Composite Cores: Pushing Heterogeneity into a Core

Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski,
Thomas F. Wenisch, and Scott Mahlke

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

Abstract

Heterogeneous multicore systems—comprised of multiple cores with varying capabilities, performance, and energy characteristics—have emerged as a promising approach to increasing energy efficiency. Such systems reduce energy consumption by identifying phase changes in an application and migrating execution to the most efficient core that meets its current performance requirements. However, due to the overhead of switching between cores, migration opportunities are limited to coarse-grained phases (hundreds of millions of instructions), reducing the potential to exploit energy efficient cores.

We propose Composite Cores*, an architecture that reduces switching overheads by bringing the notion of heterogeneity* ***within*** *a single core. The proposed architecture pairs big and little compute* μEngines *that together can achieve high performance and energy efficiency. By sharing much of the architectural state between the* μEngines*, the switching overhead can be reduced to near zero, enabling fine-grained switching and increasing the opportunities to utilize the little* μEngine *without sacrificing performance. An intelligent controller switches between the* μEngines *to maximize energy efficiency while constraining performance loss to a configurable bound. We evaluate* Composite Cores *using cycle accurate microarchitectural simulations and a detailed power model. Results show that, on average, the controller is able to map 25% of the execution to the little* μEngine*, achieving an 18% energy savings while limiting performance loss to 5%.*

1. Introduction

The microprocessor industry, fueled by Moore's law, has continued to provide an exponential rise in the number of transistors that can fit on a single chip. However, transistor threshold voltages have not kept pace with technology scaling, resulting in near constant per-transistor switching energy. These trends create a difficult design dilemma: more transistors can fit on a chip, but the energy budget will not allow them to be used simultaneously. This trend has made it possible for today's computer architects to trade increased area for improved energy efficiency of general purpose processors.

Heterogeneous multicore systems are an effective approach to trade area for improved energy efficiency. These systems comprise multiple cores with different capabilities, yielding varying performance and energy characteristics [20]. In these systems, an application is mapped to the most efficient core that can meet its performance needs. As its performance changes, the application is migrated among the heterogeneous cores. Traditional designs select the best core by briefly sampling performance on each. However, every time the application migrates between cores, its current state must be explicitly transferred or rebuilt on the new core. This state transfer incurs large overheads that limits migration between cores to a *coarse granularity* of tens to hundreds of millions of instructions. To mitigate these effects, the decision to migrate applications is done at the granularity of operating system time slices.

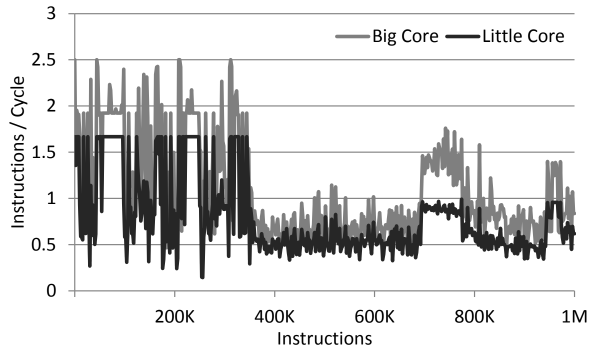
This work postulates that the coarse switching granularity in existing heterogeneous processor designs *limits* their effectiveness and energy savings. What is needed is a tightly coupled heterogeneous multicore system that can support fine-grained switching and is unencumbered by the large state migration latency of current designs.

To accomplish this goal, we propose *Composite Cores*, an architecture that brings the concept of heterogeneity to *within* a single core. A *Composite Core* contains several compute *μEngines* that together can achieve both high performance and energy efficiency. In this work, we consider a dual *μEngine Composite Core* consisting of: a high performance core (referred to as the big *μEngine*) and an energy efficient core (referred to as the little *μEngine*). As only one *μEngine* is active at a time, execution switches dynamically between *μEngines* to best match the current application's characteristics to the hardware resources. This switching occurs on a much finer granularity (on the order of a thousand instructions) compared to past heterogeneous multicore proposals, allowing the application to *spend more time on the energy efficient μEngine without sacrificing additional performance*.

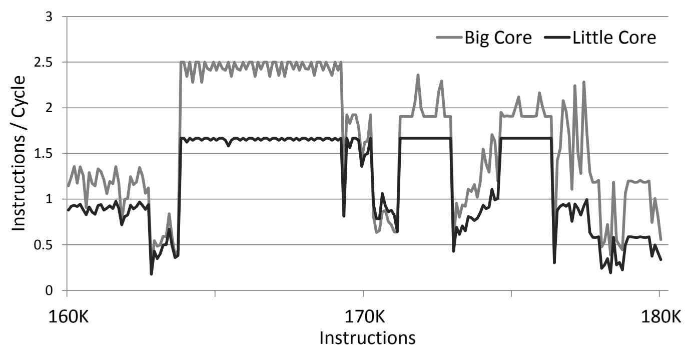
As a *Composite Core* switches frequently between *μEngines*, it relies on hardware resource sharing and low-overhead switching techniques to achieve near zero *μEngine* migration overhead. For example, the big and little *μEngines* share branch predictors, L1 caches, fetch units and TLBs. This sharing ensures that during a switch only the register state needs to be transfered between the cores. We propose a speculative register transfer mechanism to further reduce the migration overhead.

Because of the fine switching interval, conventional sampling-based techniques to select the appropriate core are not well-suited for a *Composite Core*. Instead, we propose an online performance estimation technique that predicts the throughput of the unused *μEngine*. If the predicted throughput of the unused *μEngine* is significantly higher or has better energy efficiency then the active *μEngine*, the application is migrated. Thus, the decision to switch *μEngines* maximizes execution on the more efficient little *μEngine* subject to a performance degradation constraint.

The switching decision logic tracks and predicts the accumulated performance loss and ensures that it remains within a user-selected bound. With *Composite Cores*, we allow the users or system architects to select this bound to trade off performance loss with energy savings. To accomplish this goal, we integrate a simple control loop in our switching decision logic, which tracks the current performance difference based on the performance loss bound, and a reactive model to detect the *instantaneous performance difference* via online perfor-



(a) Inst. window of length 2K over a 1M inst. interval



(b) Inst. window of length 100 over a 200K inst. interval

Figure 1: IPC Measured over a typical scheduling interval for 403.gcc

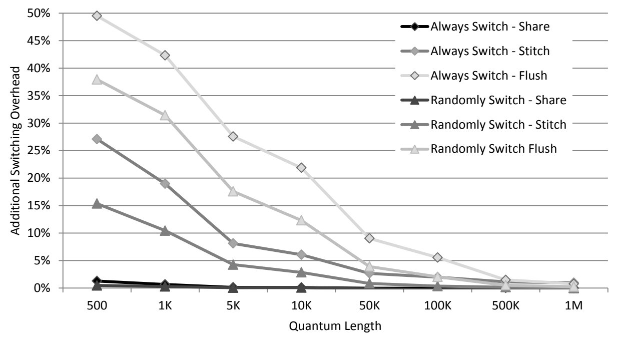


Figure 2: Migration overheads under different switching schemes and probabilities

mance estimation techniques.

In summary, this paper offers the following contributions:

- We propose *Composite Cores*, an architecture that brings the concept of heterogeneity *within* a single core. The *Composite Core* consists of two tightly coupled *μEngines* that enable fine-grained matching of application characteristics to the underlying microarchitecture to achieve both high performance and energy efficiency.
- We study the benefits of fine-grained switching in the context of heterogeneous core architectures. To achieve near zero *μEngine* transfer overhead, we propose low-overhead switching techniques and a core microarchitecture which shares necessary hardware resources.
- We design intelligent switching decision logic that facilitates fine-grain switching via predictive rather than sampling-based performance estimation. Our design tightly constrains performance loss within a user-selected bound through a simple feedback controller.
- We evaluate our proposed *Composite Core* architecture with cycle accurate full system simulations and integrated power models. Overall, a *Composite Core* can map an average of *25% of the dynamic execution* to the little *μEngine* and *reduce energy by 18%* while bounding performance degradation to at most 5%.

2. Motivation

Industry interest in heterogeneous multicore designs has been gaining momentum. Recently ARM announced a heterogeneous multicore, known as big.LITTLE [9], which combines a set of Cortex-A15 (Big) cores with Cortex-A7 (Little) cores to create a heterogeneous processor. The Cortex-A15 is a 3-way out-of-order with deep pipelines (15-25 stages), which is currently the highest performance ARM core that is available. Conversely, the Cortex-A7 is a narrow in-order processor with a relatively short pipeline (8-10 stages). The Cortex-A15 has 2-3x higher performance, but the Cortex-A7 is 3-4x more energy efficient.

In big.LITTLE, all migrations must occur through the coherent interconnect between separate level-2 caches, resulting in a migration cost of about 20 μseconds. Thus, the cost of migration requires that the system migrate between cores only at coarse granularity, on the order of tens of milliseconds. The large switching interval forfeits potential gains afforded by a more aggressive *fine-grained* switching.

2.1. Switching Interval

Traditional heterogeneous multicore systems, such as big.LITTLE, rely on coarse-grained switching to exploit application phases that occur at a granularity of hundreds of millions to billions of instructions. These systems assume the performance within a phase is stable, and simple sampling-based monitoring systems can recognize low-performance phases and map them to a more energy efficient core. While these long term low-performance phases do exist, in many applications, they occur infrequently, limiting the potential to utilize a more efficient core. Several works [27, 32, 33] have shown that observing performance at much finer granularity reveals more low-performance periods, increasing opportunities to utilize a more energy efficient core.

Figure 1(a) shows a trace of the instructions per cycle (IPC) for 403.gcc over a typical operating system scheduling interval of one million instructions for both a three wide out-of-order (big) and a two wide in-order (little) core. Over the entire interval, the little core is an average of 25% slower than the big core, which may necessitate that the entire phase be run on the big core. However if we observe the performance with finer granularity, we observe that, despite some periods of relatively high performance difference, there are numerous periods *where the performance gap between the cores is negligible*.

If we zoom in to view performance at even finer granularity (100s to 1000s of instructions), we find that, even during intervals where the big core outperforms the little on average, there are brief periods where the cores experience similar stalls and the performance gap between them is negligible. Figure 1(b) illustrates a subset of the trace from Figure 1(a) where the big core has nearly forty percent better performance, yet we can see brief regions where there is no performance gap.

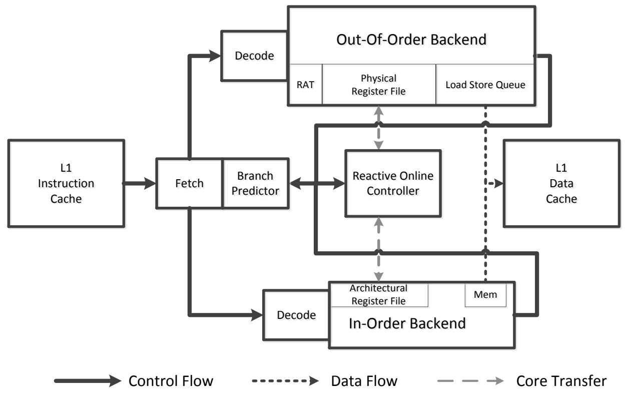


Figure 3: Microarchitectural overview of a *Composite Core*

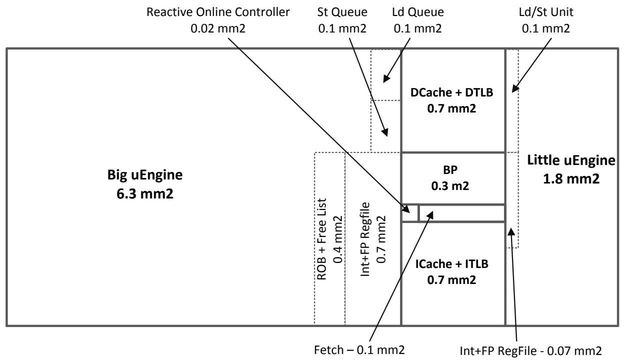


Figure 4: Estimated physical layout of a *Composite Core* in 32*nm* technology

2.2. Migration Overheads

The primary impediment to exploiting these brief low-performance periods is the cost (both explicit and implicit) of migrating between cores. Explicit migration costs include the time required to transport the core's architecturally visible state, including the register file, program counter, and privilege bits. This state must be explicitly stored into memory, migrated to the new core and restored. However, there are also a number of implicit state migration costs for additional state that is not transferred but must be rebuilt on the new core. Several major implicit costs include the extra time required to warm up the L1 caches, branch prediction, and dependence predictor history on the new core.

Figure 2 quantifies the effects of these migration overheads for different architectural implementations by measuring the additional migration overheads of switching at a fixed number of dynamic instructions, called a **quantum** or epoch. The figure shows the effects of switching at the end of every quantum with both a 100% probability (Always Switch) and with a $\frac{1}{3}$ probability (Randomly Switch). The $\frac{1}{3}$ probability is designed to weigh the instruction execution more heavily on the big core to better approximate a more typical execution mix between the big and little cores. The horizontal axis sweeps the quantum length while the vertical axis plots the added overhead due to increased switches.

The "Flush" lines correspond to a design where the core and cache state is invalidated when a core is deactivated (i.e., state is lost due to power gating), for example, ARM's big.LITTLE design. The "Stitch" lines indicate a design where core and cache state is maintained but not updated for inactive cores (i.e., clock gating of stateful structures). Finally, the "Shared" results indicate a design where both cores share all microarchitectural state (except the pipeline) and multiplex access to the same caches, corresponding to the *Composite Cores* approach. Observe that at large quanta, switching overheads are negligible under all three designs. However at small quantum lengths, the added overheads under both "Flush" and "Stitch" cause significant performance loss, while overhead under "Share" remains negligible.

These migration overheads preclude fine-grained switching in traditional heterogeneous core designs. In contrast, a *Composite Core* can leverage shared hardware structures to minimize migration overheads allowing it to target finer-grained switching, improving opportunities to save energy.

3. Architecture

A *Composite Core* consists of two tightly coupled compute *μEngines* that together can achieve high performance and energy efficiency by rapidly switching between the *μEngines* in response to changes in application performance. To reduce the overhead of switching, the *μEngines* share as much state as possible. As Figure 3 illustrates, the *μEngines* share a front-end, consisting of a fetch stage and branch predictor, and multiplex access to the same L1 instruction and data caches. The register files are kept separate to minimize the little *μEngine*'s register access energy.

As both *μEngines* require different control signals from decode, each *μEngine* has its own decode stage. Each *μEngine* has a separate back-end implementation, one striving for high performance and the other for increased energy efficiency. However, both *μEngines* multiplex access to a single L1 data cache, again to maximize shared state and further reduce switching overheads. The register file is the only state that must be explicitly transferred to switch to the opposite *μEngine*.

The big *μEngine* is similar to a traditional high performance out-of-order backend. It is a superscalar highly pipelined design that includes complicated issue logic, a large reorder buffer, numerous functional units, a complex load/store queue (LSQ), and register renaming with a large physical register file. The big *μEngine* relies on these complex structures to support both reordering and speculation in an attempt to maximize performance at the cost of increased energy consumption.

The little *μEngine* is comparable to a more traditional in-order backend. It has a reduced issue width, simpler issue logic, reduced functional units, and lacks many of the associatively searched structures (such as the issue queue or LSQ). By only maintaining an architectural register file, the little *μEngine* eliminates the need for renaming and improves the efficiency of register file accesses.

Figure 4 gives an approximate layout of a *Composite Core* system at 32nm. The big *μEngine* consumes $6.3mm^2$ and the L1 caches consume an additional $1.4mm^2$. The little *μEngine* adds an additional $1.8mm^2$, or about a 20% area overhead. However, this work assumes that future processors will be limited by power budget rather than transistor area. Finally, the *Composite Core* control logic adds an additional $0.02mm^2$ or an additional 0.2% overhead.

3.1. *μEngine* Transfer

During execution, the reactive online controller collects a variety of performance metrics and uses these to determine which *μEngine* should be active for the following quantum. If at the end of the quantum, the controller determines that the next quantum should be run on the inactive *μEngine*, the *Composite Core* must perform a switch to transfer control to the new *μEngine*. Figure 5 illustrates the sequence of events when the controller decides to switch *μEngines*.

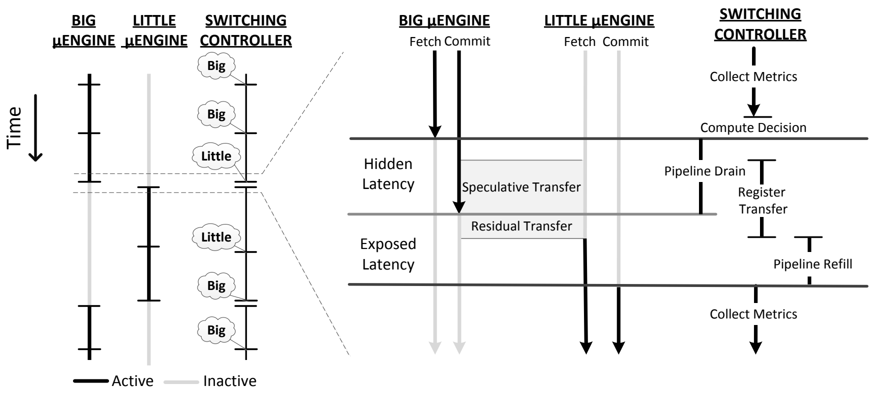


Figure 5: Mechanism of a *Composite Core* switch

As both μ*Engines* have different backend implementations, they have incompatible microarchitectural state. Therefore, when the *Composite Core* decides to switch, the current active μ*Engine* must first be brought to an architecturally precise point before control can be transferred. If the big μ*Engine* is active, it has potentially completed a large amount of work speculatively, making a pipeline flush potentially wasteful. Therefore, the *Composite Core* simply stops fetching instructions to the active μ*Engine*, and allows the pipeline to drain before switching.

As all other stateful structures have either been drained (e.g., pipeline stages) or are shared (e.g., branch predictor), the only state that must be explicitly transferred is the register file. While the active μ*Engine* is draining, the *Composite Core* attempts to speculatively transfer as much of the register state as possible to hide switching latency. Once the active μ*Engine* has completely drained, the remaining registers are transferred during the residual transfer. More details on the register transfer are given in Section 3.2.

Once the register transfer has been completed, fetch resumes with the instructions now redirected to the opposite μ*Engine*. The new μ*Engine* will incur an additional delay while its pipeline stages are refilled. Therefore the total switch latency is the sum of the pipeline drain, register transfer, and the pipeline refill delay. As the pipeline drain is totally hidden and a majority of the register file values can be speculatively transferred, the only exposed latency is the residual register transfer and the pipeline refill latency of the new μ*Engine*. As this latency is similar to that of a branch mispredict, the switching overheads behave very similarly to that of a branch misprediction recovery.

3.2. Register State Transfer

As the register file is the only architecturally visible stateful component that is not shared, its contents must be explicitly transferred during a μ*Engine* switch. This transfer is complicated by the fact that the little μ*Engine* only contains architectural registers with a small number of read and write ports while the big μ*Engine* uses register renaming and a larger multi-ported physical register file. To copy a register from the big μ*Engine* to the little, the architectural-to-physical register mapping must first be determined using the Register Allocation Table (RAT) before the value can be read from the physical register file. Typically this is a two cycle process.

When the *Composite Core* initiates a switch, the registers in the active μ*Engine* are marked as untransferred. The controller then utilizes a pipelined state machine, called the transfer agent, to transfer the registers. The first stage determines the next untransferred register, marks it as transferred, and uses the RAT to lookup the physical register file index corresponding to the architectural register. Recall that while the big μ*Engine* is draining, its RAT read ports are not needed by the pipeline (no new instructions are dispatched). The second stage reads the register's value from the physical register file. The final stage transfers the register to the inactive μ*Engine*.

To hide the latency of the register transfer, the *Composite Core* begins speculatively transferring registers before the active μ*Engine* is fully drained. Therefore, when a register value is overwritten by the draining pipeline it is again marked as untransferred. The transfer agent will then transfer the updated value during the residual transfer of Figure 5. *The transfer agent will continue to run until the pipeline is fully drained and all architectural registers have been transferred*. Once all registers have been transferred, the opposite μ*Engine* can begin execution. The process of transferring registers from the little μ*Engine* to the big μ*Engine* is similar, except there is now a single cycle register read on the little μ*Engine* and a two cycle register write on the big μ*Engine*.

4. Reactive Online Controller

The decision of when to switch is handled by the Reactive Online Controller. Our controller, following the precedent established by prior works [20, 30], attempts to maximize energy savings subject to a configurable maximum performance degradation, or slowdown. The converse, a controller that attempts to maximize performance subject to a maximum energy consumption, can also be constructed in a similar manner.

To determine the appropriate core to minimize performance loss, the controller needs to 1) estimate the dynamic performance loss, which is the difference between the observed performance of the *Composite Core* and the performance if the application were to run *entirely* on the big μ*Engine*; and 2) make switching decisions such that the estimated performance loss is within a parameterizable bound. The controller consists of three main components: a performance estimator, threshold controller, and switching controller illustrated in Figure 6.

The performance estimator tracks the performance on the active μ*Engine* and uses a model to provide an estimate for the performance of the inactive μ*Engine* as well as provide a cumulative performance estimate. This data is then fed into the switching controller, which estimates the performance difference for the following quantum. The threshold controller uses the cumulative performance difference to estimate the allowed performance drop in the next quantum for which running on the little μ*Engine* is profitable. The switching controller uses the output of the performance estimator and the threshold con-

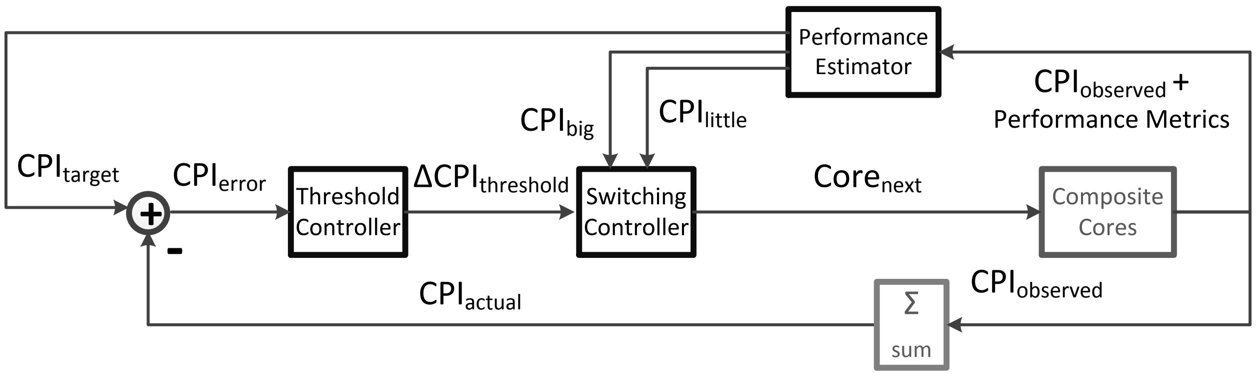


Figure 6: Reactive online controller overview

troller to determine which *µEngine* should be activated for the next quantum.

4.1. Performance Estimator

The goal of this module is to provide an estimate of the performance of both *µEngines* in the *previous* quantum as well as track the overall performance for *all* past quanta. While the performance of the active *µEngine* can be trivially determined by counting the cycles required to complete the current quantum, the performance of the inactive *µEngine* is not known and must be estimated. This estimation is challenging as the microarchitectural differences in the *µEngines* cause their behaviors to differ.

The traditional approach is to sample execution on both *µEngines* for a short duration at the beginning of each quantum and base the decision for the remainder of the quantum on the sample measurements. However, this approach is not feasible for fine-grained quanta for two reasons. First, the additional switching necessary for sampling would require much longer quanta to amortize the overheads, forfeiting potential energy gains. Second, the stability and accuracy of fine-grained performance sampling drops rapidly, since performance variability grows as the measurement length shrinks [32].

Simple rule based techniques, such as switching to the little *µEngine* on a cache miss, cannot provide an effective performance estimate needed to allow the user to configure the performance target. As this controller is run frequently, more complex approaches, such as non-linear or neural-network models, add too much energy overhead and hardware area to be practical.

Therefore the *Composite Core* instead monitors a selected number of performance metrics on the active *µEngine* that capture fundamental characteristics of the application and uses a simple performance model to estimate the performance of the inactive *µEngine*. A more detailed analysis of the performance metrics is given in Section 4.4.

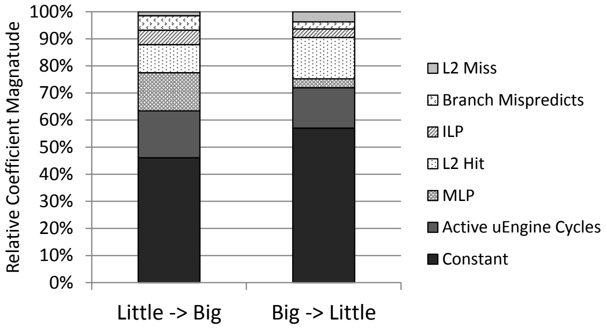


Figure 7: Magnitude of regression coefficients

4.1.1. Performance Model The performance model provides an estimate for the inactive *µEngine* by substituting the observed metrics into a model for the inactive *µEngine*'s performance. As this computation must be performed often, we chose a simple linear model to minimize computation overhead. Eq. 1 defines the model, which consists of the sum of a constant coefficient (a_0) and several input metrics (x_i) times a coefficient (a_i). As the coefficients are specific to the active *µEngine*, two sets of coefficients are required, one set is used to estimate performance of the big *µEngine* while the little *µEngine* is active, and vice versa.

$$y = a_0 + \sum a_i x_i \tag{1}$$

To determine the coefficients for the performance monitor, we profile each of the benchmarks on both the big and little *µEngine* for 100 million instructions (after a 2 Billion instruction fast-forward) using each benchmark's supplied training input set. We then utilize ridge regression analysis to determine the coefficients using the aggregated performance metrics from all benchmarks. The magnitude of each normalized coefficient for both models is shown in Figure 7, illustrating the relative importance of each metric to overall performance for each *µEngine*.

The *constant* term reflects the baseline weight assigned to the average performance of the active *µEngine* without considering the metrics. The *Active µEngine Cycles* metric scales the model's estimate based on the CPI of the active *µEngine*. *MLP* attempts to measure the levels of memory parallelism and account for the *µEngine*'s ability to overlap memory accesses. *L2 Hit* tracks the number of L2 cache hits and scales the estimate to match the *µEngine*'s ability to tolerate medium latency misses. *ILP* attempts to scale the performance estimate based on the inactive *µEngine*'s ability (or inability) to exploit independent instructions. *Branch Mispredicts* and *L2 Miss* scales the estimate based on the number of branch mispredictions and L2 cache misses respectively.

Little->Big Model: This model is used to estimate the performance of the big *µEngine* while the little *µEngine* is active. In general good performance on the little *µEngine* indicates good performance on the big *µEngine*. As the big *µEngine* is better able to exploit both MLP and ILP its performance can improve substantially over the little for applications that exhibit these characteristics. However, the increased pipeline length of the big *µEngine* makes it slower at recovering from a branch mispredict than the little *µEngine*, decreasing the performance estimate. Finally, as L2 misses occur infrequently and the big *µEngine* is designed to partially tolerate memory latency, the L2 Miss coefficient has minimal impact on the overall estimate.

Big->Little Model: While the big μ*Engine* is active, this model estimates the performance of the little μ*Engine*. The little μ*Engine* has a higher constant due to its narrower issue width causing less performance variance. As the little μ*Engine* cannot exploit application characteristics like ILP and MLP as well as the big μ*Engine*, the big μ*Engine*'s performance has slightly less impact than in the Little->Big model. L2 Hits are now more important as, unlike the big μ*Engine*, the little μ*Engine* is not designed to hide any of the latency. The inability of the little μ*Engine* to utilize the available ILP and MLP in the application causes these metrics to have almost no impact on the overall performance estimate. Additionally, as the little μ*Engine* can recover from branch mispredicts much faster, mispredicts have very little impact. Finally even though L2 misses occur infrequently, the little μ*Engine* suffers more performance loss than the big μ*Engine* again due to the inability to partially hide the latency.

Per-Application Model: While the above coefficients give a good approximation for the performance of the inactive μ*Engine*, some applications will warrant a more exact model. For example, in the case of memory bound applications like `mcf`, the large number of L2 misses and their impact on performance necessitates a heavier weight for the L2 Miss metric in the overall model. Therefore the architecture supports the use of per-application coefficients for both the Big->Little and Little->Big models, allowing programmers to use offline profiling to custom tailor the model to the exact needs of their application if necessary. However, our evaluation makes use of generic models.

4.1.2. Overall Estimate The second task of the performance estimator is to track the actual performance of the *Composite Core* as well as provide an estimate of the target performance for the entire application. The actual performance is computed by summing the observed performance for all quanta (Eq. 2). The target performance is computed by summing all the observed and estimated performances of the big μ*Engine* and scaling it by the allowed performance slowdown. (Eq. 3). As the number of instructions is always fixed, rather than compute CPI the performance estimator hardware only sums the number of cycles accumulated, and scales the target cycles to compare against the observed cycles.

$$CPI_{actual} = \sum CPI_{observed} \tag{2}$$

$$CPI_{target} = \sum CPI_{Big} \times (1 - Slowdown_{allowed}) \tag{3}$$

4.2. Threshold Controller

The threshold controller is designed to provide a measure of the current maximum performance loss allowed when running on the little μ*Engine*. This threshold is designed to provide an average per-quantum performance loss where using the little μ*Engine* is profitable given the performance target. As some applications experience frequent periods of similar performance between μ*Engines*, the controller scales the threshold low to ensure the little μ*Engine* is only used when it is of maximum benefit. However for applications that experience almost no low performance periods, the controller scales the threshold higher allowing the little μ*Engine* to run with a larger performance difference but less frequently.

The controller is a standard PI controller shown in Eq. 5. The P (Proportional) term attempts to scale the threshold based on the current observed error, or difference from the expected performance (Eq. 4). The I (Integral) term scales the threshold based on the sum of all past errors. A Derivative term can be added to minimize overshoot.

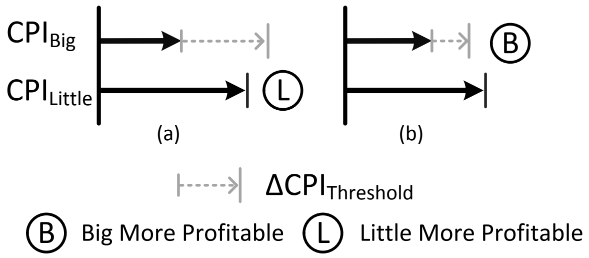


Figure 8: Switching controller behaviour: (a) If $CPI_{big} + \Delta CPI_{threshold} > CPI_{little}$ pick Little; (b) If $CPI_{big} + \Delta CPI_{threshold} < CPI_{little}$ pick Big.

However in our case, it was not included due to noisiness in the input signal. Similar controllers have been used in the past for controlling performance for DVFS [29].

The constant K_p and K_i terms were determined experimentally. The K_p term is large, reflecting the fact that a large error needs to be corrected immediately. However, this term suffers from systematically underestimating the overall performance target. Therefore the second term, K_i is introduced to correct for small but systematic under-performance. This term is about three orders of magnitude smaller than K_p, so that it only factors into the threshold when a long-term pattern is detected.

$$CPI_{error} = CPI_{target} - CPI_{actual} \tag{4}$$

$$\Delta CPI_{threshold} = K_p CPI_{error} + K_i \sum CPI_{error} \tag{5}$$

4.3. Switching Controller

The switching controller attempts to determine which μ*Engine* is most profitable for the next quantum. To estimate the next quantum's performance, the controller assumes the next quantum will have the same performance as the previous quantum. As show in Figure 8, the controller determines profitability by computing ΔCPI_{net} as shown in Eq. 6. If ΔCPI_{net} is positive, the little μ*Engine* is currently more profitable, and execution is mapped to the little μ*Engine* for the next quantum. However, if ΔCPI_{net} is negative, the performance difference between big and little is too large, making the little μ*Engine* less profitable. Therefore the execution is mapped to the big μ*Engine* for the next quantum.

$$\Delta CPI_{net} = (CPI_{Big} + \Delta CPI_{threshold}) - CPI_{little} \tag{6}$$

4.4. Implementation Details

We use several performance counters to generate the detailed metrics required by the performance estimator. Most of these performance counters are already included in many of today's current systems, including branch mispredicts, L2 cache hits and L2 cache misses. Section 4.4.1 details the additional performance counters needed in the big μ*Engine*. Due to the microarchitectural simplicity of the little μ*Engine*, tracking these additional metrics is more complicated. We add a small dependence table (described in Section 4.4.2) to the little μ*Engine* to capture these metrics.

4.4.1. Performance Counters The performance models rely heavily on measurements of both ILP and MLP, which are not trivially measurable in most modern systems. As the big μ*Engine* is already equipped with structures that exploit both ILP and MLP, we simply add a few low overhead counters to track these metrics. For ILP, a performance counter keeps a running sum of the number of instructions in the issue stage that are waiting on values from in-flight instructions.

This captures the number of instructions stalled due to serialization as an inverse measure of ILP. To measure MLP, an additional performance counter keeps a running sum of the number of MSHR entries that are in use at each cache miss. While not perfect measurements, these simple performance counters give a good approximation of the amount of ILP and MLP per quantum.

4.4.2. Dependence Table Measuring ILP and MLP on the little *µEngine* is challenging as it lacks the microarchitectural ability to exploit these characteristics and therefore has no way of measuring them directly.

We augment the little *µEngine* with a simple table that dynamically tracks data dependence chains of instructions to measure these metrics. The design is from Chen, Dropsho, and Albonesi [7]. This table is a bit matrix of registers and instructions, allowing the little *µEngine* to simply look up the data dependence information for an instruction. A performance counter keeps a running sum per quantum to estimate the overall level of instruction dependencies as a measure of the ILP. To track MLP, we extended the dependence table to track register dependencies between cache misses over the same quantum. Together these metrics allow *Composite Cores* to estimate the levels of ILP and MLP available to the big *µEngine*.

However, there is an area overhead associated with this table. The combined table contains two bits of information for each register over a fixed instruction window. As our architecture supports 32 registers and we have implemented our instruction window to match the length of the ROB in the big *µEngine*, the total table size is $2 \times 32 \times 128$ bits, 1KB of overhead. As this table is specific to one *µEngine*, the additional area is factored into the little *µEngine*'s estimate rather than the controller.

4.4.3. Controller Power & Area To analyze the impact of the controller on the area and power overheads, we synthesized the controller design in an industrial 65nm process. The design was placed and routed for area estimates and accurate parasitic values. We used Synopsys PrimeTime to obtain power estimates which we then scaled to the 32nm target technology node. The synthesized design includes the required performance counters, multiplicand values (memory-mapped programmable registers), and a MAC unit. For the MAC unit, we use a fixed-point 16*16+36-bit Overlapped bit-pair Booth recoded, Wallace tree design based on the Static CMOS design in [18]. The design is capable of meeting a 1.0GHz clock frequency and completes 1 MAC operation per cycle, with a 2-stage pipeline.

Thus, the calculations in the performance model can be completed in 9 cycles as our model uses 7 input metrics. With the added computations for the threshold controller and switching controller, the final decision takes approximately 30 cycles. The controller covers $0.02mm^2$ of area, while consuming less than $5uW$ of power during computation. The MAC unit could be power gated during the remaining cycles to reduce the leakage power while not in use.

5. Results

To evaluate the *Composite Cores* architecture, we extended the Gem5 Simulator [6] to support fast switching. All benchmarks were compiled using gcc with -O2 optimizations for the Alpha ISA. We evaluated all benchmarks by fast forwarding for two billion instructions before beginning detailed simulations for an additional one billion instructions. The simulations included detailed modeling of the pipeline drain functionality for switching *µEngines*.

We utilized McPAT to estimate the energy savings from a *Composite Core* [28]. We model the two main sources of energy loss in transistors, dynamic energy and static (or leakage) energy. We study only the effects of clock gating, due to the difficulties in estimating the performance and energy implications of power gating. Finally, as our design assumes tightly coupled L1 caches, our estimates include the energy consumption of the L1 instruction and data caches, but exclude all other system energy estimates.

Architectural Feature	Parameters
Big *µEngine*	3 wide Out-Of-Order @ 1.0GHz 12 stage pipeline 128 ROB entries 160 entry register file Tournament branch predictor (Shared)
Little *µEngine*	2 wide In-Order @ 1.0GHz 8 stage pipeline 32 entry register file Tournament branch predictor (Shared)
Memory System	32 KB L1 iCache, 1 cycle access (Shared) 32 KB L1 dCache, 1 cycle access (Shared) 1 MB L2 Cache, 15 cycle access 1024MB Main Mem, 80 cycle access

Table 1: Experimental *Composite Core* parameters

Table 1 gives more specific simulation configurations for each of the *µEngines* as well as the memory system configuration. The big *µEngine* is modeled as a 3-wide out-of-order processor with a 128-entry ROB and a 160-entry physical register file. It is also aggressively pipelined with 12 stages. The little *µEngine* is modeled to simulate a 2-wide in-order processor. Due to its simplified hardware structures the pipeline length is also shorter, providing quicker branch misprediction recovery, and it only contains a 32-entry architectural register file. The branch predictor and fetch stage are shared between the two *µEngines*.

5.1. Quantum Length

One of the primary goals of the *Composite Cores* architecture is to explore the benefits of fine-grained quanta to exploit short duration periods of low performance. To determine the optimum quantum length, we performed detailed simulations to sweep quantum lengths with several assumptions that will hold for the remainder of Section 5.1. To factor out controller inaccuracies, we assume the *µEngine* selection is determined by an oracle, which knows the performance for both *µEngines* for all quanta and switches to the little *µEngine* only for the quanta with the smallest performance difference such that it can still achieve the performance target. We also assume that the user is willing to tolerate a 5% performance loss relative to running the entire application on the big *µEngine*.

Given these assumptions, Figure 9 demonstrates the little *µEngine*'s utilization measured in dynamic instructions as the quantum length varies. While the memory-bound `mcf` can almost fully utilize the little *µEngine* at larger quanta, the remaining benchmarks show only a small increase in utilization until the quantum length decreases to less than ten thousand instructions. Once quantum sizes shrink below this level, the utilization begins to rise rapidly from approximately thirty percent to fifty percent at quantum lengths of one hundred instructions.

While a *Composite Core* is designed to minimize migration overheads, there is still a small register transfer and pipeline refill latency when switching *µEngines*. Figure 10 illustrates the performance impacts of switching *µEngines* at various quanta with the oracle targeting 95% performance relative to the all big *µEngine* case. We observe that, with the exception of `mcf`, which actually achieves a small speedup, all the benchmarks achieve the target performance at

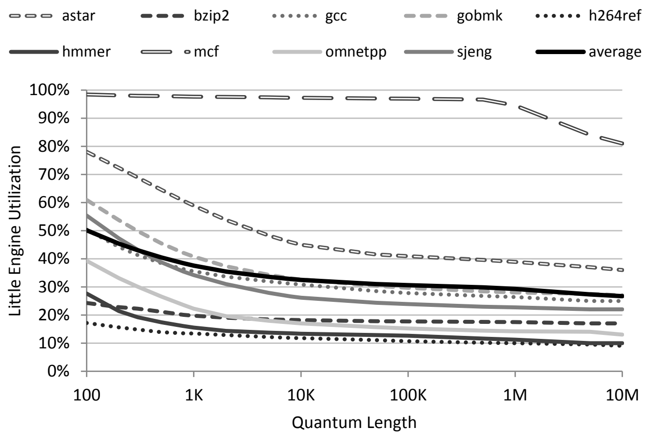


Figure 9: Impact of quantum length on little *µEngine* utilization

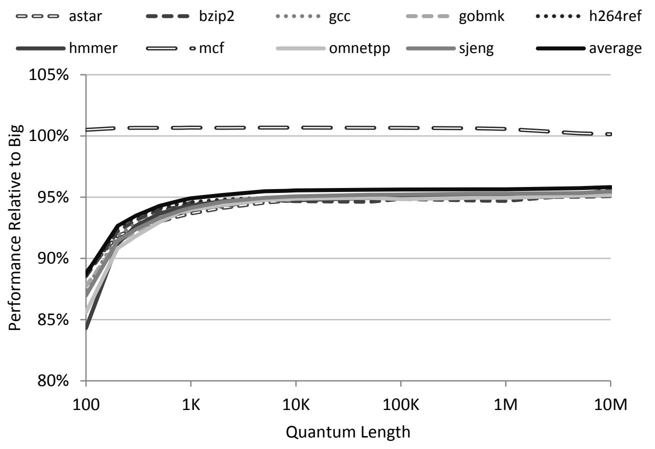


Figure 10: Impact of quantum length on overall performance with a 5% slowdown target

longer quanta. This result implies that the additional overheads of switching *µEngines* are negligible at these quanta and can safely be ignored. However, for quantum lengths smaller than 1000 instructions we begin to see additional performance degradation, indicating that the overheads of switching are no longer negligible.

The main cause of this performance decrease is the additional switches allowed by the smaller quanta. Figure 11 illustrates the number of switches per million instructions the *Composite Core* performed to achieve its goal of maximizing the little *µEngine* utilization. Observe that as the quantum length decreases, there is a rapid increase in the number of switches. In particular, for a quantum length of 1000 the oracle switches cores approximately 340 times every million instructions, or roughly every 3000 instructions.

As quantum length decreases the *Composite Core* has greater potential to utilize the little *µEngine*, but must switch more frequently to achieve this goal. Due to increased hardware sharing, the *Composite Core* is able to switch at a much finer granularity than traditional heterogeneous multicore architectures. However below quantum lengths of approximately 1000 dynamic instructions, the overheads of switching begin to cause intolerable performance degradation. Therefore for the remainder of this study, we will assume quantum lengths of 1000 instructions.

5.2. *µEngine* Power Consumption

A *Composite Core* relies on shared hardware structures to enable fine-grained switching. However these shared structures must be designed

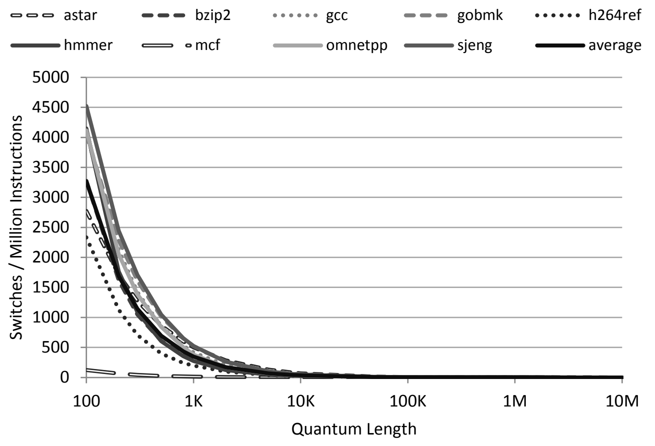


Figure 11: Impact of quantum length on *µEngine* switches

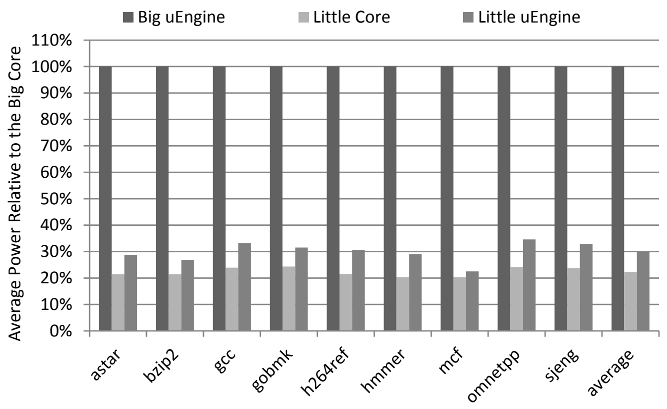


Figure 12: Average *µEngine* power relative to dedicated cores

for the high performance big *µEngine* and are over-provisioned when the little *µEngine* is active. Therefore the little *µEngine* has a higher average power than a completely separate little core. When the little *µEngine* is active, its frontend now includes a fetch engine, branch predictor, and instruction cache designed for the big *µEngine*. Also, the little *µEngine* accesses a data cache that is designed to support multiple outstanding memory transactions. While this functionality is necessary for the big *µEngine*, the little *µEngine* cannot utilize it. Finally, the leakage power of *Composite Cores* will be higher as it is comprised of two *µEngines*.

Figure 12 illustrates the average power difference between the *µEngines* and separate big and little cores. Observe that while the big *µEngine* includes the leakage of the little *µEngine*, it does not use noticeably more power than a separate big core. As the little *µEngine* is small, its contribution to leakage is minimal. However, while the little core requires only 22% of the big core's power, the shared hardware of the little *µEngine* only allow it to reduce the power to 30% of the big core. This is caused by a combination of both the leakage energy of the big *µEngine* and the inefficiencies inherent in using an over-provisioned frontend and data cache.

While the little *µEngine* of *Composite Core* is not able to achieve the same power reductions as a separate little core, this limitation is offset by *Composite Core*'s ability to utilize the little *µEngine* more frequently. As illustrated in Figure 9, a *Composite Core*, with a quantum length of 1000 instructions, executes more instructions on the little *µEngine* than a traditional heterogeneous multicore system, which has a quantum length of 10 million instructions or more. Even

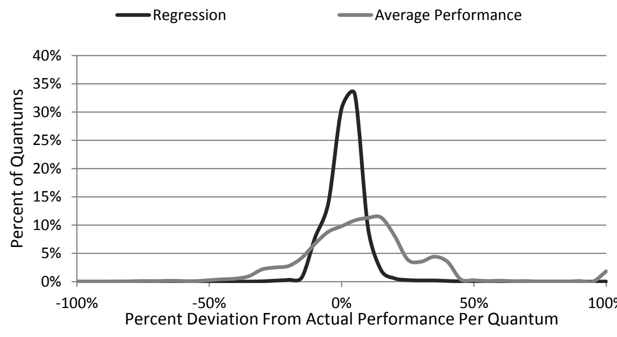


Figure 13: Distribution of Big->Little regression accuracy

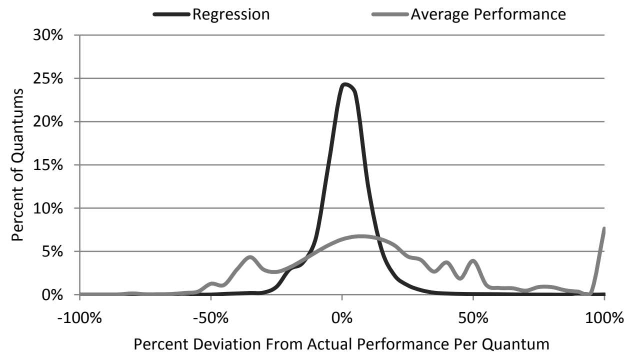


Figure 14: Distribution of Little->Big regression accuracy

after accounting for the inefficiencies of the little μ*Engine*, a *Composite Core* is still able to achieve a 27% decrease in average power compared to the traditional heterogeneous multicore approach.

5.3. Regression

While the oracle switching scheme was useful to determine the best quantum length, it is not implementable in a real system. Therefore in this section, we evaluate the accuracies of the performance model from Section 4.1. Figures 13 and 14 illustrate the accuracy for the Big->Little and Little->Big models respectively. The y-axis indicates the percent of the total quanta, or scheduling intervals. The x-axis indicates the difference between the estimated and actual performance for a single quantum. The accuracy of using a fixed estimate equal to the average performance of the inactive μ*Engine* is also given for comparison.

As the little μ*Engine* has less performance variance and fewer features, it is easier to model, and the Big->Little model is more accurate. However, the Little->Big model must predict the performance of the big μ*Engine*, which has hardware features that were designed to overlap latency, causing it to be less accurate. Also note that while the individual predictions have a larger tail, the tail is centered around zero error. Hence over a large number of quanta, positive errors are canceled by negative errors, allowing the overall performance estimate, CPI_{target} to be more accurate despite the variations in the models themselves.

5.4. Little Core Utilization

For Section 5.4-5.6 we evaluate three different switching schemes configured to allow a maximum of 5% performance degradation. The **Oracle** is the same as in Section 5.1 and picks only the best quanta to run on the little μ*Engine* so that it can still achieve its performance target. The **Perfect Past** has oracle knowledge of the past quanta only, and relies on the assumption that the next quantum has the same performance as the most recent past quantum. The realistic **Regression Model** can measure the performance of the active μ*Engine*, but must rely on a performance model for the estimated performance of the inactive μ*Engine*. This model is the same for all benchmarks and was described in Section 4.1.

Figure 15 illustrates the little μ*Engine* utilization, measured in dynamic instructions, for various benchmarks using each switching scheme. For a memory bound application, like `mcf`, a *Composite Core* can map nearly 100% of the execution to the little μ*Engine*. For applications that are almost entirely computation bound with predictable memory access patterns, the narrower width of the little μ*Engine* limits its overall utilization. However, most applications lie somewhere between these extremes and the *Composite Core* is able to

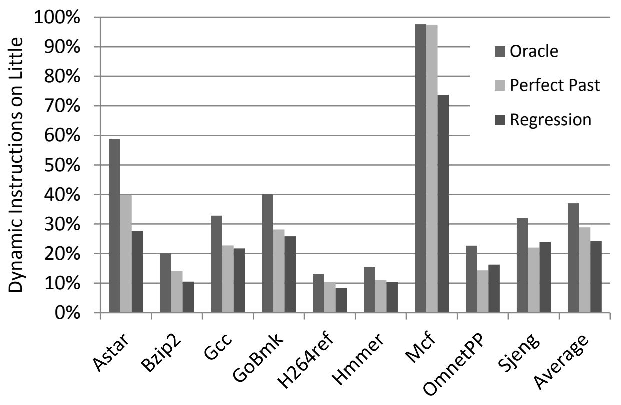


Figure 15: Little μ*Engine* utilization in dynamic instructions, for different switching schemes

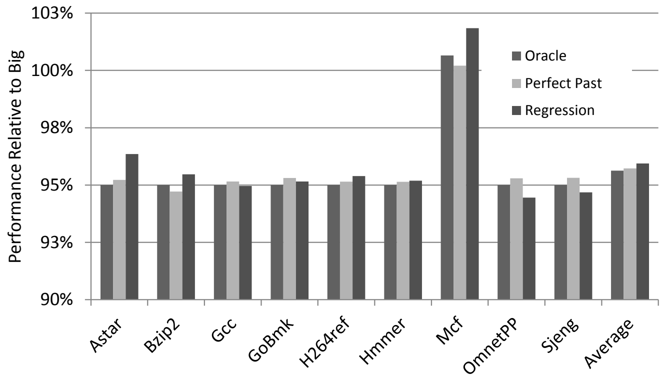


Figure 16: Performance impact for various switching schemes with a 5% slowdown target

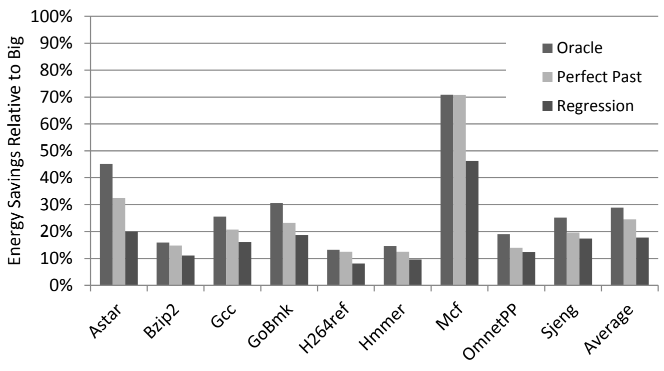


Figure 17: Energy savings for various switching schemes

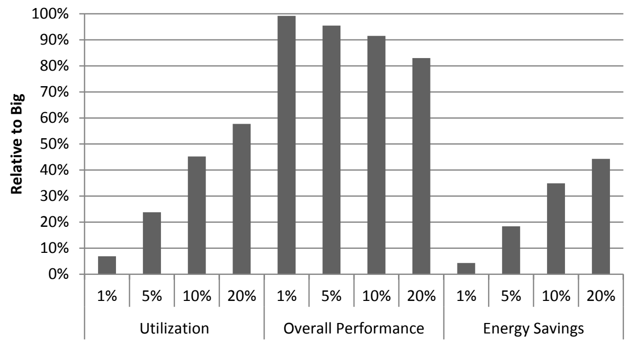


Figure 18: Slowdown sensitivity analysis

map between 20% to 60% of the instructions given oracle knowledge, with an average of 37% utilization. Given the imperfect regression model, these utilizations drop slightly, but still maintain an average utilization of 25% across all benchmarks. Finally on `omnetpp` and `sjeng`, the regression scheme actually achieves higher utilization than the perfect past, however this comes at the cost of a performance loss that is slightly below the target described in Section 5.5.

5.5. Performance Impact

Figure 16 illustrates the performance of the *Composite Core* relative to running the entire application on the big μ*Engine*. *Composite Core* is configured to allow a 5% slowdown, so the controller is targeting 95% relative performance. As `mcf` is almost entirely memory bound, the decrease in branch misprediction recovery latency actually causes a small performance speedup. All other benchmarks are at or near the target performance for all schemes. Note that the controller is designed to allow a small amount of oscillation around the exact performance target to reduce unnecessary switching, thus allowing `bzip2` to dip slightly below the target for the perfect past switching scheme. Both `omnetpp` and `sjeng` suffer from slight inaccuracies in the regression model which, when combined with the oscillation of the controller, causes their overall performance to be approximately an additional $\frac{1}{2}$% below the target performance.

5.6. Energy Reduction

Figure 17 illustrates the energy savings for different switching schemes across all benchmarks. Note that these results only assume clock-gating, meaning that both cores are always leaking static energy regardless of utilization. Again, as `mcf` is almost entirely memory bound, the *Composite Core* is able to map almost the entire execution to the little μ*Engine* and achieve significant energy savings. Overall, the oracle is able to save 29% the energy. Due to the lack of perfect knowledge, the perfect past scheme is not able to utilize the little μ*Engine* as effectively, reducing its overall energy savings to 24%. Finally, the implementable regression model achieves 18% energy savings as the additional inaccuracies in the regression model further reduce the effective utilization of the little μ*Engine*. When combined with the performance, the regression model is able to achieve a 21% reduction in EDP.

5.7. Allowed Performance Loss

As the *Composite Core* can be controlled to provide different levels of energy savings by specifying permissible performance slowdowns, the end user or OS can choose how much of a performance loss is tolerable in exchange for energy savings. Figure 18 illustrates the little μ*Engine* utilization, performance, and energy savings relative to the big μ*Engine* for various performance levels. As the system is tuned to permit a higher performance drop, utilization of the little μ*Engine* increases resulting in higher energy savings. Allowing only a 1% slowdown saves up to 4% of the energy whereas tuning to a 20% performance drop can save 44% of the energy consumed on the big μ*Engine*. This ability is particularly useful in situations where maintaining usability is essential, such as low-battery levels on laptops and cell phones.

6. Related Works

Numerous works motivate a heterogeneous multi-core design for the purposes of performance [22, 2, 4], power [20], and alleviating serial bottlenecks [10, 30, 13]. This paradigm has even begun to make its way into commercial products [9]. The heterogeneous design space can be broadly categorized into 1) designs which migrate thread context across heterogeneous processors, 2) designs which allow a thread to adapt (borrow, lend, or combine) hardware resources, and 3) designs which allow dynamic voltage/frequency scaling.

6.1. Heterogeneous Cores, Migratory Threads

Composite Cores falls within the category of designs which migrate thread context. Most similarly to our technique, Kumar et al. [20] consider migrating thread context between out-of-order and in-order cores for the purposes of reducing power. At coarse granularities of 100M instructions, one or more of the inactive cores are sampled by switching the thread to each core in turn. Switches comprise flushing dirty L1 data to a shared L2, which is slow and energy consuming. Rather than relying on sampling the performance on both cores, Van Craeynest et al. [31] propose a coarse-grained mechanism that relies on measures of CPI, MLP, and ILP to predict the performance on the inactive core.

On the other hand, Rangan et al. [27] examine a CMP with clusters of in-order cores sharing L1 caches. While the cores are identical architecturally, varied voltage and frequency settings create performance and power heterogeneity. A simple performance model is made possible by having exclusively in-order cores, and thread migration is triggered every 1000 cycles by a history-based (last value) predictor. Our solution combines the benefits of architectural heterogeneity [21], as well as those of fast migration of only register state, and contributes a sophisticated mechanism to estimate the inactive core's performance.

Another class of work targets the acceleration of bottlenecks to thread parallelism. Segments of code constituting bottlenecks are annotated by the compiler and scheduled at runtime to run on a big core. Suleman et al. [30] describe a detailed architecture and target critical sections, and Joao et al. [13] generalize this work to identify the most critical bottlenecks at runtime. Patsilaras, Choudhary, and Tuck [25] propose building separate cores, one that targets MLP and the other that targets ILP. They then use L2 cache miss rate to determine when an application has entered a memory intensive phase and map it to the MLP core. When the cache misses decrease, the system migrates the application back to the ILP core.

Other work studies the benefits of heterogeneity in real systems. Annavaram et al. [2] show the performance benefits of heterogeneous multi-cores for multithreaded applications on a prototype with different frequency settings per core. Kwon et al. [23] motivate asymmetry-aware hypervisor thread schedulers, studying cores with various voltage and frequency settings. Koufaty et al. [17] discover an application's big or little core bias by monitoring stall sources, to give preference to OS-level thread migrations which migrate a thread to a core it prefers. A heterogeneous multi-core prototype is produced by throttling the instruction retirement rate of some cores down to one instruction per cycle.

6.2. Adaptive Cores, Stationary Threads

Alternatively, asymmetry can be introduced by dynamically adapting a core's resources to its workload. Prior work has suggested adapting out-of-order structures such as the issue queue [3], as well as other structures such as ROBs, LSQs, and caches [26, 5, 1]. Kumar et al. [19] explored how a pair of adjacent cores can share area-expensive structures, while keeping the floorplan in mind. Homayoun et al. [11] recently examined how microarchitectural structures can be shared across 3D stacked cores. These techniques are limited by the structures they adapt and cannot for instance switch from an out-of-order core to an in-order core during periods of low ILP.

Ipek et al. [12] and Kim et al. [14] describe techniques to compose or fuse several cores into a larger core. While these techniques provide a fair degree of flexibility, a core constructed in this way is generally expected to have a datapath that is less energy efficient than if it were originally designed as an indivisible core of the same size.

6.3. Dynamic Voltage and Frequency Scaling (DVFS)

DVFS approaches reduce the voltage and frequency of the core to improve the core's energy efficiency at the expense of performance. However, when targeted at memory-bound phases, this approach can be effective at reducing energy with minimal impact on performance. Similar to traditional heterogeneous multicore systems, the overall effectiveness of DVFS suffers from coarse-grained scheduling intervals in the millisecond range. In addition, providing independent DVFS settings for more than two cores is costly in terms of both area and energy [16]. Finally, traditional DVFS is only effective when targeting memory-bound phases, while the *Composite Core* architecture can also target phases of serial computation, low instruction level parallelism and high branch-misprediction rates.

Despite these limitations, DVFS is still widely used in production processors today, and has been incorporated into ARM's big.LITTLE heterogeneous multicore system [9]. Similar to big.LITTLE, DVFS could easily be incorporated into a *Composite Core* design. Here the operating system would attempt to maximize energy savings by reducing the voltage for the entire *Composite Core* at a coarse granularity of multiple operating system scheduling intervals. Within these intervals, the *Composite Core* would act as an additional layer of optimization by exploiting fine-grained phases to further reduce energy consumption. This approach can designed to achieve maximum energy savings by allowing DVFS and *Composite Core* to work together to save energy by targeting both coarse-grained and fine-grained phases.

In the future, a *Composite Core* may be able to utilize heterogenity in terms of both microarchitecture and voltage/frequency scaling to further improve energy efficiency. Two competing techniques to enable fine-grained DVFS, fast on-chip regulators [16, 15] and dual-voltage rails [24, 8], have recently been proposed that promise to deliver transition latencies similar to that of a *Composite Core*. These would allow the *Composite Core* to simultaneously switch μ*Engines* and scale the operating voltage/frequency to further maximize energy savings.

7. Conclusion

This paper explored the implications of migration between heterogeneous systems at a much finer granularity than previously proposed. We demonstrated the increased potential to utilize a more energy efficient core at finer intervals than traditional heterogeneous multicore systems. We proposed *Composite Cores*, an architecture that brings the concept of heterogeneity from between different cores to *within* a core by utilizing two tightly coupled μ*Engines*. A *Composite Core* takes advantages of increased hardware sharing to enable fine-grained switching while achieving near zero migration overheads. The *Composite Core* also includes an intelligent controller designed to maximize the utilization of the little μ*Engine* while constraining performance loss to a user-defined threshold. Overall, our system can map an average of *25% of the dynamic execution* to the little μ*Engine* and *reduce energy by 18%* while maintaining a *95% performance target*.

8. Acknowledgements

This work is supported in part by ARM Ltd and by the National Science Foundation under grant SHF-1227917. The authors would like to thank the fellow members of the CCCP research group, our shepherd (Krste Asanovic), and the anonymous reviewers for their time, suggestions, and valuable feedback.

References

[1] D. Albonesi, R. Balasubramonian, S. Dropsbo, S. Dwarkadas, E. Friedman, M. Huang, V. Kursun, G. Magklis, M. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. Cook, and S. Schuster, "Dynamically tuning processor resources with adaptive processing," *IEEE Computer*, vol. 36, no. 12, pp. 49 –58, Dec. 2003.

[2] M. Annavaram, E. Grochowski, and J. Shen, "Mitigating amdahl's law through epi throttling," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005, pp. 298–309.

[3] R. Bahar and S. Manne, "Power and energy reduction via pipeline balancing," *Proc. of the 28th Annual International Symposium on Computer Architecture*, vol. 29, no. 2, pp. 218–229, 2001.

[4] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *Proc. of the 32nd Annual International Symposium on Computer Architecture*, Jun. 2005, pp. 506 – 517.

[5] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 245–257.

[6] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[7] L. Chen, S. Dropsho, and D. Albonesi, "Dynamic data dependence tracking and its application to branch prediction," in *Proc. of the 9th International Symposium on High-Performance Computer Architecture*, 2003, pp. 65–.

[8] R. Dreslinski, "Near threshold computing: From single core to many-core energy efficient architectures," Ph.D. dissertation, University of Michigan, 2011.

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

[10] M. Hill and M. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, no. 7, pp. 33 –38, 2008.

[11] H. Homayoun, V. Kontorinis, A. Shayan, T.-W. Lin, and D. M. Tullsen, "Dynamically heterogeneous cores through 3d resource pooling," in *Proc. of the 18th International Symposium on High-Performance Computer Architecture*, 2012, pp. 1–12.

[12] E. Ipek, M. Kirman, N. Kirman, and J. Martinez, "Core fusion: Accommodating software diversity in chip multiprocessors," in *Proc. of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 186–197.

[13] J. A. Joao, M. Suleman, O. Mutlu, and Y. N. Patt, “Bottleneck identification and scheduling in multithreaded applications,” in *20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 223–234.

[14] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, “Composable lightweight processors,” in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 381–394.

[15] W. Kim, D. Brooks, and G.-Y. Wei, “A fully-integrated 3-level dc-dc converter for nanosecond-scale dvfs,” *IEEE Journal of Solid-State Circuits*, vol. 47, no. 1, pp. 206 –219, Jan. 2012.

[16] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, “System level analysis of fast, per-core dvfs using on-chip switching regulators,” in *Proc. of the 14th International Symposium on High-Performance Computer Architecture*, 2008, pp. 123–134.

[17] D. Koufaty, D. Reddy, and S. Hahn, “Bias scheduling in heterogeneous multi-core architectures,” in *Proc. of the 5th European Conference on Computer Systems*, 2010, pp. 125–138.

[18] R. Krishnamurthy, H. Schmit, and L. Carley, “A low-power 16-bit multiplier-accumulator using series-regulated mixed swing techniques,” in *Custom Integrated Circuits Conference, 1998. Proceedings of the IEEE 1998*, 1998, pp. 499 –502.

[19] R. Kumar, N. Jouppi, and D. Tullsen, “Conjoined-core chip multiprocessing,” in *Proc. of the 37th Annual International Symposium on Microarchitecture*, 2004, pp. 195–206.

[20] 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 *Proc. of the 36th Annual International Symposium on Microarchitecture*, Dec. 2003, pp. 81–92.

[21] R. Kumar, D. M. Tullsen, and N. P. Jouppi, “Core architecture optimization for heterogeneous chip multiprocessors,” in *Proc. of the 15th International Conference on Parallel Architectures and Compilation Techniques*, 2006, pp. 23–32.

[22] 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 *Proceedings of the 31st annual international symposium on Computer architecture*, 2004.

[23] Y. Kwon, C. Kim, S. Maeng, and J. Huh, “Virtualizing performance asymmetric multi-core systems,” in *Proc. of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 45–56.

[24] T. N. Miller, X. Pan, R. Thomas, N. Sedaghati, and R. Teodorescu, “Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips,” in *Proc. of the 18th International Symposium on High-Performance Computer Architecture*, vol. 0, 2012, pp. 1–12.

[25] G. Patsilaras, N. K. Choudhary, and J. Tuck, “Efficiently exploiting memory level parallelism on asymmetric coupled cores in the dark silicon era,” *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 28:1–28:21, Jan. 2012.

[26] D. Ponomarev, G. Kucuk, and K. Ghose, “Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources,” in *Proc. of the 34th Annual International Symposium on Microarchitecture*, Dec. 2001, pp. 90–101.

[27] K. K. Rangan, G.-Y. Wei, and D. Brooks, “Thread motion: fine-grained power management for multi-core systems,” in *Proc. of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 302–313.

[28] L. Sheng, H. A. Jung, R. Strong, J.B.Brockman, D. Tullsen, and N. Jouppi, “Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *Proc. of the 42nd Annual International Symposium on Microarchitecture*, 2009, pp. 469–480.

[29] J. Suh and M. Dubois, “Dynamic mips rate stabilization in out-of-order processors,” in *Proc. of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 46–56.

[30] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, “Accelerating critical section execution with asymmetric multi-core architectures,” in *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009, pp. 253–264.

[31] 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*, ser. ISCA ’12, 2012, pp. 213–224.

[32] 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*, 2003, pp. 84–97.

[33] B. Xu and D. H. Albonesi, “Methodology for the analysis of dynamic application parallelism and its application to reconfigurable computing,” vol. 3844, no. 1. SPIE, 1999, pp. 78–86.