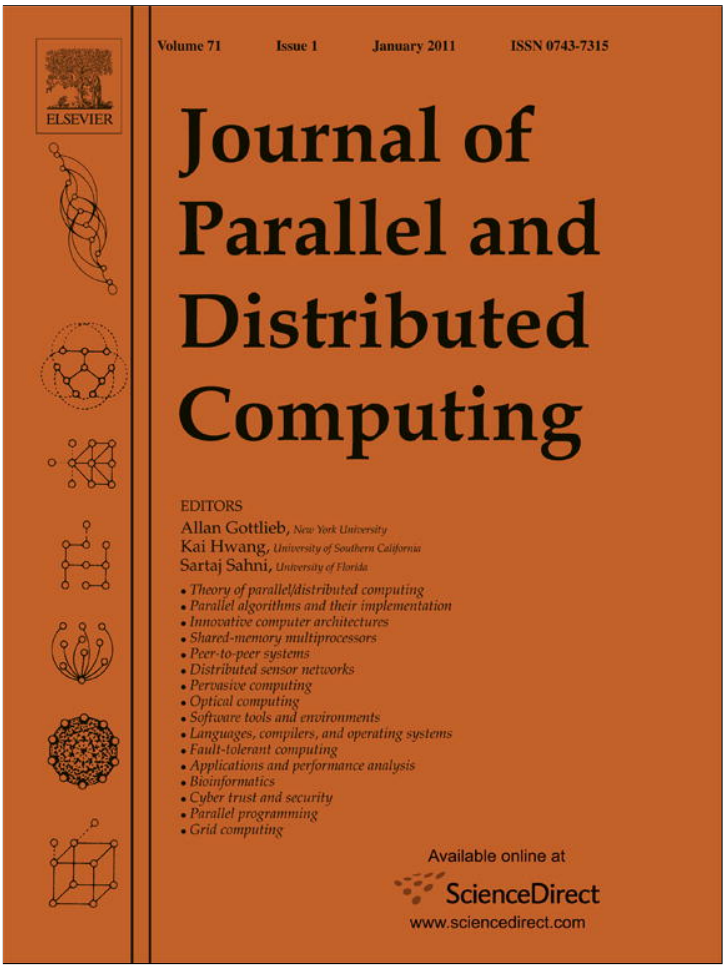


J. Parallel Distrib. Comput. 71 (2011) 114–131

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems

Juan Carlos Saez [a,*], Daniel Shelepov [b], Alexandra Fedorova [c], Manuel Prieto [a]

[a] Complutense University of Madrid, Facultad de Ciencias Físicas, Ciudad Universitaria s/n, Madrid 28040, Spain

[b] Microsoft Corporation, One Microsoft Way Redmond, WA 98052-7329, USA

[c] Simon Fraser University, School of Computing Science, 8888 University Drive, Burnaby, BC, Canada V5A 1S6

ARTICLE INFO

Article history:
Received 15 March 2010
Received in revised form
8 July 2010
Accepted 31 August 2010
Available online 21 September 2010

Keywords:
Heterogeneous multicore
Asymmetric single-ISA processors
OS scheduling
Workload characterization

ABSTRACT

Recent research has highlighted the potential benefits of single-ISA heterogeneous multicore processors over cost-equivalent homogeneous ones, and it is likely that future processors will integrate cores that have the same instruction set architecture (ISA) but offer different performance and power characteristics. To fully tap into the potential of these processors, the operating system must be aware of the hardware asymmetry when making scheduling decisions and map applications to cores in consideration of their performance characteristics. We propose a Heterogeneity-Aware Signature-Supported (HASS) scheduling algorithm that performs this mapping using per-thread architectural signatures, which are compact summaries of threads' architectural properties. We implemented HASS in OpenSolaris, and demonstrated that it always outperforms a heterogeneity-agnostic scheduler (by as much as 12.5%) for workloads exhibiting sufficient diversity. Our evaluation also includes an extensive comparison with other heterogeneity-aware schedulers to provide a more clear understanding of the pros and cons behind HASS.

1. Introduction

Single-ISA heterogeneous multicore processors [19], also known as *asymmetric*, have been proposed as a more power efficient alternative to homogeneous multicore architectures. A heterogeneous processor would consist of cores exposing the same ISA, but delivering different performance. The cores may differ in clock frequency, power consumption, and possibly in cache size and other microarchitectural features. Given a diverse workload, a heterogeneous multicore system can deliver a higher performance per watt than a homogeneous one, because threads can be matched to cores according to the relative benefit that they derive from running on different core types. For example, in a heterogeneous system with several fast and powerful cores (high clock speed, multiple-issue out-of-order pipeline) and several simple and slow cores (low clock speed, single-issue in-order pipeline) threads running memory-bound codes should typically be mapped to slow cores, because the speedup they experience on fast cores relative to slow cores is disproportionately smaller than the additional power that the fast cores consume. Power and area efficiency of heterogeneous systems have been demonstrated in numerous studies [4,19,18,20,22].

Efficiency of heterogeneous systems is maximized when applications are matched to cores according to the architectural properties of both. This matching can be conveniently performed by an operating system thread scheduler. In this paper we describe a new heterogeneity-aware scheduling algorithm (*het-aware* from now on for brevity) that employs an original methodology compared to the ones proposed in the past. Our algorithm, called Het-Aware Signature-Supported (HASS) scheduler, is based on the idea of *architectural signatures*. An architectural signature is a compact summary of architectural properties of an application. It may contain information about the application's memory access patterns, instruction-level parallelism (ILP), sensitivity to variations in the clock speed and other data. The key property of this information is that it can be efficiently interpreted by the scheduler to determine how well a given application "matches" a given core.

The architectural signature framework evaluated in this work is designed for heterogeneous systems where cores differ in the clock speed since such a system can be emulated very efficiently using existing multicore processors. To capture the properties of the application that determine its sensitivity to variations in these architectural features, the architectural signatures are based on an application's memory intensity. Memory intensity is captured by the thread's cache miss rate, which as we found can be used to model the performance of the application on cores

* Corresponding author.
E-mail addresses: jcsaezal@fdi.ucm.es (J.C. Saez), danish@microsoft.com (D. Shelepov), fedorova@cs.sfu.ca (A. Fedorova), mpmatias@dacya.ucm.es (M. Prieto).

0743-7315/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2010.08.020

with different clock speeds. We implement two versions of HASS, static (HASS-S) and dynamic (HASS-D). With the static version the architectural signature is constructed offline. In this case we obtain the application's *reuse-distance profile* [5] (a summary of the memory reuse patterns) and use it to estimate the miss rate for caches of various sizes and associativities to cater to possible systems where the application may run. With the dynamic version, the miss rate is measured online, using hardware performance counters. Performance estimates generated using cache miss rates can be used to compute the relative benefit that an application (or thread) derives from running on different cores. By comparing the relative benefits for different threads, the scheduler decides which thread is the best candidate for a particular core type.

Our architectural signature framework can be generalized to systems where cores differ in other microarchitectural features, but exploring such architectures was outside the scope of this work. Instead, we chose to address the systems that could be effectively emulated on existing hardware (as opposed to on simulators), because this enabled us to perform a more extensive and thorough evaluation than what would have been possible on a simulator.[1]

Architectural signatures allow *estimating* relative performance of threads on the cores of different types. Another alternative is to measure this performance directly, by running each thread on each possible core type. One goal of our work was to compare HASS to an algorithm based on that approach, and we were aware of two previously proposed het-aware algorithms that relied on it. They determined the best matching of threads to cores via an online performance monitoring mechanism that required running each thread on each core type [4,19]. When we implemented one of these algorithms (IPC-Driven [4] proposed by Becchi et al.), we found that this performance monitoring methodology often leads to an incorrect estimate of the relative benefit that a thread derives from running on a particular core due to the dynamic nature of program phases. Further, the necessity to periodically re-measure threads' performance on different cores creates imbalanced demand for cores of different types if there are more cores of one type than of another. This causes load imbalance and degrades the performance.

We conclude that a monitoring methodology requiring performance estimates on *all* core types is difficult to use in practice. Although het-aware scheduling algorithms show a strong potential to maximize performance of heterogeneous multicore systems, how the algorithm is designed makes a big difference, since excessively heavy online monitoring can cause prohibitive overheads. Bringing to light the problems with seemingly simple and effective monitoring methodologies and proposing heterogeneity-aware algorithms that are not susceptible to similar deficiencies are the key contributions of our work.

In evaluating HASS, we were also interested in comparing it to a relatively simple het-aware algorithm that shares fast cores among the threads in a round-robin fashion. To that end, we have designed and implemented a Het-Aware Fair Share (HAFS) algorithm that ensures that the total time spent by each thread on a given core type is proportional to the number of cores of that type in the system. Our study reveals important overheads associated with a real implementation.

[1] Evaluating a real OS implementation on a heterogeneous processor where cores differ in pipeline microarchitecture would require us to use a full-system simulator (i.e., a simulator that boots a real operating system) that can also simulate heterogeneous hardware. Despite availability of such simulators (e.g., COTSon [2]), they still run in the kilohertz range when accurate simulation is required, so performing extensive evaluation with a large number of long-running workloads would be challenging.

We have implemented the algorithms (HASS-S, HASS-D, IPC-Driven and HAFS) in the OpenSolaris operating system and evaluated them on two real multicore platforms made heterogeneous via CPU frequency scaling. We found that HASS-static improves performance by as much as 12.5% for diverse workloads (i.e., workloads where applications significantly differ from each other in their architectural properties) relative to a heterogeneity-unaware scheduler. The IPC-Driven algorithm, in contrast, improved performance by at most 7% and often even caused performance degradation. HASS-dynamic delivers performance gains of up to 12%.

We also observed that HASS did not do as well on systems with shared caches. HASS's model for estimating performance on different core types did not account for shared caches, and so the mapping of threads to cores that it performed was not always optimal. Nevertheless, HASS improved performance even in these difficult conditions, outperforming both IPC-Driven and HAFS, and never performing worse than the default scheduler.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the methodology for constructing architectural signatures. Section 4 describes the design and implementation of the evaluated algorithms. Section 5 analyzes the performance results. Section 6 summarizes our findings.

2. Background and related work

A large body of work has advocated the potential benefits of asymmetric single-ISA processors over symmetric counterparts [19,18,1,11,22]. These benefits are concisely summarized in an article by Matt Gillespie [10] of Intel, where he lays out some of the background for why this shift towards asymmetric systems is likely to happen, describes the potential variations on the hardware architectures, and the distinct challenges and opportunities. OS scheduling is one of the main challenges, and this is the focus of our paper.

Single-ISA heterogeneous systems addressed in this work derive efficiency from *core specialization*. Specialization refers to matching each core type to the thread that is able to use this core's features most effectively. Most research efforts that sought to improve the efficiency of AMP systems have exploited primarily two kinds of core specializations: the first caters to microarchitectural diversity of the workload; the second caters to diversity in thread-level parallelism (TLP).

Efficiency specialization exploits the fact that complex and powerful cores are good for running CPU-intensive applications that effectively use those processors' advanced microarchitectural features, such as out-of-order super-scalar pipelines, advanced branch prediction facilities, and replicated functional units. At the same time, simple and slow cores deliver a better trade-off between energy consumption and performance for memory-intensive applications that spend a majority of their execution time fetching data from off-chip memory and stalling the processor.

In a similar vein, TLP specialization leverages knowledge of the fact that complex and powerful cores are good for running single-threaded sequential applications because these applications cannot accelerate their performance by spreading the computation across multiple simple cores. Similarly, sequential parts of parallel applications can be also effectively accelerated if they are mapped to fast cores opportunistically. Abundant simple cores, on the other hand, are good for running highly scalable parallel applications. Because of performance/power trade-offs between complex and simple cores, it turns out to be much more efficient to run a parallel application on a large number of simple cores than on a smaller number of complex cores that consume the same power or fit into the same area.

Specialization must be aided by a thread scheduler that decides which threads to run on fast cores and which on slow cores.

Two kinds of operating system schedulers emerged to address this challenge. The first type of schedulers, among which HASS is included, targeted efficiency specialization, by assigning the most CPU-intensive threads to fast cores [19,4]. The second type targeted TLP specialization, by assigning sequential applications and sequential phases of parallel applications to run on fast cores [25].

Two of the most well-known scheduling algorithms that employed efficiency specialization have been proposed by Becchi et al. [4] and Kumar et al. [19]. Both of them assume a system with two core types (“fast” and “slow”) and rely on continuous performance monitoring to determine optimal thread-to-core assignment. Becchi’s IPC-Driven algorithm periodically samples threads’ instructions per cycle (IPC) on cores of both types to determine the relative benefit for each thread from running on the faster core. Threads that have a higher fast-to-slow IPC ratio have a priority in running on the fast core, because they are able to achieve a relatively greater speedup there. Kumar’s method uses a similar technique, except that the sampling method is made more robust by using more than one sample per thread. In addition, Kumar proposed an algorithm that tries to determine a globally optimal assignment by sampling the performance of thread groups rather than making decisions based on the IPCs of individual threads.

Both of these approaches promise significantly better performance than heterogeneity-agnostic policies according to simulation-based evaluations, but they are both difficult to use in practice. Their reliance on sampling on *all* core types means that demand for different core types will be unequal. In particular, the smaller the ratio of fast cores to slow cores, the more demand there will be to run on any given fast core for sampling purposes. This creates a workload imbalance and interferes with threads that are “legitimately” running on faster cores. We found this to be a challenging problem in implementing the IPC-Driven algorithm. Since our algorithm relies on per-thread performance profiles, it avoids performance problems related to sampling on different core types and has a much simpler implementation.

Teodorescu and Torrellas [31] developed an algorithm for optimal assignment in the context of mildly heterogeneous platforms where core differences are caused by within-die process variation. Although performance profiling is still required, a lot of overhead is avoided by assuming that a thread’s IPC is the same on all core types. The approach works well when cores are very similar to each other, but unlike our approach, it is generally inapplicable to highly heterogeneous systems.

TLP specialization was employed in our earlier algorithm called Parallelism-Aware (PA) [25] — this is the only such operating system algorithm of which we are aware. PA used the number of runnable threads as the approximation for the amount of parallelism in the application. Prior to our work on the PA algorithm, Annavaram et al. proposed a application-level AMP algorithm that caters to the application’s TLP [1], but this algorithm required modifying an application. Neither of these algorithms catered to efficiency specialization, so unlike HASS they are unable to maximize the system throughput in cases where single-threaded applications are present in the workload.

In [26] we presented CAMP, the first scheduler of which we are aware that combines both TLP and efficiency specialization. The foundation of the CAMP scheduler is the *utility factor*, a compact metric agglutinating information on TLP and efficiency of applications. Catering to TLP and efficiency enables CAMP to improve the overall system performance for a wider variety of workloads. Both HASS-D and CAMP rely on similar techniques for discovering which threads utilize complex cores most efficiently, without requiring cross-core migrations. We found that HASS-D is able to outperform CAMP for workloads consisting of single-threaded applications only because it always guarantees that threads with the high performance ratios run on fast cores, whereas CAMP fair-shares fast cores among threads in a broader *high utility* class.

Beyond employing one of these two types of specialization, several other researchers have further advocated the benefits of asymmetric multicore platforms by exploiting another type of core specialization (not previously discussed in this paper). For example, Mogul et al. [22] described a scheduler that temporarily switches a thread to run on a slow core when the thread is executing a system call. By using system calls as a heuristic for thread assignment, this scheduler completely avoids any monitoring overhead (or the need to generate architectural signatures), but it only applies to workloads dominated by system calls.

Balakrishnan et al. [3] implemented a simple het-aware scheduler in Linux that ensures that fast cores never go idle before slow cores. Li et al. [20] proposed AMPS, a het-aware algorithm for Linux that makes sure that the load on each core is proportional to its power and that fast cores are never underutilized. While these schedulers mitigate the effects of performance asymmetry, they are not meant to improve the efficiency.

In this work, we use relative speedups to maximize *system-wide* performance. However, in the event that some processes have a higher priority than others or in scenarios where the system needs to deliver QoS guarantees, the relative speedup could be used as a *complementary* metric to provide better service for prioritized applications with a minimal effect on performance. For example, the scheduler might decide to run low-priority CPU-intensive threads on fast cores rather than high-priority memory-intensive ones, simply because “wasting” fast cores on running memory-intensive instruction streams may lead to significant overall degradation of system performance. Therefore, the relative speedup could be also used to make a trade-off between QoS and system-wide performance.

Due to practical reasons, our evaluation has been limited to asymmetric systems in which cores just differ in performance due to different clock speeds. In this scenario, algorithms for DVFS-based systems and for those specific single-ISA asymmetric systems address a very similar problem from different angles. While the former asks the question: “How to find the best frequency for a given application?”, the latter asks: “Given a fixed set of frequencies, where to map existing applications?” There is a key difference, however, between existing DVFS algorithms (such as [8], which is the closest one to ours) in that they rely on being able to *adjust frequency of individual cores* and observe performance of an application under different frequency settings in order to achieve their goals. In our setting, this would be equivalent to running each thread on each core type (and this is what an earlier proposed IPC-Driven algorithm does), but we found that this causes very large performance degradation and as a result simply does not work in our setting. For this reason, existing DVFS algorithms do not address the problem that we are solving. Other DVFS-based algorithms, such as [15], which do not require running applications at different DVFS settings, assume that performance of the application scales with CPU frequency (which we showed not to be the case) and so they do not tackle the same problem that we do either.

Finally, we want to highlight that the design space of heterogeneous multicore systems also includes other popular architectures that exhibit both performance and functional asymmetry, such as the IBM’s CELL BE or systems that combine homogeneous processors with an accelerator (a GPUs or even an FPGAs). Despite the continuous progress made by compiler technology over the last few years, mapping application onto these architectures is still extremely labor intensive. In most applications, developers must explicitly partition programs into different kernels, which are compiled to run on a particular core type. Then, either the application

developer or the underlying runtime system assign and schedules these kernels on the underlying system [24]. On single-ISA heterogeneous systems, the same binary can run on all cores types making software development much simpler. In fact, the workloads studied in this paper consist of single-threaded applications whose respective programs have been coded without any knowledge about the underlying architecture. Nevertheless, even if we abstract away from the fact that on heterogeneous-ISA multicores scheduling is done for the most part by the application runtime, the scheduling algorithms themselves are usually quite different. For instance, thread migrations, which are essential in our investigated algorithms, can be performed easily on single-ISA systems. However, these kinds of operations are not straightforward on heterogeneous-ISA systems, even if binaries themselves have different implementations of the same kernel for different core types. On the other hand, some recent studies demonstrate that their effectiveness can for the most part be leveraged by data-parallel regular codes [32] widely prevalent in the HPC domain, but is much more difficult to harness by other codes (such as servers). So while there is certainly a huge market for architectures like Cell and CPU–GPU systems, single-ISA asymmetric architectures just target a different domain, which may not necessarily be addressed by those other systems.

3. Architectural signatures

An architectural signature is a summary of the architectural properties of an application. HASS relies on the ability to estimate the relative performance of threads on different core types. To that end, the signature must enable it to predict a thread's performance based on the features of the core. As explained earlier, in this work we focus on systems where cores differ in clock frequency (as on the evaluation platforms used in this paper), a parameter expected to play a prominent role in the future heterogeneous systems.

To predict performance variations due to clock frequency, we must consider the application's degree of *memory intensity* [9]. An application with a high rate of memory accesses is likely to stall the core often, so the clock frequency will not have a significant effect on performance. Memory intensity can be approximated by a thread's miss rate [33]. The static version of HASS estimates the miss rate from an application's reuse-distance profile, which is obtained offline prior to executing the application. The dynamic version of HASS measures the miss rate online. Relative performance of threads on cores with different frequencies is then estimated online by the scheduler using a simple performance model.

The static version is more appropriate for environments like embedded systems, where application inputs are typically known *a priori*, and so the architectural signatures can be obtained for all typical executions of the workload. The dynamic version is more appropriate for dynamic and highly phased workloads, whose runtime properties are too variable for capturing offline.

The remainder of this section is structured as follows. In Sections 3.1 and 3.2, we explain how the signatures are constructed for the static and dynamic version of HASS, respectively. Then, in Section 3.3, we explain how the scheduler estimates the relative performance of applications on different core types. Section 3.4 is devoted to describing how architectural signatures could be extended for multithreaded applications. In Section 3.5, we analyze the implications of the presence of shared caches in our signature-based framework and suggest how it can be improved for these scenarios.

3.1. Static signatures

Static signatures rely on a reuse-distance profile. A reuse distance is defined as the number of intervening memory accesses between two consecutive accesses to the same memory location. A reuse-distance profile is the distribution of reuse distances. From this profile we can accurately estimate the application's last-level cache miss rates for any cache configuration [5,12,27,29] that can be encountered at runtime. *These estimated miss rates make up the contents of the static signature.*

Since reuse-distance profiles are mostly microarchitecture independent, our statically generated architectural signatures are microarchitecture independent as well.[2] Thanks to this property, out of all hardware platforms where it is possible to run the binary, we should be able to select any single one to construct a signature usable by all.

The signature should be available to the OS at scheduling time, so the ideal place to hold it is in the application binary itself. For evaluation covered in this paper we have not implemented the binary embedding scheme, and have instead hard-coded a limited set of signatures into the kernel.

To construct the signature, we need to obtain the reuse-distance profile, which is collected via offline profiling. Such a profiling can be done, for example, as part of the feedback-directed optimization phase of the application development, which can be set up with little or no involvement from the programmer. All that needs to be done is to execute a program once with the profiler (see below) that will generate the signature and embed it into the binary. The responsibility of the developer, then, is to make sure that the thread exhibits "typical" behavior during this signature run. If it is impossible to do so in one run, the developer can do several runs (for example with different inputs) and combine the results into one signature. In this work, we construct profiles using Pin, a binary instrumentation framework from Intel [21], along with a custom extension to Pin, MICA [13]. A more detailed account can be found in our previous work [27]. Once the profile is collected, we estimate (also offline) cache misses for a limited set of realistic last-level cache configurations (we do not account for different first- and mid-level cache configurations, because we found that accounting for these details did not significantly affect the signatures' accuracy). These estimations, collected in a matrix, comprise the architectural signature. We support 11 different last-level cache sizes (powers of two from 16K to 16M) and four set associativities (4, 8, 16 and 32), so the matrix has 44 values.

Shown in Table 1 is an example signature for the benchmark `art` from the SPEC CPU2000 suite. (Columns for sizes 16K to 128K are omitted, because these values were in this case exactly the same as that for 256K (427 misses).) Each integer in the matrix cell represents the expected number of misses per 4096 instructions (the number 4096 was selected to speed up calculations at scheduling time).

When signatures are generated offline, capturing the differences between various phases is not impossible [14], but certainly more difficult.[3] Using multiple signatures for an application, representing its different program phases, may lead to improving the dynamic thread-to-core assignments further, but at the expense of extra complexity, due to phase detection, and potentially higher runtime overheads, due to additional thread migrations. We found that, in practice, the average behavior captured by a single signature is good enough to effectively guide scheduling decisions with low runtime overhead.

[2] LLC miss rates eventually depend on both reuse-distance profiles and specific microarchitectural features such as pre-fetching mechanisms, cache replacement policies an so forth. Nevertheless, for our purpose, i.e. guiding scheduling decisions on asymmetric systems, these statically generated signatures are enough to model the memory behavior of the applications.

[3] Capturing the phased behavior of applications offline would require partitioning the program into phases that behave differently enough to have a different signature.

Table 1
The architectural signature for `art`.

Set. assoc.	Cache size						
	256K	512K	1M	2M	4M	8M	16M
4	427	412	325	157	42	6	1
8	427	418	332	131	21	1	0
16	427	424	337	107	11	0	0
32	427	426	336	86	5	0	0

3.2. Dynamic signatures

Dynamic signatures are constructed quite simply by measuring the application's last-level cache miss rate online. The advantage of dynamic signatures relative to static ones is that they adapt to the variations in the program input and to program phase changes. Phase-change adaptability of the HASS-D algorithm is described in detail in Section 4.3. The disadvantage of the dynamic signature scheme is that it is not as easily adaptable to systems where different cores have different-sized last-level caches; on these systems a scheduler would need to run each thread on each core type, which, as will be shown later, degrades the performance. Therefore, another alternative for constructing dynamic signatures is to use dynamically estimated reuse-distance profiles (as in the system RapidMRC [30]), and as in the static case use these reuse-distance profiles to estimate the miss rates. Since our experimental systems had uniform cache sizes across cores, we relied on the first method, where the miss rates are measured directly online, rather than obtained via dynamically estimated reuse-distance profiles.

3.3. Using signatures for scheduling

At runtime the architectural signature is used to estimate a thread's performance on each type of core present in the system. To accomplish this, we calculate a hypothetical completion time for some constant number of instructions. Two separate components of completion time are considered: execution time and stall time. Execution time is the amount of time it takes to execute the instructions assuming a constant number of cycles per instruction. To compute the execution time we assume a cost of 1.5 cycles per instruction and factor in the clock speed. These two parameters are machine dependent so their values must be appropriately chosen for the hardware platform in question.

We approximate the stall time by the number of cycles used for servicing the last-level cache misses. Although this is a coarse approximation, it gives reasonable accuracy, because memory access time dominates other stalls [17]. To estimate this, we need memory access latency (discoverable by the OS) and the miss rate that we obtain from the signature. Note that since we are assuming a constant memory latency, the presence of non-uniform memory access (NUMA) can reduce the accuracy of estimates. Although we did not have a chance to investigate this effect comprehensively, we observed that in our case the presence of NUMA on one of the experimental platforms did not prevent the algorithm from performing successfully.

The resultant sum of both time components gives us an abstract "completion time" metric. For actual scheduling, we focus on the ratio of completion times calculated for different types of cores, to which we refer to as the *Speedup Factor*. More precisely, if $P(t, C)$ denotes the performance for a given thread t on core C, and C_1, C_2 are two different core types such that C_1 is faster than C_2, then by convention we define the Speedup Factor as $\mathrm{SF}(t, C_1, C_2) = \frac{P(t,C_2)}{P(t,C_1)}$.

To summarize, we predict the performance of different threads on different cores based on threads' caching behavior and cores' frequency. This allows the OS to distinguish cores by their relative desirability for different threads. We have tested the accuracy of this method on cores that differ in frequency by using Dynamic Voltage and Frequency Scaling (DVFS) facilities available on most modern processors. DVFS allows the operating system to control the clock speed of the cores. Fig. 1 shows how well real speedup factors match predicted speedup factors for some of our test configurations (described in Section 5). As evident, the estimation method is successful in separating memory-intensive threads (which are less sensitive to changes in frequency and therefore concentrated toward lower left) from CPU-bound threads (upper right), but is less precise in characterizing memory intensity.

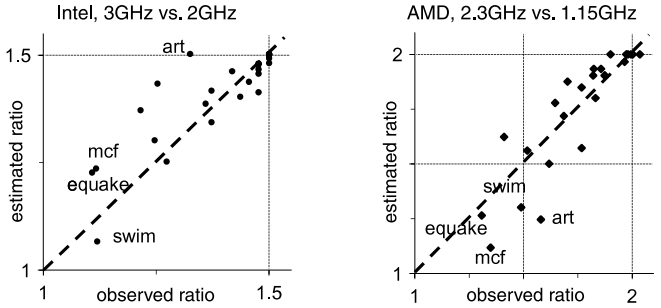


Fig. 1. Signature-estimated performance ratios vs. observed ratios. Some outliers are labeled. Perfectly accurate estimations would have all points on the diagonal line.

3.4. Multithreaded applications

Although the signature-based framework evaluated in this work was designed for single-threaded applications, there are no inherent barriers to extending it for multithreaded applications. In that case, the signature would be generated per thread—threads would be identified by the function that a thread executes. In scenarios where threads perform a different type of work (and thus have different architectural properties) despite executing the same function, an online method for signature generation would be preferred. The important point is that almost no changes would have to be done in the scheduler itself, because it already uses threads as schedulable entities associated with an architectural signature. This study, where for simplicity we use single-threaded applications in our experiments, evaluates the effectiveness on a het-aware scheduling algorithm assuming that per-thread signatures are known. Performing this evaluation was our key objective.

3.5. A reflection on shared caches

Wrapping up the discussion of architectural signatures, we would like to reflect on shared caches. On shared-cache architectures (including SMT), performance is affected not only by the frequency of the core and the properties of the application, but by cache access patterns of co-scheduled threads. Our existing method for estimating performance does not account for effects of shared caches. In our evaluation, this caused performance benefits to diminish when shared caches were present. Modeling shared cache effects is an orthogonal and well-studied problem. Existing models of miss rates in shared caches are based on input data very similar to reuse-distance profiles (used to construct our signatures) [7], and this presents a good opportunity to extend our signature-based model to account for cache sharing. We have devised a method that performs optimal co-scheduling of threads on shared-cache architectures based on the threads' reuse-distance profiles [33]. This algorithm is able to find the optimal thread schedule most of the time, performing within 1% of the oracular algorithm that always picks the optimal assignment. Integrating this cache-aware scheduling algorithm with het-aware algorithms is an interesting avenue for future work.

4. The algorithms

In this section we provide the description of the design of all the evaluated algorithms, and we also highlight the main challenges we had to face when creating real-world implementations of these.

Section 4.1 is devoted to introducing a few key abstractions that the implementations of all the investigated algorithms rely on. Sections 4.2–4.5 describe the HASS-S, HASS-D, IPC-Driven, and HAFS algorithms, respectively.

4.1. Core types and partitions

The first key abstraction used by all the algorithms is the *core type*. Each *core type* has a unique combination of features such as clock frequency or cache hierarchy. We assume that the configuration of the system is static and so the features of the cores do not change dynamically. A system must have at least two core types to be heterogeneous. Our implementations, however, assumed two core types only: "fast" and "slow". Apart from simplifying system design, earlier work demonstrated that having just two core types is sufficient to extract optimal benefits from heterogeneous systems [18]. Nevertheless, our performance models are general enough for HASS-S and HASS-D to be adapted to systems with a larger number of core types.

Since future many-core systems may contain a very large number of cores, load balancing and accounting may become costly. To manage a large number of cores in a scalable way, all the implemented algorithms rely on core *partitions*: sets of cores of the same type. Each core must belong to exactly one partition. Note, however, that there may be more than one partition including cores of an specific type (e.g. several partitions containing slow cores might be present in the system). Henceforth, we will use the term *fast partition* to refer to any partition that consists of fast cores. Conversely, the term *slow partition* will refer to a partition including slow cores.

4.2. The HASS-S algorithm

A key goal in the design of HASS was scalability, because future multicore processors may be built with hundreds or thousands of cores. Scalability mainly manifests in two aspects of the algorithm design: the lack of global locks, and the scheduling decision logic that relies only on local information. As we describe the algorithm, we will point out the particular features that ensure scalability.

As stated previously, HASS-S relies on core partitions. The scheduler maintains a counter of runnable threads for each partition (threads either currently running or ready to be run). This counter is the primary partition-wide contention point, as it has to be fully synchronized. In HASS-S, a partition is the widest locking scope during normal operation of the scheduler. This partition-based design enables us to manage a large number of cores in a scalable way.

When threads enter the system, the operating system estimates their performance on fast and slow cores according to the attributes of both core types (the method described in Section 3.3). The ratio between these estimates is the aforementioned speedup factor, which approximates the fast-to-slow speedup that a thread would experience when running on a fast core without ever being pre-empted in favor of other threads.

To assign a thread to a specific partition, the scheduler goes through the list of all partitions and estimates that thread's performance in each partition using the speedup factor and the current number of runnable threads per core in the partition. The scheduler assumes that the CPU time will be shared equally among all threads within the partition. After that, the scheduler selects the partition with the highest expected performance and assigns the thread there. This process is called *regular assignment*. Note that regular assignment has linear complexity with respect to the number of partitions, so there should be a balance between the number of partitions and the number of cores in each partition. An assignment of threads to partitions is not only done initially, but is repeated every time a thread accumulates a certain amount of CPU time on its current partition, in case the current partition becomes non-optimal, i.e., when the number of threads in a partition changes. This repeated assignment is called a *refresh*. By having the refresh period tied to CPU time rather than wall clock time, we avoid increasing the absolute number of refreshes as the load factor grows. If there is no change in the system load, the refresh assignment can be skipped.

Algorithm 1 An algorithm for *regular assignment* and *optimistic rebinding* in HASS-S

Definitions: F is the set of threads assigned to fast cores, S is the set of threads assigned to slow cores. FP and SP are the sets of fast and slow partitions, respectively. t is a runnable thread.
Require: $(t \in F \bigcup S)$
Ensure: t is assigned to a partition that improves its performance.
  success ← **false**
  {First of all, try a *regular assignment*}
  p_{cur} ← partition where t is currently assigned to
  if $((t \in F)$ **and** $(nthreads(p_{cur}) \leq ncores(p_{cur}))$ **then**
    {The expected performance of t is already optimal}
    success ← **true**
  else {Find a better target partition to improve performance}
    p_{target} ← find partition $p_s \in (FP \bigcup SP)$ with maximum expected performance for t
    if *expected_performance*(t, p_{target}) > *expected_performance*(t, p_{cur}) **then**
      move t to p_{target}
      success ← **true**
    end if
  end if
  {Try *optimistic rebinding* if no better partition was found so far}
  if not *success* **then**
    $t_{partner}$ ← find thread $t_{partner}$ such that the swap of t with $t_{partner}$ improves system performance
    if $t_{partner}$ was found **then**
      swap t and $t_{partner}$
    end if
  end if

Algorithm 1 shows the pseudo-code for the regular assignment and refresh. *Optimistic rebinding*, also shown in this listing is discussed in the following paragraphs. For reasons of conciseness we do not provide the pseudo-code for *expected_performance*. The procedure for estimating the performance on different core types is described in Section 3.3.

Load balancing and core assignment within partitions can be done according to regular OS policies, which can also be tailored to emphasize scalability (we do not discuss these techniques). Between partitions, however, there is no direct load balancing. Instead threads will converge to a balanced load distribution, with more powerful partitions potentially receiving higher loads. This also allows a situation where a thread is waiting in a queue while there is an idle core somewhere in the system. To prevent such occurrences, it is forbidden to move a thread to fully loaded or overloaded partitions when some partitions are less than fully loaded (underloaded).

The greedy thread assignment approach described so far has a potential problem where threads may become locked in a sub-optimal assignment and further optimization can only be accomplished by cooperative action between two threads (swapping) rather than by a greedy decision w.r.t. one thread. There is a mechanism to perform such a swapping, and it is called *optimistic rebinding*. A scheduler may decide to use optimistic rebinding instead of

Algorithm 2 Event-driven migrations in HASS-D

Definitions: F is the set of threads assigned to fast cores, S is the set of threads assigned to slow cores, t is a runnable thread whose SF has changed.
Require: $(F \neq \emptyset) \wedge (S \neq \emptyset) \wedge (t \in F \bigcup S)$
$\wedge (t \in F \Rightarrow (\forall u \in F - \{t\}, \forall v \in S : SF(u) \geq SF(v)))$
$\wedge (t \in S \Rightarrow (\forall u \in F, \forall v \in S - \{t\} : SF(u) \geq SF(v)))$
Ensure: $(\forall u \in F, \forall v \in S : SF(u) \geq SF(v))$
if $t \in F$ **then**
  $t_{sc} \leftarrow$ find thread t_{sc} in S with max SF
  if $SF(t) < SF(t_{sc})$ **then**
    {swap threads t and t_{sc}}
    $\langle F, S\rangle \leftarrow \langle F - \{t\} + \{t_{sc}\}, S - \{t_{sc}\} + \{t\}\rangle$
  end if
else $\{t \in S\}$
  $t_{fc} \leftarrow$ find thread t_{fc} in F with min SF
  if $SF(t) > SF(t_{fc})$ **then**
    {swap threads t and t_{fc}}
    $\langle F, S\rangle \leftarrow \langle F - \{t_{fc}\} + \{t\}, S - \{t\} + \{t_{fc}\}\rangle$
  end if
end if

the regular assignment during a refresh, if it fails to find a good target partition for a thread. The scheduler then has to find a partner for the thread in some partition with "good" potential performance and swap the target thread with the partner. The scheduler only triggers the swap if it confirms that the swap will actually increase the performance of the target thread as well as the overall system performance. This is done by comparing the speedup factors of the target thread and of the potential partner. The search for a partner can be slow when the target partition has many threads or when there are a lot of partitions. Therefore, our algorithm forgoes exhaustive search and instead uses randomized search with a limited amount of probing.

The partitioning scheme allows the scheduler to avoid global synchronization during scheduling. Instead, threads can lock one partition at a time when doing a refresh (for reading the runnable threads counter), migrating between partitions or entering/leaving runnable states (for updating the runnable counter). Using read/write locks can further decrease the pressure on this contention point.

4.3. The HASS-D algorithm

HASS-D, the dynamic version of HASS-S, estimates the speedup factor online, by periodically sampling threads' last-level-cache (LLC) miss rates and using them as the input to the performance algorithm described in Section 3.3. The fact that the speedup factor in HASS-D is not known when the thread first arrives, and that it can change dynamically throughout the thread's lifetime, dictates different algorithms for assignment of threads to cores than those used in HASS-S. For example, HASS-D cannot perform the same regular assignment as HASS-S, because when the thread arrives its speedup factor is not yet known. Likewise, subsequent reassignment of threads in HASS-D is driven by dynamic changes in the speedup factor rather than the changes in the load. The main focus of this section, therefore, is to describe the algorithm used in HASS-D to assign threads to cores.

When a new thread enters the system, HASS-D assigns it a *default* speedup factor,[4] since no information about its actual speedup factor is available. The initial mapping of newly created threads is performed such that fast partitions are populated before slow partitions and the load balance across the cores is preserved. As soon as the thread begins to run, HASS-D begins to monitor its last-level cache miss rate (on whatever core it was assigned to run), and then uses that miss rate to estimate its speedup factor as described in Section 3.3.

As threads run, two things happen: speedup factors for newly arrived threads become known, or speedup factors of old threads (as they enter into different phases of execution) change. The scheduler must map threads to cores according to their speedup factors, and to that end it follows the so-called *event-driven migration* procedure shown in Algorithm 2 and described below.

Event-driven migrations ensure that the system adheres to the two rules: (1) All threads in fast partitions have a higher SF than the thread with maximum SF running in a slow partition (2) load balance must be preserved.

In order to enforce Rule 1, the scheduler must check that the thread with minimum SF on fast cores (t_{fc}) has a higher SF than the thread with highest SF on slow cores (t_{sc}). This rule may be broken either when a change in the SF of a thread takes place or in the event that a thread transitions between a runnable and a non-runnable state. In the former scenario, the scheduler enforces the rule by swapping t_{fc} and t_{sc} when needed. In the latter case, the migration of one the aforementioned threads when necessary is enough to guarantee that the two rules of HASS-D hold true.

The scheduler maintains per-partition lists of runnable threads sorted by SF to simplify the selection of the optimal thread(s) to migrate (or swap): t_{fc} and t_{sc}. For efficiency reasons, fast partitions' lists are kept sorted in an ascending order by SF, while a descending order by SF is preferred for thread lists in slow partitions. As a result, finding the optimal candidate in either case has linear complexity with respect to the number of partitions.

HASS-D measures LLC miss rates for each thread continuously using performance counters, and the values are sampled every 20 timer ticks (roughly 200 ms on our experimental system). We keep a running average of the values observed at different periods and we discard the first values collected immediately after the thread starts or after it is migrated to another core in order to correct for cold-start effects causing the miss rate to spike intermittently after migration.

We also use a phase-detection mechanism that seeks to capture coarse-grained phases rather than fine-grained ones, in an attempt to reduce the number of unnecessary migrations. Updating SF estimations during abrupt phase changes may cause frequent expensive migrations, which may end up being unnecessary if the phase change is not lasting. Instead, SF estimations are updated exclusively once a thread enters a phase exhibiting stable behavior.

To detect stable phases, we use a light-weight mechanism based on a *phase transition threshold* parameter (12% in our experimental platform). When the moving average is recorded, it is compared with the previous average measured over the previous interval. If the two differ by more than the transition threshold, a phase transition is indicated. Two or more sampling intervals containing no indicated phase transition signal a stable phase.

4.4. The IPC-Driven algorithm

To compare HASS to an existing het-aware algorithm, we chose to implement the IPC-Driven algorithm proposed previously by Becchi and Crowley [4], an algorithm that combined good results, applicability to general purpose systems and specification completeness. In the original work [4] the IPC-Driven scheduler was simulated. We created the first real implementation of the IPC-Driven algorithm.

The IPC-Driven algorithm assumes two types of cores: ("fast") and ("slow"). The assignment is done based on IPC ratios, which

[4] For this default value, we opted to choose the lowest speedup factor attainable in an attempt to avoid that threads with a relatively low estimated speedup factor and legitimately assigned to fast cores are evicted from fast partitions when new threads enter the system.

Algorithm 3 IPC-Driven's thread swapping mechanism

Definitions: F and S are the sets of threads assigned to fast and slow cores, respectively. F_p and S_p are the sets of *pinned* threads on fast and slow cores, respectively. t is a runnable thread whose *IPC-ratio* has changed or a thread that has just entered the *pinned* state.

Require: $(F_p \subseteq F) \wedge (S_p \subseteq S) \wedge (F_p \neq \emptyset) \wedge (S_p \neq \emptyset) \wedge (t \in F_p \bigcup S_p) \wedge (t \in F_p \Rightarrow (\forall u \in F_p - \{t\}, \forall v \in S_p : \textit{IPC-ratio}(u) \geq \textit{IPC-ratio}(v))) \wedge (t \in S_p \Rightarrow (\forall u \in F_p, \forall v \in S_p - \{t\} : \textit{IPC-ratio}(u) \geq \textit{IPC-ratio}(v)))$

Ensure: $(\forall u \in F_p, \forall v \in S_p : \textit{IPC-ratio}(u) \geq \textit{IPC-ratio}(v))$

```
if t ∈ F_p then
    t_sc ← find thread t_sc in S_p with max IPC-ratio
    if IPC-ratio(t) < IPC-ratio(t_sc) then
        {swap threads t and t_sc}
        ⟨F, S, F_p, S_p⟩ ← ⟨F − {t} + {t_sc}, S − {t_sc} + {t}, F_p − {t} + {t_sc}, S_p − {t_sc} + {t}⟩
    end if
else {t ∈ S_p}
    t_fc ← find thread t_fc in F_p with min IPC-ratio
    if IPC-ratio(t) > IPC-ratio(t_fc) then
        {swap threads t and t_fc}
        ⟨F, S, F_p, S_p⟩ ← ⟨F − {t_fc} + {t}, S − {t} + {t_fc}, F_p − {t_fc} + {t}, S_p − {t} + {t_fc}⟩
    end if
end if
```

determine the relative benefit of running a thread on a particular core type. IPC ratios in the IPC-drive algorithm are synonymous with the Speedup Factor in the HASS algorithms, but in this section we will use the term IPC ratio to follow the original definition of the authors.

The key idea behind the IPC-Driven algorithm is very similar to HASS-D: IPC-Driven like HASS-D also relies on event-driven migrations, and the procedures for event-driven migrations in the two algorithms are very similar. The key difference is that IPC-Driven *requires running each thread on both core types* to estimate its IPC ratio, while HASS-D only needs to run a thread on one (any) core type to estimate its speedup factor. As we will see later, the need to run a thread on both core types creates load imbalance, which causes performance degradation, and often results in inaccurate estimates of the IPC ratios. We will provide more discussion and explanation of this phenomenon in the experimental section. In the rest of this section, we complete the description of the IPC-Driven algorithm.

A thread with a high ratio between the IPC on the fast core and the IPC on the slow core is expected to benefit from the fast core. The scheduler periodically samples threads' IPC on both core types and examines the IPC ratios of threads running on fast and slow cores. If the smallest IPC ratio among the threads running on the fast core is smaller than the highest IPC ratio among the threads running on the slow cores, the threads with the corresponding ratios are swapped. This part of the IPC-Driven algorithm is very similar to the event-driven migration algorithm in HASS-D. It is shown in Algorithm 3.

As in HASS, cores are organized into partitions according to their types. The original algorithm assumed only two types of cores, but our implementation is a generalization for n different types.

Just like HASS-D, IPC-Driven periodically re-estimates the IPC ratio when a thread is deemed to have entered a new phase. New program phases are detected by the changes in the program's IPC that exceed a certain `ipc_threshold`. Whenever a program enters a new IPC phase, the IPC ratios relative to the thread's most recent "home" core type are re-measured. However unlike

Algorithm 4 HAFS's thread swapping mechanism

Definitions: F and S are the sets of threads assigned to fast and slow cores, respectively. F_x and S_x are the sets of *expired* threads on fast and slow cores, respectively. t is a runnable thread that has just entered the *expired* state. *BC* stands for *balance counter*.

Require: $(F_x \subseteq F) \wedge (S_x \subseteq S) \wedge (F_x \neq \emptyset) \wedge (S_x \neq \emptyset) \wedge (t \in F_x \bigcup S_x) \wedge (((t \in F_x \Rightarrow (F_x = \{t\})) \wedge (t \in S_x \Rightarrow (S_x = \{t\})))$

Ensure: $((F_x = \emptyset) \vee (S_x = \emptyset))$ $\wedge$ t has been swapped with t_x, the *oldest* expired thread in the opposite core type

```
if t ∈ F_x then
    t_x ← find thread t_x in S_x with max BC
    {swap threads t and t_x}
    ⟨F, S, F_x, S_x⟩ ← ⟨F − {t} + {t_x}, S − {t_x} + {t}, F_x − {t}, S_x − {t_x}⟩
else {t ∈ S_x}
    t_x ← find thread t_x in F_x with min BC
    {swap threads t and t_x}
    ⟨F, S, F_x, S_x⟩ ← ⟨F − {t_x} + {t}, S − {t} + {t_x}, F_x − {t_x}, S_x − {t}⟩
end if
```

HASS-D, re-estimating the IPC ratio requires migrating a thread to another core (and as we show in the experimental section, this is the main cause for performance differences between HASS-D and IPC-Driven). This is done via forced migrations where a thread is switched to run in a partition of the opposite core type to its most recent one for a period of time called `refresh_period`. Additionally, a forced migration is triggered for threads that have just entered the system (after a warm-up period) in order to initialize the IPC ratio. Note also that those newly created threads are assigned in the first place to the partition with the lowest number of runnable threads per core.

In order to limit the number of forced migrations and to allow the system to stabilize between two consecutive thread swaps, a thread must run on a new core for a period of time equal to a `swap_inactivity` period before another forced migration is allowed. A thread that has been assigned to a particular core and is eligible for swapping is said to be in a *pinned state*. A thread whose IPC ratio is in the process of being updated is said to be *refreshing*. The performance of the IPC-Driven algorithm is sensitive to the settings of the aforementioned parameters (`refresh_period`, `swap_inactivity period`, etc.), and so we have carried out an exhaustive evaluation of the parameter space and picked the ones that yielded the best overall performance. `Refresh_period` was set to 30 ms, `ipc_threshold` to 10%, `swap_inactivity` period to 1.5 s and `warm_up` period to 200 ms.

4.5. The HAFS algorithm

HAFS is an implementation of a heterogeneity-aware round-robin scheduling policy. The goal of this algorithm is to ensure that fast cores are shared equally among threads. The current implementation supports systems with two types of cores: fast and slow.

On the high level, the HAFS algorithm works as follows. It assigns threads to slow and fast partitions so as to preserve load balance across the cores, and then periodically migrates the threads among fast and slow partitions to ensure that fast cores are shared equally among the threads. HAFS relies on two mechanisms: *Inter-partition swaps* and *balance counters*. Inter-partition swaps is a mechanism for cross-partition migrations that ensures that migrations do not disturb load balance. Balance counters is a mechanism that ensures that fast cores are shared equally among the threads. We first explain how inter-partition swaps work, and then describe the balance counters.

Suppose that a thread must be migrated from one partition to another. Simply enqueueing this thread in a runqueue of a core in

the target partition could cause load imbalance if there is a large number of migrations going one way. To prevent load imbalance, the scheduler never migrates a thread from one partition to another unless there is a *candidate* thread that needs to be migrated in the opposite direction. *Swapping* threads among partitions, rather than performing one-way migrations, is guaranteed to preserve load balance.

A thread may be migrated without a swap if there are idle cores in a fast partition, since one goal of HAFS is to keep the fast cores busy. Furthermore, if a fast partition is overloaded and there are idle cores in a slow partition, the algorithm will also migrate a thread into that slow partition without requiring a swap. This preserves load balance.

Balance counters are used to achieve fair sharing. The scheduler associates with each thread a "balance" counter to track the deviation between the number of cycles a thread has been running in slow partitions compared to fast partitions. When that counter reaches a certain threshold,[5] the scheduler sets the thread as "expired" and marks it as a candidate for migration onto the opposite core type. When a matching candidate thread wishing to migrate in the other direction appears, the two threads are swapped. Candidates are swapped in the FIFO order, so no thread gets "stuck" in a slow partition longer than any other thread. The swapping mechanism, which ensures that fast cores are shared equally, is illustrated in Algorithm 4.

If there are no matching candidates for swapping, an "expired" thread will keep running in the old partition. For example, if the number of threads is smaller than or equal to the number of fast cores, all the threads will keep running on fast cores without ever being migrated to slow cores.

Inter-partition swaps and balance counters ensure that fast cores are shared equally among threads and that the load balance is preserved at the same time. Another important property of HAFS is that it does not require global communication across all cores when making scheduling decisions. This property of the algorithm suggests that it has good scalability properties, which will be especially relevant on future many-core systems with potentially hundreds of cores.

5. Results and discussion

This section is divided into four parts. In Section 5.1 we describe our experimental platform in detail and introduce the heterogeneous configurations used for the evaluation. Then, in Section 5.2, we introduce the benchmarks and workloads employed. Section 5.3 is devoted to describing our experimental methodology. Finally, in Section 5.4, we analyze the performance results of all the investigated schedulers and report our main findings.

5.1. Experimental platform

We chose OpenSolaris as the platform to implement the scheduling algorithms due to its powerful profiling framework DTrace [6]. We used two machines for our experiments. One was an Intel Xeon X5365 server with four dual-core chips. A pair of cores on a chip shared a 4 MB L2 cache. Another was an AMD Opteron 8356 with four quad-core chips. Cores on the same chip shared a 2 MB L3 victim cache; per-core 512 KB L2 caches were private.

We configured our test systems to be heterogeneous by setting the cores to run at different frequencies using DVFS. Since we wanted to get the most heterogeneous setting out of our platforms, the frequency of fast and slow cores was set to the minimum and maximum frequency levels, respectively. More specifically, on our heterogeneous Intel-based platform, fast cores operate at 3.0 GHz, while slow cores run at 2.0 GHz. Conversely, on the AMD platform, fast and slow cores were set to run at 2.3 GHz and 1.15 GHz, respectively.

In our experiments we used three heterogeneous configurations: (1) AMD-2,2, — two fast cores, two slow cores, each on its own chip and exclusive L3$ per core; (2) Intel-2,2 — two fast cores, two slow cores, each on its own chip and exclusive L2$ per core; and (3) AMD-12,4 — four fast cores and twelve slow cores. In some configurations we used fewer cores than available (AMD-2,2, Intel-2,2) in order to avoid any performance effects due to cache sharing (to that end, we had to use at most one core per chip). Conversely, the AMD-12,4 configuration, where all of the cores are used, is subject to cache interference effects.

5.2. Benchmarks

Our workloads consist of several single-threaded applications drawn from the SPEC CPU2000 suite that expose a wide range of behaviors concerning the efficiency of pipeline utilization. Although reuse-distance profiles for HASS-S had to be collected on Linux (Pin does not run on OpenSolaris), we ensured that the benchmark binaries compiled for Linux-x86 were sufficiently similar to the binaries compiled for Solaris-x86 by using the same compiler version and flags.

We opted not to include multithreaded applications in our workloads because our investigated algorithms only seek to deliver efficiency specialization rather than TLP (thread-level parallelism) specialization. Other algorithms whose main goal is to deliver TLP specialization only, such as "PA" [25], have proved beneficial for workloads containing both parallel and sequential applications but are unable to deliver performance gains for workloads consisting of single-threaded applications only [26].

In our experimental evaluation we used two workload sets: #1 and #2. Benchmarks included in each set are shown in Table 2(a) and (b). Workload set #1 includes eight workloads with four applications each. Workload set #2 consists of eight application sets, each including up to sixteen different benchmark programs.

Set #1 includes three categories of workloads. The first category is highly heterogeneous (HH), and consists of a pair of highly CPU-bound benchmarks and a pair of memory-bound benchmarks. Workloads in this category (HH1, HH2 and HH3) show the most diversity in terms of applications' architectural properties, and so they will have the most performance improvements from het-aware algorithms. In each HH workload, the first two benchmarks are CPU intensive with virtually any cache size, and the second pair is memory intensive. These workloads, especially when running on the AMD-2,2 and Intel-2,2 configurations, enable us to assess the effectiveness of each investigated scheduler under the most favorable (most heterogeneous) conditions, since in those configurations an effective scheduling will result in mapping the two CPU-intensive instruction streams on the two available fast cores. The second category of workloads included in set #1 is moderately heterogeneous (MH). These workloads include benchmarks representing the whole spectrum of memory intensity, with less extreme differences between the benchmarks. In general, however, the first two benchmarks in each workload are less memory intensive than the last two on the majority of our configurations. These MH workloads are expected to benefit less from het-aware scheduling than HH workloads. Finally, in the lightly heterogeneous category (LH) we have the workload consisting of the four copies of the same application (`wupwise`). We report the data on only one workload in this category, because we did not observe particularly interesting effects for homogeneous workloads.

[5] There is actually a positive and a negative threshold. The former controls the number of cycles a thread should spent in slow partitions without being migrated, whereas the latter performs the same control in fast partitions.

Table 2
Multi-application workloads (a) Set #1. (b) Set #2.

(a) Workload set #1 Categories	Benchmarks
HH1	sixtrack, crafty, mcf, equake
HH2	gzip, sixtrack, mcf, swim
HH3	mesa, perlbmk, equake, swim
LH1	wupwise, wupwise, wupwise, wupwise
MH1	vortex, twolf, fma3d, art
MH2	gap, parser, applu, vpr
MH3	apsi, ammp, lucas, mgrid
MH4	bzip2, gcc, wupwise, art
(b) Workload set #2 Categories	**Benchmarks**
W1	sixtrack, crafty, eon, gzip, twolf, mesa, parser, bzip2, gap, vortex, ammp, mgrid, gcc, apsi, vpr, wupwise
W2	sixtrack, crafty, twolf, perlbmk, mesa, parser, bzip2, gap, vortex, ammp, mgrid, apsi, vpr, wupwise, fma3d, art
W3	gzip, bzip2, parser, gap, vortex, ammp, mgrid, gcc, apsi, vpr, wupwise, fma3d, art, applu, swim, lucas
W4	eon, gzip, perlbmk, gap, mgrid, gcc, apsi, vpr, wupwise, fma3d, art, applu, swim, lucas, mcf, equake
W5	gzip, sixtrack, crafty, perlbmk, gap, mgrid, apsi, vpr, wupwise, fma3d, art, applu, swim, lucas, mcf, equake
W6	parser, gap, vortex, ammp, mgrid, gcc, apsi, vpr, wupwise, fma3d, art, applu, swim, lucas, mcf, equake
W7	sixtrack, crafty, twolf, mesa, gap, mgrid, apsi, vpr, wupwise, fma3d, art, applu, swim, lucas, mcf, equake
W8	sixtrack(x2), crafty(x2), art(x2), applu(x2), swim(x2), lucas(x2), mcf(x2), equake(x2)

In all experiments, the total number of applications was set to match the number of cores in the heterogeneous platform, since this is how runtime systems typically configure the number of threads when only CPU-bound applications are used [23]. Four copies of each benchmark from set #1's "base" workloads were used on the AMD-12,4 configuration to make use of the sixteen cores available, while only one copy is needed on the Intel-2,2 and AMD-2,2 configurations to keep all cores busy.

For the AMD-12,4 configuration, we also report the performance of workloads in set #2, which are larger and more diverse than those in set #1. These application sets, which include benchmark programs with a wide range of speedup factors, are shown in Table 2(b) and appear sorted in ascending order by memory intensity. This way, the workloads range from W1 (the least memory intensive) which is made up of both CPU-intensive and mildly memory-intensive applications, to W8 (the most memory-intensive workload), which contains up to twelve highly memory-intensive programs.

5.3. Experimental methodology and metrics

For a given test we launch a predetermined number of benchmarks, and as individual copies terminate, they are immediately restarted. Thus we keep the workload constant and measure the average completion time of every benchmark. Our goal is to minimize the mean of normalized completion times across all benchmarks. Each benchmark runs at least three times, so there are at least three completion time values.

Our original goal was to compare the completion times achieved with HASS to the completion times achieved with the native OpenSolaris scheduler, but we found that completion times under the native scheduler were highly variable (standard deviation was as high as 23%) and thus not suitable for comparison. This is due to the fact that the native scheduler is not heterogeneity aware and thus migrates threads between different core types at infrequent and arbitrary intervals. Therefore, the fraction of time that a thread spends on a particular core type varies significantly from one run to another. Achieving a low standard deviation is not possible in these conditions.

Instead we compare completion times of the algorithms to a composite metric, to which we refer to as the *default metric*. To compute a completion time for a benchmark using the default metric we run the benchmark bound to a specific type of core (e.g., the "fast" type) while the rest of the benchmarks in the workload are running on other cores. Then we repeat the same measurement while the benchmark is bound to the core of the other type (e.g., "slow"). We then average the completion times on fast and slow cores, and the resulting value is used to approximate the default completion time. This metric gives us the expected completion time of a benchmark over a large number of trials if the benchmark were randomly bound to a core at the start of its execution and kept running on that core until completion. This is a good approximation of how the default scheduler operates, because it tries to minimize migration of threads from one core to another in order to maintain cache affinity.

The default metric could be too pessimistic on systems with sustained loads, where new threads are constantly arriving as old threads finish. As threads running on faster cores retire more often, faster cores will be available for assignment more often. To compensate, we also compare our algorithms to HAFS, which keeps the fast cores busy. It is important to understand though, that the performance achieved with HAFS will be better than with a heterogeneity-agnostic default scheduler, because HAFS is specifically designed to keep the fast cores busy. In summary, while we do not use real completion times for the native scheduler, we understand that they are no worse than the default metric, and are somewhat worse than those obtained with HAFS.

For performance comparison we report completion times normalized to the default metric for each benchmark in workload set #1 as well as the geometric mean for all its benchmarks. Due to space constraints, however, we only show the geometric mean of normalized completion times for all benchmarks in workload set #2. Apart from performing experiments with workloads where the number of running benchmarks matches the number of cores, we have also studied scenarios where more than one thread per core was used, and the results (omitted from this paper but reported in our previous work [28]) were qualitatively similar.

In addition to the results obtained with the various algorithms we also show the results obtained with the best static assignment. A static assignment is decided at the beginning of execution and never changed thereafter. The best static assignment is obtained by testing all possible static assignments and picking the one with the best performance. The best static assignment is the theoretical upper bound for the performance that can be achieved with our implementation of HASS-S.

5.4. Performance analysis

This section is organized as follows: In Section 5.4.1 we evaluate the performance of HASS-S. In Section 5.4.2 we explain

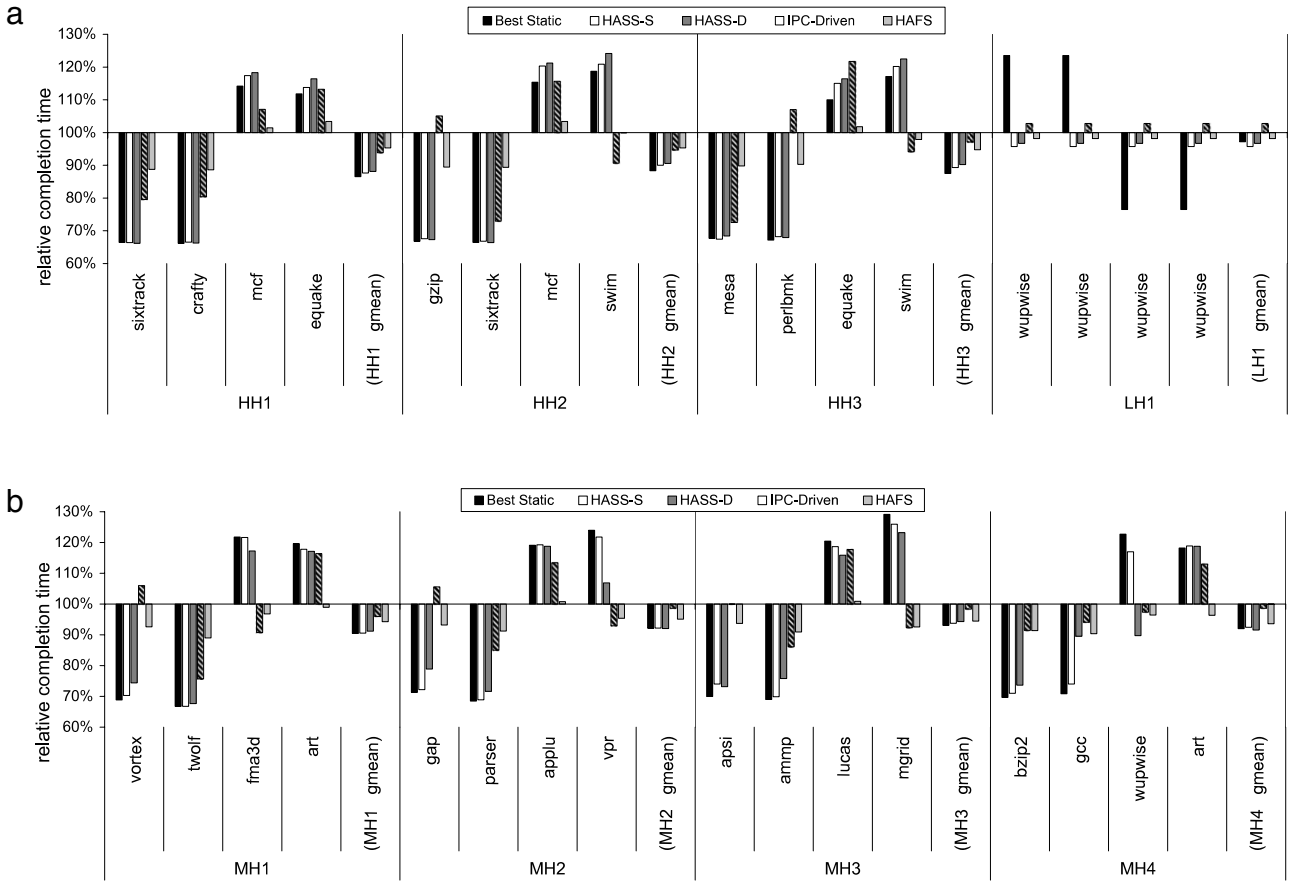


Fig. 2. Completion times relative to the default metric for workload set #1 on the AMD-2,2 configuration. Bars above 100% represent slow down, and below 100% represent speedup. (a) HH and LH workloads, (b) MH workloads.

the performance results of HASS-D. Sections 5.4.3 and 5.4.4 are devoted to analyzing IPC-Driven and HAFS. In Section 5.4.5, we briefly discuss overall performance numbers for the different schedulers across different heterogeneous configurations. Finally, in Section 5.4.6 we summarize our main findings.

5.4.1. Evaluation of HASS-S

First we analyze the behavior of HASS-S on the AMD-2,2 configuration. This is the configuration where we expect to see the best results, because (1) this configuration is more "heterogeneous" than the Intel-2,2 configuration, since the difference in the speeds of fast and slow cores is greater than on Intel-2,2, and (2) this configuration is not subject to cache sharing (unlike the AMD-12,4 system), which our algorithms do not handle. Fig. 2a and b show the completion times for the different workloads relative to the default metric (lower numbers are better). The types of workloads are shown on the bottom of the chart. The completion times are shown for each application individually as well as for the entire workload as the geometric mean.

As can be expected, HASS-S performed especially well with the HH workloads, where the mean speedup was as much as 12.5% for the {`sixtrack`, `crafty`, `mcf`, `equake`} workload and reached 10% and 11% for the other HH workloads. In all these cases HASS-S achieved its theoretical upper bound. We traced the execution of the benchmarks with DTrace and confirmed that HASS-S actually chooses the mapping of threads to cores that corresponds to the best static assignment.

As expected, the performance improvements were more modest for the MH workloads: 10%, 8%, 6% and 8% for each of the four workloads and 8% on average on the AMD-2,2 system. The reason is that there are smaller differences in the CPU speed sensitivities among different applications, so the optimization opportunities are smaller. Despite the similarity in the applications' signatures in the MH workload, HASS-S was still able to pick the right candidates for running on the fast cores, matching again the best static assignment.

To understand the source of performance improvements from het-aware scheduling, we examine the relative completion times for individual benchmarks within the workload. For HH and MH workloads we see that the first two applications in the workload (recall that these are CPU-bound applications) usually speed up under het-aware scheduling (i.e., they experience lower completion times), while the second two applications (the memory-bound type) slow down. Since the speedup experienced by the CPU-bound applications is greater than the slow down experienced by the memory-bound applications, the workload as a whole experiences an improvement in performance. This points to the inherently "discriminative" nature of heterogeneity-aware scheduling, which may make it inappropriate to situations where the goal is to optimize the performance of individual applications. But when the goal is to optimize the workload as a whole, the het-aware policy does its job.

Examining completion times for individual applications offers another way to check whether HASS-S was able to pick the right candidates to run on the fast cores. Ideally, we want to see the same applications experiencing the speedup with HASS-S as with the best static assignment. For HH and MH workloads we see that this is always the case. HASS-S is able to determine which two applications are CPU intensive and assign them to run on fast cores, matching the best static assignment.

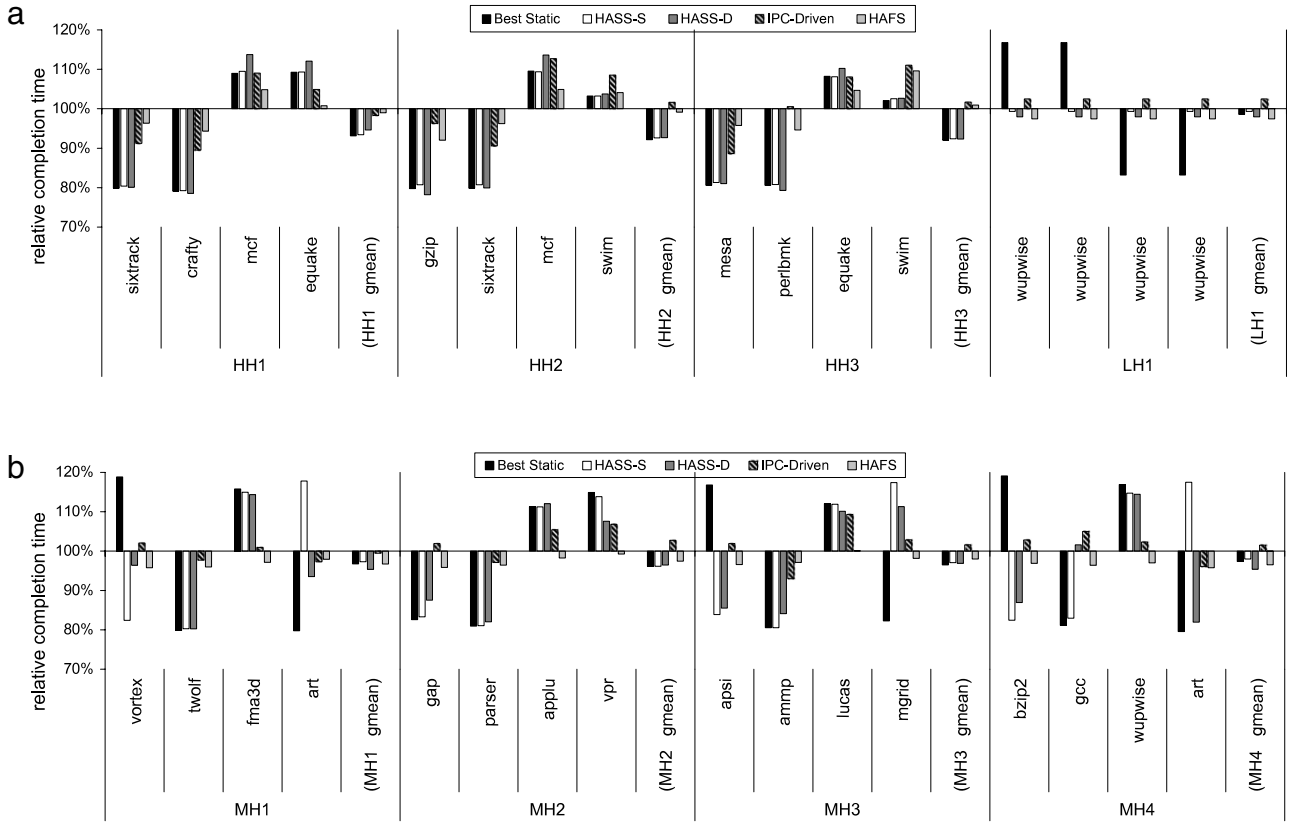


Fig. 3. Completion times relative to the default metric for workload set #1 on the Intel-2,2 configuration (a) HH and LH workloads, (b) MH workloads.

The speedup that HASS-S achieves relative to HAFS is more modest: on average 6% for HH workloads and 2% for MH workloads. This implies that for MH workloads, a simple het-aware round-robin scheduler can perform almost as well as more complex algorithms, but for HH workloads a more sophisticated assignment policy is necessary to optimize performance.

We also note that HASS-S always outperforms the IPC-Driven algorithm. This was a surprising finding, because the IPC-Driven algorithm, in contrast to HASS-S, is phase aware and so it could fine tune the thread assignment as the workload goes through different phases of execution. We provide the explanation for this unexpected result in Section 5.4.3.

The other investigated phase-aware scheduler, HASS-D, is not subjected to the same limitations that IPC-Driven suffers from, since performance monitoring in this scheduler does not require cross-core migrations. As a result, HASS-D performs within 1% range of HASS-S on the AMD-2,2 configuration.

Turning to the Intel-2,2 configuration (Fig. 3a and b), we note that the range of performance improvements from het-aware scheduling is smaller on this system. This is expected, because this hardware platform is less "heterogeneous" than the AMD-2,2 configuration. This indicates that sophisticated het-aware algorithms are more appropriate for systems with a high degree of heterogeneity among the cores as opposed to systems where performance differences across the cores are small.[6]

On the Intel-2,2 configuration, most benchmarks exhibited a more CPU-bound nature than on the AMD system (probably because the Intel system had larger L2 caches), and so there was less distinction in the CPU speed sensitivities among the benchmarks, especially those in the MH category. As a result, HASS-S often picked a different set of applications to run on fast cores than those picked by the best static assignment. Nevertheless, the differences in the sensitivities were so small that picking the "wrong" applications did not have a large impact on performance—in many cases it did not matter which applications would be chosen to run on fast cores. As a result, HASS-S performed only 0.5% worse (on average) than the best static assignment. This demonstrates that the HASS-S algorithm is robust even in these conditions difficult for optimization.

Finally, we examine the performance of workload set #1 and #2 on the AMD-12,4 configuration (Figs. 4 and 5). We expected to see the smallest performance improvements here, because there are proportionally fewer fast cores than on the other systems (only a quarter of the cores is fast as opposed to one half on the other configurations), and also because there is cache sharing.

Examining the results for the HH workloads (Fig. 4a) we note that HASS-S did not always pick the same application to run on the fast cores as that which was picked by the best static assignment. For example, in the workload {`sixtrack`, `crafty`, `mcf`, `equake`} HASS-S assigned the four copies of `sixtrack` to run on the four fast cores, while the best static assignment picked `crafty`. The differences in sensitivities of `crafty` and `sixtrack` are so small that it is difficult for HASS-S to make this distinction. At the same time, failure to make this distinction does not have a large effect on performance, so HASS-S underperformed the best static assignment only by 2%.

A similar phenomenon (not picking the same applications to run on the fast cores as those picked by the best static assignment) can be observed for the MH workloads (Fig. 4b). It is important to note, however, that HASS-S has never made an incorrect choice of running memory-bound applications on the fast cores: it has always correctly picked the CPU-bound applications. The reason

[6] For example, systems that exhibit small variations in the frequencies among the cores due to fabrication process variation would probably benefit less from sophisticated het-aware scheduling algorithms than explicitly heterogeneous systems.

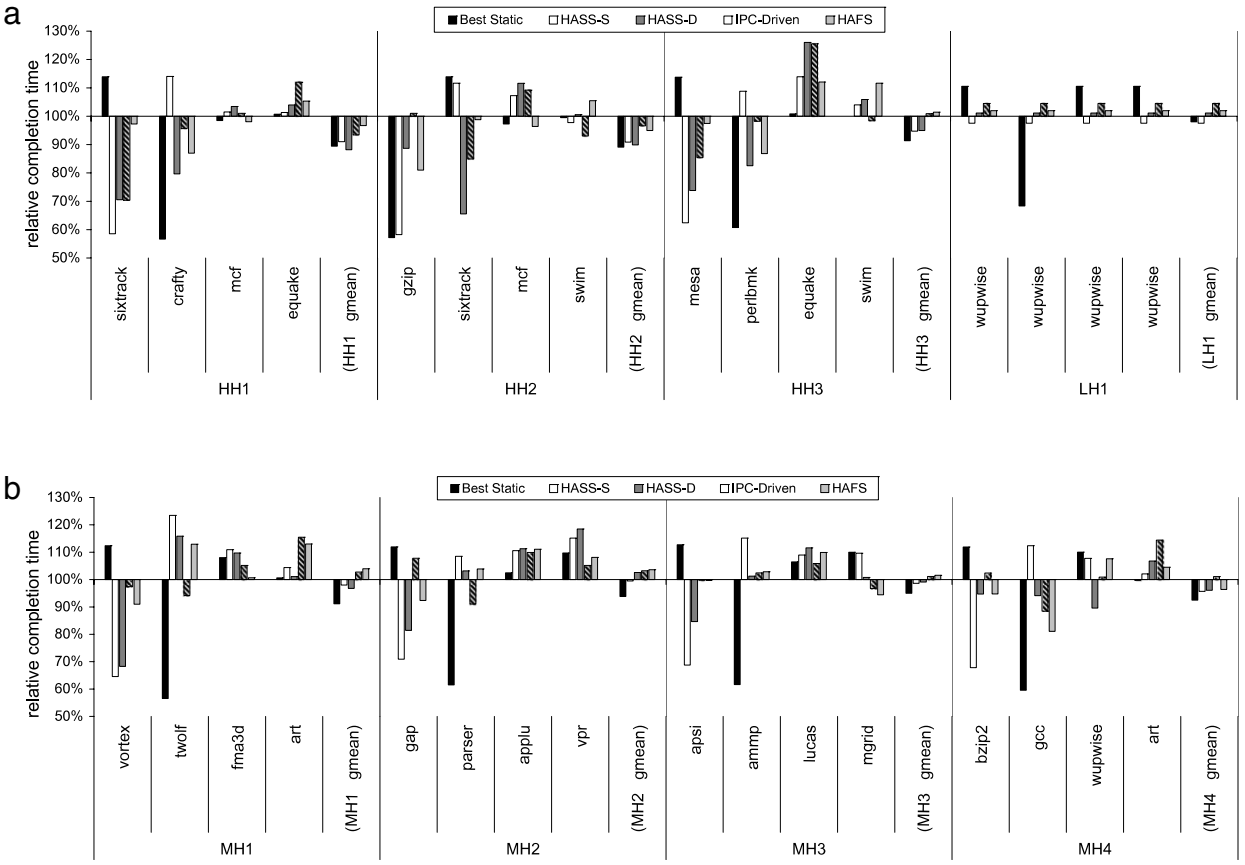


Fig. 4. Completion times relative to the default metric for workload set #1 on the AMD-12,4 configuration, base workload multiplied by 4 (16 benchmarks in total). (a) HH and LH workloads, (b) MH workloads.

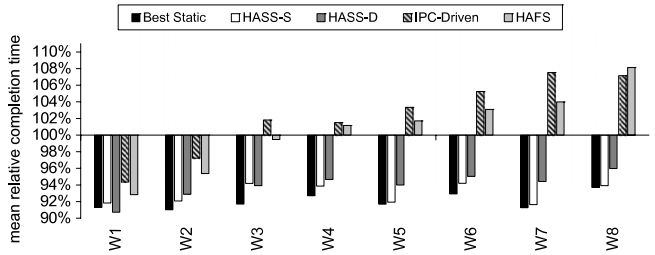


Fig. 5. Geometric mean of completion times relative to the default metric for workloads in set #2 on the AMD-12,4 configuration.

for not picking the sames ones as the best static assignment is that the signatures for CPU-bound applications were difficult to distinguish from one another in this moderately heterogeneous workload.

The results for the MH workloads on the AMD-12,4 configuration offer an opportunity to observe the effects of cache sharing, indicating the importance of accounting for this phenomenon in scheduling algorithms. Consider, for instance, the workload {`vortex`, `twolf`, `fma3d`, `art`}. HASS-S chose to run the four copies of `vortex` on fast cores, while according to the best static assignment the four copies of `twolf` should not have been picked. In theory, this "mistake" should not have had much impact on performance, because the sensitivities of `vortex` and `twolf` are very similar. In reality, this "mistake" caused HASS-S to underperform the best static assignment by 7% (although still doing better than default). The reason is cache sharing. `Twolf` is a very cache-sensitive application. That is, its performance suffers when it shares a cache with an aggressive co-runner that generates a lot of cache misses. In this workload, such aggressive applications are `fma3d` and `art`. By running `twolf` on fast cores, as was done under the best static assignment, `twolf` is isolated to run on a separate chip from other benchmarks (recall that the four fast cores in AMD-12,4 are placed on a separate chip), and so it avoids sharing the per-chip last-level cache with the aggressive co-runners. But when `twolf` runs in a slow partition, where the 12 cores are spread across the three chips, it risks sharing a cache with the aggressive `art` or `fma3d`. These results indicate the importance of incorporating the awareness of shared caches into scheduling algorithms for multicore systems.

We now focus our attention on the performance numbers of workload set #2 on the AMD-12,4 configuration (shown in Fig. 5). Since each workload in this set includes up to sixteen different applications and exhibits a wide diversity in speedup factors (as on the HH workloads), a significant improvement over the default metric can be potentially achieved by het-aware schedulers. Performance gains delivered by HASS-S are very close to the best static's counterparts for workloads W1, W5, W7 and W8. In fact, execution traces obtained with Dtrace revealed that those gains stemmed from the fact that HASS-S actually chooses the mapping of threads to cores that corresponds to the best static assignment. In the remaining workloads, HASS-S picked a different set of applications to run on fast cores to best static's counterparts but because the divergences between the speedup factors of the benchmarks mapped to fast cores in both cases are small, HASS-S's "wrong" application mappings do not have a large impact on performance (at most 2.5% for the W3 workload).

We must also highlight that, on the AMD-12,4 configuration, the estimation model used by HASS-S is affected by the presence of shared caches. Essentially, the LLC miss rate of the applications

may vary due to the sharing of the cache with other threads and, as a result, offline-estimated ratios used by HASS-S may not approximate so accurately the observed ratios during the execution. In other words, the fact that the miss rate may decrease because of cooperative data sharing or increase due to cache contention may lead to overestimation or underestimation of the ratios, respectively. However, previous researchers [33] observed (and so did we) that the *quality* of the miss rate does not change significantly no matter whether the thread shares a cache or runs solo: i.e., if the thread's miss rate is low relative to other threads when it runs solo, its value relative to other threads will stay low when it shares the cache even though it may increase by tens or hundreds of percent relative to its solo value. Similarly, if the thread's miss rate is high it will stay high relative to other threads, regardless of sharing. For that reason, HASS-S is still able to effectively distinguish between memory-intensive and CPU-intensive applications so it correctly classifies the threads even when core and thread counts increase.

To conclude this section, we must highlight that we also performed scalability and overhead analysis of HASS-S. These results, which are omitted from this paper, were reported in our prior work [28]. Overall, our evaluation indicates that HASS-S has overheads comparable to those of the native scheduler and that it scales well on our experimental systems.

5.4.2. Evaluation of HASS-D

Comparing HASS-D to HASS-S, we see that the former performs within 1%, 2.5% and 3.5% range of the latter on the AMD-2,2, Intel-2,2 and AMD-12,4 configurations, respectively. Traces of the execution of the workloads, collected by means of Dtrace, lead us to conclude that both algorithms perform the same thread-to-core mappings for the vast majority of the execution. The reason behind this behavior is two-fold. First, while most applications involved in the evaluation exhibit several program phases, we have not found any application from the SPEC suite alternating between large CPU-intensive phases and large memory-intensive phases. Since the program-phase-detection engine of HASS-D has been deliberately designed to filter out short-term program phases in an attempt to reduce the number of thread migrations, this algorithm captures primarily long-term phases. For most applications, this leads HASS-D to detect one large phase that encompasses nearly the entire execution interspersed with a few shorter phases. Second, both algorithms rely on threads' last-level-cache miss rates to estimate the relative benefit from running a given thread on fast cores rather than on slow cores, so they obtain similar estimates and perform thread assignments accordingly. In summary, the fact that applications do not exhibit many long-term distinct program phases in conjunction with a common model for performance estimates used by both schedulers makes them perform similarly. Furthermore, it is worth highlighting that both algorithms are exposed to similar mispredictions and, when present, they fail to figure out the optimal assignments (see MH1 and MH3 workloads in Fig. 3b). However, those minor mispredictions do not affect the overall performance significantly.

As opposed to HASS-S, HASS-D is phase aware. Supposedly, being aware of program phases would enable HASS-D to enforce better thread-to-core mappings throughout the execution. Although adjusting thread-to-core assignments dynamically may improve the system-wide efficiency on AMP systems, it may also introduce performance degradation due to additional thread migrations. The negative impact on performance due to these additional migrations may be especially pronounced for highly memory-intensive applications (such as `mcf`, `equake` and `swim`), whose performance suffers significantly when their cache state needs rebuilding after migrations (at least in the private levels of the cache hierarchy). In particular, HASS-S usually outperforms HASS-D when highly memory-intensive programs are included in the workload, such as for workloads W4–W8 in set #2 (Fig. 5). Note, however, that not only does migration overhead affect HASS-D but it has also a negative impact in the performance of any scheduler triggering a non-negligible number migrations, such as IPC-Driven and HAFS. For those schedulers, overhead of as much as 7% over the default metric is introduced for the aforementioned workloads.

5.4.3. Evaluation of the IPC-Driven algorithm

We now turn our attention to the IPC-Driven algorithm. The results for the three hardware configurations and the different workloads are shown in Figs. 2a–4b. The overall (unexpected) conclusion is that the IPC-Driven algorithm performs worse than HASS and the best static assignment. We expected the IPC-Driven algorithm to work better, since unlike the other approaches it relies on a real measured speedup factor rather than estimating it. Despite the careful tuning of configurable parameters in the algorithm (the results are shown for the best combination of parameter values), we could not make the IPC-Driven algorithm match the performance of HASS and of the best static assignment. We discovered that the unexpectedly low performance of the IPC-Driven algorithm is due to two problems: (1) inaccurate estimation of the relative benefit that threads derive from running on different core types, and (2) overhead due to migrations performed as part of dynamic performance monitoring.

We illustrate the first problem by analyzing the performance of the HH workload {`sixtrack`, `crafty`, `mcf`, `equake`} on the AMD-2,2 configuration. For that workload, the IPC-Driven algorithm achieves the performance improvement of only 6% over default—recall that HASS-S has achieved a 12% performance improvement for this workload! To understand the root cause of the problem we analyzed how the IPC-Driven algorithm performed thread assignments relative to HASS-S.

Both HASS-S and the best static assignment mapped the two frequency-sensitive applications `sixtrack` and `crafty` to the fast cores, and the two memory-bound applications `mcf` and `equake` to the slow cores. The IPC-Driven algorithm, on the other hand, mapped `mcf` to the fast core roughly 51% of the time, pushing `crafty` to run on the slow core in the meantime (these data were obtained with Dtrace). Although `mcf` does have some high-IPC phases when it makes sense to map it to the fast core (see Fig. 7a), those phases last only 25% of `mcf`'s execution time, not 51%. So 26% of the time `mcf` is not being mapped to the right core, which degrades the performance.

The reason for this suboptimal mapping has to do with the unstable nature of phase changes. When `mcf` runs on a fast core during a high-IPC phase and a phase change is detected, it is migrated to a slow core to refresh its IPC ratio. However, as it runs on the slow core, the phase change (and the decrease in the IPC) continues, and so the IPC degradation reflects not only the lower clock frequency of the slow core but the fact that the program has entered an even more memory-bound phase. Ideally, we want the IPC ratio to be computed from the IPCs measured during the *same* program phase. But since each IPC measurement takes a while to perform (the program must run on each core at least several milliseconds in order to amortize for cold cache effects), it is impossible to guarantee that the program will not change a phase during the measurement. As a result, the estimated IPC ratio is inaccurate.

In this particular example we observed that the IPC-Driven algorithm estimated much higher IPC ratios than what could be obtained on this hardware. Specifically, `mcf`'s IPC ratio between the fast and slow cores was computed to be as high as 2.2 and 2.5 on some occasions. On this configuration, however, the highest possible IPC ratio can be 2.0, because the difference between the frequencies of fast and slow cores is a factor of two. Since the

(incorrectly estimated) ratio is too high, the algorithm erroneously decides that `mcf` derives a far more significant benefit from running on the fast core than in reality. As a result, `mcf` is assigned to a fast partition, when in fact it would be more optimal to assign it to a slow partition.

It is very difficult to ensure that the IPCs used to compute the ratio belong to the same phase. Phase changes are difficult to predict at runtime. The problem gets worse if the number of core types, and hence the number of IPC measurements that must be done, is large (recall that the ratio has to be computed for each class of processors). Increasing the `ipc_threshold` did not help, because no single value worked well for all applications.

The reason why this problem did not occur in the original (simulated) evaluation of the IPC-Driven algorithm [4] is that IPC refreshing was not simulated in the same way as it would happen on a real system. IPCs used to compute ratios were obtained from offline IPC traces, and so in contrast with real systems IPCs always corresponded to the same program phase. In other words, in the earlier simulation-based evaluation it was assumed that the IPCs on different core types can be obtained *instantaneously*, while in reality this could not be accomplished.

Another reason why the IPC-Driven algorithm performed worse than expected is the overhead associated with forced thread migrations, which were performed as part of online monitoring. Recall that the IPC-Driven algorithm must periodically refresh the IPC of all threads on all core types. In order to do so, the scheduler forcefully migrates each thread to the cores of different types for IPC measurement. Unfortunately, this creates load imbalance in the system, because as a result of these migrations some cores may have more threads wanting to run on them than others. As a result of a load imbalance, threads running on ''overloaded'' cores experience longer *CPU wait times* that threads on ''underloaded'' cores. That is, they spend more time waiting in the CPU runqueue until the core to which they are assigned becomes available. This causes their performance to degrade.

To illustrate this phenomenon we have measured to what extent longer CPU wait times affect the performance under the IPC-Driven algorithm. The CPU wait time is the difference between the wall clock completion time and the total CPU time (computed as the sum of user and system CPU times). So if the CPU wait times were negligible (as it should be on our configuration where the number of threads never exceeds the number of cores), the wall clock time would be roughly equal to the CPU time. Fig. 7b shows the amount (in percent) by which the wall clock time exceeds the CPU time for the AMD-2,2 configuration (results for other configurations are omitted, but they are qualitatively similar). The difference in the wall clock time relative to the CPU time is the overhead due to load imbalance. It can be seen that the IPC-Driven algorithm sacrifices a few percentage points of performance due to load imbalance for almost every workload.

Migration overhead was not detected in the original paper on the IPC-Driven algorithm [4], perhaps because runqueue contention was modeled differently than in a real scheduler. The paper did not provide sufficient detail about this part of the simulation. Increasing the `ipc_threshold` and `swap_inactivity` period alleviates migration overhead, but at the expense of making the algorithm less phase aware.

In summary, the particular monitoring methodology used in the IPC-Driven algorithm (and in another het-aware algorithm [19]) suffered from several significant problems. The problems stemmed from the fact that the measurements had to be performed on *every* type of core in the system. Addressing these problems would require a fundamental redesign of the IPC-Driven algorithm. Essentially, we have done this in some way by implementing HASS-D. This algorithm is not subjected to these problems since it estimates performance ratios from measurements obtained on a single core, as opposed to multiple cores. As a result, HASS-D outperforms IPC-Driven across the board.

5.4.4. Evaluation of the HAFS algorithm

In evaluating HAFS we are first of all interested in investigating whether HAFS accomplishes fair sharing of fast cores among applications. To demonstrate HAFS's fairness property we ran an experiment consisting of several instances of the same applications (we chose `mgrid` from the SPEC CPU2000 suite) and measured the fraction of time that each instance spends running on fast cores. (Running four identical applications that have identical completion times simplified data analysis.) We varied the number of concurrent instances from three to ten. The experiments were run on the AMD-2,2 configuration. Fig. 8a shows the results. It can be observed that the fast-core CPU clock cycles are shared equally among the concurrently running instances. When the number of concurrent instances (or threads) equals or exceeds the number of cores, each thread spends 50% of its time running on a fast core. When the number of concurrent instances is three, each thread spends roughly 66% of its time on a fast core, because there are fewer threads competing for fast cores and each one is entitled to its own share.

These results demonstrate two nice properties of the algorithm: first, it ensures fairness in sharing heterogeneous CPU resources. Second, it ensures stable and predictable completion times. Recall that with the native scheduler, completion times were highly variable.

The second question we were interested in investigating has to do with performance overhead due to thread migrations performed by HAFS. Thread migrations are an integral part of HAFS or any implementation of the het-aware round-robin algorithm. Threads *must* be migrated between cores of different types to accomplish fair sharing of fast cores at fine-granular intervals. Although the migration mechanism in HAFS does not cause load imbalance, migrations may still degrade the performance due to disturbing threads' cache affinity [16]. When a thread is moved from one core to another it loses the cache state accumulated in the old core's private caches, and in the old shared cache if the new core does not share a cache with the old one. These overheads have not been investigated in the earlier work and we use HAFS as a tool to study them.

To that end, we compare the performance of HAFS with an *ideal-round-robin* (ideal-RR) metric. The ideal-RR metric is computed by combining the previously measured completion times on all core types such that the total time spent on each core type is proportional to how many of those cores are present in the system. Essentially, the ideal-RR metric estimates the completion time for a benchmark in conditions where the cores of different types are shared equally, but when there are no overheads due to migrations. Comparing completion times estimated with the ideal-RR metric to completion times obtained under HAFS enables us to evaluate the migration overheads in HAFS.

Fig. 8b shows HAFS completion normalized to the ideal-RR completion times for the AMD-2,2 configuration (the results for the other configurations are omitted, but we describe them in the text). The increase in HAFS completion times relative to ideal-RR is the migration overhead. We see that the migration overhead is significant, but not prohibitively large. On this configuration the overhead reached at most 6% for some memory-bound applications. On the AMD-12,4 configuration, where competition for cache was more severe, the overhead reached 25% for one memory-bound application (`twolf`), but hovered around 5%–10% for the rest of the applications.

All in all, the migrations required to deliver fair sharing of fast and slow cores do cause overhead. But despite this overhead, HAFS outperforms default on AMD-2,2 and Intel-2,2 (by 5% and 2% on average, respectively) and breaks even with default on the AMD-12,4 configuration (on average for workload set #1, see Fig. 4a–b). We found that workloads including highly memory-intensive applications, such as W4–W8 from workload set #2, are the most extreme cases, where migration overhead translates into performance degradation with respect to the default metric.

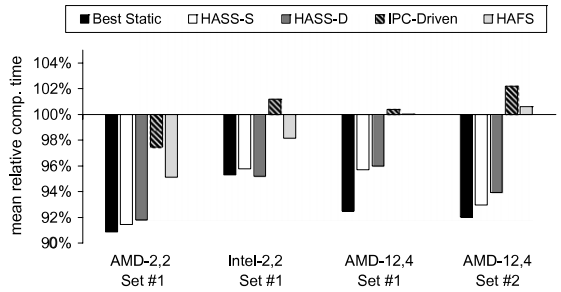


Fig. 6. Overall reduction in completion time relative to the default metric across configurations and workload sets.

5.4.5. Overall results

Fig. 6 shows the overall reduction in completion time over the default metric delivered by all the evaluated schedulers across the different heterogeneous configurations and workloads. These numbers reveal that the performance of HASS-S and HASS-D is very close to the best static's in all cases except one: workload set #1 on AMD-12,4. In the latter case, the fact that four instances of each applications are used when running workload set #1 on AMD-12,4 (sixteen cores) makes the estimated speedup factors of the applications closer, and, as result, the entire workload set is less heterogeneous than the others. This fact further underscores that both versions of HASS deliver greater performance gains when the workload exhibits enough heterogeneity, and more importantly, that these gains can be still obtained when the number of threads and cores increases.

5.4.6. Summary

In summary, the results presented in this section lead us to make the following conclusions:

- HASS-S is an effective and robust het-aware scheduling algorithm that is able to differentiate among benchmarks with different architectural properties and assign CPU-intensive applications to fast cores and memory-intensive applications to slow cores.
- It is more difficult for HASS-S to distinguish among the sensitivities of applications whose signatures are very similar, as would be the case with two CPU-intensive applications. While in this case HASS-S often does not match the best static assignment, the performance impact is small, because the wrongly classified applications have very similar architectural features.
- Cache sharing has an important impact on performance and so it is crucial to incorporate shared cache awareness in HASS-S or any other scheduling algorithm for multicore systems.
- Cross-core migrations required for performance ratio measurements in IPC-Driven often lead to inaccurate IPC ratios and disrupt the load balance of the system. We have also showed that HASS-D, the other phase-aware algorithm, is not subjected to these problems since it estimates performance ratios from measurements obtained on a single core, as opposed to multiple cores. As a result, HASS-D outperforms IPC-Driven across the board, and so does HASS-S.
- In most cases, HASS-S delivers slightly better performance gains that its dynamic version, HASS-D. This was an unexpected finding, because HASS-D, as opposed to HASS-S, is phase aware and so it can adjust thread-to-core mappings dynamically as applications in the workload go through different program phases. Unfortunately, the additional number of migrations triggered by HASS-D introduce overheads that may significantly reduce the benefits coming from phase-aware thread assignments. Furthermore, we have observed that migration overhead also has a negative impact on the performance of other schedulers like IPC-Driven and HAFS, which trigger a non-negligible number of migrations as well, and the presence of highly memory-intensive applications further aggravates this issue.
- The performance improvements from het-aware scheduling are especially pronounced on systems where the difference in CPU speeds among the cores of different types is large.
- Fair sharing of fast cores with HAFS comes at a cost, but in most cases the benefits justify it.

Although the performance improvements achieved on our experimental system were quite significant, we believe that on a "real" heterogeneous system they would be even greater. Real heterogeneous systems are likely to have more drastic differences among the cores of different types (e.g., differences in the pipeline

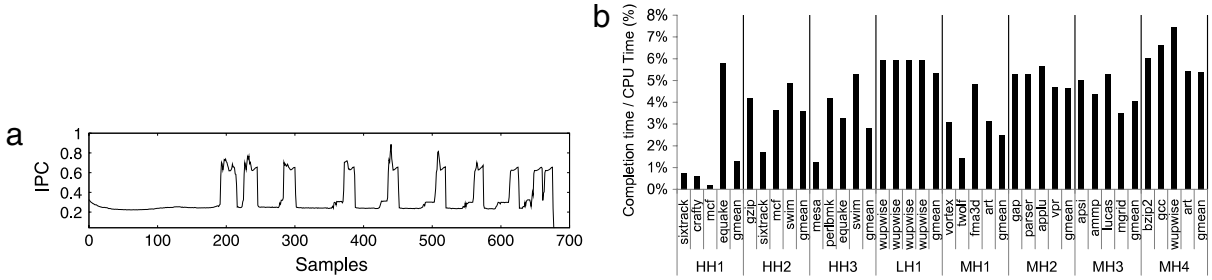


Fig. 7. (a) IPC over time for `mcf` on the Intel system. (b) The amount by which the wall clock completion time with the IPC-Driven algorithm exceeds the CPU time (user + system) for the AMD-2,2 configuration.

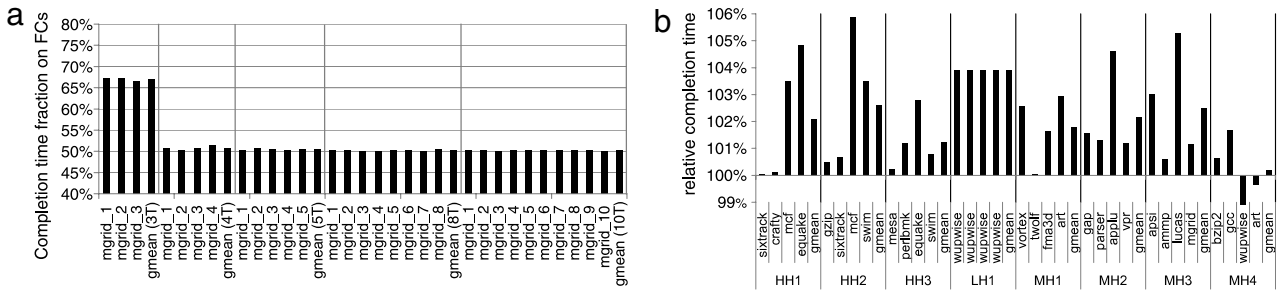


Fig. 8. (a) Fair sharing of fast-core cycles with HAFS. (b) Completion times under HAFS normalized to ideal-RR on the AMD-2,2 configuration.

microarchitecture [18]), and we have shown that more heterogeneous hardware renders greater performance improvements from het-aware scheduling.

Furthermore, in future heterogeneous multicore systems, memory access will likely remain as the major performance-limiting factor (as recently found in [17]). Therefore, we believe that the architectural signature scheme can still rely on LLC misses, at least as a first approximation to estimate the efficiency on different cores with different cache hierarchies and microarchitectures, and so speedup factors could also be predicted in a similar way. Exploring this avenue of research is an interesting direction for future work.

6. Conclusions

We presented HASS, a new scheduling algorithm for single-ISA heterogeneous multicore systems. The novelty of HASS is in relying on architectural signatures for estimating relative benefits that threads derive from running on different core types. We presented two versions of HASS, HASS-S and HASS-D, which rely on statically and dynamically generated architectural signatures, respectively.

HASS consistently improves the performance over a heterogeneity-agnostic scheduler when the workload lends itself to heterogeneity-related optimizations. It is robust even in the conditions where performance improvements are difficult to obtain. Benefits from het-aware scheduling algorithms are especially pronounced for workloads where there is a large disparity between applications' architectural properties and on systems with large differences in the speed among different types of cores. When no performance improvements can be expected due to the nature of the workload, HASS never does worse than the heterogeneity-agnostic scheduler.

Contrary to our expectations, our implementation of HASS, both the static and the dynamic version, performed better than the IPC-Driven algorithm that relied on actual measured speedup factors as opposed to the estimated ones. We discovered that the IPC-Driven algorithm suffered from inaccuracies and overheads stemming from the need to measure performance on multiple core types. As a result, HASS-S and HASS-D outperformed the IPC-Driven algorithm for every workload we have measured. For the sake of providing a more comprehensive experimental evaluation we compared all algorithms to a round-robin heterogeneity-aware algorithm HAFS. We found that HAFS outperforms the heterogeneity-agnostic default scheduler, but fails to match the performance of more sophisticated het-aware algorithms for highly heterogeneous workloads.

When comparing both versions of HASS, we found that the usage of offline collected architectural signatures rather than online ones incurs a lot less overhead at runtime, and this leads the static version to deliver greater performance gains. However, in the event signatures are not either available – i.e. not embedded in the application binary – or are not highly representative throughout the execution – e.g. when the application shows large and fairly distinct program phases – online estimated signatures can be effectively used to fill this gap. For that reason, a hybrid version of HASS, which relies on static signatures and resorts to using dynamic ones when they are not either present or representative enough, would deliver performance gains to a wider range of applications.

Overall, we conclude that using architectural signatures rather than direct measurement of performance of each thread on each core time results in less overhead and delivers greater performance gains.

Acknowledgments

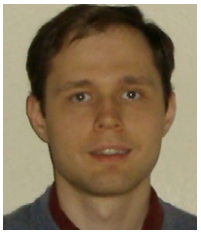
This research was funded by the Spanish government's research contracts TIN2008-005089 and the Ingenio 2010 Consolider ESP00C-07-20811, by the HIPEAC[2] European Network of Excellence, by the National Science and Engineering Research Council of Canada (NSERC) under the Strategic Project Grant program and by Sun Microsystems. Juan Carlos Saez is supported by a MEC FPU fellowship grant.

References

[1] M. Annavaram, E. Grochowski, J. Shen, Mitigating Amdahl's law through EPI throttling, in: ISCA'05.
[2] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, D. Ortega, COTSon: infrastructure for full-system simulation, SIGOPS Oper. Syst. Rev. 43 (1) (2009) 52–61.
[3] S. Balakrishnan, R. Rajwar, M. Upton, K. Lai, The impact of performance asymmetry in emerging multicore architectures, SIGARCH Comput. Archit. News 33 (2) (2005) 506–517.
[4] M. Becchi, P. Crowley, Dynamic thread assignment on heterogeneous multiprocessor architectures, in: Proc. of ACM Computing Frontiers'06.
[5] E. Berg, E. Hagersten, StatCache: a probabilistic approach to efficient and accurate data locality analysis, in: ISPASS'04.
[6] B.M. Cantrill, M.W. Shapiro, A.H. Leventhal, Dynamic instrumentation of production systems, in: USENIX ATEC'04.
[7] D. Chandra, F. Guo, S. Kim, Y. Solihin, Predicting inter-thread cache contention on a chip multi-processor architecture, in: HPCA'05.
[8] G. Dhiman, T.S. Rosing, Dynamic voltage frequency scaling for multi-tasking systems using online learning, in: ISLPED'07: Proceedings of the 2007 International Symposium on Low Power Electronics and Design, pp. 207–212.
[9] V.W. Freeh, D.K. Lowenthal, F. Pan, N. Kappiah, R. Springer, B.L. Rountree, Analyzing the energy-time trade-off in high-performance computing applications, IEEE TPDS. 18 (6) (2007) 835–848.
[10] M. Gillespie, Preparing for the second stage of multi-core hardware: asymmetric (Heterogeneous) cores, Intel White Paper, 2008.
[11] M.D. Hill, M.R. Marty, Amdahl's law in the multicore era, IEEE Computer 41 (7) (2008) 33–38.
[12] M.D. Hill, A.J. Smith, Evaluating associativity in CPU caches, IEEE TC 38 (12) (1989) 1612–1630.
[13] K. Hoste, L. Eeckhout, Microarchitecture-independent workload characterization, IEEE Micro 27 (3) (2007) 63–72.
[14] M.C. Huang, J. Renau, J. Torrellas, Positional adaptation of processors: application to energy reduction, in: ISCA'03.
[15] C. Isci, A. Buyuktosunoglu, C.Y. Cher, P. Bose, M. Martonosi, An analysis of efficient multi-core global power management policies: maximizing performance for a given power budget, in: MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 347–358.
[16] V. Kazempour, A. Fedorova, P. Alagheband, Performance implications of cache affinity on multicore processors, in: Euro-Par'08.
[17] D. Koufaty, D. Reddy, S. Hahn, Bias scheduling in heterogeneous multi-core architectures, in: Proc. of Eurosys'10.
[18] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, D.M. Tullsen, Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction, in: MICRO 36.
[19] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, K.I. Farkas, Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance, in: ISCA'04.
[20] T. Li, D. Baumberger, D.A. Koufaty, S. Hahn, Efficient operating system scheduling for performance-asymmetric multi-core architectures, in: ICS'07.
[21] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in: PLDI'05.
[22] J.C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, V. Talwar, Using asymmetric single-ISA CMPs to save energy on operating systems, IEEE Micro 28 (3) (2008) 26–41.
[23] R. van der Pas, The OMPlab on sun systems, in: Proc. of IWOMP'05.
[24] J.P. Perez, P. Bellens, R.M. Badia, J. Labarta, CellSs: making it easier to program the cell broadband engine processor, IBM J. Res. Dev. 51 (2007) 593–604.
[25] J.C. Saez, A. Fedorova, M. Prieto, H. Vegas, Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors, in: Proc. of ACM Computing Frontiers'10.
[26] J.C. Saez, M. Prieto, A. Fedorova, S. Blagodourov, A comprehensive scheduler for asymmetric multicore systems, in: Proc. of ACM Eurosys'10.
[27] D. Shelepov, A. Fedorova, Scheduling on heterogeneous multicore processors using architectural signatures, in: Workshop on the Interaction between Operating Systems and Computer Architecture.
[28] D. Shelepov, J. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. Huang, S. Blagodurov, V. Kumar, HASS: a scheduler for heterogeneous multicore systems, SIGOPS Oper. Syst. Rev. 43 (2009) 66–75.
[29] A.J. Smith, A comparative study of set associative memory mapping algorithms and their use for cache and main memory, IEEE Trans. Softw. Eng. 4 (2) (1978) 121–130.
[30] D.K. Tam, R. Azimi, L.B. Soares, M. Stumm, RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations, in: ASPLOS'09, pp. 121–132.
[31] R. Teodorescu, J. Torrellas, Variation-aware application scheduling and power management for chip multiprocessors, in: ISCA'08.

[32] R. Vuduc, A. Chandramowlishwaran, J. Choi, M.E. Guney, A. Shringarpure, On the limits of gpu acceleration, in: Second USENIX Workshop on Hot Topics in Parallelism.
[33] S. Zhuravlev, S. Blagodurov, A. Fedorova, Addressing cache contention in multicore processors via scheduling, in: ASPLOS'10.

Juan Carlos Saez obtained an M.Sc. degree in computer science from the Complutense University of Madrid and a B.A. degree in music from Teresa Berganza Professional Music Conservatory, both in 2006. He is now a Ph.D. student in computer science at the Complutense University. His research interests include energy-aware and cache-aware task scheduling on multicore and many-core processors. His recent research activities focus on operating system scheduling on heterogeneous multicore processors, exploring new techniques to deliver better performance per watt, and quality of service on these systems. His work is supported by the Spanish government through an FPU fellowship grant.



Daniel Shelepov graduated from Simon Fraser University (SFU), Vancouver, Canada, in 2008 with a B.Sc. in computer science. During his senior year he led a research effort on heterogeneity-aware scheduling under supervision from Alexandra Fedorova. He currently works in the Internet Explorer team at Microsoft.

Alexandra Fedorova is an assistant professor of computer science at Simon Fraser University (SFU) in Vancouver, Canada. She earned her Ph.D. from Harvard University in 2006, where she completed a thesis on operating system scheduling for multicore processors. Concurrently with her doctorate studies, Fedorova worked at Sun Microsystems Research Labs, where she investigated transactional memory and operating systems. She is the lead inventor of three US patents. At SFU, Fedorova co-founded the SYNAR (Systems, Networking and Architecture) research lab. Her research interests span operating systems and virtualization platforms for multicore processors, with a specific focus on scheduling. Recently she started a project on tools and techniques for parallelization of video games, which has led to the design of a new language for this domain.

Manuel Prieto received his Ph.D. in computer science from the Complutense University of Madrid (UCM) in 2000. He is now Associate Professor in the Department of Computer Architecture at UCM and serves as Vice-Dean for External Relations and Research at the Faculty of Informatics. His research interests lie in the areas of parallel computing and computer architecture. Most of his activities have focused on leveraging parallel computing platforms and on complexity-effective microarchitecture design. His current research addresses emerging issues related to chip multiprocessors, with a special emphasis on the interaction between the system software and the underlying architecture. He has co-written numerous articles in journals and for international conferences in the field of parallel computing and computer architecture. He is a member of the ACM.