Bias Scheduling in Heterogeneous Multi-core Architectures

David Koufaty Dheeraj Reddy Scott Hahn
Intel Labs
{david.a.koufaty, dheeraj.reddy, scott.hahn}@intel.com

Abstract

Heterogeneous architectures that integrate a mix of big and small cores are very attractive because they can achieve high single-threaded performance while enabling high performance thread-level parallelism with lower energy costs. Despite their benefits, they pose significant challenges to the operating system software. Thread scheduling is one of the most critical challenges.

In this paper we propose bias scheduling for heterogeneous systems with cores that have different microarchitectures and performance. We identify key metrics that characterize an application bias, namely the core type that best suits its resource needs. By dynamically monitoring application bias, the operating system is able to match threads to the core type that can maximize system throughput. Bias scheduling takes advantage of this by influencing the existing scheduler to select the core type that bests suits the application when performing load balancing operations.

Bias scheduling can be implemented on top of most existing schedulers since its impact is limited to changes in the load balancing code. In particular, we implemented it over the Linux scheduler on a real system that models microarchitectural differences accurately and found that it can improve system performance significantly, and in proportion to the application bias diversity present in the workload. Unlike previous work, bias scheduling does not require sampling of CPI on all core types or offline profiling. We also expose the limits of dynamic voltage/frequency scaling as an evaluation vehicle for heterogeneous systems.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management - Scheduling; C.1.3 [*Processor Architectures*]: Other Architecture Styles - Heterogeneous (hybrid) systems

EuroSys'10, April 13–16, 2010, Paris, France.

General Terms Algorithms, Performance

1. Introduction

Advances in semiconductor technology have enabled processor manufacturers to integrate more and more cores on a chip. Most multi-core processors consist of identical cores, where each core implements sophisticated microarchitecture techniques, such as superscalar and out-of-order execution, to achieve high single-thread performance. This approach can incur in high energy costs due to the power inefficiency of these techniques. Alternatively, a processor can contain many simple, low-power cores, possibly with in-order execution. This approach, however, sacrifices single-thread performance and benefits only applications with thread-level parallelism.

A heterogeneous system integrates a mix of *big* and *small* cores, and thus can potentially achieve the benefits of both [1, 2, 7–9, 12, 13, 20]. Despite their significant benefits in power and performance, heterogeneous architectures pose significant challenges to operating system design [4], which has traditionally assumed homogeneous hardware. Key among these challenges is scheduling. Homogeneous systems simplify the scheduler design by providing a uniform computing capability on each core.

Heterogeneous architectures can be classified into two types: *functional asymmetry* and *performance asymmetry*. Functional asymmetry refers to architectures where cores have different or overlapping instructions sets. For example, some cores may be general-purpose while others perform fixed functions such as encryption and decryption. Performance asymmetry refers to architectures where cores differ in performance (and power) due to differences in microarchitecture or frequency.

In this paper, we propose bias scheduling for performance asymmetric heterogeneous systems with cores that have different microarchitectures. We identify key metrics that characterize the potential benefits of scheduling an application on a big core over a small core, and the core type that best suits the resource needs of the application, namely its bias. By dynamically monitoring application bias, the scheduler is able to match threads to the core type that maximizes sys-

tem throughput. Bias scheduling can be easily implemented on top of any scheduler. We implemented bias scheduling on top of the Linux® scheduler on a system that models asymmetry accurately and found that it can improve system performance significantly. Moreover, these gains are proportional to the application bias diversity in the workload.

Our work is novel in two ways. First, our proposal makes dynamic scheduling decisions based on online application monitoring without requiring sampling performance on the different core types [3, 13] or offline profiling [15]. Second, to our knowledge, ours is the first implementation and evaluation of heterogeneity on a real system that models microarchitectural differences accurately by changing the internal workings of the cores. Previous work has been based on simulations or cores running at different frequencies, which as we will show, have significantly different profiles than cores with different microarchitectures and, therefore, challenge the applicability of these results.

The rest of this paper is organized as follows. Section 2 demonstrates the importance of scheduling threads on the right core type to maximize performance. Section 3 introduces the notion of *bias* and how it can be computed dynamically. Section 4 describes the asymmetric system and why it is novel. Section 5 presents the bias scheduling algorithm. Sections 6 and 7 discusses the experimental setup, implementation and results. Section 8 discusses related work and Section 9 presents our conclusions.

2. Background

Throughout this paper, we focus on a performance asymmetric heterogeneous system with two core types that have different microarchitectures, but the same algorithms can be applied to architectures with simpler sources of asymmetry like frequency or cache sizes. We focus on only two cores types for several reasons. First, general purpose heterogeneous designs with two core types already capture most of the power/performance benefits from asymmetry. Second, co-designing two cores for a new processor design already challenges design resources and it is unlikely that co-designing more cores would be commercially viable. Finally, it is unclear what designs would be desirable with more core types. The two core types we chose are representative of typical high performance out-of-order cores and power efficient in-order cores. The high performance *big core* has an area size significantly higher than the low performance *small core*.

Cores in a performance asymmetric heterogeneous system do not provide a uniform computing capability for each process. Indeed, while one application might see an exceedingly good performance boost when executing in a more *powerful* core, others might not see any benefit because they cannot take advantage of the additional hardware resources available to them.

To illustrate this point, Figure 1 shows the performance of the SPEC CPU2006 [17] components on a big core normalized to their performance on a small core. The big and small cores are representative of state of the art out-of-order and in-order cores, respectively. While the absolute values are not relevant, their distribution shows a significant variation among applications. On one side there are applications like *gamess* and *libquantum* that can harness over a 2x speedup on the big core, while on the other end of the spectrum applications like *mcf* and *astar* are only able to gain a modest 10%.

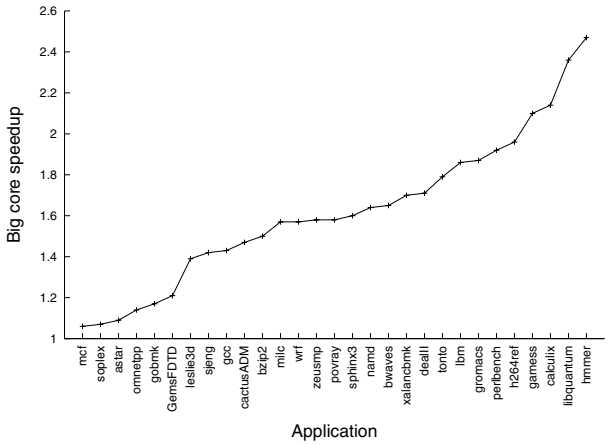


Figure 1: Speedup of a big core over a small core running each of the SPEC CPU2006 components

Most systems typically run a variety of processes concurrently, very often from different applications. It is rare, if not impossible, to be running a collection of identical or homogeneous processes exclusively, and even in that case it is likely that applications will go through different phases. The performance of these processes on a heterogeneous system will heavily depend on the scheduling choices made by the operating systems. Consider for example a simple two core heterogeneous system. If this system is running *mcf* and *libquantum* concurrently, the best choice would be to schedule *mcf* on a small core since it gets a very small gain from the big core. Conversely, *libquantum* should be scheduled on a big core given it can effectively exploit the added performance. Making the opposite choice will undoubtedly result in lower overall performance. While the figure above shows the relative performance of big and small cores for whole applications, this performance is nevertheless not static. An important aspect of any new scheduler design is to be able to adapt to the dynamic characteristics of the application as it goes through different phases of the computation.

This is a fundamental shift from typical scheduling policies on homogeneous systems were the application performance on each individual core is consistently the same, subject to small variations due to contention and locality of shared system resources [11, 19]. The intrinsic gap in core performance requires a different approach. Previous work on this area can be divided in two camps. The first one uses on-

line profiling of clocks per instruction by periodically sampling each thread on both core types and making a scheduling decision based on it until the next sample is taken. The second approach uses offline profiling of the application to give a static hint to the scheduler. We discuss the shortcomings of these approaches in detail in Section 8.

Our approach differs significantly from both of these. While we perform online monitoring of thread performance, we do not sample its execution on both core types. Instead we monitor critical performance data using standard low overhead performance monitoring hardware that the operating system can use to select a *preferred* core type for a thread, which we refer as application bias. By continuously monitoring the system, the scheduler can make dynamic decisions to assign threads to cores to maximize system throughput.

3. Application bias

The reasons that directly affect the performance of a core are many. We divide them into two broad categories. The first category consists of performance differences caused by microarchitectural choices in the core. Examples include out-of-order versus in-order core designs, the size of resources allocated (TLB, execution units, or private caches) or any other microarchitectural feature (branch predictors, pipeline depth, or unit latency). The second category consists of the performance effect of resources outside the core. These are typically relatively long latency events and include access to shared caches and memory, and I/O operations.

In the rest of this section we investigate these effects in detail and how they can be used to implement effective scheduling policies.

3.1 CPI breakdown

The clocks per instruction (*CPI*) metric is very useful to understand application performance. CPI can be broken down [6] into individual components to create a CPI stack. A CPI stack describes where instructions spend their time in the hardware. A detailed CPI stack would include the source of many stall types in the pipeline, including TLB misses, cache misses, branch mispredictions, resource allocation stalls and memory references, among others. Without this breakdown, CPI by itself in one core cannot be used to predict the CPI on another core since the source of stalls might remain constant (e.g. long latency dependent operations) or change when moving from one core to the other. However, a detailed understanding of the causes of core stalls is unnecessary for our purposes. We simply classify core stall cycles broadly into two classes:

1. *Internal stalls*: these are caused by the lack of resources internal to the core (an execution unit, a load port), competition on those resources (a TLB or private cache miss) or natural inaccuracies on them (branch misprediction). Most of these events are short in duration and can often be hidden by out-of-order microarchitectures. However, they are numerous and include many sources, leading to pipeline stalls even in the most CPU-bound tests.
2. *External stalls*: these are caused by access to resources external to the core. They include shared last level caches, memory and I/O. These events are significantly less frequent than internal stalls. However, their latency can be orders of magnitude larger than internal stalls.

The rest of the cycles are execution cycles where the core is actually executing instructions instead of waiting for resources.

3.2 Bias

Recall from Figure 1 that different applications behave significantly different when executing on cores of different type. Although this graph shows a continuum in the performance benefits of executing on a big core instead of a small core, it is obvious that applications on the left of the spectrum (modest speedups close to 1.0) will not benefit from executing on a big core as much as those on the right of the spectrum (large speedups close to 2.0).

We define application bias as the type of core that the operating system would *prefer* to run threads of the application at a particular time. More specifically:

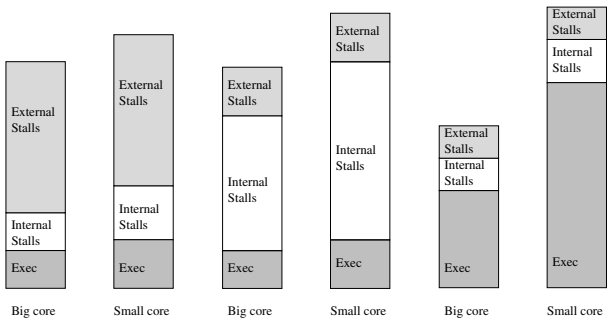
- A thread has a *small core bias* if its speedup from running on a big core compared to a small core would be modest.
- A thread has a *big core bias* if its speedup from running on a big core compared to a small core would be large.

Of course, the definition of what constitutes a modest or large speedup is dependent on the characteristics of the cores. For example, a 10% speedup might be considered large when comparing cores that differ only in cache sizes, but would be considered modest when comparing an out-of-order core with an in-order core. To select values for these speedups, an operating system might compare the performance of both cores on a battery of tests (like the ones shown later in Figure 3) and select a specific range in the curve to map a large speedup (e.g. the upper quartile).

Application bias is not static. While an application might have a certain bias overall, it can change as the application goes through different phases of the computation. Different threads from the same application might have different bias.

Previous work has focused on trying to figure out these ratios dynamically [3, 13] by sampling the CPI on all core types. However, sampling requires each thread to periodically switch cores, making this a very expensive proposition as thread migrations can negatively impact performance [15]. It also assumes a uniform application behavior during sampling and between samples.

Instead, we propose estimating the application bias using the information already available from the analysis of the types and causes of internal and external stalls.



(a) CPI dominated by external stalls, small core bias

(b) CPI dominated by internal stalls, small core bias

(c) CPI dominated by execution cycles, big core bias

Figure 2: CPI breakdown for several classes of applications. Cycles are broken down into the two major stall sources and execution cycles. The gap between big and small core CPI is biggest when CPI is dominated by execution cycles.

3.3 Bias and stall correlation

External and internal stalls can be used as a strong predictor of the application bias. We first discuss why this correlation might exist and then confirm this hypothesis with empirical data from the heterogeneous system in Section 6.1.

An application whose CPI is dominated by external stalls in a big core will undoubtedly be dominated by external stalls in a small core. The small core is unlikely to perform any better for obvious reasons, its simpler execution engine will not cope with memory latency better. Conversely, if an application is dominated by external stalls in a small core, what are the chances that it is also dominated by external stalls in a big core? To answer this question, we need to look at the continuous increase in the gap between core performance and memory performance. While out-of-order execution and memory speculation can partially hide the latency of external stalls, it is increasingly difficult for cores to hide it, so much so that for some applications they completely fail [18]. We argue that while the big core is able to hide some of this latency, if the application is dominated by external stalls on any core, it will also be dominated by external stalls in the other core. This situation is depicted in Figure 2-(a). While the absolute amount of external stalls is smaller in the big core, the fact that the CPI is dominated by external stalls make the relative speedup of the big core modest. The key is finding at what point is the CPI dominated by external stalls. In such cases, large external stalls is a predictor of small core bias.

Internal stalls present an interesting challenge. Given the nature of the big core, it is likely that it has more resources than the small core and that internal stalls will grow in the small core as shown in Figure 2-(b), but this might not always be the case. Not all sources of internal stalls are relevant to us. The choices in microarchitecture determine both the relevant stalls and their direction of growth between cores. For example, take two designs that are identical except that the small core has smaller caches and a larger TLB. The metrics that are more relevant for this case are those measuring TLB and cache stalls. However, their direction of change between the cores is not the same: on average, the small core has *more* cache related stalls but *fewer* TLB related stalls.

For this reason, it is preferable to leave it to core architects to deliver an abstraction for the type of internal stall metric that is relevant to the processor. When co-designing the cores, architects know what the key differences between microarchitectures are and can pass this information to the operating system. This abstraction could be a combination of existing events that measure the relative efficiency of the code on the key resource differences between the core microarchitectures. For example, a *hetero internal stalls* event would measure a combination of TLB and cache stalls in the previous processor example and *instruction starvation cycles* on our experimental platform, as we will demonstrate in Section 6, freeing the operating system developer from the specifics of the event.

Still, even without this abstraction, operating system designer can improve scheduling by analyzing the underlying microarchitectures. In Section 6 we explain how the microarchitecture choices in the experimental platform affect internal stalls, and why our selection for the internal stalls metric is a predictor of small core bias.

In summary, applications with CPI dominated by either internal or external stalls have a small core bias. If neither internal nor external stalls dominate the CPI, the application is dominated by execution cycles. Given the nature of the two core types, the big core is very likely to outperform the small core significantly, therefore we consider this type of application to have a big core bias.

Execution cycles could also be smaller for an application in a small core. This would be the case of a small core containing specialized accelerators. Extending our framework to this case is not particularly challenging but requires hardware feedback identifying the fraction of cycles spent on the accelerators.

4. Heterogeneous system

A perfect evaluation heterogeneous system would consist of a mix of cores with different microarchitectures, for example pairing a high-performance core such as the Intel® Xeon® Processor X5560 featuring the out-of-order core codenamed Nehalem with a low-power core such as the in-order Intel® Atom™ Processor. Unfortunately, this type of system is not available commercially. Therefore, researchers usually validate their ideas using other approaches. The first approach uses simulations. Simulations have the disadvantage of being slow, subject to many assumptions and lack the accuracy of a real system. The alternative is to emulate the target system using dynamic frequency/voltage scaling (*DVFS*). By

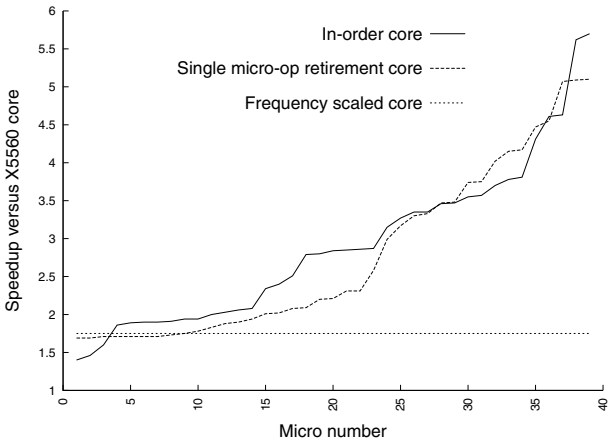


Figure 3: Speedup of a X5560 core over different small cores. Each point represents one of 40 micro-test. Speedups are sorted independently to show curve profile.

running cores at different frequencies, the researcher is able to emulate a system with cores that achieve different performance.

In this paper, we argue that using DVFS to emulate heterogeneous systems overly simplifies the real challenges in heterogeneous systems and might in fact lead to the wrong conclusions. Instead, we configure some of the cores of an X5560 processor as small using proprietary tools to enable a debug mode that reduces instruction retirement from four to one micro-op per cycle. Even though retirement is throttled at one micro-op, the CPI can be lower due to core stalls or higher due to micro-op fusion [21]. We believe this type of asymmetry is more representative than DVFS used in previous studies [2, 14, 15]. To evaluate this claim, we designed a set of 40 micro-tests that test common core features like execution bandwidth, branch misprediction penalty, private cache latency and so on. They explicitly avoid using any shared resources like last level cache and memory. Therefore, they exclusively test the core microarchitecture.

Figure 3 shows the relative performance of these micro-tests. Each line represents the speedup of a *big* 2.8 GHz core in the X5560 processor over a *small* core as represented by:

- A real in-order core running at 2.8 GHz.
- A X5560 core running in single micro-op retirement mode at 2.8 GHz.
- A X5560 core running at 1.6 GHz (DVFS)

Each curve is sorted independently because we are interested in showing the curve profile, not the exact performance delta between the different cores. With frequency scaling, execution cycles do not change at all since only memory appears slower, therefore core performance is a constant given by the frequency ratio. The in-order core shows a dramatically different profile. First, the performance degradation is significantly higher than that of a frequency scaled core. Second, the degradation is not constant and varies with the core features exercised by the micro-test.

On the other hand, the profile of the core in single micro-op retirement mode is very close to the profile of the the in-order core, where different micro-tests are impacted differently depending on the microarchitectural differences. On average, their performance is within 6% of each other, although individual tests vary significantly more.

This fundamental difference between the heterogeneous system used in our work and previous work that uses DVFS, forces the operating system designer to consider factors beyond caches and memory when assessing the relative performance of the cores. In particular, we will demonstrate that the metrics typically used in previous papers that model heterogeneity using DVFS, namely last level cache misses, are insufficient to explain the performance differences between core types in heterogeneous systems.

5. Algorithm Design

Our general approach consists of dynamically computing the bias of each thread and influencing the existing scheduler to select threads with the best bias possible when doing migrations. We avoid any intrusions on other parts of the scheduler that decide when to run a thread or when to perform load balancing operations, easing implementation.

5.1 Computing bias

To compute application bias we use performance monitoring hardware available in modern processors to measure internal and external stalls. Since the performance counters are a shared core resource, we virtualize the counters in the operating system in order to keep track of counts for each thread. On each context switch, a snapshot of the value of the counters is taken. The counts for the thread being switched out is updated with the increment in counter value since the previous snapshot was taken at the last context switch [11].

Several strategies are possible to periodically update the thread metrics. One approach is to keep cumulative counts since the thread creation. This has the advantage of producing fairly stable and slow changing results, but has the drawback that threads that change phases often will not get their bias updated fast enough. A second approach is to update the bias based solely on the last quantum of the execution of the thread. This has the advantage of adapting quickly to phase changes, but in practice we found it leads to excessive thread migrations and performance loss.

Our design uses a running average of the metrics over a sliding instruction window. As performance monitoring data is collected on a quantum, the running average is updated with the new data plus a fraction of the running average inversely proportional to the number of instructions executed in the quantum. This approach produces a gradual metric shift and can be fine tuned to identify bias changes.

Application bias changes as the amount of stalls fall below or climb above predefined thresholds, switching an application between small or big core bias. These thresholds determine when a thread is being dominated by either type of stalls and correlates well with its expected bias as show in Section 3.3. While we chose to have a system-wide threshold for each metric, it is possible to have per-application threshold that would shift some of the burden to an application layer.

5.2 Bias scheduling

Bias scheduling is not a from the ground-up scheduler design. Rather, bias scheduling is a technique that influences how an existing scheduler selects the core where a thread will run. It does not change the existing scheduler in ways that would dictate when to run a thread or change any system properties that the scheduler is trying to maintain such as fairness, responsiveness or real time constraints. Bias scheduling works by preferably scheduling applications on the core type that best suits its bias.

The operating system is first modified with several key changes to support performance asymmetry [14]. These changes include faster-core first scheduling, which enables maximum performance by scheduling first to idle big cores, and asymmetry-aware load balancing, which schedules work in big cores proportional to an estimated ratio of big/small performance. These changes are non-essential for bias scheduling, but are required to do a fair evaluation when the workloads have idle time.

Bias scheduling can be performed on top of any existing scheduling algorithm. Our design tries to minimize changes to the existing scheduler by focusing on two areas: imbalanced systems and balanced system.

- When the system is imbalanced, the scheduler tries to migrate a thread from the busiest core to the idlest core. We do not change the way in which these cores are selected. Once they have been identified, if the cores have different types, we migrate the required load that has the highest bias towards the destination core. If the cores have the same type, bias is irrelevant. In some instances the scheduler cannot find a thread with an appropriate bias, and defaults to migrating any thread just as it does without bias.
- On a balanced system, our changes are more extensive. Load balance is periodically checked and nothing is done if the system is balanced. In such case, we modify load balancing to inspect the load on the runqueue of the critical big cores looking for a thread that has a small core bias. If such thread is found, it then searches the small cores looking for a thread that has a big core bias. It then performs a thread swap on their runqueues. This process is started only in big cores since we assume there are fewer of them, but it works similarly if started on a small core.

Inspecting every thread in the runqueues of every small or big core can be an expensive proposition. Our implementation saves much of this effort by designing data structures that maintain a small sample of the last *N* threads run on each core as explained in Section 6.2.

Finally, we do not modify the initial thread allocation. While it is possible to allow for user annotations or historic tables for initial scheduling, that is complementary to our approach. In practice, however, we have found that threads show bias fairly quickly, so it might not be necessary to improve initial allocation.

Bias scheduling only hooks into the existing scheduler during load balancing. It is also limited to influence the core where a thread will be run based on bias and has no effect whatsoever on when the thread is run, making it transparent to other parts of the scheduler that select what to run based on priorities, fairness and other scheduler design considerations. Therefore, implementation on top of any existing general purpose scheduler design is simplified significantly.

5.3 Frequency scaling and power consumption

A common approach in modern systems is to apply dynamic voltage/frequency scaling (*DVFS*) to reduce power. If cores can be individually scaled, it opens up the possibility of improved scheduling on this form of performance asymmetric heterogeneous system.

Our design can be changed to support DVFS. First, notice that DVFS is just a special case of our general model. Since there are no microarchitectural differences, the internal stalls will be constant in cycles across all cores (recall that internal stalls are expressed as a fraction of CPI). Therefore, external stalls become the only predictor required. Using only externals stalls allows the operating system to effectively place memory-starved threads on lower frequency cores, or lowering the frequency of a core if all threads running on the core have a small core bias, leaving the high frequency cores free for CPU-bound threads. However, it becomes necessary to support more than two bins as shown in [7].

Previous work has shown the power savings that can be achieved with DVFS. Unfortunately, although similar savings are possible with the addition of heterogeneous cores, our prototype prevents us from measuring any type of power savings. We are currently developing another prototype that would enable accurate power measurements by using real asymmetric cores.

6. Experimental setup

All of our experiments were conducted on a Supermicro® X8DTN board with two Intel® Xeon® Processor X5560 running at 2.8 GHz. Each X5560 processor has four cores sharing an 8 MB last level cache and an integrated memory controller. We assume that the big core has an area size equal to three small cores (3:1 ratio), therefore we configure three of the X5560 cores as small. To keep the core to cache area in

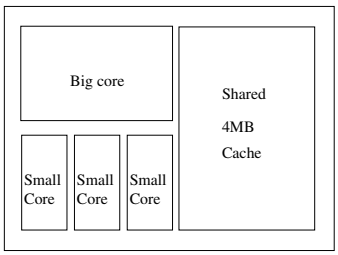


Figure 4: Processor configuration. Each socket contains a single big core and three small cores sharing a cache.

line with current shipping processors (2MB per core on the X5560) we reconfigure the last level cache to 4MB (2MB per big core area equivalent).

The resulting processor consists of one big and three small cores as shown in Figure 4. We refer to this heterogeneous system with one processor as *1B3S* and the one with two processors as *1B3S+1B3S*. In some instances we also report results for a homogeneous system with four big cores on a single processor that we refer as *4B*.

6.1 Estimating stalls

Accurately measuring stalls in an out-of-order core is challenging because most of these events occur in parallel and can be hidden by speculation. Some processors have added performance counters to estimate those stalls. In our system, we lack a precise count or are required to collect many events that exceed the number of hardware counters available.

Given those constraints, we opted for a more empirical approach for our evaluation. We do not need to precisely know the amount of cycles spent on each stall type, instead we just need to know if they are above or below certain thresholds that would qualify the application as small or big core biased.

We performed extensive evaluation of the performance counters available in our system and found a significant correlation between certain counters and an application relative performance on big and small cores.

To estimate the amount of external stalls, we use the number of requests serviced outside the core (last level cache and memory), we call them offcore requests. Offcore requests are critical because they are very likely to cause core stalls while waiting for the last level cache or memory to respond. Figure 5 shows the average number of offcore requests per 1K instructions for the SPEC CPU2006 components. On the secondary axis we also plot the speedup obtained from the big core. This graph clearly shows a good correlation between offcore requests and big core speedup. Indeed, the applications that benefit the most from big cores (e.g., *gromacs* to *hmmer* on the right side) are among those with the lowest amount of offcore requests. On the other hand, the applications that benefit the least from the big core (e.g., *mcf* to *gcc* on the left) have the highest amount of offcore requests.

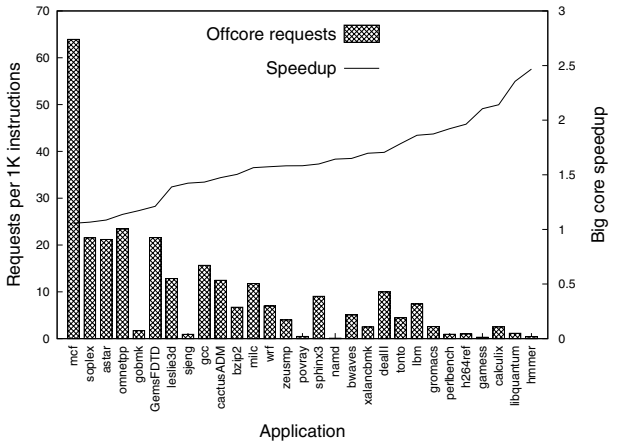


Figure 5: Correlation between offcore requests and big core speedup. Offcore requests are those that are serviced outside of the core: the last level cache and memory.

However, there are several outliers to this rule, most notably *gobmk* and *sjeng*. As we will see shortly, these two applications are dominated by internal stalls.

In order to estimate internal stalls, recall that the difference between the cores in our experimental platform is the speed at which they retire instructions. If the cores are kept busy, the performance will suffer significantly on a small core. If, on the other hand the cores are idling due to lack of instructions, the performance of the two cores will be closer. For this reason we compute the number of cycles when the front end of the machine is not delivering micro-ops to the back end. This is referred to as instruction starvation.

Figure 6 shows the correlation of instruction starvation with the big core speedup. Once again, the correlation is strong. More importantly, it shows that the two applications were the external stalls correlation was weak, *gobmk* and *sjeng*, are affected significantly by internal stalls.

The results from Figures 5 and 6 demonstrate that, unlike previous work based on DVFS, the performance delta between cores with different microarchitectures cannot be explained solely using a single memory-centric metric like last level cache misses. While such metric might cover the majority of applications, other factors intrinsic to the core microarchitectures (e.g. internal stalls) become relevant.

While our analysis to find the performance event metrics that best describe the stalls is mostly empirical and that more accurate metrics are possible, we believe that the results in Section 7 demonstrate that there is sufficient merit to this approach.

6.2 Scheduler implementation

We implemented bias scheduling in the Linux kernel 2.6.27.5. On every context switch, the metrics for external and internal stalls, *offcore requests* and *instruction starvation* are updated with a running average over the instruction window.

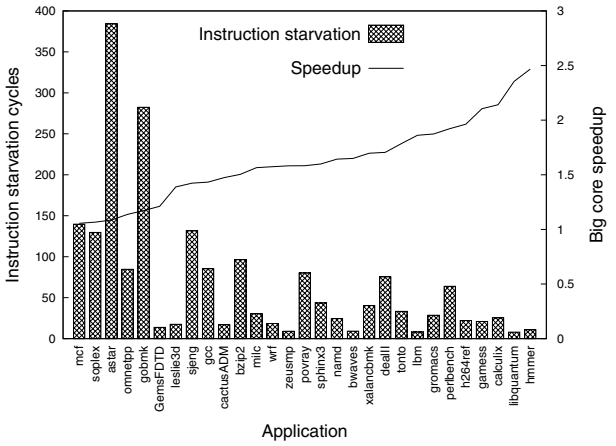


Figure 6: Correlation between front end instruction starvation and big core speedup. The front end of the machine is starved when there are no instructions to be issue to the execution engine.

We normalize metrics to the number of instructions retired, expressing them per 1K instructions. For offcore requests we only count the number of demand reads and requests for ownership (RFO) since those are the ones likely to stall the core. Other type of offcore requests like prefetches or writebacks are not included since they do not stall the core. Instruction starvation is measured as the difference between micro-ops issued stall cycles and resource stalls (i.e. cycles where no micro-ops were issued but not because of lack of resources).

Migrations during normal load balancing in Linux are accomplished by searching the runqueue for a thread that can be moved from the busiest core (i.e, it is not running or pinned). We modify it to select the thread in the runqueue of the busiest core that has the highest bias towards the idlest core, but only when the core types are different. For simplicity, if no such thread can be found, we rescan the runqueue but ignore bias.

When the system is balanced we find the candidates for migration by locating two threads on different core types with bias opposite to the core they are running on. One caveat is that in practice, this could often be a running thread, which will normally not be moved in the Linux kernel. For this reason, on every bias migration we activate the kernel migration thread on both cores, which will deactivate the running thread and allow its migration. There is a handshake protocol observed so that each core is only involved in one swap operation at a time.

One of the key operations that is repeatedly done in our algorithm is to find the thread with the highest (or lowest) bias. Given that this has to be done potentially on every core in the system and that the data resides within each thread data structure, it could lead to a very slow process. In our implementation we maintain a small circular history buffer on each runqueue that tracks the bias of recently run threads. On each context switch, the outgoing thread updates the history buffer with its current bias. When a thread is migrated from a runqueue, the history buffer is cleared. If there are traces of a recently run thread with the appropriate bias in the history buffer of a runqueue, this thread becomes a candidate for bias migration.

From a memory perspective, our algorithm requires very modest additions. We store per thread performance events for four counters: instructions retired, offcore requests (demand reads plus RFO), micro-ops issued stall cycles and resource stalls [10]. This requires 4x8 bytes per thread. Additionally, the bias history buffer is kept per core. The metrics used to compute bias are offcore requests per 1K instructions and instruction starvation per 1K instructions. Each metric is safely represented with 16 bits in a 10 entry buffer. In term of complexity, the total code added or modified is 4200 lines, of which 3200 are in a supporting role (performance monitoring) and other general purpose asymmetry awareness in the operating system such as core type detection, proportional load balancing and near-idle performance optimizations [14]. Only about 1000 lines of code are needed to implement bias scheduling, and that includes significant amounts of comments, conditional compilation and debug instrumentation.

7. Results

To evaluate bias scheduling, we compare its performance against the Linux scheduler (*stock*) using multiple combinations of SPEC CPU2006 components. The first set consists of heterogeneous workloads running a mix of different applications. The second set consists of homogeneous workloads, either a parallel application or multiple copies of a serial application.

7.1 Heterogeneous workloads

Each heterogeneous workload consists of multiple components of the SPEC CPU2006 benchmark, one copy of one component and three copies of another. Since each instance finishes at a different time, we avoid idle cores by repeatedly running the component until each has run at least once. We report execution times for the first run.

To select applications we divided them in two groups according to their big core speedups above or below 1.5, resulting in 12 applications under 1.5 and 17 over it. Eight workloads were chosen with one application from each group with markedly different speedups, which would be the cases where we would expect to find significant diversity in bias. Three other workloads where chosen from applications very close the middle, were it is unlikely to find a strong bias to either core. Table 1 shows the two components of the eleven workloads and their big/small core speedups.

Based on the results from Figures 5 and 6 we set the thresholds for these metrics at 130 starvation cycles and

3-copies	Speedup	1-copy	Speedup
soplex	1.07	hmmer	2.47
GemsFDTD	1.21	gromacs	1.87
leslie3d	1.39	dealII	1.71
gcc	1.43	perlbench	1.92
sjeng	1.42	lbm	1.86
mcf	1.06	libquantum	2.36
astar	1.09	calculix	2.14
omnetpp	1.14	xalancbmk	1.70
gobmk	1.17	wrf	1.57
cactusADM	1.47	zeusmp	1.58
milc	1.57	bwaves	1.65

Table 1: Workload composition. Each workload consists of three copies of the first application and one copy of the second. Their respective big over small core speedups are also listed.

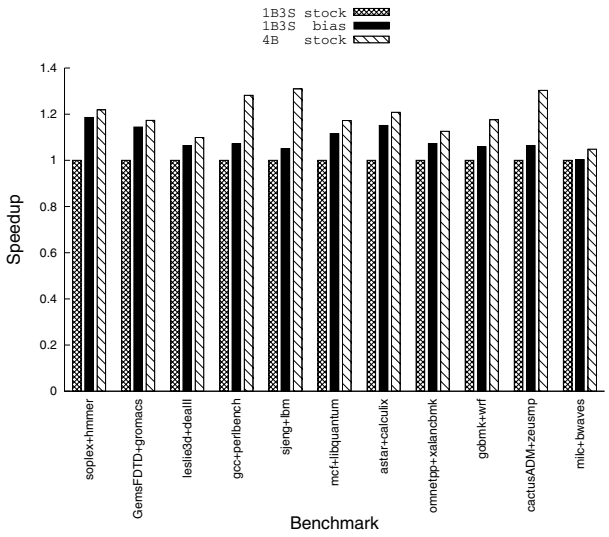


Figure 7: Performance of heterogeneous workloads with bias scheduling relative to the stock scheduler on 1B3S. An upper bound with the stock scheduler on 4B is also shown.

10 offcore requests per 1K instructions. We also set the instruction window size to 16 billion. In Sections 7.3 and 7.4 we discuss how to tune these parameters.

Figure 7 shows the performance results of bias scheduling (*bias*) vs. the Linux scheduler on the 1B3S system. We do not show results for a 2B system of the same area since the performance benefits of heterogeneous systems have already been demonstrated elsewhere. Instead, we show the 4B performance as an upper bound on performance. Execution time is calculated using the geometric mean of the execution times of all applications. One caveat is that the Linux scheduler shows significant variability on a performance asymmetric system due to the random chance of choosing better application to core mappings. Therefore the stock 1B3S times are a sample of the performance of the Linux scheduler on 1B3S.

In these workloads, bias scheduling is able to improve performance of the 1B3S system by an average of 9%, with several workloads getting more than 15%. Notice also that the performance of bias scheduling on 1B3S comes very close to the performance of the 4B homogeneous system. When the workload includes applications that benefit modestly from big cores, the performance difference between the 1B3S and 4B systems is less than 5%, indicating our algorithm is very near the optimal since the 4B performance cannot be achieved by the 1B3S system ever. The exception are those workloads where both applications have large big core speedups (e.g. *sjeng+lbm* have big core speedups over 1.42), in this case it is impossible for the 1B3S to come close to the 4B homogeneous system. This result is by itself significant, because it goes to show the value of a heterogeneous system paired with an asymmetry-aware operating system. In those workloads where there is a clear differentiation in bias (the top eight in Table 1), results are much better than in those with a weaker bias differentiation, 11% vs. 4%, respectively.

Surprisingly, *milc+bwaves* does not show any benefit because its performance on the 1B3S system is about the same as in the 4B system, which contradicts their big core speedups of over 1.5 from Table 1. These results highlight the problem of profiling an application in isolation, as we did to select workloads. When the application is run concurrently with others, its profile changes. Indeed, when *milc+bwaves* run together they suffer from a significant increase in contention for shared resources, transforming them from a mildly core-bound individual applications, to a memory-intensive combined workload.

When looking at the performance of each individual thread we discover that the threads that are fairly sensitive to core type are able to speedup an average of 65% as shown in Figure 8. At the same time, the performance of the other threads (not shown) remains relatively unchanged, with a 5% slowdown. This goes to show that the bias-aware scheduling is able to trade off a small performance degradation in one thread for a very significant performance speedup in another.

The bias scheduling gains are highly dependent on the ratio of big to small core count. The performance upside of having a big core is increasingly diluted as the number of small cores grows. Although not shown here due to space constraints, when we execute the same tests on a *1B1S* system, the performance gains more than double to 20% on average.

In all the cases we have tested, the runtime overhead of bias scheduling is very small, less than 1%, even in those tests that have no bias. We have not observed any performance loss in a single processor configuration, although it is possible to find more performance upside by improving the accuracy of the predictors and thresholds. We have also

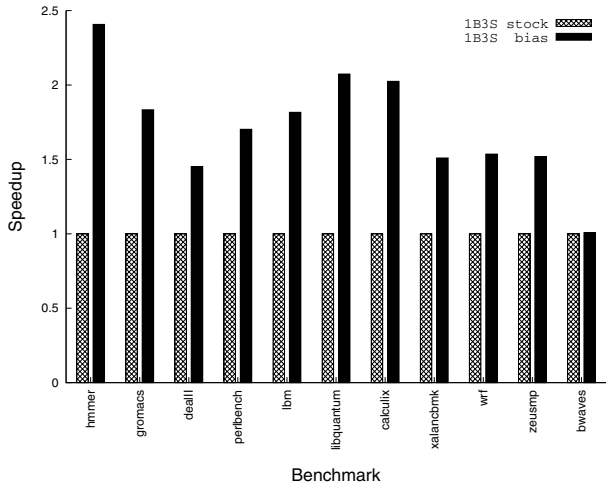


Figure 8: Performance of big core biased application with bias scheduling relative to the stock Linux scheduler on 1B3S.

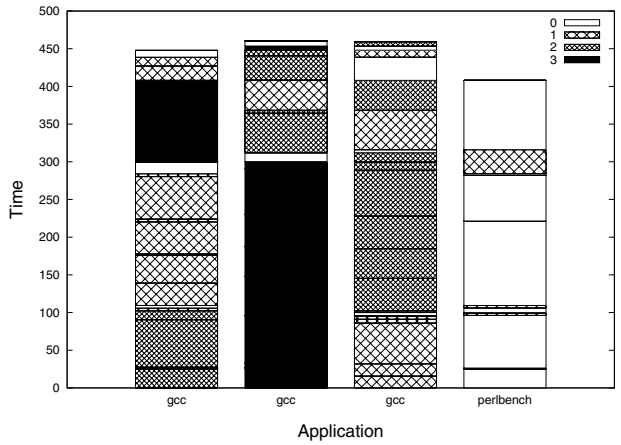


Figure 9: Core distribution profile of *gcc*+*perlbench* on 1B3S. Each column represents one thread, and the patterned boxes are the runqueue where it was at the time. Core 0 is the big core.

found that the performance variability observed in the Linux scheduler in performance asymmetric systems almost disappears in our scheduler.

One of the key aspects of our algorithm is its ability to dynamically adapt to changes in the application profile. Figure 9 shows an example of this behavior on *gcc*+*perlbench*. For each thread we show the core where it is queued (most likely running, but not necessarily). A core switch is clearly seen by the change in pattern vertically. When the application changes phase, we see a corresponding change in the core allocated to it, in particular we see how *perlbench* is mostly allocated to the big core (core 0) because is has very few stalls for most of its life. But when it becomes bound by stalls, the scheduler is able to dynamically switch other threads to the big core from *gcc* that are able to exploit it better, at least temporarily. Sometimes the phase changes occur within a single thread. In the case of *perlbench*, it executes three Perl scripts which have different characteristics at runtime. In this case, the input set determines application profile at runtime.

7.2 Homogeneous workloads

In homogeneous workloads every core in the system is running very similar threads, either the result of running multiple instances of the same serial application or multiple threads from one parallel application. Bias scheduling should have a very minimal effect on these workloads since processes should have similar bias overall. Any gains would come exclusively from exploiting dynamic changes in bias as applications change phases.

In the first experiment we run eight components of SPEC CPU2006 that have been parallelized with the Intel® compiler. These loop parallel applications synchronize often across all threads. In the second experiment, we run multiple

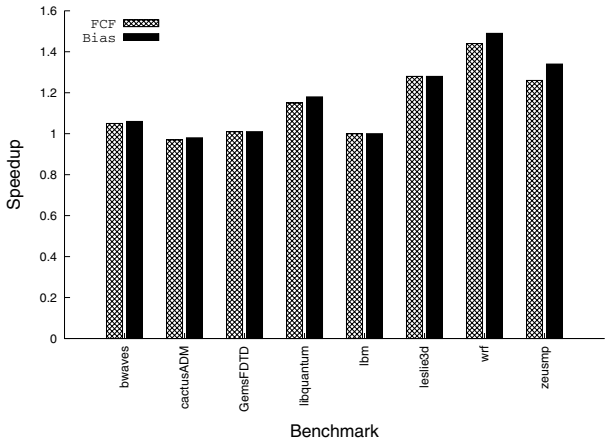


Figure 10: Performance of parallel homogeneous workloads on faster-core first (FCF) and bias scheduling relative to the stock Linux scheduler on 1B3S.

copies of the same application that are not synchronized at all.

Figure 10 shows the performance of bias scheduling on the parallel application relative to the Linux scheduler. Unlike the heterogeneous workloads, these workloads contain a significant amount of idle time and often benefit from faster-core first (*FCF*) scheduling [14] because threads synchronize at the end of a parallel loop. Therefore, we show the performance benefit of FCF by itself, followed by the additional gains due to bias scheduling.

As expected, five of the workloads show less than a 1% improvement with bias scheduling. However, the remaining three show modest additional gains of 3% to 8% over the Linux scheduler. While the threads are homogeneous, they

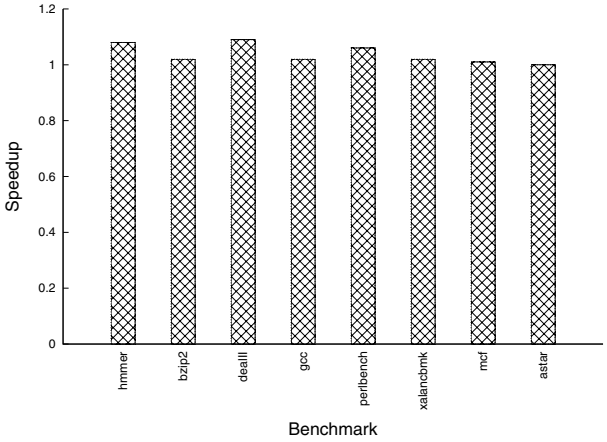


Figure 11: Performance of the multiple-copy homogeneous workloads relative to the stock Linux scheduler on 1B3S. Each workload runs four copies of the application.

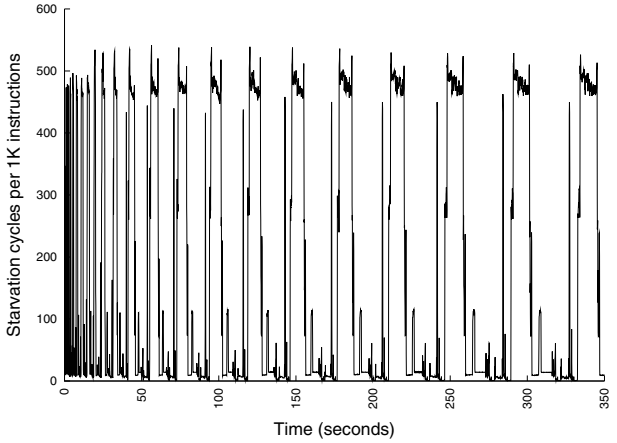


Figure 12: Execution profile of *dealII*. Each sample is 100ms.

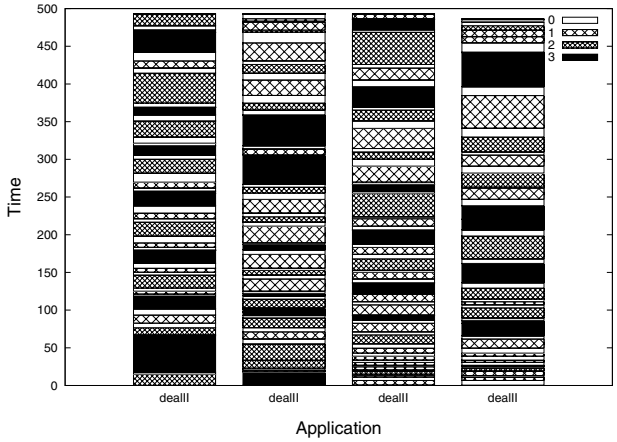


Figure 13: Core distribution profile of multiple copies of *dealII* on 1B3S. Each column represents one thread, and the patterned boxes are the runqueue where it was at the time. Core 0 is the big core.

reach phase changes at different times due to the difference in core performance, leading to these small gains.

Figure 11 shows the performance of bias scheduling on the multiple-copy workloads relative to the Linux scheduler. On average they gain 5%, with three workloads gaining more than 8%. These workloads are more likely to reach phase changes at different times since the applications are running independently until completion, while the parallel homogeneous workloads synchronize before each serial section.

In particular, *dealII* demonstrates one of the arguments in favor of dynamically adjusting to phases in the computation. Figure 12 shows a time based profile of instruction starvation cycles in *dealII* during its lifetime. While overall *dealII* is big core biased, there are times during its execution when it is not. Indeed, Figure 13 shows the distribution of the cores were each copy executed. The big core gets distributed across the different copies of *dealII* as they switch bias due to phase changes, leading to a 9% speedup.

7.3 Stall thresholds

One of the challenges with our approach is to find the conditions in which a thread is considered dominated by external or internals stalls.

The first issue is figuring out the events that should be used to correlate the stalls. Our approach has been to use detailed knowledge of the two microarchitectures to figure out events that best represent how stalls affect performance in the two cores. We argued in Section 3.3 that future heterogeneous system should relieve the operating system designer from this task by providing a counter abstraction that is specific to the system.

The second, more pragmatic issue is to find the thresholds at which the bias is considered small core or big core. While we chose to perform this process manually, it is not difficult to automate it. The operating system or the user could run a battery of tests to compute small to big core speedups. This provides the range of speedups expected for a variety of applications relevant to the system. Specific points in this range can be chosen to represent big or small bias. Correlating this range with the sampled performance events would allow for automatic generation of the stall thresholds. Notice that this has to be done once on the system using representative applications, and only to estimate the effects of stalls on different application profiles. Once they are set, there is no need to recalibrate the system unless the relevant workloads change.

7.4 Instruction window

The optimal instruction window size has to balance conflicting goals. On one hand, a fine grain window would allow for fast detection of phase changes, but can also result in too

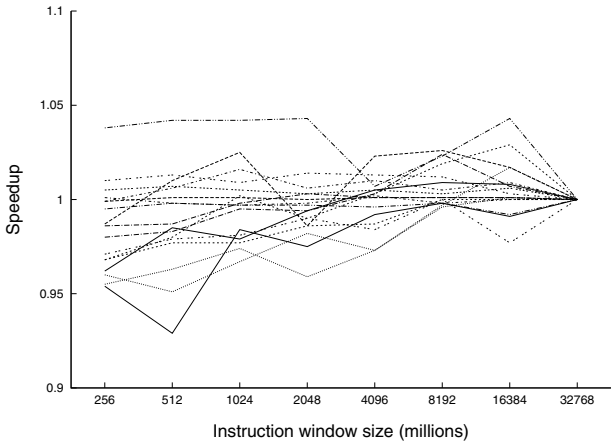


Figure 14: Bias scheduling performance with varying instruction window size. Each line corresponds to one workload presented on this paper. Data is normalized to the performance with 32 billion instructions.

many migrations. A coarse window, on the other hand, reacts slower to phase changes and can delay or completely miss migration opportunities. Two things factor into the optimal instruction window size:

- frequency and length of phase changes, and whether consecutive phase changes straddle bias thresholds.
- thread migration cost, including cache refills.

Figure 14 shows the sensitivity of all the workloads from Sections 7.1 and 7.2 to the instruction window size. While all workloads show a change in performance as the instruction window changes, this change is not uniform and there is not an optimal point. Many applications are very uniform, therefore their performance is very similar in all configurations and the number of migrations triggered by bias is very small. On the other hand, irregular applications show more sensitivity, with the number of migrations growing two orders of magnitude as the instruction window size shrinks.

Fortunately, on average, all the large instruction window sizes (2 to 32 billion instructions) have very similar performance, and outperform the small instruction window sizes (up to 1 billion) by up to 3%. While the small instruction window tends to detect phase changes faster, it is also susceptible to excessive migrations when those phases are short. Using performance monitoring profiles, we found that for these workloads a large instruction window size works slightly better. Consecutive phases of the application often lie within the same side of the thresholds, leading to a coarser change in bias.

7.5 NUMA systems

Modern systems spread memory across multiple memory controllers distributed around the system, leading to non-uniform memory access (*NUMA*) times that depend on the proximity of the memory controller to the core making the access request.

NUMA systems require several operating system optimizations, particularly in memory allocation and load balancing. Process memory might be interleaved across all memory controllers or preference could be given to the local memory controller. While load balancing is typically done across all nodes in the system, some operating systems might allow limits on migrations to remote nodes to improve locality. These policies can adversely impact each other. For example, if memory allocation is local, load balancing across nodes will result in a performance penalty for processes that suddenly find all of their memory remote.

Unfortunately, no single NUMA policy is best for all workloads. The memory allocation policy trades-off memory latency vs. memory bandwidth, while the scheduling policy balances locality vs. idle cores. Operating systems usually have a default policy that is a good compromise between the two but allow users or systems administrators to modify this policy on a per application basis (e.g. numactl and taskset in Linux).

Bias scheduling in a NUMA system follows the same principles of the multiple scheduler domains: preference is given to maintain processes within their current domain to improve locality, but domains are traversed hierarchically from the bottom looking for bias migration opportunities. With interleaved memory allocation, there is no memory latency penalty when migrating threads across nodes. With local memory allocation, there is a penalty when migrating to a remote node. Therefore we evaluate the performance of bias scheduling in these two scenarios.

Figure 15 shows the performance of bias scheduling on a two node system with one big and three small cores on each (*1B3S+1B3S*) on four workloads. Workloads have been chosen by doubling the work on some of the workloads from Section 7.1.

The first observation is the performance of the Linux scheduler under the two memory allocation policies. The *local stock* performance is 3%-13% better than the *interleaved stock* performance for three workloads. Not surprisingly, the worst case is *libquantum+mcf*, since *mcf* is the application most sensitive to memory latency.

Regardless of the memory allocation policy in effect, bias scheduling is able to improve performance in all cases by 6%-20% with one exception. In particular, bias scheduling delivers an average speedup of 8% with interleaved memory since node migration does not increase memory latency. On the other hand, bias scheduling does not always result in a performance gain when using local memory allocation. Again, the workload containing *mcf* is actually hurt by 7% due to the increased memory latency as evidenced by its loss of 13% when going from local to interleaved memory.

A side effect not shown in the figure is that bias scheduling increases the variability in execution times in memory

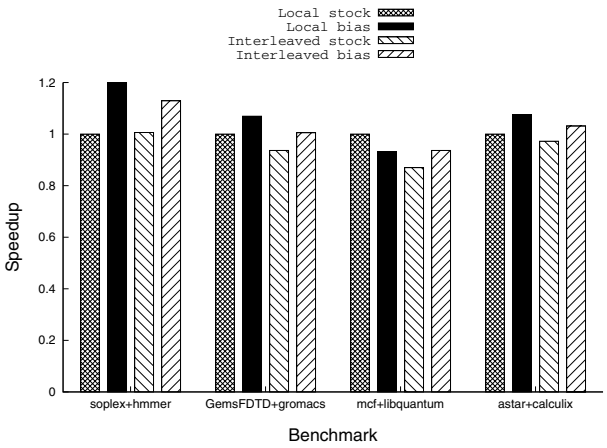


Figure 15: Performance of bias scheduling on a two socket NUMA heterogeneous system (1B3S+1B3S) with local and interleaved memory allocation policies. All data is relative to the stock scheduler with local memory allocation.

sensitive workloads when the memory allocation is local. With interleaved memory the gap between fastest and slowest execution of an application in a workload is less than 1%, while with local allocation this gap becomes as large as 25%.

Based on these results, bias scheduling should always be used in NUMA systems with interleaved memory allocation. However, the best overall performance in NUMA system is often achieved by using local memory. In that case, bias scheduling still delivers performance gains on average, but it can hurt performance for applications that are very sensitive to the increased memory latency as workloads migrate to remote nodes. In that case, it is desirable that users (or the scheduler) back off from cross node migrations. Indeed, when we modify the scheduler to prevent cross node migrations we eliminate any performance losses, but we are subject to the randomness of the initial node allocation for each application. The latter can be addressed by using thread affinity masks to control initial thread placement.

8. Related Work

Previous work on scheduling for performance asymmetric heterogeneous architectures have focused on either online CPI sampling [3, 13] or offline profiling [15]. Our approach does not require sampling CPI on all core types and overcomes the limitations of offline profiling.

Kumar et al. [13] demonstrated the performance benefits of heterogeneous architectures. Using simulation, they examined heuristics to dynamically adjust scheduling based on sampling the performance of permutations of thread to core assignments with different strategies, followed by a steady phase. Using simulations Becchi and Crowley [3] demonstrated that dynamic thread assignment outperforms any static assignment in heterogeneous architectures. However, they resort to sampling the CPI of each thread by executing a sampling phase on the two core types that are triggered by a rapid variation in CPI.

Shelepov et al. [15] propose generating offline application signatures using cache misses that can be used by the scheduler to match the application characteristics to the different core types. Our approach differs in several ways. First, we showed than on real asymmetric systems there are other factors beyond cache misses that are critical for scheduling, namely internal stalls. Second, their approach is limited in practice because input sets are not necessarily known beforehand and they can cause significant changes in application behavior. Finally, signatures are static and do not account for phase changes in the workloads. Sondag et at. [16] propose using offline analysis to identify and group similar application phases coupled with dynamic monitoring that extrapolates the behavior of a representative phase to determine core assignment for other phases in the group.

Previous research have proposed scheduling policies that detect application phases in systems with identical cores or with different frequencies. DeVuyst et at. [5] studied scheduling policies to adapt to thread execution phases on a homogeneous processor with simultaneous multi-threading using sampling and electron policies. Ghiasi et al. [7] proposed using performance counters to predict thread performance at a given core frequency and guide scheduling to reduce power consumption with minimum performance loss. Our design applied to a DVFS system is similar to this work, but our framework extends beyond frequency asymmetry.

9. Conclusions

Performance asymmetric heterogeneous architectures impose new challenges to scheduler design since the computing capability of each core is no longer uniform. Moreover, different applications can benefit differently from big cores.

To address this problem we have proposed bias scheduling. By decomposing the sources of core stalls that limit application performance we defined metrics that can be correlated with the core type that is best suited for the application. Application bias can be monitored dynamically by the operating system to guide scheduling decisions that maximize performance. Bias scheduling uses this information to influence load balancing to give preference to migrations where the core matches the application bias. In particular, load balancing is modified to select threads that have the most bias towards the target core during migrations and balanced systems are periodically inspected for threads with bias opposite to the core they are scheduled on. Since bias scheduling is limited to changes in load balancing, it can be implemented reasonably well over most schedulers.

While finding metrics that correlate well with application bias is possible in existing systems, this poses a serious challenge to the operating system designer. A better alternative is for architects of future systems to hide the underlying details by abstracting the bias metrics that are more relevant to each

specific system. Moreover, the inclusion of precise counts could allow us to unify and compare the different sources of stalls.

By comparing the speedups of cores with real microarchitectural differences and a single core running at different frequencies, we show the limits of commonly used DVFS as an evaluation vehicle for heterogeneous systems. Therefore, our evaluation is performed in a system that accurately models cores with different microarchitectures.

We implemented bias scheduling on top of the Linux scheduler. Using bias to select the preferred core type for each application during load balancing, the scheduler is able to deliver performance gains that are proportional to the amount of bias diversity in the workload. For heterogeneous workloads with a clear bias differentiation, performance is improved by an average of 11% on a 1B3S system. For workloads with threads of similar bias, the gains are smaller, as expected. Even in homogeneous workloads, bias scheduling can take advantage of phase changes during execution to allocate the big cores to the threads that can exploit it best temporarily, leading to gains of up to 9%. In NUMA systems, bias scheduling delivers solid performance gains under both local as well as interleaved memory allocations policies. Still, it has the potential for increased average memory latency as threads migrate across nodes, which can be eliminated by preventing such migrations.

References

[1] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttling. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 298–309, June 2005.

[2] 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*, pages 506–517, June 2005.

[3] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 29–40, 2006.

[4] F. A. Bower, D. J. Sorin, and L. P. Cox. The impact of dynamically heterogeneous multicore processors on thread scheduling. *IEEE Micro*, 28(3):17–25, 2008.

[5] M. DeVuyst, R. Kumar, and D. M. Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors. In *Proc. of the 20th International Parallel and Distributed Processing Symposium*, Apr. 2006.

[6] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A top-down approach to architecting CPI component performance counters. *IEEE Micro*, 27(1):84–93, 2007.

[7] S. Ghiasi, T. Keller, and F. Rawson. Scheduling for heterogeneous processors in server systems. In *Proceedings of the 2nd Conference on Computing Frontiers*, pages 199–210, May 2005.

[8] M. Gschwind. The Cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, June 2007.

[9] R. A. Hankins, G. N. Chinya, J. D. Collins, P. H. Wang, R. Rakvic, H. Wang, and J. P. Shen. Multiple instruction stream processor. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 114–127, June 2006.

[10] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*. Intel Corporation, 2009.

[11] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multi-core systems. *IEEE Micro*, 28(3):54–66, 2008.

[12] 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 IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Dec. 2003.

[13] 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 *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 64–75, June 2004.

[14] T. Li, D. Baumberger, D. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multicore architectures. In *Proc. of the 2007 ACM/IEEE Conference on Supercomputing*, Nov. 2007.

[15] 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(2):66–75, Apr. 2009.

[16] T. Sondag, V. Krishnamurthy, and H. Rajan. Predictive thread-to-core assignment on a heterogeneous multi-core processor. In *Proceedings of the 4th workshop on Programming Languages and Operating Systems*, pages 1–5, 2007.

[17] Standard Performance Evaluation Corporation. The SPEC CPU2006 benchmark. http://www.spec.org/cpu2006.

[18] R. Stets, L. A. Barroso, K. Gharachorloo, and B. Verghese. A detailed comparison of TPC-C versus TPC-B. In *Third Workshop on Computer Architecture Evaluation Using Commercial Workloads*, 2000.

[19] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 47–58, 2007.

[20] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proc. of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 156–166, June 2007.

[21] O. Weschler. Inside Intel Core Microarchitecture. White Paper, Intel Corporation, 2006.