Exploring Fine-Grained Heterogeneity with Composite Cores

Andrew Lukefahr, *Student Member, IEEE*, Shruti Padmanabha, *Student Member, IEEE*, Reetuparna Das, *Member, IEEE*, Faissal M. Sleiman, *Student Member, IEEE*, Ronald G. Dreslinski, *Member, IEEE*, Thomas F. Wenisch, *Member, IEEE*, and Scott Mahlke, *Fellow, IEEE*

Abstract—Heterogeneous multicore systems—comprising multiple cores with varying performance and energy characteristics—have emerged as a promising approach to increasing energy efficiency. Such systems reduce energy consumption by identifying application phases and migrating execution to the most efficient core that meets performance requirements. However, the overheads of migrating between cores limit opportunities 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 migration overheads by bringing heterogeneity *into* a core. *Composite Cores* pairs a big and little compute μ*Engine* that together achieve high performance and energy efficiency. By sharing architectural state between the μ*Engines*, the migration overhead is reduced, enabling fine-grained migration and increasing the opportunities to utilize the little μ*Engine* without sacrificing performance. An intelligent controller migrates the application between μ*Engines* 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, *Composite Cores* are able to map 30% of the execution time to the little μ*Engine*, achieving a 21% energy savings while maintaining 95% performance.

Index Terms—Adaptive architecture, heterogeneous processors, hardware scheduling, fine-grain phases

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, making it possible for computer architects to trade increased area for improved energy efficiency.

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 [19]. 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 to a new core. Traditional designs select the best core by briefly sampling performance on each core. However, every time the application is migrated to a new core, its current state must be explicitly transferred or rebuilt on the new core. This state transfer incurs large overheads that limit the migration between cores to a *coarse granularity* of tens to hundreds of millions of instructions. To mitigate these effects, the migration is only done at the granularity of operating system time slices.

This work postulates that the coarse-grained migration 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 migration and is unencumbered by the large state transfer latency of current designs. To accomplish this goal, we propose *Composite Cores*, an architecture that brings the concept of heterogeneity *into* a single core. A *Composite Core* contains several compute μ*Engines* that together can achieve both high performance and energy efficiency. In this work, we consider a dual μ*Engine Composite Core* consisting of: a high performance pipeline (referred to as the big μ*Engine*) and an energy efficient pipeline (referred to as the little μ*Engine*). As only one μ*Engine* is active at a time, execution migrates dynamically between μ*Engines* to best match the current application's characteristics to the hardware resources. As this occurs on a much finer granularity (on the order of a thousand instructions) compared to past heterogeneous multicore proposals it allows the

• *A. Lukefahr, S. Padmanabha, R. Das, F.M. Sleiman, R.G. Dreslinski, T.F. Wenisch and S.Mahlke are with the Advanced Computer Architecture Laboratory, University of Michigan, Ann Arbor, MI 48109. E-mail: lukefahr,shrupad,reetudas,sleimanf,rdreslin,twenisch,mahlke@umich.edu*

Manuscript received April 10, 2014; revised January 13, 2015.

application to *spend more time on the energy efficient μEngine without sacrificing additional performance*.

As a *Composite Core* migrates frequently between *μEngines*, it relies on hardware resource sharing and low-overhead transfer techniques to achieve near zero migration overhead. For example, both *μEngines* share branch predictors, L1 caches, fetch units and TLBs. This sharing ensures that during a migration only the register state needs to be transfered between *μEngines*. We further propose optimizations to a *Composite Core* that both improve energy savings and simplify physical layout.

Because of the fine migration 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 than the active *μEngine*, the application is migrated. Thus, the decision to migrate *μEngines* maximizes execution on the more efficient little *μEngine* subject to a performance degradation constraint.

The migration 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 decision logic, which maintains the current performance within the allowed performance bound, and a reactive model to detect the *instantaneous performance difference* via online performance estimation techniques.

In summary, this paper offers the following contributions:

- We propose *Composite Cores*, an architecture that brings the concept of heterogeneity *into* 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 migration in the context of heterogeneous core architectures. To achieve near zero *μEngine* transfer overhead, we propose low-overhead migration techniques and a core microarchitecture which shares necessary hardware resources.
- We design intelligent migration decision logic that facilitates fine-grain migration 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 study the performance and energy implications of several architectural designs of a *Composite Core* with the goal of maximizing both energy savings and physical layout. We propose the addition of a small L0 filter cache [15] for the little *μEngine*, as well as evaluate the effects of various migration techniques.
- 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 21%* while bounding performance degradation to at most 5%.

2 Motivation

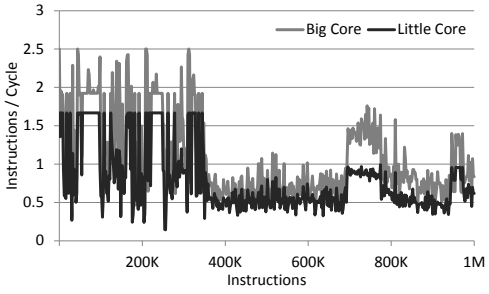
Industry interest in heterogeneous multicore designs has been gaining momentum. ARM's heterogeneous multicore, known as big.LITTLE [9], combines a set of Cortex-A15 (Big) cores with Cortex-A7 (Little) cores to create a heterogeneous processor. The Cortex-A15 is a 3-way out-of-order with deep pipelines (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, this overhead requires that the system migrate between cores only at coarse granularity, on the order of tens of milliseconds. The large scheduling interval forfeits potential gains afforded by a more aggressive *fine-grained* migration.

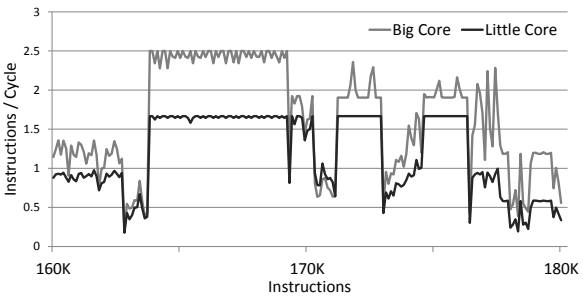
2.1 Migration Interval

Traditional heterogeneous multicore systems, such as big.LITTLE, rely on coarse-grained migration to exploit application phases that are 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. Prior works have shown that observing performance at finer granularity reveals more low-performance periods, increasing opportunities to utilize a more energy efficient core [26], [32].

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 25% slower on average than the big core, which may necessitate that the entire phase be run on the big core. However, when observing the performance with finer granularity, we observe that there are numerous



(a) Instruction window of length 2K over 1M instructions



(b) Instruction window of length 100 over a 200K instructions

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

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 again see brief regions with minimal performance gap.

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.

3 Architecture

A *Composite Core* consists of tightly-coupled big and little compute μ*Engines* that can achieve high performance and energy efficiency by rapidly migrating between the μ*Engines* in response to changes in application performance. To reduce the overhead of migration, the μ*Engines* share as much state as possible. As Figure 2 illustrates, the μ*Engines* 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 μ*Engine*'s register access energy. However, as both μ*Engines* require different control signals from decode, each μ*Engine* has its own decode stage.

As the μ*Engines* target different performance and energy tradeoffs, each μ*Engine* has a separate back-end implementation. The big μ*Engine* is a highly-pipelined superscalar 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. It 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 μ*Engine* features a reduced issue width, simpler issue logic, reduced functional units, and omits many of the associatively searched structures (such as the issue queue or LSQ). By only maintaining an architectural register file, the little μ*Engine* eliminates the need for renaming and improves the efficiency of register file accesses.

As a base design, both μ*Engines* multiplex access to a single L1 data cache, again to maximize shared state and reduce migration overheads. However, sharing a cache between two backends might impact cache access latencies for one or both μ*Engines*. Therefore we explore a design where the L1 data cache is tightly integrated with the big μ*Engine*, maintaing its access latency at the expense of the little μ*Engine*. We then introduce an L0 data cache to filter many of the little μ*Engine*'s access, alleviating it's latency penalty.

Figure 3 gives an approximate layout of a *Composite Core* system with an L0 at 32nm. The big μ*Engine* consumes $6.3mm^2$ and the L1 caches consume an additional $1.4mm^2$. The little μ*Engine* without an L0 adds an additional $1.8mm^2$, or about a 20% area overhead. The L0 cache consumes another $0.05mm^2$, but alleviates some of the difficulties in routing two high-performance access paths to the same cache. Finally, the *Composite Core* control logic adds an additional $0.02mm^2$ or 0.2% overhead.

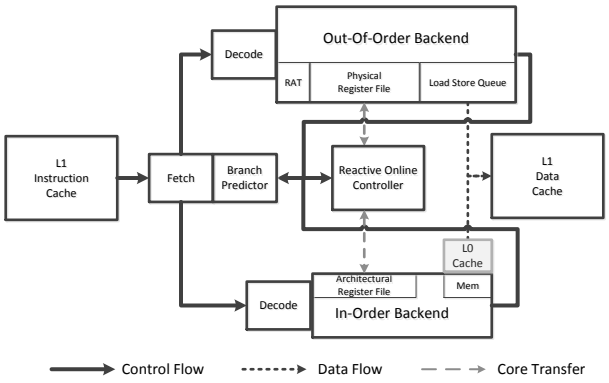


Fig. 2. Microarchitectural overview of a *Composite Core*. The optional L0 cache is shown in grey.

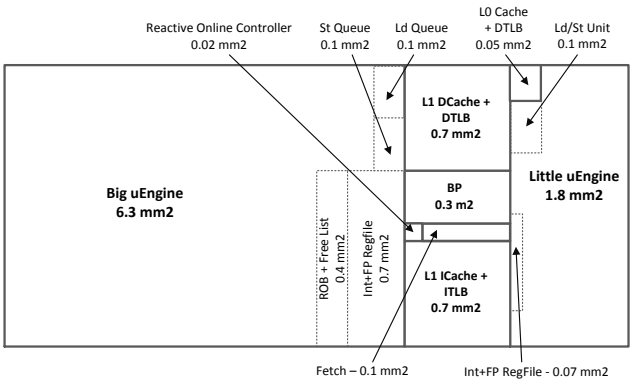


Fig. 3. Estimated physical layout of a *Composite Core* in 32nm technology.

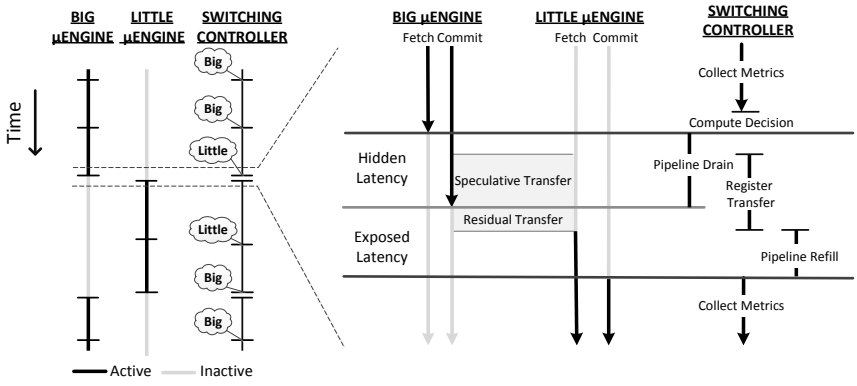


Fig. 4. Overview of a *Composite Core* migration when utilizing pipeline draining.

3.1 μ*Engine* Transfer

During execution, the Reactive Online Controller collects a variety of performance metrics and uses these to determine the μ*Engine* to activate for the following quantum. If, at the end of the quantum, the controller determines that the following quantum should be run on the inactive μ*Engine*, the *Composite Core* must migrate control to the new μ*Engine*. As both μ*Engines* have different backend implementations, they have incompatible microarchitectural state. Therefore before migration, the current active μ*Engine* must first be brought to an architecturally precise point for control to be transferred.

The simplest approach, shown in Figure 4, is to drain the active μ*Engine*'s pipeline. While the pipeline is draining, the register contents can be speculatively transferred to the inactive μ*Engine*. If draining instructions update a previously-transferred register, the register must again be transferred during the residual transfer. Once the register transfer is complete, fetch resumes dispatching instructions to the newly active μ*Engine*, which must refill its pipeline before committing instructions. As the pipeline drain hides a majority of the register transfer latency, 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 total exposed migration overheads are roughly equivalent to a branch misprediction recovery.

If, while draining the active μ*Engine*, a long latency instruction stalls the pipeline, draining may take a long time and negate much of the potential energy savings of the migration. A second approach is to immediately flush (or squash) all speculative state in the pipeline, and immediately transfer control to the inactive μ*Engine*. While this allows a quicker migration, it is potentially wasteful as the pipeline may contain a large amount of completed work that we do not wish to flush.

A third option is a hybrid approach. When a migration is triggered, the active μ*Engine* is allowed to drain, or commit instructions, as long as there are instructions ready to commit. In this way, completed instructions are allowed to commit rather than being squashed. However, if commit stalls for any reason, the pipeline will be squashed and execution migrated to the new μ*Engine* as desired.

The energy implications of all three options are evaluated in Section 5.8.

3.2 L0 Cache

As mentioned previously, the L1 data cache should be placed in physical proximity to the big μ*Engine*, with the goal of maximizing its performance. As this

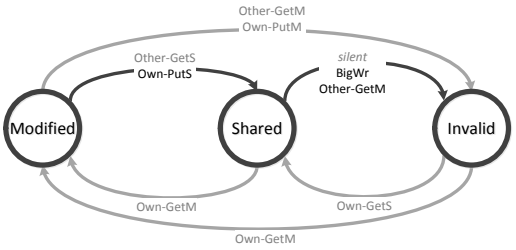


Fig. 5. L0's Writeback-On-Migrate Protocol. Bold Transitions have been added to the standard MSI protocol [28].

increases access latencies for the little μ*Engine*, we provisioned it with a private 2kB L0 data cache. As this cache is both near the little μ*Engine* and small, hits provide both low-latency access *and* increased energy efficiency. However, as a L0 miss incurs increased latency and energy consumption, the L0 hit rate must be kept high to provide these benefits. Additionally, the little μ*Engine* accesses the L0 through its own TLB, which is kept in sync with the big μ*Engine*'s TLB. The hit rate and energy savings are further discussed in Section 5.7.

We designed the coherence protocol between the L1 and L0 specifically to prevent the L0 from causing a slowdown when running on the big μ*Engine*. As such, the L1 is inclusive of the L0, preventing the L0 from removing cache blocks from the L1 and causing additional miss latencies for big. Additionally, to prevent the big μ*Engine* from accessing stale data directly from the L1, all dirty blocks in the L0 are flushed back to the L1 before a migration from the little to big μ*Engine* occurs. Finally, as both μ*Engines* are not simultaneously active, the big μ*Engine* will not access the L1 while the little μ*Engine* and L0 are active.

The exact coherence protocol is a modified MSI protocol, [28], called Writeback-On-Migrate (WOM), shown in Figure 5. WOM contains all original MSI transitions, shown in grey, plus three additional transitions, shown in bold. When a migration is triggered, the L0 uses the $Own-PutS$ signal to self-downgrade all blocks to Shared, flushing dirty data back to the L1. After a migration to the big μ*Engine*, the L0 must monitor big's writes and invalidate it's copy of any block which the write updates. While the L1 can already issue the $Other-GetM$ signal to evict a Modified block, as a condition of inclusivity, it must also be able to evict a Shared block using the same $Other-GetM$ signal.

4 Reactive Online Controller

The decision of when to migrate μ*Engines* is handled by the Reactive Online Controller. Our controller, following the precedent established by prior works [19], [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 migration 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 migration 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 migration 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 migration controller uses the output of the performance estimator and the threshold controller to determine which μ*Engine* 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 μ*Engines* in the *previous* quantum as well as track the overall performance for *all* past quanta. While the performance of the active μ*Engine* can be trivially determined by counting the cycles required to complete the current quantum, the performance of the inactive μ*Engine* is not known and must be estimated. This estimation is challenging as the microarchitectural differences in the μ*Engines* cause their behaviors to differ.

The traditional approach is to sample execution on both μ*Engines* 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 migration 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 migrate-to-little-on-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

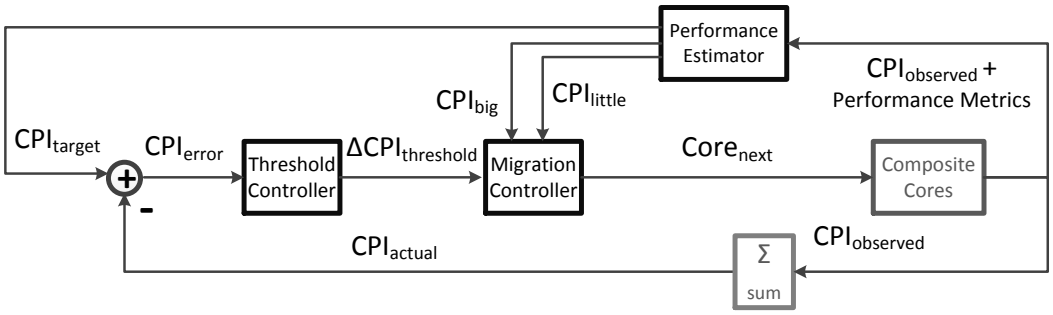


Fig. 6. Overview of the Reactive online controller.

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.

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.

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 *μEngine* is active, this model estimates the performance of the little *μEngine*. The little *μEngine* has a higher constant due to its narrower issue width causing less performance variance. As the little *μEngine* cannot exploit application characteristics like ILP and MLP as well as the big *μEngine*, the big *μEngine*'s performance has slightly less impact than in the Little→Big model. L2 Hits are now more important as, unlike the big *μEngine*, the little *μEngine* is not designed to hide any of the latency. The inability of the little *μEngine* 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 *μEngine* can recover from branch mispredicts much faster, mispredicts have very little impact. Finally even though L2 misses occur infrequently, the little *μEngine* suffers more performance loss than the big *μEngine* 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 *μEngine*, 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 *μEngine* 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.

$$CPI_{actual} = \sum CPI_{observed} \quad (2)$$

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

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 *μEngine*. This threshold is designed to provide an average per-quantum performance loss where using the little *μEngine* is profitable given the performance target. As some applications experience frequent periods of similar performance between *μEngines*, the controller scales the threshold low to ensure the little *μEngine* 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 *μEngine* 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. 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.

$$CPI_{error} = CPI_{target} - CPI_{actual} \quad (4)$$

$$\Delta CPI_{threshold} = K_p CPI_{error} + K_i \sum CPI_{error} \quad (5)$$

4.3 Migration Controller

The migration controller attempts to determine which *μEngine* 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 shown in Figure 7, the controller determines profitability by computing ΔCPI_{net} as shown in Eq. 6. If ΔCPI_{net} is positive, the little *μEngine* is currently more profitable, and execution is mapped to the little *μEngine* for the next quantum. However, if ΔCPI_{net} is negative, the performance difference between big and little is too large, making the little *μEngine* less profitable. Therefore the execution is mapped to the big *μEngine* for the next quantum.

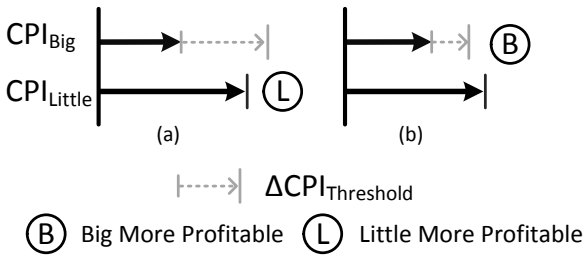


Fig. 7. Migration controller behaviour: (a) If $CPI_{big} + \Delta CPI_{threshold} > CPI_{little}$ pick Little; (b) If $CPI_{big} + \Delta CPI_{threshold} < CPI_{little}$ pick Big.

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

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 *μEngine*. Due to the microarchitectural simplicity of the little *μEngine*, tracking these additional metrics is more complicated. We add a small dependence table (described in Section 4.4.2) to the little *μEngine* 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 *μEngine* 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 μ*Engine* 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 μ*Engine*.

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 μ*Engine*, the total table size is $2 \times 32 \times 128$ bits, 1KB of overhead. As this table is specific to one μ*Engine*, the additional area is factored into the little μ*Engine*'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 [17]. 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 migration 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 to support fast migration [6]. All benchmarks were compiled using gcc with -O2 optimizations for the Alpha ISA. We fast forwarded all benchmarks for two billion instructions before beginning detailed simulations for an additional one billion instructions. The simulations included detailed modeling of the pipeline drain/flush functionality for migrating between μ*Engines*.

We utilized McPAT to estimate the energy savings from a *Composite Core* [27]. 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 power gating at these granularities. Finally, as our design assumes tightly coupled L1 caches, our estimates include the energy consumption of the L1 instruction and data caches (and L0 data cache when included), but neglect all other system energy estimates.

Component	Parameters
Big μ*Engine*	3 wide Out-Of-Order @ 1.0GHz 12 stage pipeline 128 ROB entries 160 entry register file Tournament branch predictor (Shared)
Little μ*Engine*	2 wide In-Order @ 1.0GHz 8 stage pipeline 32 entry register file Tournament branch predictor (Shared)
Memory System	2KB L0 dcache, 1 cycle access (Little Only) 32KB L1 dCache, 2 cycle access (Shared) 32KB L1 iCache, 2 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 μ*Engines* as well as the memory system configuration. The big μ*Engine* 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 μ*Engine* is modeled to simulate a 2-wide in-order processor with a 32-entry architectural register file. Its simplified hardware structures shorten the pipeline depth, providing quicker branch misprediction recovery. The branch predictor and fetch stage are shared between the two μ*Engines*. Results from Section 5.1-5.6 do not include an L0, while Section 5.7-5.9 do.

5.1 Quantum Length

One of the primary goals of the *Composite Cores* architecture is to exploit short duration phases of low performance using fine-grained quanta. 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 achieve an upper bound, we assume the μ*Engine* selection is determined by an oracle, which knows the performance for both μ*Engines* for all quanta and migrates to the little μ*Engine* 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 μ*Engine*.

Given these assumptions, Figure 8 demonstrates the little μ*Engine*'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 sizes shrink below this level, the utilization begins to rise from approximately thirty percent to fifty percent at quantum lengths of one hundred instructions.

While a *Composite Core* is designed to minimize overheads, migration still incurs a small register transfer and pipeline refill latency. Figure 9 illustrates the performance impacts of these migrations at various quanta with the oracle targeting 95% performance relative to the all big *μEngine* case. We observe that, with the exception of `mcf`, all the benchmarks achieve the target performance at longer quanta. This result implies that the additional overheads of migration are negligible at these quanta and can safely be ignored. However, for quantum lengths around 1000 instructions we begin to see additional performance degradation, indicating that the migration overheads are no longer negligible.

The main cause of this performance decrease is the additional migrations allowed by the smaller quanta. Figure 10 illustrates the number of migrations 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 migrations. In particular, for a quantum length of 1000 the oracle migrates 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 migrate more frequently to achieve this goal. Increased hardware sharing allows the *Composite Core* to migrate at a much finer granularity than traditional heterogeneous multicore architectures. However below quantum lengths of approximately 1000 dynamic instructions, the overheads of migration begin to cause intolerable performance degradation. Therefore for the remainder of this study, we fix the quantum length to 1000 instructions.

5.2 *μEngine* Power Consumption

A *Composite Core* relies on shared hardware structures to enable fine-grained migration. However these shared structures must be designed for the high performance big *μEngine* and are over-provisioned when the little *μEngine* is active. The little *μEngine* frontend now includes a fetch engine, branch predictor, and instruction cache designed for the big *μEngine*. Also, the shared L1 data cache, designed to support the big *μEngine*'s need for multiple outstanding memory transactions, is not effectively utilized by the little *μEngine*'s simple pipeline. Finally, the leakage power

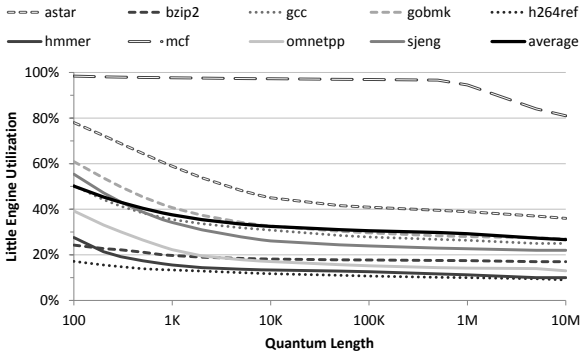


Fig. 8. Impact of quantum length on instructions mapped to the little *μEngine*. Note the increase as quantum length approaches 1K.

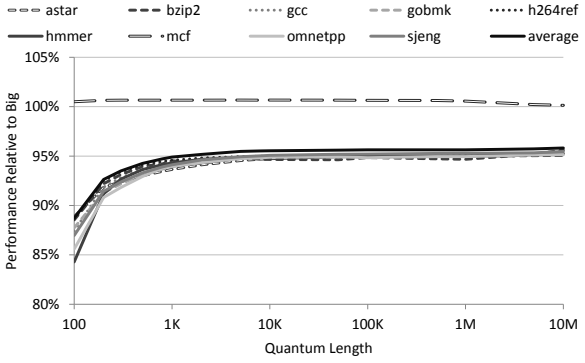


Fig. 9. Impact of quantum length on overall performance with a 5% slowdown target. The overheads remain negligible until 1K.

of *Composite Cores* will be higher as neither *μEngine* can be power gated.

Figure 11 illustrates the average power when the complete application is run on the specified core or *μEngine*. While the little core provides lower performance levels, it consumes almost 6x less power than the big core. Observe that there are minimal overheads for the big *μEngine*, as the leakage power of little is small relative to the dynamic power of big. However, as the little *μEngine* incurs additional leakage power from the big *μEngine* and uses the less-

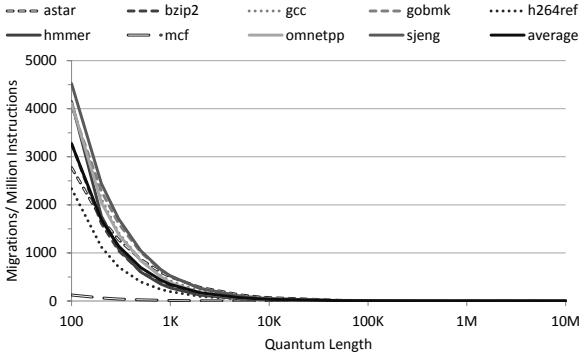


Fig. 10. Impact of quantum length on *μEngine* migrations.

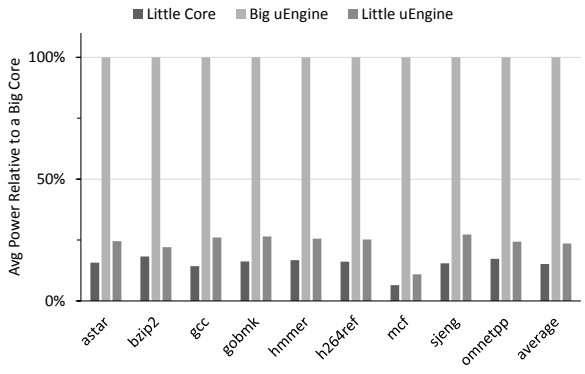


Fig. 11. Average μ*Engine* power relative to dedicated cores. The little μ*Engine* consumes 10% higher power than a little core.

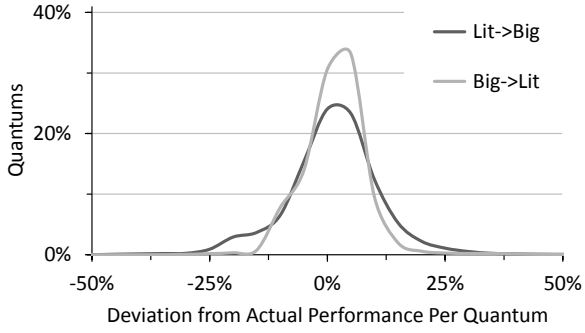


Fig. 12. Accuracy of regression models. 95% ($Big \rightarrow Lit$) and 82% ($Lit \rightarrow Big$) of the quanta have $\leq 10\%$ error.

efficient frontend and data caches, it's average power is about 10% higher than the little core, reducing it's power consumption advantage to 4x. While the little μ*Engine* of a *Composite Core* is not able to achieve the same average power as a separate little core, this limitation is offset by *Composite Core*'s ability to utilize the little μ*Engine* more frequently.

5.3 Regression

As the Reactive Online Controller, of Section 4.1, relies on a model to estimate performance, in this section we evaluate its accuracy. Figure 12 illustrates the accuracy of both the Big→Little and Little→Big models. The y-axis indicates the percent of the total quanta, or scheduling intervals. The x-axis indicates the difference (or error) between the estimated and actual performance for a single quantum. Higher peak values indicate a greater percentage of quantums with 0% Deviation. Wider curves indicate a greater amount of error is being introduced.

As the little μ*Engine* has less performance variation and fewer features, it is easier to model, causing the Big→Little model to be more accurate, with 95% of the quanta having ¡10% error. On the other hand, the Little→Big model must rely on the little μ*Engine*'s limited features to predict the performance of the big μ*Engine*, which has advanced hardware features

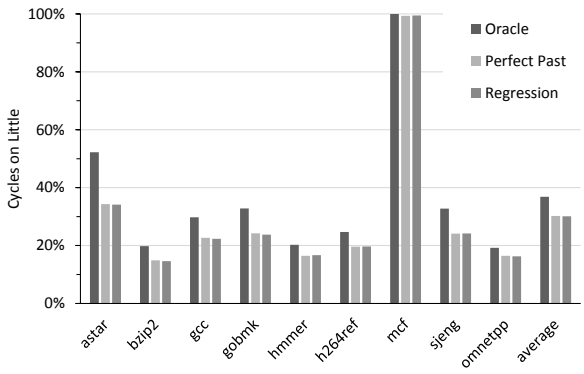


Fig. 13. Cycles on the Little μ*Engine*, for different migration schemes. Note the similarities between Perfect Past and Regression.

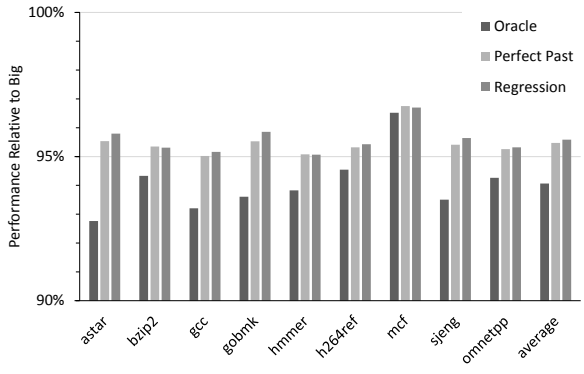


Fig. 14. Performance impact for various migration schemes with a 5% slowdown target. Oracle switching incurs 1% switching overheads, while Perfect Past and Regression adapt to maintain performance targets.

designed to overlap latency nonlinearly. This causes it to have a slightly lower percentage (82%) that are within 10% error. Also note that, as the error is centered around zero, over a large number of quanta, positive errors are canceled by negative errors. This allows the overall performance estimate, CPI_{target} to be more accurate despite the variations in the models themselves.

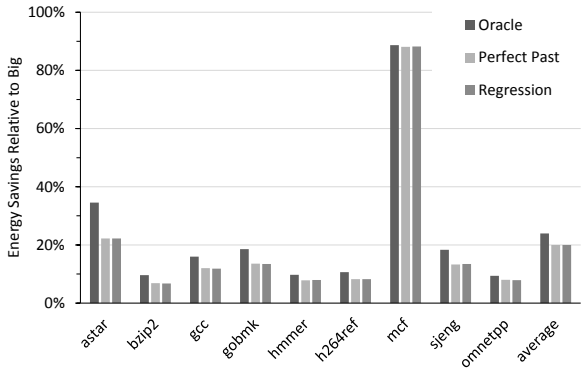


Fig. 15. Energy savings for various migration schemes.

5.4 Little Core Utilization

For Section 5.4-5.6 we evaluate three different migration 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 *μEngine* 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 *μEngine*, but must rely on a performance model for the estimated performance of the inactive *μEngine*. This model is described in Section 4.1.

Figure 13 illustrates the percent of cycles spent on the little *μEngine* for various benchmarks using each migration scheme. For a memory bound application, like `mcf`, a *Composite Core* can map nearly 100% of the execution to the little *μEngine*. For applications that are almost entirely computation bound with predictable memory access patterns, the narrower width of the little *μEngine* limits its overall utilization. However, most applications lie somewhere between these extremes and the *Composite Core* is able to map between 20% to 55% of the cycles on the little *μEngine* given oracle knowledge, with an average of 37%. Given the imperfect regression model, the average drops to 30%.

5.5 Performance Impact

Figure 14 illustrates the performance of the *Composite Core* relative to running the entire application on the big *μEngine*. *Composite Core* is configured to allow a 5% slowdown, so the controller is targeting 95% relative performance. As can be observed, Oracle falls a few percentage points below the target. This is due to the migration overheads, which the oracle neglects. However, the controller is able to compensate for these overheads, and maintains a target performance at or slightly above the target. As `mcf` is almost entirely memory bound, the controller runs almost the entire application on the little *μEngine* and is actually able to beat the 95% target.

5.6 Energy Reduction

Figure 15 illustrates the energy savings for different migration 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 *μEngine* and achieve significant energy savings. Overall, the oracle is able to save 24% the energy of a Big Core. Due to the lack of perfect knowledge, the perfect past and regression schemes are not able to utilize the little *μEngine* as effectively, reducing their overall energy savings to 20%. Interestingly, the addition of a regression model causes an almost negligible impact on energy savings.

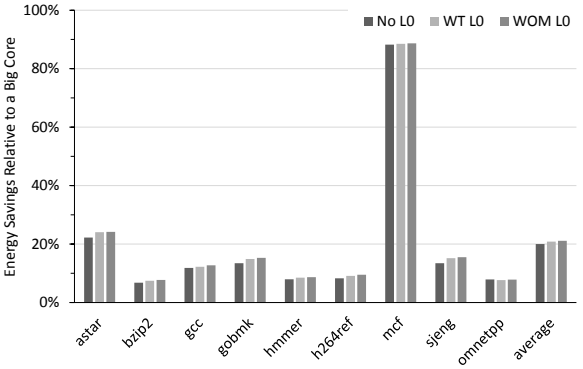


Fig. 16. Impact of additional L0 filter Cache. The WOM (Writeback-On-Migrate) protocol achieves higher energy savings than both No L0 and WT (WriteThrough).

5.7 L0 Cache

As mentioned in Section 3, the physical layout of a *Composite Core* is difficult without impacting cache access latencies. Therefore, we evaluate the impact of adding a small L0 data cache to the little *μEngine*. As the L1 is inclusive of the L0 and the L0's access is single cycle, it effectively filters many of little's accesses to the L1, which has a two-cycle access. While the hit rates vary from as low as 67%, `mcf`, to 94%, `bzip2`, the average hit rate is 86%, decreasing the average memory access time and yielding a small performance improvement. This allows the controller to schedule more quanta on the little *μEngine*.

However, migration is now made more complicated as the *μEngines* no longer share a single data cache. One solution is to make the L0 use a WriteThrough (WT) protocol, which would propagate any update the L1. Figure 16 illustrates that WT's performance improvement coupled with the decrease in access energy for the L0 yields an additional 1% energy savings on average over a No L0 design. However, WT caches cause a significant amount of traffic between the L0 and the L1. As the big *μEngine* is not active while the L0 is being accessed, a Writeback-On-Migrate (WOM) protocol writes back dirty data to the L1 only on migration to the big *μEngine*, reducing the traffic. This allows the L0 cache with WOM to achieve slightly higher savings over WT, increasing the overall energy savings to 21%.

5.8 Migration Technique

Another potential impact on the energy savings is how the *μEngine* is brought to an architecturally stable point before migration. Figures 9 and 14 assume the active *μEngine* is flushed when migration occurs. However, as discussed in Section 3.1, three options need to be considered: draining, flushing, and finishing completed instructions. Figure 17 illustrates the impact of these three techniques. Drain, which re-

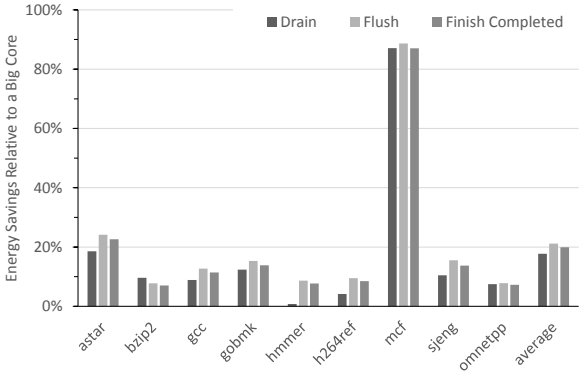


Fig. 17. Impact of different migration techniques. Flushing the pipeline is more effective than either draining or waiting for all completed instructions to commit.

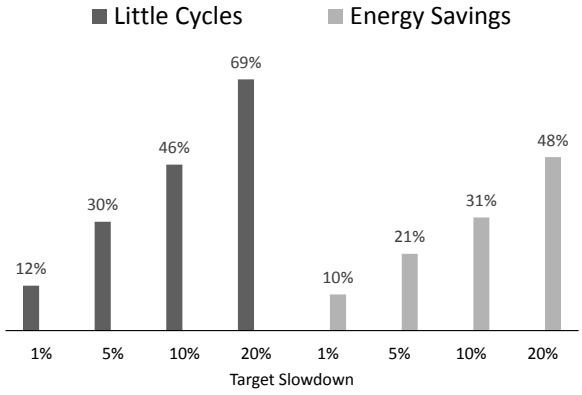


Fig. 18. Slowdown sensitivity analysis. Note the large energy savings (10%) for a minimal target slowdown (1%).

mains on the energy-intensive big *μEngine* the longest while waiting for the pipeline to empty, achieves the least energy savings. The next option, Flush, flushes the pipeline and migrates to the more energy efficient *μEngine* faster. The final option, Finish Completed, allows the pipeline to drain until commit stalls, then flushes. Counter-intuitively, Flush actually achieves higher energy savings than Finished Completed, implying that it is more energy efficient to squash a few completed instructions on the energy-intensive big *μEngine* to allow quicker migration to the more energy-efficient little *μEngine* than it is continue to clock the big *μEngine* while it commits instructions.

5.9 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 *μEngine* cycles as a percent of total runtime and corresponding energy savings for various performance levels using a *Composite Core* with an L0. As the system is tuned to permit a higher performance drop, utilization of the little *μEngine* increases resulting in higher energy savings. Allowing only a 1% slowdown saves up to 10% of the energy whereas tuning to a 20% performance slowdown can save 48% of the energy consumed on the big *μEngine*.

6 Related Work

Numerous works motivate a heterogeneous multicore design for the purposes of performance [21], [2], [4], power [19], and alleviating serial bottlenecks [10], [30], [13]. 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

Similar to our technique, Kumar et al. [19] consider migrating thread context between out-of-order and in-order cores for the purposes of reducing power. At granularities of 100M instructions, the performance of an inactive core is sampled by briefly migrating the thread to the core. Rather than sampling, PIE relies on a model using measures of CPI, MLP, and ILP to predict the performance on the inactive core [31].

Rangan et al. [26] examine fine-grained thread migration using a history-based predictor in a cluster of in-order cores sharing an L1 cache. Varied voltage and frequency settings are used to create performance and power heterogeneity. Fallin et al. uses customized atomic blocks to examine extremely fine-grained heterogeneity[8]. Our solution combines the benefits of both architectural heterogeneity [20] and fast migration of only register state, and contributes a more sophisticated mechanism to estimate the inactive core's performance. Rather than react to performance changes, Padmanabha et al. rely on a predictive model for a *Composite Cores* architecture [23].

Another class of work targets the acceleration of parallel applications. Segments of code constituting bottlenecks are annotated by the compiler and migrated at runtime to a big core. Suleman et al. [30] accelerate the critical sections, and Joao et al. [13] generalize this work to identify more potential bottlenecks at runtime. Patsilaras et al. [24] propose one core that targets MLP and another that targets ILP, and use L2 cache miss rate to determine memory intensive phases and map them to the MLP core.

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. [22] motivate asymmetry-aware thread schedulers. Koufaty et al. [16] 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.

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], ROBs, LSQs, and caches [25], [5], [1]. Kumar et al. [18] explored how conjoined cores can share area-expensive structures, while keeping the floorplan in mind. Homayoun et al. [11] examined how microarchitectural structures can be shared across 3D stacked cores. Ipek et al. [12] and Kim et al. [14] describe techniques to fuse several cores into a larger core. While these techniques provide a fair degree of flexibility, a core constructed in this way often has a datapath that is less energy efficient than an indivisible core of the same size.

6.3 Dynamic Voltage and Frequency Scaling

DVFS approaches reduce the voltage/frequency of the core, improving the core's energy efficiency at the expense of performance. Like traditional heterogeneous multicore systems, the overall effectiveness of DVFS is limited to scheduling intervals in the millisecond range. When targeted at memory-bound phases, this approach can be effective at reducing energy with minimal impact on performance. Unlike DVFS, the *Composite Core* architecture can also target phases of serial computation, low instruction level parallelism and high branch-misprediction rates.

DVFS is widely used in today's processors, including ARM's big.LITTLE heterogeneous multicore system [9]. Similarly to big.LITTLE, DVFS could 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. Within these intervals, the *Composite Core* controller can act as an additional optimization layer by exploiting fine-grained phases. This approach can be 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.

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 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 *30% of the execution time* to the little μ*Engine* and *reduce energy by 21%* while maintaining a *95% performance target*.

8 Acknowledgements

This work is supported in part by ARM Ltd and the National Science Foundation, grant SHF-1217917.

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] C. Fallin, C. Wilkerson, and O. Mutlu, "The heterogeneous block architecture," Carnegie Mellon University, Tech. Rep., March 2014.

[9] P. Greenhalgh, "Big.little processing with arm cortex-a15 & cortex-a7," Sep. 2011, http://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 *17th 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] J. Kin, M. Gupta, and W. Mangione-Smith, "The filter cache: An energy efficient memory structure," in *Proc. of the 30th Annual International Symposium on Microarchitecture*, Dec. 1997, pp. 184–193.

[16] 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.

[17] 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.

[18] 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.
[19] 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.
[20] 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.
[21] 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.
[22] 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.
[23] S. Padmanabha, A. Lukefahr, R. Das, and S. Mahlke, "Trace based phase prediction for tightly-coupled heterogeneous cores," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46, 2013, pp. 445–456.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, 1st ed. Morgan & Claypool Publishers, 2011.
[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.

Andrew Lukefahr received his BS in electrical and computer engineering from the University of Missouri - Columbia in 2010, his MSE in computer science and engineering from the University of Michigan in 2012, and is currently working toward his PhD in computer science and engineering. His research interests include energy efficiency, heterogeneous architectures, task scheduling, and control systems.

Shruti Padmanabha received her BTech in computer engineering from Birla Institude of Technology and Science - Pilani in 2011, her MSE in computer science and engineering from the Univeristy of Michigan in 2013, and is currently working toward her PhD in computer science and engineering. Her research interests include energy efficient processor design, memory architectures, and heterogeneous multicore systems.

Reetuparna Das received her PhD in Computer Science and Engineering from Pennsylvania State University in 2010. She is currently a Research Scientist at the University of Michigan and a researcher in residence for the Center for Future Architecture Research (CFAR). She is a member of the IEEE. Her research interests include energy-efficient mobile architectures, near-data processing for big-data applications, and on-chip interconnection networks.

Faissal M. Sleiman received his BE in computer and communications engineering from the American University of Beirut in 2008, his MSE in computer science and engineering from the University of Michigan in 2010, and is currently working toward his PhD in computer science and engineering. His research interests include energy-efficient cache and core architectures focusing on hybrid designs.

Ronald G. Dreslinski received a BSE in electrical engineering and a BSE in computer engineering, a MSE and PhD in computer science and engineering from the University of Michigan, Ann Arbor. He is currently a research scientist at the University of Michigan and is a member of the IEEE. His research focuses on architectures that enable emerging low-power circuit techniques.

Thomas F. Wenisch received a PhD in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA in 2007. He is an associate professor of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, as well as a member of IEEE. His research interests include computer architecture, server and data center energy efficiency, smartphone architecture, and multiprocessor systems.

Scott Mahlke received the PhD degree in electrical engineering from the University of Illinois at Urbana-Champaign in 1997. Currently, he is a professor in the Electrical Engineering and Computer Science Department at the University of Michigan. He leads the Compilers Creating Custom Processors Research (CCCP) Group, focusing on the areas of compilers for multicore processors, heterogenous processors, and reliable system design. He was awarded the Young Alumni Achievement Award from the University of Illinois in 2006 and the Most Influential Paper Award from the International Symposium on Computer Architecture in 2007. He is a fellow of the IEEE.