Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE)

Kenzo Van Craeynest[•*] Aamer Jaleel[†] Lieven Eeckhout[•] Paolo Narvaez[†] Joel Emer[†‡]

Ghent University[•]
Ghent, Belgium
{kenzo.vancraeynest,
lieven.eeckhout}@ugent.be

Intel Corporation, VSSAD[†]
Hudson, MA
{aamer.jaleel,paolo.narvaez,
joel.emer}@intel.com

MIT[‡]
Cambridge, MA

Abstract

Single-ISA heterogeneous multi-core processors are typically composed of small (e.g., in-order) power-efficient cores and big (e.g., out-of-order) high-performance cores. The effectiveness of heterogeneous multi-cores depends on how well a scheduler can map workloads onto the most appropriate core type. In general, small cores can achieve good performance if the workload inherently has high levels of ILP. On the other hand, big cores provide good performance if the workload exhibits high levels of MLP or requires the ILP to be extracted dynamically.

This paper proposes Performance Impact Estimation (PIE) as a mechanism to predict which workload-to-core mapping is likely to provide the best performance. PIE collects CPI stack, MLP and ILP profile information, and estimates performance if the workload were to run on a different core type. Dynamic PIE adjusts the scheduling at runtime and thereby exploits fine-grained time-varying execution behavior. We show that PIE requires limited hardware support and can improve system performance by an average of 5.5% over recent state-of-the-art scheduling proposals and by 8.7% over a sampling-based scheduling policy.

1 Introduction

Heterogeneous multi-cores can enable higher performance and reduced energy consumption (within a given power budget) by executing workloads on the most appropriate core type. Recent work illustrates the potential of heterogeneous multi-cores to dramatically improve energy-efficiency and power-efficiency [2, 3, 9, 17, 18, 19, 20, 21, 30]. Commercial offerings include CPU and GPU integration, e.g., Intel's Sandy Bridge [12], AMD's Fusion [1], and NVidia's Tegra [25]; or CPU plus accelerators, e.g., IBM's Cell [16]. Other commercial products integrate different CPU core types on a single chip, e.g., NVidia's Kal-El [26] which integrates four performance-tuned cores along with one energy-tuned core, and ARM's big.LITTLE chip [10], which integrates a high-performance big core with a low-energy small core on a single chip. The latter two examples are so-called single-ISA heterogeneous multi-cores, which means that the different core types implement the same instruction-set architecture (ISA); single-ISA heterogeneous multi-cores are the main focus of this paper.

A fundamental problem in the design space of single-ISA heterogeneous multi-core processors is how best to schedule workloads on the most appropriate core type. Making wrong scheduling decisions can lead to suboptimal performance and excess energy/power consumption. To address this scheduling problem, recent proposals use workload memory intensity as an indicator to guide application scheduling [2, 3, 9, 17, 22, 30]. Such proposals tend to schedule memory-intensive workloads on a small core and compute-intensive workloads on a big core. We show that such an approach causes suboptimal scheduling when memory intensity alone is not a good indicator for workload-to-core mapping.

In general, small (e.g., in-order) cores provide good performance for compute-intensive workloads whose subsequent instructions in the dynamic instruction stream are mostly independent (i.e., high levels of inherent ILP). On the other hand, big (e.g., out-of-order) cores provide good performance for workloads where the ILP must be extracted dynamically or the workload exhibits a large amount of MLP. Therefore, scheduling decisions on heterogeneous multi-cores can be significantly improved by taking into account how well a small or big core can exploit the ILP and MLP characteristics of a workload.

This paper proposes Performance Impact Estimation (PIE) as a mechanism to select the appropriate workload-to-core mapping in a heterogeneous multi-core processor.

[*]A large part of this work was performed while Kenzo Van Craeynest was an intern at Intel/VSSAD.

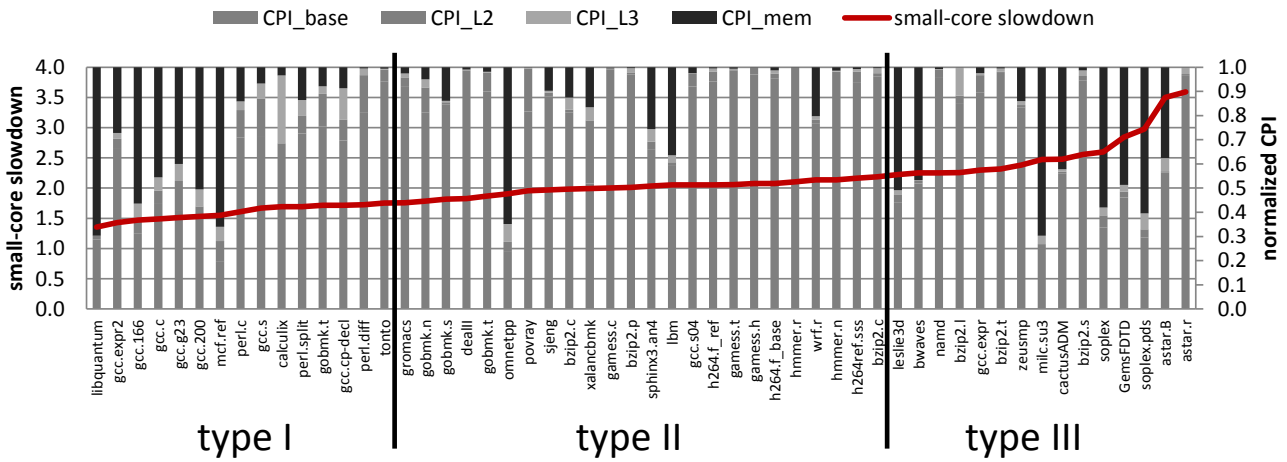

Figure 1. Normalized big-core CPI stacks (right axis) and small-core slowdown (left axis). Benchmarks are sorted by their small-versus-big core slowdown.

The key idea of PIE is to estimate the expected performance for each core type for a given workload. In particular, PIE collects CPI stack, MLP and ILP profile information during runtime on any one core type, and estimates performance if the workload were to run on another core type. In essence, PIE estimates how a core type affects exploitable MLP and ILP, and uses the CPI stacks to estimate the impact on overall performance. Dynamic PIE scheduling collects profile information on a per-interval basis (e.g., 2.5 ms) and dynamically adjusts the workload-to-core mapping, thereby exploiting time-varying execution behavior. We show that dynamically collecting profile information requires minimal hardware support: five 10-bit counters and 64 bits of storage.

We evaluate PIE scheduling using a large number of multi-programmed SPEC CPU2006 workload mixes. For a set of scheduling-sensitive workload mixes on a heterogeneous multi-core consisting of one big (out-of-order) and one small (in-order) core, we report an average performance improvement of 5.5% over recent state-of-the-art scheduling proposals. We also evaluate PIE scheduling and demonstrate its scalability across a range of heterogeneous multi-core configurations, including private and shared last-level caches (LLCs). Finally, we show that PIE outperforms a sampling-based scheduling by an average of 8.7%.

2 Motivation

Efficient use of single-ISA heterogeneous multi-cores is dependent on the underlying workload scheduling policy. A number of recent proposals use memory intensity as an indicator to guide workload scheduling [2, 3, 9, 17, 22, 30]. This policy is based on the intuition that compute-intensive workloads benefit more from the high computational capabilities of a big core while memory-intensive workloads execute more energy-efficiently on a small core while waiting for memory.

To correlate whether memory intensity is a good indicator to guide workload scheduling, Figure 1 compares the slowdown for SPEC CPU2006 workloads on a small core relative to a big core (left y-axis), to the normalized CPI stack [5] on a big core (right y-axis). The normalized CPI stack indicates whether a workload is memory-intensive or compute-intensive. If the normalized CPI stack is memory dominant, then the workload is memory-intensive (e.g., mcf), else the workload is compute-intensive (e.g., tonto).

The figure illustrates workloads grouped into three categories on the x-axis: workloads that have reasonable slowdown ($<1.75\times$) on the small core (type-I workloads), workloads that have significant slowdown ($>2.25\times$) on the small core (type-III), and the remaining workloads are labeled as type-II. Making correct scheduling decisions in the presence of type-I and III workloads is most critical: making an incorrect scheduling decision, i.e., executing a type-III workload on a small core instead of a type-I workload, leads to poor overall performance, hence we label type-I and III as *scheduling-sensitive* workloads.

The figure shows that while memory intensity alone can provide a good indicator for scheduling some memory-intensive workloads (e.g., mcf) onto a small core, such practice can significantly slowdown other memory-intensive workloads (e.g., soplex). Similarly, some compute-intensive workloads (e.g., astar.r) observe a significant slowdown on a small core while other compute-intensive workloads (e.g., calculix) have reasonable slowdown when executing on a small core. This behavior illustrates that memory intensity (or compute intensity) alone is not a good indicator to guide application scheduling on heterogeneous multi-cores.

The performance behavior of workloads on small and

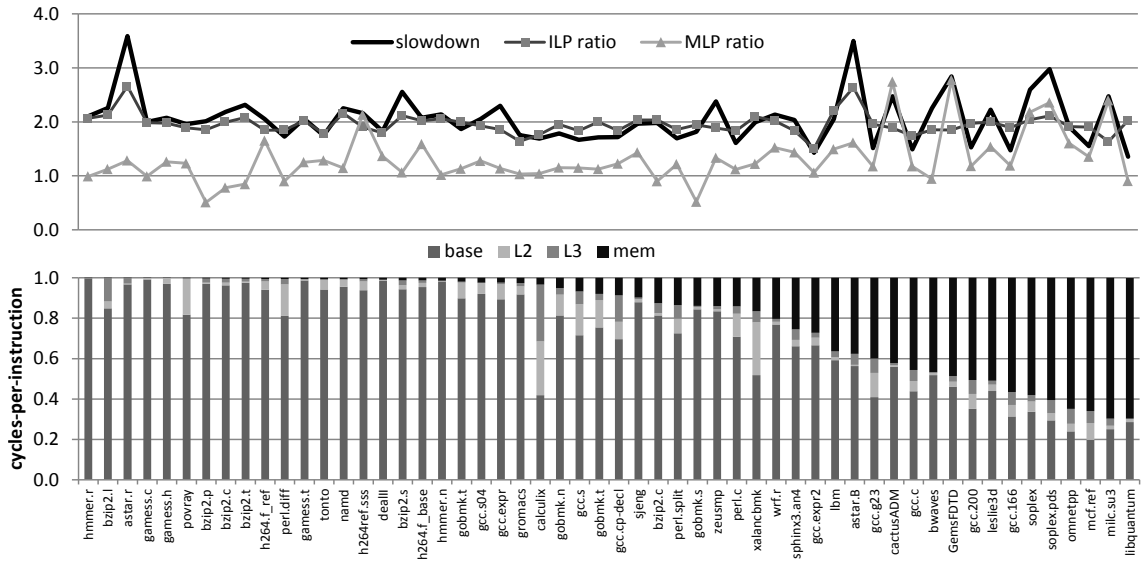

Figure 2. Correlating small-core slowdown to the MLP ratio for memory-intensive workloads (right-hand side in the graph) and to the ILP ratio for the compute-intensive workloads (lefthand side in the graph). Workloads are sorted by their normalized memory CPI component (bottom graph).

big cores (Figure 1) can be explained by the design characteristics of each core. Big cores are particularly suitable for workloads that require ILP to be extracted dynamically or have a large amount of MLP. On the other hand, small cores are suitable for workloads that have a large amount of inherent ILP. This implies that performance on different core types can be directly correlated to the amount of MLP and ILP prevalent in the workload. For example, consider a memory-intensive workload that has a large amount of MLP. Executing such a memory-intensive workload on a small core can result in significant slowdown if the small core does not expose the MLP. On the other hand, a compute-intensive workload with large amounts of ILP may have a reasonable slowdown on a small core and need not require the big core.

To quantify this, Figure 2 illustrates slowdown and the loss in MLP (or ILP) when scheduling a workload on a small core instead of a big core. The workloads are sorted left-to-right based on memory intensity (inferred from the normalized CPI stack). We use *MLP ratio* to quantify MLP loss and *ILP ratio* to quantify ILP loss. MLP and ILP ratios are defined as follows:

$$MLP_{ratio} = MLP_{big}/MLP_{small} \quad (1)$$

$$ILP_{ratio} = CPI_{base\_big}/CPI_{base\_small} \quad (2)$$

with MLP defined as the average number of outstanding memory requests if at least one is outstanding [4], and CPI_{base} as the base (non-miss) component of the CPI stack. The key observation from Figure 2 is that MLP ratio correlates with slowdown for memory-intensive applications (righthand side of the graph). Similarly, ILP ratio correlates with slowdown for compute-intensive workloads (lefthand side of the graph).

In summary, Figures 1 and 2 indicate that memory intensity alone is not a good indicator for scheduling workloads on a heterogeneous multi-core. Instead, scheduling policies on heterogeneous multi-cores must take into account the amount of MLP and ILP that can be exploited by the different core types. Furthermore, the slowdowns (or speedups) when moving between different core types can directly be correlated to the amount of MLP and ILP realized on a target core. This suggests that the performance on a target core type can be estimated by predicting the MLP and ILP on that core.

3 Performance Impact Estimation (PIE)

A direct approach to determine the best scheduling policy on a heterogeneous multi-core is to apply sampling-based scheduling [2, 18, 19, 33]. Sampling-based scheduling dynamically samples different workload-to-core mappings at runtime and then selects the best performing mapping. While such an approach can perform well, it introduces performance overhead due to periodically migrating workloads between different core types. Furthermore, these overheads increase with the number of cores (and core types). To address these drawbacks, we propose *Performance Impact Estimation (PIE)*.

The key idea behind PIE is to estimate (not sample) workload performance on a different core type. PIE accom-

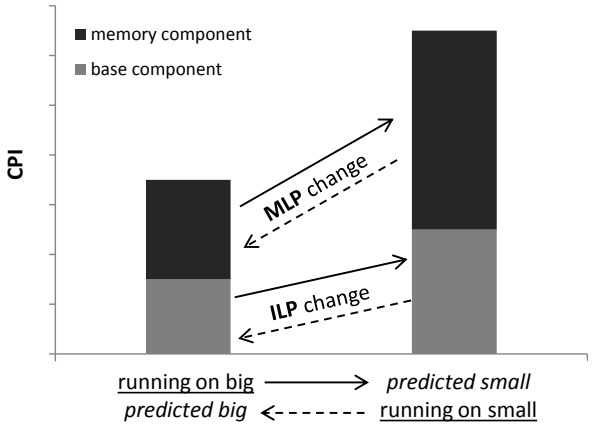

Figure 3. Illustration of the PIE model.

plishes this by using CPI stacks. We concentrate on two major components in the CPI stack: the base component and the memory component; the former lumps together all non-memory related components:

$$CPI = CPI_{base} + CPI_{mem}. \quad (3)$$

Figure 2 illustrated that MLP and ILP ratios provide good indicators on the performance difference between big and small cores. Therefore, we use MLP, ILP, and CPI stack information to develop our PIE model (see Figure 3). Specifically, we estimate the performance on a small core while executing on a big core in the following manner:

$$\begin{aligned} CPI_{small} &= \widetilde{CPI}_{base\_small} + \widetilde{CPI}_{mem\_small} \\ &= \widetilde{CPI}_{base\_small} + CPI_{mem\_big} \times MLP_{ratio}. \end{aligned} \quad (4)$$

Similarly, we estimate the performance on a big core while executing on a small core as follows:

$$\begin{aligned} CPI_{big} &= \widetilde{CPI}_{base\_big} + \widetilde{CPI}_{mem\_big} \\ &= \widetilde{CPI}_{base\_big} + CPI_{mem\_small}/MLP_{ratio}. \end{aligned} \quad (5)$$

In the above formulas, $\widetilde{CPI}_{base\_big}$ refers to the base CPI component on the big core estimated from the execution on the small core; $\widetilde{CPI}_{base\_small}$ is defined similarly. The memory CPI component on the big (small) core is computed by dividing (multiplying) the memory CPI component measured on the small (big) core with the MLP ratio. The remainder of this section details on how we predict the base CPI components as well as the MLP ratio, followed by an evaluation of the PIE model. Section 4 then presents dynamic PIE scheduling, including how we collect the inputs to the PIE model during runtime by introducing performance counters.

3.1 Predicting MLP

The memory CPI component essentially consists of three contributors: the number of misses, the latency per (isolated) miss, and the number of simultaneously outstanding misses (MLP). In this paper, we assume that the big and small cores have the same cache hierarchy, i.e., the same number of cache levels and the same cache sizes at each level. In other words, we assume that the number of misses and the latency per miss is constant across core types[1]. However, MLP varies across core types as big cores and small cores vary in the amount of MLP that they can exploit. We now describe how we estimate MLP on the big core while running on the small core; and vice versa, we estimate MLP on the small core while running on the big core. Combining these MLP estimates with measured MLP numbers on the current core type enables predicting the MLP ratio using Formula 1, which in its turn enables estimating the memory CPI components on the other core type, using Formulas 4 and 5.

3.1.1 Predicting big-core MLP on small core

Big out-of-order cores implement a reorder buffer, non-blocking caches, MSHRs, etc., which enables issuing independent memory accesses in parallel. The maximum MLP that a big core can exploit is bound by the reorder buffer size, i.e., a necessary condition for independent long-latency load misses to be issued to memory simultaneously is that they reside in the reorder buffer at the same time. We therefore estimate the big-core MLP as the average number of memory accesses in the big-core reorder buffer. Quantitatively, we do so by calculating the average number of LLC misses per instruction observed on the small core (MPI_{small}) multiplied by the big-core reorder buffer size:

$$MLP_{big} = MPI_{small} \times ROB\_size. \quad (6)$$

Note that the above estimate does not make a distinction between independent versus dependent LLC misses; we count all LLC misses. A more accurate estimate would be to count independent LLC misses only, however, in order to simplify the design, we simply count all LLC misses.

3.1.2 Predicting small-core MLP on big core

Small in-order cores exploit less MLP than big cores. A stall-on-miss core stalls on a cache miss, and hence, it does not exploit MLP at all — MLP equals one. A stall-on-use core can exploit some level of MLP: independent loads between a long-latency load and its first consumer can be issued to memory simultaneously. MLP for a stall-on-use core thus equals the average number of memory accesses between a long-latency load and its consumer. Hence, we

[1]If the cache hierarchy is different, then techniques described in [14] can be used to estimate misses for a different cache size.

estimate the MLP of a stall-on-use core as the average number of LLC misses per instruction on the big core multiplied by the average dependency distance D between an LLC miss and its consumer. (Dependency distance is defined as the number of dynamically executed instructions between a producer and its consumer.)

$$MLP_{small} = MPI_{big} \times D. \tag{7}$$

Again, in order to simplify the design, we approximate D as the dependency distance between any producer (not just an LLC miss) and its consumer. We describe how we measure the dependency distance D in Section 4.3.

3.2 Predicting ILP

The second CPI component predicted by the PIE model is the base CPI component.

3.2.1 Predicting big-core ILP on small core

We estimate the base CPI component for the big core as one over the issue width W_{big} of the big core:

$$\widetilde{CPI}_{base\_big} = 1/W_{big}. \tag{8}$$

A balanced big (out-of-order) core should be able to dispatch approximately W_{big} instructions per cycle in the absence of miss events. A balanced core design can be achieved by making the reorder buffer and related structures such as issue queues, rename register file, etc., sufficiently large to enable the core to issue instructions at a rate near the designed width [7].

3.2.2 Predicting small-core ILP on big core

Estimating the base CPI component for a small (in-order) core while running on a big core is more complicated. For ease of reasoning, we estimate the average IPC and take the reciprocal of the estimated IPC to yield the estimated CPI. We estimate the average base IPC on the small core with width W_{small} as follows:

$$\widetilde{IPC}_{base\_small} = \sum_{i=1}^{W_{small}} i \times P[IPC = i]. \tag{9}$$

We use simple probability theory to estimate the probability of executing i instructions in a given cycle. The probability of executing only one instruction in a given cycle equals the probability that an instruction produces a value that is consumed by the next instruction in the dynamic instruction stream (dependency distance of one):

$$P[IPC = 1] = P[D = 1]. \tag{10}$$

Likewise, the probability of executing two instructions in a a given cycle equals the probability that the second instruction does not depend on the first, and the third depends on either the first or the second:

$$\begin{aligned} P[IPC = 2] = &(1 - P[D = 1]) \times \\ &(P[D = 1] + P[D = 2]). \end{aligned} \tag{11}$$

This generalizes to three instructions per cycle as well:

$$\begin{aligned} P[IPC = 3] = &(1 - P[D = 1]) \times \\ &(1 - P[D = 1] - P[D = 2]) \times \\ &(P[D = 1] + P[D = 2] + P[D = 3]). \end{aligned} \tag{12}$$

Finally, assuming a 4-wide in-order core, the probability of executing four instructions per cycle equals the probability that none of the instructions depend on a previous instruction in a group of four instructions:

$$\begin{aligned} P[IPC = 4] = &(1 - P[D = 1]) \times \\ &(1 - P[D = 1] - P[D = 2]) \times \\ &(1 - P[D = 1] - P[D = 2] - P[D = 3]). \end{aligned} \tag{13}$$

Note that the above formulas do not take non-unit instruction execution latencies into account. Again, we used this approximation to simplify the design, and we found this approximation to be accurate enough for our purpose.

3.3 Evaluating the PIE Model

Figure 4 evaluates the accuracy of our PIE model. This is done in two ways: we estimate big-core performance while executing the workload on a small core, and vice versa, we estimate small-core performance while executing the workload on a big core. We compare both of these to the actual slowdown. (We will describe the experimental setup in Section 5.) The figure shows that we achieve an average absolute prediction error of 9% and a maximum error of 35% when predicting speedup (predicting big-core performance on the small core). The average absolute prediction error for the slowdown (predicting small-core performance on the big core) equals 13% with a maximum error of 47%. More importantly, PIE accurately predicts the relative performance differences between the big and small cores. This is in line with our goal of using PIE for driving runtime scheduling decisions.

As a second step in evaluating our PIE model, we consider a heterogeneous multi-core and use PIE to determine the workload-to-core mapping. We consider all possible two-core multi-programmed workload mixes of SPEC

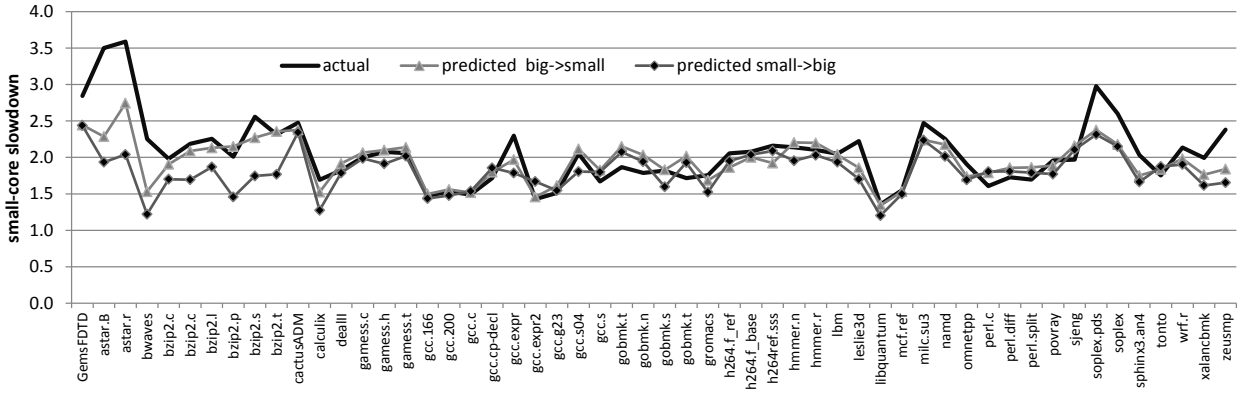

Figure 4. Evaluating the accuracy of the PIE model.

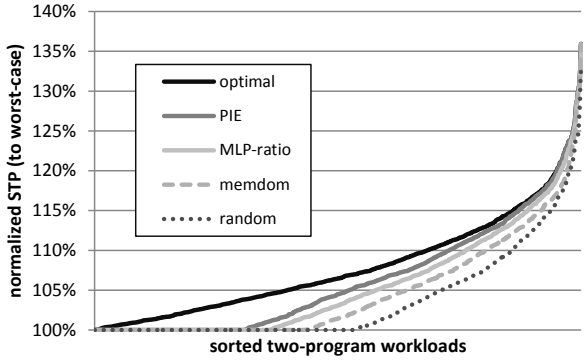

Figure 5. Comparing scheduling policies on a two-core heterogeneous multi-core.

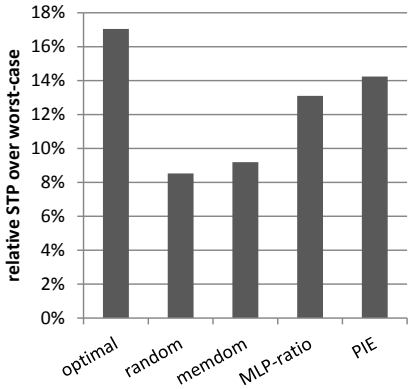

Figure 6. Comparing different scheduling algorithms for type-I and type-III workload mixes assuming a static setup.

CPU2006 applications and a two-core system with one big core and one small core and private LLCs. Further, benchmarks are scheduled on a given core and stay there for the remainder of the execution (static scheduling).

Figure 5 reports performance (system throughput or weighted speedup) relative to worst-case scheduling for all workload mixes; we compare PIE scheduling against random and memory-dominance (memdom) scheduling. Memory-dominance scheduling refers to the conventional practice of always scheduling memory-intensive workloads on the small core.

PIE scheduling chooses the workload-to-core mapping by selecting the schedule that yields the highest (estimated) system throughput across both cores. PIE scheduling outperforms both random and memory-dominance scheduling over the entire range of workload mixes. Figure 6 provides more detailed results for workload mixes with type-I and type-III workloads. PIE outperforms worst-case scheduling by 14.2%, compared to random (8.5%) and memory-dominance scheduling (9.2%). Put differently, PIE scheduling achieves 84% of optimal scheduling, compared to 54% for memory-dominance and 50% for random scheduling.

The PIE model takes into account both ILP and MLP. We also evaluated a version of PIE that only takes MLP into account, i.e., ILP is not accounted for and is assumed to be the same on the big and small cores. We refer to this as MLP-ratio scheduling. Figures 5 and 6 illustrate the importance of taking both MLP and ILP into account. MLP-ratio scheduling improves worst-case scheduling by 12.7% for type-I and III workloads, compared to 14.2% for PIE. This illustrates that accounting for MLP is more important than ILP in PIE.

So far, we evaluated PIE for a heterogeneous multi-core with one big and one small core (e.g., ARM's big.LITTLE design [10]). We now evaluate PIE scheduling for heterogeneous multi-cores with one big core and multiple small cores, as well as several big cores and one small core (e.g., NVidia's Kal-El [26]); we assume all cores are active all the time. Figure 7 shows that PIE outperforms memory-dominance scheduling by a bigger margin even for these heterogeneous multi-core design points than for the one-big, one-small multi-core system.

4 Dynamic Scheduling

So far, PIE scheduling was evaluated in a static setting, i.e., a workload is scheduled on a given core for its entire execu-

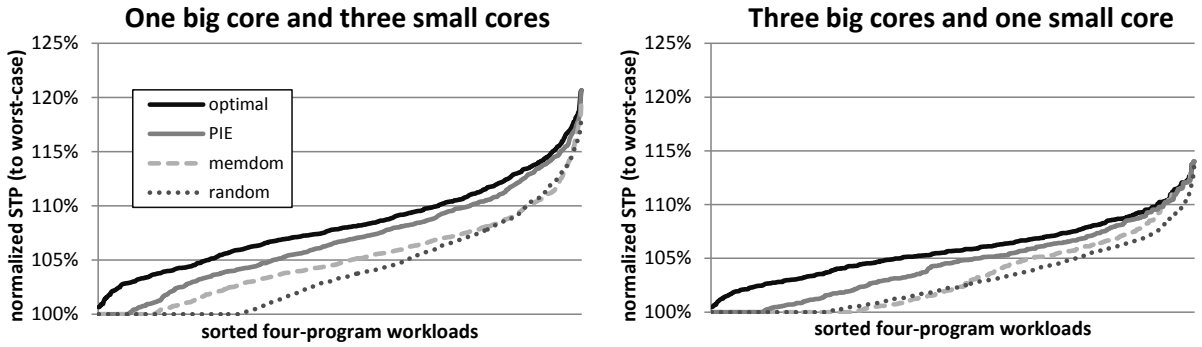

Figure 7. Evaluating PIE for heterogeneous multi-core with one big and three small cores (left graph), and three big cores and one small core (right graph).

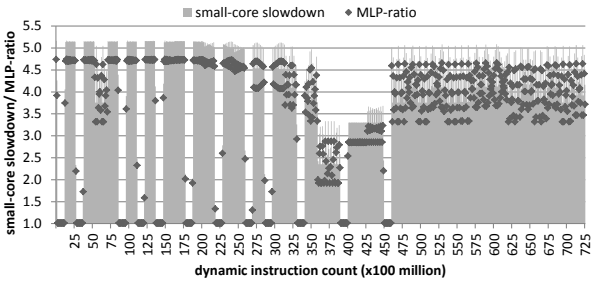

Figure 8. Dynamic execution profile of libquantum.

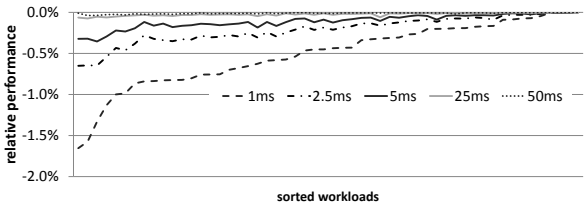

Figure 9. Migration overhead for a shared LLC.

tion. There is opportunity to further improve PIE scheduling by dynamically adapting to workload phase behavior. To illustrate this, Figure 8 shows big-core and small-core CPI and MLP as a function of time for libquantum from SPEC CPU2006. The key observation here is that, although the average slowdown is high for the small core compared to the big core, the small core achieves comparable performance to the big core for some execution phases. For libquantum, approximately 10% of the instructions can be executed on the small core without significantly affecting overall performance. However, the time-scale granularity is relatively fine-grained (few milliseconds) and much smaller than a typical OS time slice (e.g., 10 ms). This suggests that dynamic hardware scheduling might be beneficial provided that rescheduling (i.e., migration) overhead is low.

4.1 Quantifying migration overhead

Dynamic scheduling incurs overhead for migrating workloads between different cores. Not only does migration incur a context switch, it also incurs overhead for warming hardware state, especially the cache hierarchy. A context switch incurs a fixed cost for restoring architecture state. To better understand the overhead due to cache warming, we consider a number of scenarios to gain insight on cache hierarchy designs for low migration overheads at fine-grained dynamic scheduling.

Shared LLC. Figure 9 quantifies the performance overhead of migrating a workload every x milliseconds, with x varying from 1 ms to 50 ms. Migration overhead is measured by configuring two identical cores to share a 4MB LLC. Workloads are rescheduled to a different core every x ms. Interestingly, for a 2.5 ms migration frequency, the performance overhead due to migration is small, less than 0.6% for all benchmarks. The (small) performance overhead are due to (private) L1 and L2 cache warmup effects.

Private powered-off LLCs. The situation is very different in case of a private LLC that is powered off when migrating a workload. Powering off a private LLC makes sense in case one wants to power down an entire core and its private cache hierarchy in order to conserve power. If the migration frequency is high (e.g., 2.5 ms), Figure 10 reports severe performance overhead for some workloads when the private cache hierarchy is powered off upon migration. The huge performance overheads are because the cache looses its data when powered off, and hence the new core must re-fetch the data from main memory.

Private powered-on LLCs. Instead of turning off private LLCs, an alternative is to keep the private LLCs powered on and retain the data in the cache. In doing so, Figure 11 shows that performance overhead from frequent migrations is much smaller and in fact even leads to substantial performance benefits for a significant fraction of the benchmarks. The performance benefit comes from having a larger effective LLC: upon a miss in the new core's private LLC, the

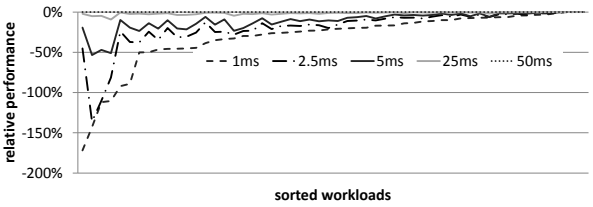

Figure 10. Migration overhead for private powered-off LLCs.

Figure 11. Migration overhead for private powered-on LLCs.

data is likely to be found in the old core's private LLC, and hence the data can be obtained more quickly from the old core's LLC through cache coherency than by fetching the data from main memory.

4.2 Dynamic PIE Scheduling

Having described the PIE model, we now describe Dynamic PIE scheduling. PIE scheduling is applicable to any number of cores of any core type. However, to simplify the discussion, we assume one core of each type. We assume as many workloads as there are cores, and that workloads are initially randomly scheduled onto each core. Furthermore, we assume that workload scheduling decisions can be made every x milliseconds.

To strive towards an optimal schedule, PIE scheduling requires hardware support for collecting CPI stacks on each core, the number of misses, the number of dynamically executed instructions, and finally the inter-instruction dependency distance distribution on the big core. We discuss the necessary hardware support in the next section.

During every time interval of x milliseconds, for each workload in the system, PIE uses the hardware support to compute CPI stacks, MLP and ILP on the current core type, and also predicts the MLP and ILP for the same workload on the other core type. These predictions are then fed into the PIE model to estimate the performance of each workload on the other core type. For a given performance metric, PIE scheduling uses these estimates to determine whether another scheduling decision would potentially improve overall system performance as compared to the current schedule. If so, workloads are rescheduled to the predicted core type. If not, the workload schedule remains intact and the process is repeated the next time interval.

Note that PIE scheduling can be done both in hardware and software. If the time interval of scheduling workloads to cores coincides with a time slice, then PIE scheduling can be applied in software, i.e., the hardware would collect the event counts and the software (e.g., OS or hypervisor) would make scheduling decisions. If scheduling decisions would need to be made at smaller time scale granularities, hardware can also make the scheduling decisions, transparent to the software [10].

4.3 Hardware support

PIE scheduling requires hardware support for collecting CPI stacks. Collecting CPI stacks on in-order cores is fairly straightforward and is implemented in commercial systems, see for example Intel Atom [11]. Collecting CPI stacks on out-of-order cores is more complicated because of various overlap effects between miss events, e.g., a long-latency load may hide the latency of another independent long-latency load miss or mispredicted branch, etc. Recent commercial processors such as IBM Power5 [23] and Intel Sandy Bridge [12] however provide support for computing memory stall components. PIE scheduling also requires the number of LLC misses and the number of dynamically executed instructions, which can be measured using existing hardware performance counters. In other words, most of the profile information needed by PIE can be readily measured on existing hardware.

PIE scheduling requires some profile information that cannot be collected on existing hardware. For example, while running on a big core, PIE requires the ability to measure the inter-instruction dependency distance distribution for estimating small-core MLP and ILP. The PIE model requires the dependency distance distribution for a maximum dependency distance of W_{small} only (where W_{small} is the width of the small core). For a 4-wide core, this involves four plus one counters: four counters for computing the dependency distance distribution up to four instructions, and one counter for computing the average distance.

The PIE model requires that the average dependency distance D be computed over the dynamic instruction stream. This can be done by requiring a table with as many rows as there are architectural registers. The table keeps track of which instruction last wrote to an architectural register. The delta in dynamic instruction count between a register write and subsequent read then is the dependency distance. Note that the table counters do not need to be wide, because the dependency distance tends to be short [8]; e.g., four bits per counter can capture 90% of the distances correctly. In summary, the total hardware cost to track the dependency distance distribution is roughly 15 bytes of storage: 4 bits times the number of architectural registers (64 bits for x86-64), plus five 10-bit counters.

5 Experimental Setup

We use CMP$im [13] to conduct the simulation experiments in this paper. We configure our simulator to model heterogeneous multi-core processors with big and small cores. The big core is a 4-wide out-of-order processor core; the small core is a 4-wide (stall-on-use) in-order processor core[2]. We assume both cores run at a 2 GHz clock frequency. Further, we assume a cache hierarchy consisting of three levels of cache, separate 32 KB L1 instruction and data caches, a 256 KB L2 cache and a 4 MB last-level L3 cache (LLC). We assume the L1 and L2 caches to be private per core for all the configurations evaluated in this paper. We evaluate both shared and private LLC configurations. We consider the LRU replacement policy in all of the caches unless mentioned otherwise; we also consider a state-of-the-art RRIP shared cache replacement policy [15]. Finally, we assume an aggressive stream-based hardware prefetcher; we experimentally evaluated that hardware prefetching improves performance by 47% and 25% on average for the small and big cores, respectively.

We further assume that the time interval for dynamic scheduling is 2.5 ms; this is small enough to benefit from fine-grained exploitation of time-varying execution behavior while keeping migration overhead small. The overhead for migrating a workload from one core to another (storing and restoring the architecture state) is set to 300 cycles; in addition, we do account for the migration overhead due to cache effects.

We consider all 26 SPEC CPU2006 programs and all of their reference inputs, leading to 54 benchmarks in total. We select representative simulation points of 500 million instructions each using PinPoints [27]. When simulating a multi-program workload we stop the simulation when the slowest workload has executed 500 million instructions. Faster running workloads are reiterated from the beginning of the simulation point when they reach the end. We report system throughput (STP) [6] (also called weighted speedup [31]) which quantifies system-level performance or aggregate throughput achieved by the system.

6 Results and Analysis

We evaluate dynamic PIE scheduling on private and shared LLCs with LRU, and a shared LLC with RRIP replacement. We compare PIE scheduling to a sampling-based strategy [2, 18, 19, 33] that assumes running a workload for one time interval on one core and for the next time interval on the other core. The workload-core schedule that yields the highest performance is then maintained for the next 10 time intervals, after which the sampling phase is reinitiated.

[2]We also ran experiments with a 2-wide in-order processor and found the performance for the 2-wide in-order processor to be within 10% of the 4-wide in-order processor, which is a very small compared to the 200%+ performance difference between in-order versus out-of-order processor performance. Hence, we believe that our conclusions hold true irrespective of the width of the in-order processor.

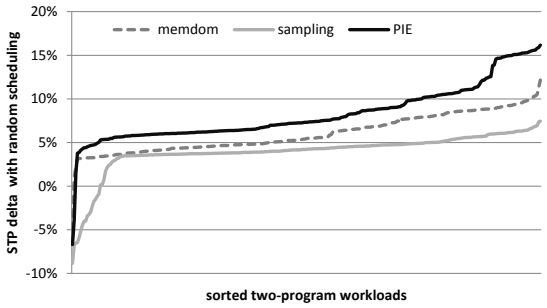

Figure 12. Relative performance (STP) delta over random scheduling for sampling-based, memory-dominance and PIE scheduling, assuming private LLCs.

6.1 Private LLCs

We first assume that each core has its private LLC. Figure 12 quantifies the relative performance over random scheduling for sampling-based, memory-dominance and PIE scheduling. PIE scheduling clearly outperforms the other scheduling strategies by a significant margin. Across the type-I and III workload mixes, we report an average 5.5% and 8.7% improvement in performance over memory-dominance and sampling-based scheduling, respectively. The improvement over memory-dominance scheduling comes from two sources: PIE is able to more accurately determine the better workload-to-core mapping, and in addition, PIE can exploit fine-grain phase behavior, unlike memory-dominance scheduling. PIE also improves upon sampling-based scheduling, because PIE does not incur any overhead from sampling because it (accurately) estimates the performance impact of a workload reschedule, and hence, it can more quickly and better adapt to fine-grain phase behavior.

6.2 Shared LLC

With shared LLCs, Figure 13 shows similar conclusions to private LLCs: PIE outperforms random, sampling-based and memory-dominance scheduling. For the type-I and III workload mixes, we obtain an average 3.7% and 6.4% improvement in performance over memory-dominance and sampling-based scheduling, respectively. The performance improvement is slightly lower for private LLCs though. The reason is that none of the scheduling strategies anticipate conflict behavior in the shared LLC, and, as a result, some of the scheduling decisions may be partly offset by negative conflict behavior in the shared LLC. Further, in the case of sampling-based scheduling, LLC performance changes when switching between core types (as a result of sampling) because the access patterns change, which in turn changes

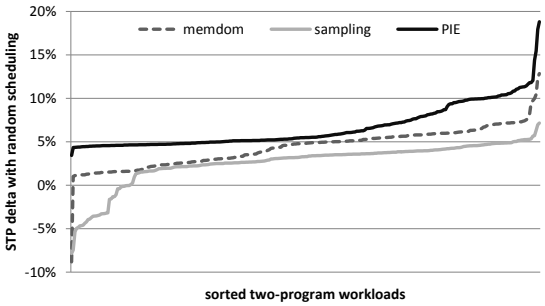

Figure 13. Relative performance (STP) delta over random scheduling for sampling-based, memory-dominance and PIE scheduling, assuming an LRU-managed shared LLC.

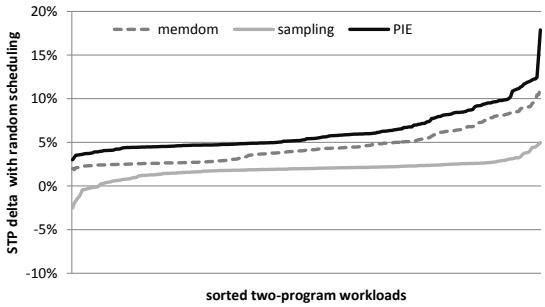

Figure 14. Relative performance (STP) delta over random scheduling for sampling-based, memory-dominance and PIE scheduling, assuming an RRIP-managed shared LLC.

overall performance; in other words, sampling is particularly ineffective in case of a shared LLC.

6.3 RRIP-managed shared LLC

So far, we assumed an LRU cache replacement policy. However, it has been shown that LRU is not the most effective shared cache management policy; a state-of-the-art shared cache replacement policy is RRIP [15] which significantly improves LLC performance by predicting the re-reference behavior of cache blocks. The results for PIE scheduling applied to an RRIP-managed LLC are shown in Figure 14. For the type-I and III workload mixes, PIE scheduling improves performance by 2.4% and 7.8% over memory-dominance and sampling-based scheduling, respectively.

An interesting observation to make from Figure 14 is that an intelligent shared cache management policy such as RRIP is able to reduce the performance hit observed for some of the workloads due to scheduling. A large fraction of the workloads observe a significant performance hit under sampling-based scheduling (and a handful workloads under memory-dominance scheduling) for an LRU-managed shared LLC, see bottom left in Figure 13; these performance hits are removed through RRIP, see Figure 14. In other words, a scheduling policy can benefit from an intelligent cache replacement policy: incorrect decisions by the scheduling policy can be alleviated (to some extent) by the cache management policy.

7 Related Work

Heterogeneous multi-cores desings vary from single-ISA cores only varying in clock frequency, to single-ISA cores differing in microarchitecture, to cores with non-identical ISAs. Since we focus on single-ISA heterogeneous multi-cores, we only discuss this class of heterogeneity.

Kumar et al. [18] made the case for heterogeneous single-ISA multi-core processors when running a single application: they demonstrate that scheduling an application across core types based on its time-varying execution behavior can yield substantial energy savings. They evaluate both static and dynamic scheduling policies. In their follow-on work, Kumar et al. [19] study scheduling on heterogeneous multi-cores while running multi-program workloads. The dynamic scheduling policies explored in these studies use sampling to gauge the most energy-efficient core. Becchi and Crowley [2] also explore sample-based scheduling. Unfortunately, sample-based scheduling, in contrast to PIE, does not scale well with increasing core count: an infrequent core type (e.g., a big core in a one-big, multiple-small core configuration) quickly becomes a bottleneck.

Bias scheduling [17] is very similar to memory-dominance scheduling. It schedules programs that exhibit frequent memory and other resource stalls on the small core, and programs that are dominated by execution cycles (and hence low fraction of stalls) on the big core. Thresholds are used to determine a program's bias towards a big versus small core based on these stall counts.

HASS [30] is a static scheduling policy, the key motivation being scalability. Chen and John [3] leverage offline program profiling. An obvious limitation of static/offline scheduling is that it does not enable exploiting time-varying execution behavior. PIE on the other hand is a dynamic scheduling algorithm that, in addition, is scalable.

Several studies [9, 30, 32] explore scheduling in heterogeneous systems by changing clock frequency across cores; the core microarchitecture does not change though. Such studies do not face the difficulty of having to deal with differences in MLP and ILP across core types. Hence, memory-dominance based scheduling is likely to work well for such architectures.

Age-based scheduling [20] predicts the remaining execution time of a thread in a multi-threaded program and schedules the oldest thread on the big core. Li et al. [21] evaluate the idea of scheduling programs on the big core first, before scheduling programs on the small cores, in order to make sure the big power-hungry core is fully utilized.

Chou et al. [4] explored how microarchitecture techniques affect MLP. They found that out-of-order processors can better exploit MLP compared to in-order processors. We show that MLP and ILP are important criteria to take into account when scheduling on heterogeneous multi-cores, and we propose the PIE method for doing so.

Patsilaras et al. [28, 29] study how to best integrate an MLP technique (such as runahead execution [24]) into an asymmetric multi-core processor, i.e., should one integrate the MLP technique into the small or big core, or both? They found that if the small core runs at a higher frequency and implements an MLP technique, the small core might become more beneficial for exploiting MLP-intensive workloads. Further, they propose a hardware mechanism to dynamically schedule threads to core types based on the amount of MLP in the dynamic instruction stream, which they estimate by counting the number of LLC misses in the last 10K instructions interval. No currently shipping commercial processor employs runahead execution; also, running the small core at a high frequency might not be possible given current power concerns. We therefore take a different approach: we consider a heterogeneous multi-core system as a given — we do not propose changing the architecture nor the frequency of either core type — and we schedule tasks onto the most appropriate core type to improve overall performance while taking both MLP and ILP into account as a criterion for scheduling.

8 Conclusions

Single-ISA heterogeneous multi-cores are typically composed of small (e.g., in-order) cores and big (e.g., out-of-order) cores. Using different core types on a single die has the potential to improve energy-effiency without sacrificing significant performance. However, the success of heterogeneous multi-cores is directly dependent on how well a scheduling policy maps workloads to the best core type (big or small). Incorrect scheduling decisions can unnecessarily degrade performance and waste energy/power. With this in mind, this paper makes the following contributions:

- We show that using memory intensity alone as an indictator to guide workload scheduling decisions can lead to suboptimal performance. Instead, scheduling policies must take into account how a core type can exploit the ILP and MLP characteristics of a workload.
- We propose the Performance Impact Estimation (PIE) model to guide workload scheduling. The PIE model uses CPI stack, ILP and MLP information of a workload on a given core type to estimate the performance on a different core type. We propose PIE models for both small (in-order) and big (out-of-order) cores.
- Using the PIE model, we propose dynamic PIE scheduling. Dynamic PIE collects CPI stack, ILP and MLP information at run time to guide workload scheduling decisions.
- We show that the use of shared LLCs can enable high frequency, low-overhead, fine-grained scheduling to exploit time-varying execution behavior. We also show that the use of private LLCs can provide similar capability as long as the caches are not flushed on core migrations.

We evaluate PIE for a variety of systems with varying core counts and cache configurations. Across a large number of scheduling-sensitive workloads, we show that PIE scheduling is scalable to any core count and outperforms prior work by a significant margin.

In this paper, we focused on using PIE scheduling to improve the weighted speedup metric for a heterogeneous multi-core system. The evaluations were primarily done for multi-programmed workload mixes. However, PIE scheduling can also be applied to improve multi-threaded workload performance. Furthermore, when multiple workloads contend for the same core type, PIE scheduling can be extended to optimize for fairness. Exploring these extensions is part of our on-going work.

Acknowledgements

We thank David Albonesi, the VSSAD group, Socrates Demetriades, Krishna Rangan, and the anonymous reviewers for their constructive and insightful feedback. Kenzo Van Craeynest is supported through a doctoral fellowship by the Agency for Innovation by Science and Technology (IWT). Additional support is provided by the FWO projects G.0255.08 and G.0179.10, and the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295.

References

[1] AMD. The future is fusion: The industry-changing impact of accelerated computing. http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf, 2008.

[2] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. *Journal of Instruction-Level Parallelism (JILP)*, 10:1–26, June 2008.

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

[4] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of ISCA*, pages 76–87, June 2004.

[5] P. G. Emma. Understanding some simple processor-performance limits. *IBM Journal of Research and Development*, 41(3):215–232, May 1997.

[6] S. Eyerman and L. Eeckhout. System-level performance metrics for multi-program workloads. *IEEE Micro*, 28(3):42–53, May/June 2008.

[7] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2), May 2009.

[8] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of MICRO*, pages 236–245, Dec. 1992.

[9] S. Ghiasi, T. Keller, and F. Rawson. Scheduling for heterogeneous processors in server systems. In *Proceedings of the Second Conference on Computing Frontiers (CF)*, pages 199–210, May 2005.

[10] P. Greenhalgh. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7: Improving energy efficiency in high-performance mobile platforms. http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf, Sept. 2011.

[11] T. R. Halfhill. Intel's tiny Atom. *Microprocessor Report*, 22:1–13, Apr. 2008.

[12] Intel. 2nd generation Intel Core vPro processor family. http://www.intel.com/content/dam/doc/white-paper/core-vpro-2nd-generation-core-vpro-processor-family-paper.pdf, 2008.

[13] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A Pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), held in conjunction with ISCA*, June 2008.

[14] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely Jr., and J. Emer. CRUISE: Cache Replacement and Utility-aware Scheduling. In *Proceedings of ASPLOS*, pages 60–71, March 2011.

[15] A. Jaleel, K. Theobald, S. C. Steely Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of ISCA*, pages 60–71, June 2010.

[16] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49:589–604, July 2005.

[17] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 125–138, Apr. 2010.

[18] 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 *Proceedings of MICRO*, pages 81–92, Dec. 2003.

[19] 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 ISCA*, pages 64–75, June 2004.

[20] N. B. Lakshminarayana, J. Lee, and H. Kim. Age based scheduling for asymmetric multiprocessors. In *Proceedings of Supercomputing: the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, Nov. 2009.

[21] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of Supercomputing: the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, Nov. 2007.

[22] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *Proceedings of HPCA*, pages 1–12, Jan. 2010.

[23] A. Mericas. Performance monitoring on the POWER5 microprocessor. In L. K. John and L. Eeckhout, editors, *Performance Evaluation and Benchmarking*, pages 247–266. CRC Press, 2006.

[24] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of HPCA*, pages 129–140, Feb. 2003.

[25] NVidia. The benefits of multiple CPU cores in mobile devices. http://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf, 2010.

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

[27] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *Proceedings of MICRO*, pages 81–93, Dec. 2004.

[28] G. Patsilaras, N. K. Choudhary, and J. Tuck. Design tradeoffs for memory-level parallelism on a asymmetric multi-core system. In *Proceedings of the Third Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA), held in conjunction with ISCA*, June 2010.

[29] G. Patsilaras, N. K. Choudhary, and J. Tuck. Efficiently exploiting memory level parallelism on asymmetric coupled cores in the dark silicon era. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8, Jan. 2012.

[30] D. Shelepov, J. C. S. Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. HASS: A scheduler for heterogeneous multicore systems. *Operating Systems Review*, 43:66–75, Apr. 2009.

[31] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for simultaneous multithreading processor. In *Proceedings of ASPLOS*, pages 234–244, Nov. 2000.

[32] S. Srinivasan, L. Zhao, R. Illikal and R. Iyer. Efficient Interaction between OS and Architecture in Heterogeneous Platforms. In *ACM SIGOPS Operating Systems Review*, Vol 45, Issue 1, Jan. 2011.

[33] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *Proceedings of PACT*, pages 29–40, Nov. 2010.