multiple cores with different capabilities into a single heterogeneous processor, yielding varying performance and energy characteristics. Heterogeneous multi-cores trade increased area to provide higher performance and reduced energy consumption by matching an application's performance and energy requirements to the most appropriate core type. Researchers [34, 2, 5, 14, 15, 27] have demonstrated the potential benefits of such designs in terms of realizing more energy efficient systems. Commercially available processors include ARM's big.LITTLE [9], which consists of one high performance out-of-order Big core and a lower performance, but much more energy efficient in-order Little core and Nvidia's Kal-El [21], which combines four high performance and one energy efficient cores.

The energy efficiency attained in heterogeneous systems is determined in part by the efficiency of the scheduling or switching mechanism, that dynamically guides program execution to the most energy efficient core, while maintaining a performance target. Existing techniques are *reactive* in nature - they distinguish performance phase changes in applications by sampling for a brief period and assume that the performance will remain stable until the following sampling phase. Metrics like performance [15], load memory intensity [5, 2], available instruction and memory level parallelism (ILP, MLP) [34], and branch misprediction rates are some of the factors used to evaluate the microarchitectural requirements of an application and thus determine the best fitting core. A reactive controller dynamically monitors performance on the active core at the granularity of an instruction slice or *quantum.* Performance on another core is either monitored by sampling briefly on it [15] or by modeling it using observed performance metrics [18, 34]. This measured or modeled performance on the available core options is compared and contingent upon the desired performance target, the controller maps execution on the most efficient core.

A reactive controller is well suited for traditional heterogeneous systems that make switching decisions at *coarse-grained* application phases of tens to hundreds of millions of instructions. However, better energy efficiency can be achieved by slicing applications into micro-phases of a few hundred instructions. More low-performance phases exist at such fine instruction granularities [20], increasing oppor-

tunities to utilize more energy efficient alternatives. *Fine-grained* phase changes in *general-purpose* programs can be exploited by coupling heterogeneous general-purpose cores in a system that is unencumbered by large transfer overheads between core switches. Recently proposed Composite Cores [18] has implemented a general-purpose architecture that enables switching at a fine granularity. It is composed of two compute *backends* that lie on opposite ends of the energy-performance spectrum, a high-performing *Big* out-of-order backend and an energy efficient *Little* inorder backend. Together, they aim to achieve both high performance and energy efficiency. By sharing the caches, TLB, and pipeline frontends (including branch predictors and the fetch unit), the switching overhead can be reduced to *near zero*. We have assumed the Composite Core architecture for our implementation.

The fundamental assumption of reactive controllers however, that the performance observed during sampling is stable over the subsequent intervals, fails at fine granularities because of the high variation between performance in consecutive quanta. We observe that the average performance difference between consecutive quanta is almost an order of magnitude higher for fine-grain quanta over coarse-grain.

We propose a controller that is able to make accurate phase change predictions at a fine granularity of hundreds of instructions, thus realizing the benefits offered by fine-grained heterogeneous systems more effectively over existing techniques. This work employs a *predictive approach* to scheduling by finding and exploiting repeatable phase behavior in an application, while maintaining a performance loss target. Prior work [28, 29, 16, 33] has shown that programs exhibit repeatable phase-based behavior across different microarchitecture metrics at a coarse granularity. Our goal is to extend that intuition to finer-grained phases of a few hundred instructions. However, direct application of those works at finer granularities is infeasible due to the eccentric nature of fine-grained quanta and large associated computation overheads.

Our proposed controller architecture divides the application's execution into recurring sequences of instructions at run-time, referred to as *super-traces*. A *super-trace* is identified by a combination of backward branches, or *backedges*, that appear together. Backedges are useful in identifying loop and function boundaries in a dynamic instruction stream, which represent regular and recurring control independent blocks of code (˜50 instructions). A *super-trace* captures inter-dependencies between blocks that could affect performance by combining consecutive backedges until a minimum length requirement is met.

We propose a predictive trace-based switching controller for this architecture that can dynamically learn regular fine-grained phase behavior and predict an upcoming phase change based on a phase's individual history and the current program context. This is used to preemptively migrate the execution to a more suitable backend, improving phase-to-backend mappings and increasing energy efficiency.

In summary, this paper offers the following contributions:

- We observe that accuracy of reactive scheduling approaches used in existing heterogeneous architectures decreases at fine switching granularities (hundreds of instructions).
- We leverage the concept of performance micro-phases, and exploit their regular, repeatable behavior to develop a predictive scheduling technique.
- We define a *super-trace* that can capture program phases effectively at the granularity of hundreds of instructions. We design a complexity effective hardware controller to predict *both* which *super-trace* is likely to be executed next in the dynamic instruction stream *and* which core is more suitable to execute it.
- We compare the accuracy achieved by our proposed scheme over existing methods of phase detection, such as that implemented by Sherwood et. al [31], and instruction quantum based approaches [18].
- We analyze the proposed system with cycle accurate full system simulations on a fine-grain heterogeneous multicore [18]. Overall, with a performance degradation constraint of 5%, our phased-based predictive approach to scheduling maps an average of 28% of program execution time onto the little backend, nearly 50% more than what an existing state-of-art design can achieve. The overall energy consumption is reduced by 15% as compared to full execution on the big core and by 60% over existing techniques.

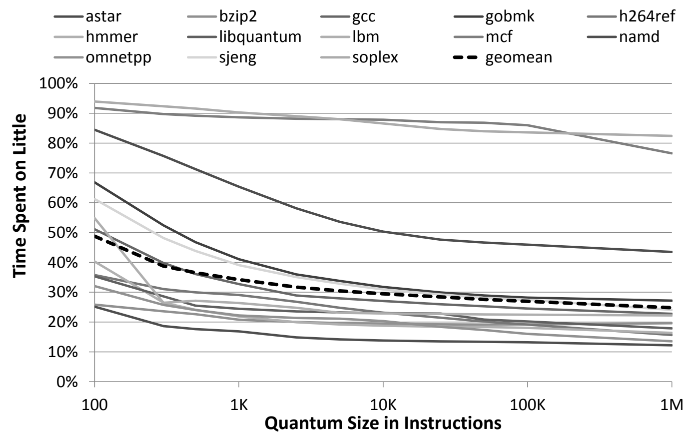


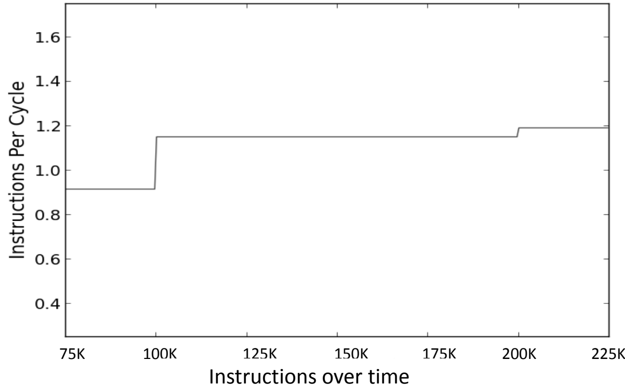
Figure 1: Potential increase in time spent on the little core with reduction in quantum size

2. MOTIVATION

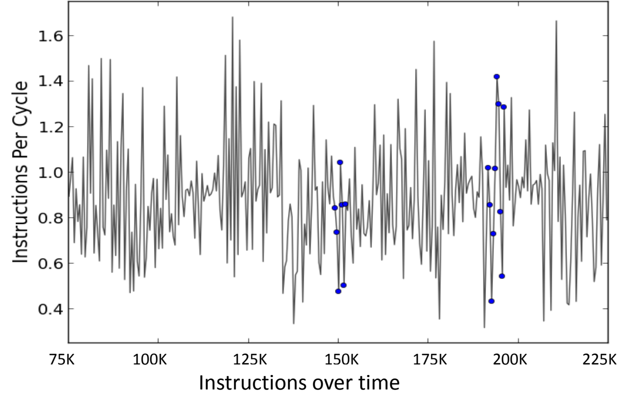
Switching at fine granularities exposes additional opportunities for energy savings. However, sampling or quantum based reactive approaches to scheduling used in existing designs lead to less efficient phase to core mapping, especially as the granularity of switching is reduced.

2.1 Fine-Grained Switching

Traditional single-ISA heterogeneous multicore systems rely on coarse-grained switching to exploit application phases that occur at a granularity of hundreds of millions to billions of instructions [34]. These phases usually represent distinct application tasks with stable average performance levels throughout their execution [30]. Discrete performance changes are observed between phases and simple sampling-based monitoring systems can recognize low-performance phases and map them to a more energy-efficient core. These long term low-performance phases occur infrequently in many applications however, limiting the potential to utilize a more efficient core. Conversely, several works [23, 35, 36] have



(a) IPC sampled every 100K instructions (Coarse) shows stable performance between quanta



(b) IPC sampled every 500 instructions (Fine) shows highly variant performance between quanta

Figure 2: Variance of IPC in *gcc* over 300K instructions

shown that performance in general-purpose applications varies rapidly when observed at a much finer granularity. This reveals more low-performance phases, thereby increasing opportunities to utilize a more energy efficient core.

The benefits of reducing the switching granularity to hundreds of instructions is quantitatively shown by Figure 1. The figure sweeps quantum lengths, from hundred to a million instructions per quantum. For every quantum size, it shows the maximum percentage of time (in cycles) that can potentially be spent on Little, subject to an allowable 5% performance degradation. Migration costs are ignored in this oracle study. We observe that time spent on Little increases by 40% on average as the switching granularity reduces from 1M to 1K instructions, and a further 45% from 1K instructions to 100 instructions. Time spent on the Little, more energy efficient backend is translated to energy savings. A reduction in granularity by four orders of magnitude can thus be translated into higher energy efficiency gains.

2.2 Inaccuracy of the Reactive Controller

Existing dynamic controllers react to changes in performance with a delay of a sampling phase. While this cost is amortized in traditional coarse-grained systems, sampling intervals subsume entire quanta in a fine-grained system. Fine-grained reactive controllers assume that consecutive quanta have similar performance and react to changes in performance with a delay of at least one quantum.

Figure 2 shows the performance (in Instructions Per Cycle, IPC) seen over 300K instructions of *gcc*. The performance is sampled every quantum of 100K instructions in Figure 2(a). While the figure shows a large performance change after the first quantum (at 100K instructions), subsequent quanta display stable performance, validating a reactive approach. Figure 2(b) shows the same 300K instructions of *gcc*, but this time, performance is sampled at a fine granularity of 500 instructions. Besides demonstrating the additional low performance phases observable at this granularity, the figure also illustrates the inaccuracy imposed by assuming consecutive quanta behave similarly. Some points have been circled on the figure to help make this observation more clear. Drastic performance changes can be seen between consecutive circled quanta, implying that a quantum based reactive controller will provide inefficient schedules.

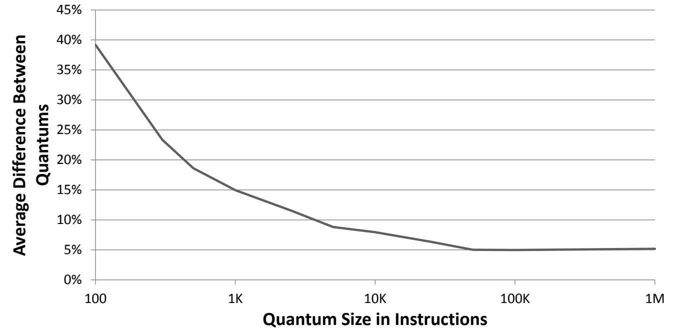


Figure 3: Average performance difference between consecutive quanta increases rapidly with quantum size

Figure 3 shows the average difference in performance that is observed between consecutive quanta for benchmarks in the SPEC06 suite, across quantum sizes. At a coarse granularity, consecutive quanta differ for each other by approximately 5% with a maximum standard deviation of 0.2 for *omnetpp* from the mean, implying they have similar performance. But this difference increases inversely with switching granularity, to nearly 40% for quanta of a 100 instructions, with a maximum standard deviation of 1.6 for gcc. This demonstrates that the previous quantum's performance is not a good indicator of future behavior for quanta with less than 10K instructions.

3. TRACE-BASED PHASE PREDICTOR

General purpose applications typically exhibit irregular dynamic code behavior but often times follow regular code structure. A controller which can dynamically learn to recognize these regular code sequences or *super-traces* (defined in Section 3.2) can preemptively map code to backend for maximum energy efficiency.

While it is possible to use a compiler to detect regular micro phases in a program based on static control flow profiling [5, 27], it cannot capture regularities imposed by data flow patterns. We employ a cheap hardware mechanism to build our *super-traces* and use a simple correlation-based prediction table to predict them.

3.1 Overview

Figure 4 illustrates an overview of the predictive approach

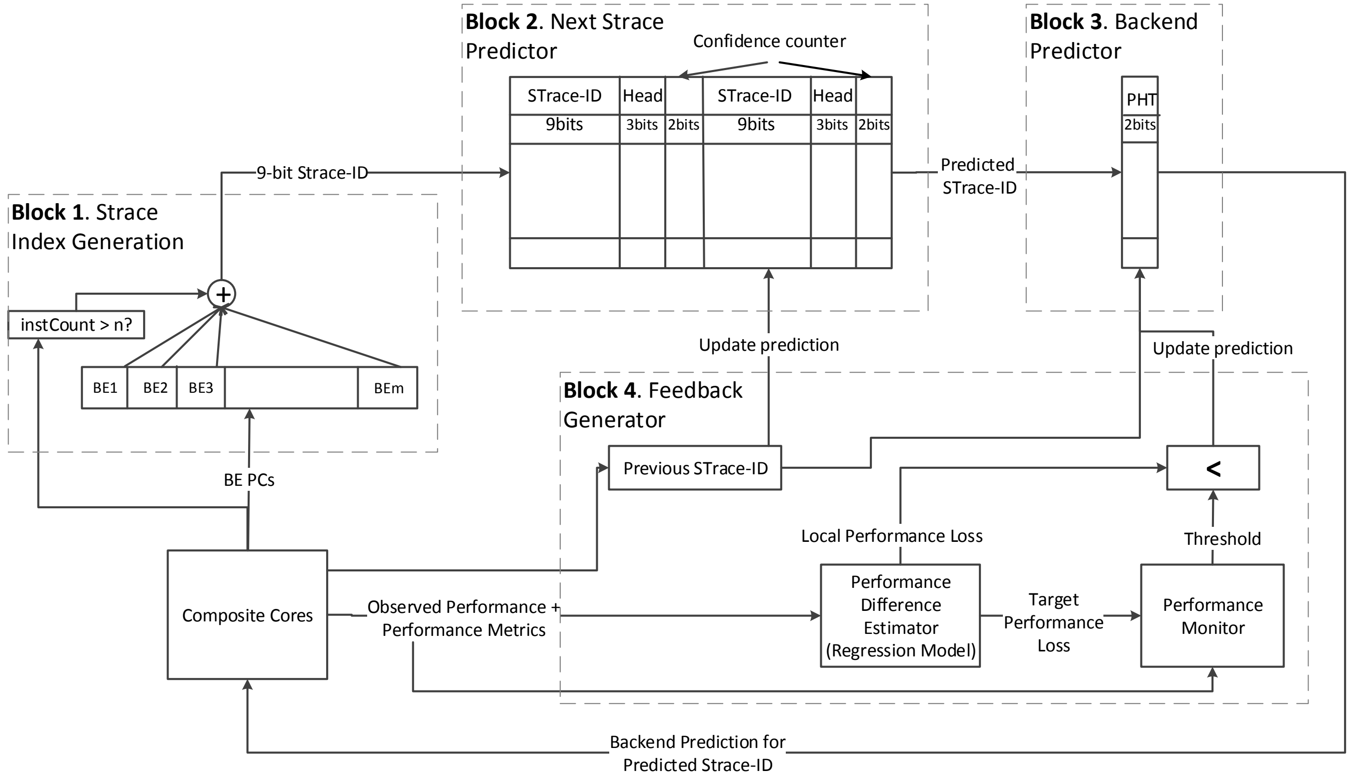


Figure 4: **Overview of the Predictive Trace Controller.** Backedge PCs seen by Composite Cores are hashed to index into the next-*super-trace* predictor (Block 1). This index references an entry from the backend predictor PHT (Block 2) to decide whether migration of execution is required. The feedback generator (Block 3) monitors dynamic performance and updates the tables accordingly

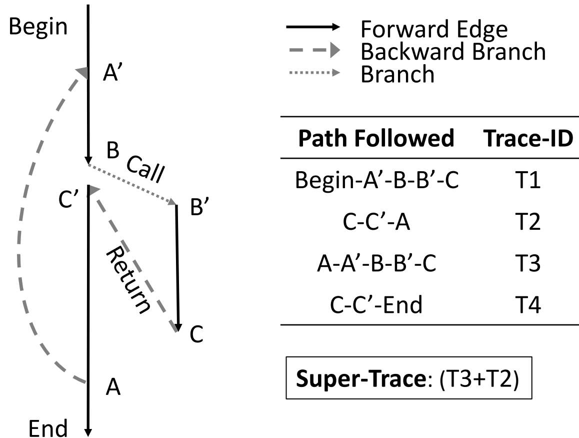
for scheduling *super-traces* onto the most efficient backend. Block 1 (Section 3.2) involves dynamically defining *super-trace* boundaries and creating pseudo-unique *super-trace*-IDs. Block 2 (Section 3.3) illustrates a correlation-based table for predicting future *super-traces*. Block 3 (Section 3.4) shows the *super-trace*-to-core-backend predictor table. Block 4 is the feedback mechanism that updates the prediction tables with correct values, in case of mispredictions.

3.2 Building Super-Traces

In order to have predictable behavior, switching boundaries should enclose intervals that occur repetitively. To identify a *super-trace*, we leverage a concept similar to that of traces [8] or frames [22]. Traces are defined as sequences of instructions or basic blocks that have a high likelihood of executing back to back, despite the presence of intervening control instructions. These can be identified both statically and dynamically, covering roughly 70% of dynamic instructions [22]. The controller used in this work is organized around traces that are defined at backedge boundaries. A backedge is defined as a control instruction (branches, function calls and returns) that branches to a negatively placed Program Counter(PC) ($PC_{target} < PC_{current}$). They capture the most frequently occurring circular paths in a dynamic instance of code (loops, for example). Since either a function call or its return will be a backedge, traces also account for function bodies. The intuition behind using backward branches is that their target PCs act as global re-convergent points [1, 24, 25]. The control re-convergence point for a particular instruction is defined as a future dynamic instruction that will be eventually reached, regardless of the outcome of any non-exceptional intervening control flow. Traces delineated by these chosen points act as control independent code blocks in trace processors [25] and dynamic multi-threading processors [1]. By ignoring the intervening branches (which account for 93% of the total static branches [6]) between these re-convergence points future, traces can be predicted with higher accuracy. Another advantage of using backedges as switching points is that mispredicted backedges cause a pipeline flush, in order to recover from wrongly speculated instructions. This partially hides the pipeline drain imposed by the architecture in case the thread chooses to migrate.

Backedges occur frequently in the SPEC benchmark suite, occurring once every 53 instructions on average. Existing fine-grain heterogeneous cores aren't capable of switching cores at such a granularity. Hence backedge traces are merged together until a minimum instruction length boundary has been crossed. This block constrained by the number of instructions in it is referred to as a *super-trace*. For fine-grained switching this minimum length was experimentally found to be 300 instructions per *super-trace*, and can go upto 10K instructions. Below this minimum length, the switching costs imposed by our core architecture negate energy benefits. Figure 5 pictorially describes the runtime formation of traces with the help of an example loop with function calls within its body. The return from the function at point C and the return to the loop header at point A are backedges

that the hardware observes, dynamically creating traces T2 and T3. If there are around 300 instructions in both T2 and T3 cumulatively, then the hardware defines the sequence (T2+T3) as one *super-trace*. The *super-trace* (T2+T3) is representative of this loop and its characteristics will determine which backend it should be run on in the future.



Path Followed	Trace-ID
Begin-A'-B-B'-C	T1
C-C'-A	T2
A-A'-B-B'-C	T3
C-C'-End	T4

Figure 5: Defining *super-trace* boundaries dynamically

3.3 Predicting Super-Traces

A correlation-based predictor is used to predict the next *super-trace*. A variety of works [19, 12] have demonstrated the efficiency of path based multi-branch predictors. The strength of using such a predictor lies in its ability to capture program context by using path based correlation.

As *super-traces* are limited by a maximum instruction length, the number of backedges per *super-trace* is variable. For example, for a *super-trace* length of 300 instructions, this number varies between 4 for *lbm* and 20 for *mcf* on average, with an average of 12 backedges per *super-trace* across all the benchmarks. A *super-trace* made of 1000 instructions has approximately 35 backedges on average across all benchmarks. Ideally, the *super-trace*-ID used to index into the prediction table should be a concatenation of all the backedge PCs that form it. But practical limitations like hardware overheads mandate a restriction on the number of backedges that can be used to uniquely identify a *super-trace*. We performed sensitivity studies for accuracy of prediction using the last 12, 15, 21, and 30 backedges in a *super-trace* to form its ID. For our analysis, keeping the last 15 backedges was sufficiently accurate, providing low aliasing with minimal overheads. We used an indexing mechanism similar to [12] to create a pseudo-unique *super-trace*-ID, as shown in Figure 6. This technique gives higher priority to a more recently seen backedge by hashing more of its bits as compared to the older backedges. The intuition behind this approach is that more recently executed branches are more likely to indicate what code is executed next. The specific parameters of the index generation mechanism were determined experimentally.

The generated *super-trace*-ID is used to index into a two-way associative prediction table that consists of two possible successors of this trace (Figure 4). Since the next *super-trace* prediction is made when the last instruction of the previous *super-trace* (the backedge) is committing, the header PC for the next *super-trace* is known. To leverage this information, the last three bits (excluding the byte offset) of the next PC is stored along with the hashed *super-trace*-ID for each *super-trace*. This head PC acts as a tag to select one of the two potential successors in the table. A 2-bit saturating counter per successor ID is used to select replacements candidates. In case of an incorrect prediction, the table is updated with the correct successor. Either the entry with the lower counter value is demoted, or deleted if the counter is zero. In case of a correct prediction, the confidence counter is incremented by 1. Effects on accuracy of the predictor when using different predictor table sizes by using different different configurations of backedges to create the index is shown in Section 5.

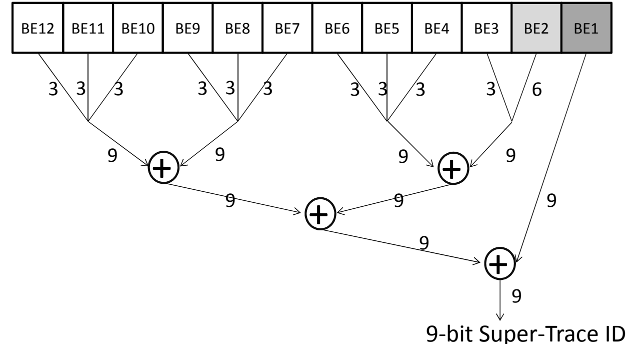


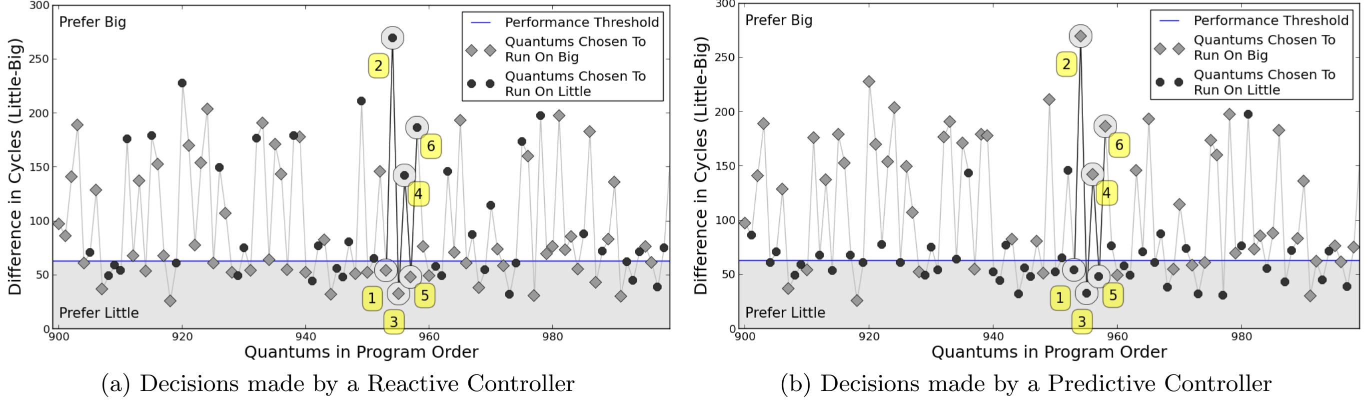
Figure 6: Super-trace-ID generation using illustrated bits from the 12 last seen backedges. More priority is given to more recent (darker-gray) backedges.

3.4 Scheduling Super-Traces

The controller is built on the hypothesis that behavior of a recurring trace can be estimated based on its individual characteristics, in conjunction to the context (*super-trace*) in which it appears. The controller leverages these two pieces of information to map a *super-trace* to the most efficient backend.

A simple 2-bit saturating counter is used to predict whether a *super-trace* should be executed on the Big or Little backend (Figure 4). The *super-trace*-ID outputted by the *super-trace* predictor is used to index into a Pattern History Table (PHT) which steers the program execution to either backend. We compared more complex predictors like two level adaptive local and global predictors (Section 5). We found that the accuracy gains achieved from higher sophistication were not significant enough to warrant the extra hardware.

The feedback to this backend predictor is given by a performance model that captures the microarchitectural characteristics of the *super-trace*. A threshold controller provides an average per-*super-trace* performance loss threshold below which it is currently profitable to switch to Little, given the performance target. A tuned Proportional-Integral (PI) feedback control loop [26] scales this threshold by observing the proximity of current performance to the target performance setpoint. This performance monitoring is assisted by a linear regression model which estimates the target performance (only Big) and observes current performance (Big+Little). We employ the linear regression model used in [18], which estimates a *super-trace*'s performance on the inactive backend using performance metrics such as number of cache misses, branch mispredicts, the ILP and



(a) Decisions made by a Reactive Controller

(b) Decisions made by a Predictive Controller

Figure 7: Illustration of the lower accuracy shown by a reactive controller vs that of a predictive controller for a fine-grained system with 300 instructions per quantum running gcc. Ideally, points with low performance differences (area shaded grey) could be run on Little for energy gains with little performance loss, while the rest should be run on the Big backend.

MLP inherent in the code and its dependency on previously executed *super-traces*. These metrics succinctly capture the characteristics of individual *super-traces* and can be used to determine the resources that can best extract its performance efficiently. For example, if a *super-trace* is a compute-intensive loop with high ILP, it is always better to run it on the Big out-of-order backend. However, if a *super-trace* is a memory intensive function, back to back dependent load misses can potentially stall the core. Such a *super-trace* will consume less energy if run on Little. At the end of execution of a *super-trace*, its observed performance is compared to the threshold, and accordingly its PHT entry is updated to show its tendency towards execution on *Big* or *Little*. More details are covered in Section 4.

3.5 Illustrative Example

Figure 7 pictorially shows the inaccuracy incurred when using the last quantum to predict the next quantum's behavior. The graph plots the cycle difference observed between Big and Little for each quantum of 300 instructions for *gcc*. A large cycle difference implies that the quantum consumes lesser cycles to execute on Big as compared to Little, implying that the 'out-of-orderness' was useful for the instructions in it. These quanta, if mis-scheduled on Little, incur heavy performance losses. The points with lower cycle difference are quanta that took approximately equal number of cycles on either backend, implying that their performance is low irrespective of the backend they execute on. These quanta are good candidates to run on Little, as they can run on a more energy efficient backend without hurting performance. The performance threshold line shows the oracle threshold - all points above this line should be run on Big (diamonds) and all other points on Little (circles), in order to maintain a performance loss of 5% while maximizing energy benefits.

The reactive controller 7(a) assumes that a quantum behaves as per the last one and hence makes a few wrong decisions compared to the oracle. The circled points illustrate this effect. A low cycle difference on point 1 dupes the controller into scheduling the next quantum on Little. This is a mistake, and this mistake propagates to point 3, which is mis-scheduled on Big. By mis-scheduling quanta, the reactive controller slows down by 1780 cycles in this window of execution. Using a predictive based approach instead, assumes that a quantum behaves similarly to its last few instances and makes the decisions shown in 7(b). Such a controller follows the oracle decision schedule more closely. The marked points 1-6 are scheduled properly this time. Also, the circled points (Little decisions) are clustered tightly around the threshold line. This causes the predictive controller to lose only 470 cycles due to mispredictions.

3.6 Overheads of the Controller

The predictor described in this section adds minimal hardware and energy overheads. Each entry of the predictor table contains enough bits to accommodate 2 *super-trace*-IDs (9bits), each with 3 bits for next head PC and 2 bits for a confidence counter, along with 2 bits for the backend predictor PHT, totalling 1.875kB of storage. This table is accessed once at every *super-trace* boundary. We used CACTI [32] to estimate the power (0.033W) and area (0.012mm^2) overheads of this table and found them to be negligibly small in comparison to the whole core.

4. METHODOLOGY & HARDWARE DETAILS

The architecture we use to validate our scheme is closely modeled after the Composite Core design, proposed by Lukefahr et al. [18]. Switching intervals of hundreds of instructions makes necessary a tightly-coupled heterogeneous multicore system that is unencumbered by the large state migration latency of traditional designs. The Composite Core architecture consists of two tightly coupled core backends (modeled after ARM's big.LITTLE), with different performance and energy characteristics. It consists of two separate pipelines that share a common frontend and access to the memory system. Structures like the caches and branch predictors are shared, so as to avoid the overheads of rebuilding architecture state on every switch. These architecture is designed to ensure that only the register file has to be explicitly transferred between the backends on migration.

The high performing Big backend is modeled as a highly pipelined 3 wide superscalar out-of-order processor with a large ROB and LSQ, loosely based on ARM's A15 core. Its higher superscalar width and its ability to reorder instructions, allows it to run faster, achieving about twice the per-

Architectural Feature	Parameters
Big backend	3 wide superscalar @ 1.2GHz 12 stage pipeline 128 entry ROB 160 entry register file
Little backend	2 wide superscalar @ 1.2GHz 8 stage pipeline 32 entry register file
Memory System	32 KB L1 iCache (Shared) 32 KB L1 dCache (Shared) 1 MB L2 Cache 1024MB Main Mem

Table 1: Experimental Core Parameters

formance of Little. The energy efficient Little is modeled as a 2 wide in-order processor, based on ARM's A7 core. The simpler and smaller hardware resources used by an inorder processor allows it to consume a much lower energy per instruction than the out-of-order pipeline, consuming 1/3rd as much total energy. Also, due to a shorter pipeline length, it provides a quicker branch misprediction recovery.

The online linear regression model is based on the one used in [18]. Performance defining metrics like observed IPC, ILP, MLP, cache misses, branch mispredicts, dependencies on previous *super-traces*, are fed into a linear equation with constant multipliers (precomputed using SPEC's train input sets). These metrics are monitored online at run-time using existing performance counters and some additional storage tables. Prior work estimates the overheads to be negligible (30 cycle computation, consuming 5μW power and 0.02mm^2 area). Computations do not lie on critical path and are shown to be fairly accurate.

We used the Gem5 simulator [3], running a Linux OS, to perform cycle-accurate simulations to model performance. In order to estimate static and dynamic energy consumption of the core and L1 caches, we utilized the McPAT modeling framework [17]. A core is assumed to be clock gated when inactive, since power gating adds significant overheads that are not scalable at fine-granularity switching. Little uses a frontend that is provisioned for the bigger out-of-order Big core, and hence it pays an added overhead in terms of energy as compared to a custom designed Little core. Table 1, gives a detailed description of the configurations used, as per [18]. We evaluate our scheme using the SPEC 2006 benchmark suite[7]. Benchmarks were compiled using gcc with -O2 optimizations for the Alpha ISA. All evaluations were done over 100 million instructions, after fast-forwarding for 2 billion instructions.

5. RESULTS

In this section we analyze the *super-trace* based predictive controller and compare it to the current state-of-art techniques. For the rest of the study, we assume the user will tolerate a performance loss of 5% in exchange for maximum energy savings possible.

5.1 Prediction Accuracies

Phase detection in programs is a well researched problem for coarse granules. Sherwood et al. [30, 31] define a phase footprint by the frequency of execution of basic blocks within a quantum. They create hashes of branches along with the number of instructions between branches to capture individual basic blocks. Such a phase ID tracks the instruction foot-

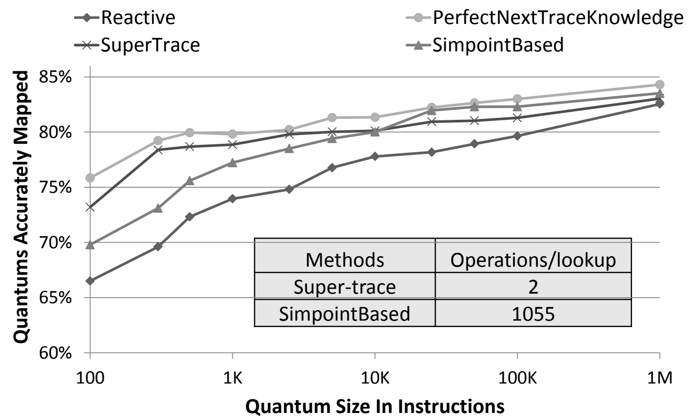


Figure 8: Comparison of accuracies of predictions. SimpointBased uses a prediction scheme similar to the coarse-grained methods shown in [31]. The enclosed table compares computation overheads.

print of the frequently executed blocks that dominate the behavior of that phase, thus distinguishing coarse-grained phases. They define a hardware implementation to dynamically build and predict phases and performance changes at 10 M instruction granularities. We modify this technique to use only backedges instead of all branches, to maintain a common baseline for comparison, and use a LRU-based phase signature table of 32 entries. To predict the next phase, they suggest a Run Length Encoding Predictor ([31, 16]), which predicts the next phase using the current phase and the number of times it occurs successively.

In this work, we define the accuracy of a controller to be the percentage of quanta or *super-traces* for which the controller picks the same backend as chosen by the Oracle. The **Oracle** knows the behavior for every quantum throughout the program's execution. The static Oracle sorts all traces based on $(cycles_{Little} - cycles_{Big})$ and selects the ones with lowest cycle differences to run on Little. This allows us to maximize the number of traces run on Little under a specific performance loss constraint. The performance loss incurred per trace is accumulated until the performance target threshold is reached, beyond which the remaining quanta are scheduled to run on Big. The Oracle assumes instant switching between backends with zero overhead. The Oracle's optimality can be increased by considering dynamic switching costs, but complexity of this solution grows exponentially with number of traces (100 millions).

Figure 8 summarizes the accuracies that can be obtained across various trace granularities using controllers studied in this work. In order to isolate the accuracy of prediction of each kind of predictor from the noise added by migration costs, we assume zero switching overheads and no threshold controller in the accuracy study. The accuracies shown in the figure are a measure of how well a controller can predict a phase and a phase change, i.e. predict when execution should shift from one core backend to the other. In this work's context, a correct Big/Little decision is more critical to the final aim of energy efficiency, rather than accuracy of phase prediction (which is 70% accurate on average at a fine-granularity on average). As expected, a reactive controller starts a sharp downward curve around 10K instructions granularity in Figure 3.

The PerfectNextTraceKnowledge Oracle assumes perfect

knowledge of the ID of the oncoming *super-trace*, and has to predict which backend to run it on (i.e. Block #2 in Figure 4 is perfect). Finally our scheme, SuperTrace, uses the controller mentioned in Section 3 to classify *super-traces* as Big or Little. The difference between these two lines demonstrates the inaccuracies added by the next *super-trace* predictor. Benchmarks like *gobmk* (57%) and *sjeng* (64%) show the highest inaccuracy at a fine granularity since both these benchmarks have among the highest number of branches and branch misprediction rates in the SPEC2006 suite [4]. For example, *sjeng* involves continually exploring through a variation tree for every point in its optimization algorithm [10], leading to poor path predictability.

The last line (SimpointBased) on this graph is the accuracy shown by our modified version of the scheme by Sherwood et al. [31] to identify phases. We see that this technique performs well at coarser granularities (beyond 100K instructions/quantum). But as the granularity of switching reduces, this method's inaccuracy reduces and is 10% lesser than that of the *super-trace*-based predictor at fine granularity. No significant accuracy improvements were found even when infinitely sized signature tables were used. At finer-granularities, the path followed by previous code is equally critical to predicting future blocks' behavior. The previous work considers only the frequency of basic-blocks executed, and not the actual path through them. Our method of prioritizing recently encountered backedges in the *super-trace*-identifying hash (Section 3.2) is more suited in this domain. Also, the phase matching implementation (calculating Manhattan distances between phase vectors) involves heavy computation in their implementation, including 1K adds/subtracts and compares. This is infeasible to perform at the frequency demanded by our fine-grained architecture (every few hundred instructions).

This study demonstrates that at a coarse-granularity of 100K instructions or more, a reactive controller works equally as well as a predictive one. This is because stable performance phases can be observed at much coarser-granularities. However, in the realm of fine-grained heterogeneity, which affords additional energy savings as compared to coarser-grained quanta, more information about code behavior is needed to make precise code to backend mappings. A prediction-based controller that implicitly keeps track of code specific resource requirements such as the one described in this paper is capable of increasing prediction accuracy and thus the energy gains obtained.

As noted earlier, the overheads imposed on migration between cores subsume energy gains below 300 instruction traces. We limit our switching architecture to this minimum granularity. The rest of the section shows the benefits of a predictive controller specifically for this fine-granularity.

5.1.1 *Case Study - gcc*

In order to demonstrate the effectiveness of a *super-trace*-based backend predictor over existing techniques, Figure 9 plots the cycle differences between executing a quantum on the Little vs the Big backends. A high cycle difference implies that the quantum gains performance benefits on the big out-of-order core and hence should be run on Big. A low cycle difference implies that the quantum is agnostic of underlying microarchitecture and will perform similarly on either core. Running these on Little offers opportunity to conserve energy without losing significant performance.

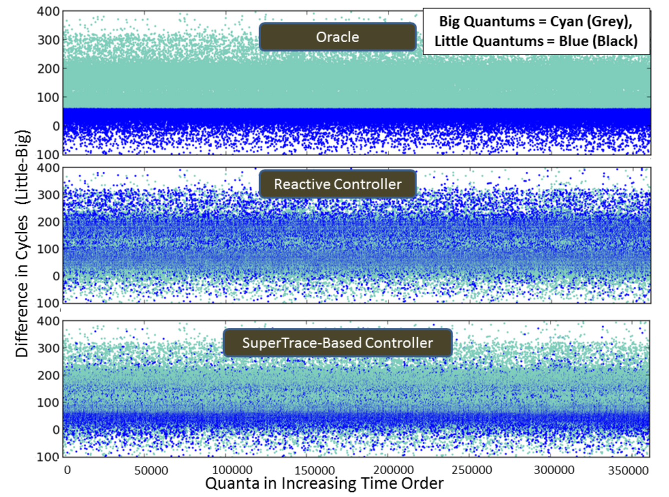


Figure 9: Case Study: A *super-trace*-based predictor follows the oracle schedule more closely over existing techniques, for hard to predict *gcc*

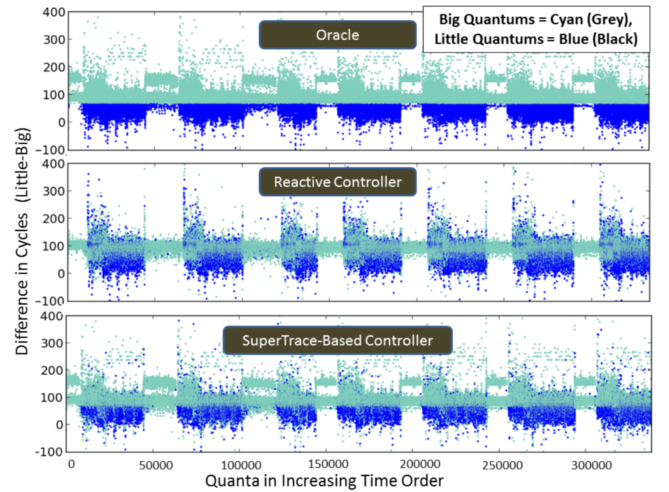


Figure 10: Case Study: A *super-trace*-based predictor follows the oracle schedule more closely over existing techniques, for compute intensive *h264ref*

The blue(black) and cyan(grey) points show the decisions taken per quantum by each controller over the entire run of *gcc*. Blue represents a quantum that was picked to go on Big and cyan represents those that went on Little. We pick *gcc* as an example because of the highly unpredictable data and control flow that it possesses. It displays many phases changes of similar performance through its execution. The oracle schedules 41% of its overall execution on Little.

The first illustration in Figure 9 is the schedule chosen by the oracle. It chooses a threshold, such that all quanta with a cycle difference of above that threshold are run on Big to conserve performance and all below it are run on Little, offering opportunity for energy saving. The second illustration is the schedule made by a reactive oracle; there are many quanta with a high cycle difference that are picked to execute on Little. It gets only 49% of the predictions correct, over the execution window, compromising the overall performance of the system. The third illustration shows the

decisions made by the predictive controller. The threshold set by the oracle is more or less learnt by the predictor, leading to a more accurate (66%) mapping schedule. There are some paths taken in *gcc* that always pick one core over the other. For example, the third most frequently seen *super-trace* has hard to correctly predict branch behavior. Such a *super-trace* is a Little backend candidate, because Little, with its smaller pipeline has faster branch recovery. But there are other paths throughout its execution that have extremely unpredictable data flow patterns. It is difficult for our mechanism to predict the behavior of such paths, which reduces its accuracy.

5.1.2 Case Study - h264ref

This case study considers *h264ref* which is a compute-intensive benchmark, and hence chooses to run on Little only 30% of time in the oracle case. It has modest branch mispredicts rates and is not memory intensive. Figure 10 shows the decisions made by all three controllers over the execution of this benchmark, similar to the previous study. There are noticeable performance phases that can be observed in the figure during the benchmark's run. Some individual paths taken in a coarse-grained phase always perform worse than the others; these are Little candidates. After experiencing a high performance (cyan) phase, the reactive controller misspeculates the start of the low performance (blue) phase, and sluggishly reacts to the phase change, incurring performance losses and an accuracy of 85%. The predictive controller on the other hand, learns to distinguish the low performance *super-traces* from the high performance ones yielding an increased accuracy of 89%.

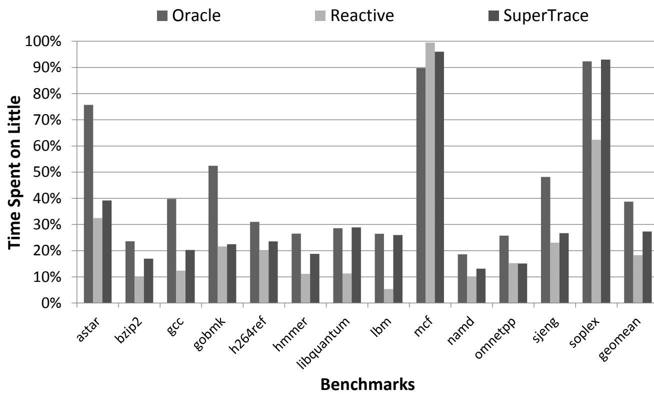


Figure 11: *super-trace*-based predictor increases percentage of overall execution time spent on the Little backend

5.2 Time Spent On Little

The amount of energy saved on a Composite Core is proportional to the time spent on Little. This is measured by the fraction of total cycles spent doing work on Little (Figure 11). The Oracle bar shows the maximum possible Little utilization assuming instant switching subject to a 5% performance loss (defined in Section 3) at a granularity of 300 instructions. The second bar shows Little utilization obtained by a reactive controller, averaging to around 18% overall. The *super-trace* predictive controller spends more than 27% of its execution time on Little, bridging the gap between the Oracle and previous work by nearly half. Memory intensive benchmarks like *soplex* spend more time on Little, because stalling the pipeline for a load is cheaper in terms of energy on a smaller, in-order core. In memory-intensive *mcf*, Little benefits from the prefetching action that arises from running a trace on Big before squashing it, on a switch. The statically derived oracle is oblivious to these implications of switching between backends, owing it to assign lower execution time on Little. Benchmarks like *astar*, *sjeng* and *gobmk*, have very high branch misprediction rates [4]. Because of the unpredictable control flow, combined with unpredictable data flow patterns, it is difficult to predict whether the upcoming *super-trace* is a Little or Big candidate. Hence, even though the oracle shows good potential for such benchmarks, it is hard for a predictor to learn and make accurate predictions. Benchmarks which mainly involve computation (with minimum stalls due to memory and branch mispredictions) like *hmmer* and *bzip2* show a 68% and 71% improvement over reactive in Little utilization, This is because the predictor learns the behavior of the few memory intensive paths seen in them and preemptively switches to Little for these.

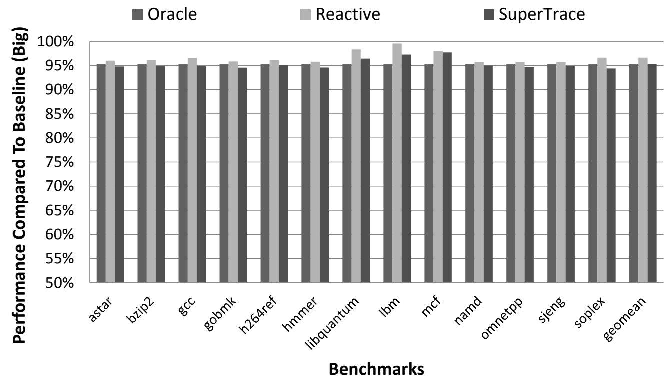


Figure 12: Overall performance loss of 5% is honored by each controller

5.3 Performance of the System

Figure 12 shows the performance degradation that each of the controllers incurs with a Composite Cores style architecture. Recall that the user sets a performance loss threshold of 5% overall, and this bound is honored by each of the controllers, while optimizing for energy consumption. There is a trade-off between time spent on Little (Figure 11) and performance (Figure 12). In *soplex*, our predictor overestimates time on Little compared to oracle, at a performance penalty. *omnetpp* falls short of the target, because of extra unnecessary switching decisions made by the controller, imposing additional migration overheads.

5.4 Energy Consumption

Figure 13 compares the energy that can be conserved by running the benchmarks used in this study on tightly coupled heterogeneous cores. While a core is inactive, it is assumed to be clock gated. The overheads imposed by the 2kB predictor table used by the predictive controller are negligible compared to the core's dynamic energy, and is ignored in this study. *omnetpp* is one benchmark that conserves lesser energy using our scheme, owing to its longer execution time. On an average, the predictive controller saves total core consumption energy by 60% over the previous work.

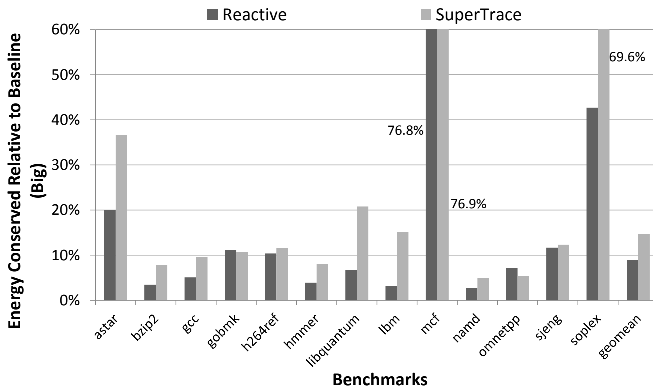


Figure 13: *super-trace*-based predictor conserves higher overall energy on a tightly coupled heterogeneous system as compared to the baseline (big) core and existing controllers

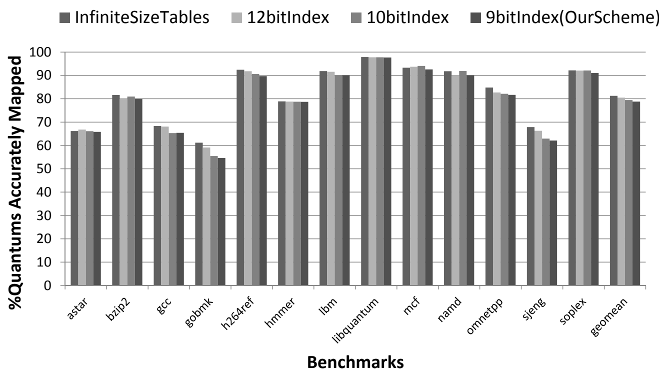


Figure 14: Larger predictor table sizes show slightly higher prediction accuracies

The remainder of this section covers the sensitivity analysis that led us to our design choices of table sizes and the types of predicting schemes.

5.5 Effect of Size of Predictors

Figure 14 shows the effect of predictor table size on the accuracy of backend prediction. The first bar shows the accuracy achieved by the *super-trace*-based predictor if there is no limitation on the table size. This ensures no aliasing due to the hashing performed in the ID generation stage, and no capacity conflicts in the predictor tables. The remaining three bars show the accuracies for three different table sizes, indexed by *super-trace*-IDs of 12, 10 and 9 bits respectively. As described in Section 3, the size of a table indexed by 12 bits is 18kB, while it is 4kB for a 10 bit index and 1.875kB for a 9 bit index. To optimize for both accuracy as well as hardware overhead, we choose a prediction table indexed by a 9 bit index for the rest of the study.

5.6 Effect of Backend Predictor Variations

In this section, we show the sensitivity (Figure 15) of a fine-grained phase predictor to the different adaptations of the core predictor (Block #3 in Figure 4). To isolate its effect, we assume a perfect next *super-trace* predictor in this study. We compare our method to a two-level adaptive predictor, which indexes into the PHT using history bits. The first bar shows such a predictor indexed into by 4 bits of history local to each *super-trace*-ID. The second bar keeps a

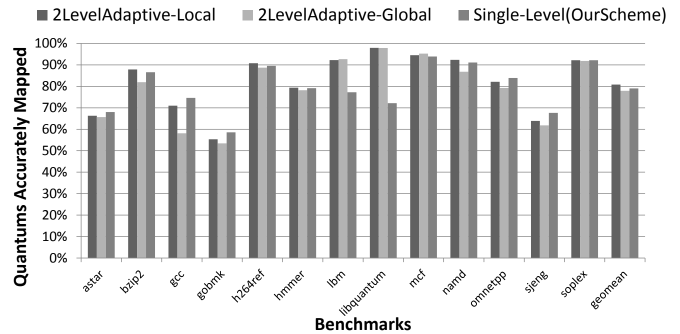


Figure 15: More sophisticated backend predicting schemes offer moderately higher prediction accuracies

10 bit global history to index into the PHT. The third bar is our approach, a single-level implementation, with each *super-trace*-ID accessing a 2 bit saturating counter. We see that the global predictor does worse generally, since it is hard to capture the performance changes on a per *super-trace* basis on a global level, at such granularities. The local history predictor does well on regular benchmarks like *bzip2*, *h264ref* and *lbm* because they have well defined performance phases. With a local history table indexed by a 9-bit index however, such a predictor adds a 14% hardware overhead to the implementation. Irregular benchmarks like *astar* and *gcc* have super-traces that have stable behavior across execution for these benchmarks, barring a few, sparsely located instances that show performance deviations attributable to irregular loads. These blips are handled well by a 2-bit saturating counter, resulting in good accuracies. Again, to achieve best trade-off for implementation complexity and accuracy, we choose a single-level backend predictor in this work.

6. RELATED WORKS

6.1 Heterogeneous Cores

Kumar et al. [14] consider migrating thread context between single-ISA implementations of out-of-order and in-order cores for the purposes of reducing power. They consider both sampling based and static approaches to scheduling coarse-grained quanta of 100M instructions. Sampling phases are triggered every 2M instructions on each inactive core and the most energy-efficient one is chosen. Lukefahr et al. [18] use a reactive approach to scheduling at a fine-granularity and propose the tightly coupled heterogeneous architecture that we model to validate our scheme. The inaccuracies incurred by such a controller limit their scope to 1000 instruction quanta. Becchi and Crowley [2] also show that an IPC driven sampling based scheduling scheme outperforms a static based approach. Rather than relying on sampling the performance on both cores, Van Craeynest et al. [34] propose a coarse-grained mechanism that relies on measures of CPI, MLP, and ILP to predict the performance on the inactive core. Bias scheduling [13] is another scheme that determines a program's bias towards a specific core configuration based on dynamic internal and external stalls due to architecture variations and resource availabilities. Recent work [37] use a regression model based approach to schedule webpages with varying, diverse characteristics to underlying heterogeneous architecture.

Chen and John [5] use offline profiling to leverage the relationship between inherent characteristics of a program and

its resource usage, in order to determine the appropriate core to run on. Such static approaches fail to capture execution behavior that varies with time. HASS [27] is another such static assignment scheme that tags applications with information regarding which core to run on, using offline analysis.

Another type of heterogeneity is achieved by scaling the voltage and frequency on cores at runtime, in order to gain energy savings. It is widely used in production processors today, and has been incorporated into ARM's big.LITTLE heterogeneous multicore system [9]. Such systems work well for conserving energy on memory bound applications.

6.2 Program Phase Detection and Prediction

Programs go through stable performance phases, both at a coarse granularity of millions of instructions [28] and at finer granularities [35, 36]. At a coarse granularity, hardware techniques have been proposed which dynamically identify and predict phase changes. In [33] program phases are defined to be the instruction working set of the program for a specific interval. Program phase changes are detected based on the relative differences between the instructions touched during execution. Another technique uses execution frequencies of basic blocks to define a phase [29, 31]. Phases are classified based on subroutines using a call stack in [11]. These methods are effective in identifying coarse grained phases in applications. However, as shown in this work, some of them break down at finer granularities.

At the other end of the granularity spectrum are traces - instruction sequences that possess regular control and data flow patterns. Trace processors [24] apply data and control flow prediction at trace boundaries in order to distribute execution resources among fine-grained slices of instructions. Dynamic Multithreading Processors [1] identify fine-grained phases at procedure calls and loop boundaries. The approach in [6] identifies global control reconvergence points using backedges. These schemes, conversely, deal with instruction phases of tens of instructions which lie below the scope of switching granularity offered by our architecture.

7. CONCLUSION

General purpose applications exhibit frequent performance phases, which at a fine-granularity, can have radically different control flow and data access patterns. This work demonstrates that heterogeneous systems operating at such granularities can gain energy efficiency by identifying these micro-phases and mapping them to hardware customized to their characteristics. In this work, we show the limitations of using a reactive approach to scheduling quanta when making scheduling decisions at granularities of less than 10K instructions. A predictive controller which foresees an oncoming phase change and accordingly migrates a thread is more precise. We propose a *super-trace*-based predictive controller that can develop and predict the behavior patterns of frequently occurring code sections in the program at a fine granularity of hundreds of instructions. This enables applications to run 28% of their execution on the Little, more energy-efficient backend, nearly 50% more compared to existing techniques. The predictive scheduler reduces energy consumption of an out-of-order processor by 15% with negligible hardware and energy overheads on a tightly coupled heterogeneous system.

8. ACKNOWLEDGEMENTS

This work is supported in part by ARM Ltd and by the National Science Foundation under grant SHF-1217917. The authors would like to thank the fellow members of the CCCP research group, and the anonymous reviewers for their time, suggestions, and valuable feedback.

9. REFERENCES

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 226–236, 1998.

[2] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers*, pages 29–40. ACM, 2006.

[3] N. Binkert et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, Aug. 2011.

[4] S. Bird, A. Phansalkar, L. K. John, A. Mericas, and R. Indukuru. Performance characterization of spec cpu benchmarks on intel's core microarchitecture based processor. In *SPEC Benchmark Workshop*, 2007.

[5] J. Chen and L. K. John. Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the 46th Annual Design Automation Conference*, pages 927–930. ACM, 2009.

[6] J. D. Collins, D. M. Tullsen, and H. Wang. Control flow optimization via dynamic reconvergence prediction. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 129–140, 2004.

[7] S. P. E. Corporation. Spec 2006, 2006. http://www.spec.com/cpu2006/.

[8] D. Friendly, S. Patel, and Y. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 173–181, June 1998.

[9] P. Greenhalgh. Big.little processing with arm cortex-a15 & cortex-a7, Sept. 2011. http://www.arm.com/files/downloads/big_LITTLE_Final.pdf.

[10] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[11] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 157–168. IEEE, 2003.

[12] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 14–23, 1997.

[13] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European conference on Computer systems*, pages 125–138. ACM, 2010.

[14] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Microarchitecture,*

2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on, pages 81–92, 2003.

[15] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ACM SIGARCH Computer Architecture News*, volume 32, page 64. IEEE Computer Society, 2004.

[16] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 278–289. IEEE, 2005.

[17] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009.

[18] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 317–328, 2012.

[19] R. Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 15–23, 1995.

[20] H. H. Najaf-abadi and E. Rotenberg. Architectural contesting. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 189–200. IEEE, 2009.

[21] NVidia. Variable smp -a multi-core cpu architecture for low power and high performance, 2011. http://www.nvidia.com/content/PDF/tegra_white_papers/Variable-SMP-A-Multi-Core-CPU-Architecture-for-LowPower-and-High-Performance-v1.1.pdf.

[22] S. J. Patel and S. S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, June 2001.

[23] K. K. Rangan, G.-Y. Wei, and D. Brooks. Thread motion: fine-grained power management for multi-core systems. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 302–313. ACM, 2009.

[24] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 138–148. IEEE, 1997.

[25] E. Rotenberg, Q. Jacobson, and J. Smith. A study of control independence in superscalar processors. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 115–124. IEEE, 1999.

[26] D. Sellers. An overview of proportional plus integral plus derivative control and suggestions for its successful application and implementation. In *Proceedings for the 2001 International Conference on Enhanced Building Operations*, 2001.

[27] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. Hass: a scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review*, 43(2):66–75, 2009.

[28] T. Sherwood and B. Calder. Time varying behavior of programs. 1999.

[29] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, New York, NY, USA, 2002. ACM.

[30] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *Micro, IEEE*, 23(6):84–93, 2003.

[31] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. *ACM SIGARCH Computer Architecture News*, 31(2):336–349, 2003.

[32] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, Technical Report 2001/2, Compaq Computer Corporation, 2001.

[33] J. E. Smith and A. S. Dhodapkar. Dynamic microarchitecture adaptation via co-designed virtual machines. In *Solid-State Circuits Conference, 2002. Digest of Technical Papers. ISSCC. 2002 IEEE International*, volume 1, pages 198–199. IEEE, 2002.

[34] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA '12, pages 213–224, 2012.

[35] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–97, 2003.

[36] B. Xu and D. H. Albonesi. Methodology for the analysis of dynamic application parallelism and its application to reconfigurable computing. volume 3844, pages 78–86. SPIE, 1999.

[37] Y. Zhu and V. J. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. *Network*, 8000:6000, 2013.