HASS: A Scheduler for Heterogeneous Multicore Systems

Daniel Shelepov[1]
dsa5@cs.sfu.ca

Juan Carlos Saez Alcaide[2]
jcsaezal@fdi.ucm.es

Stacey Jeffery[3]
sjeffery@uwaterloo.ca

Alexandra Fedorova[1]
fedorova@cs.sfu.ca

Nestor Perez[1]
npa5@sfu.ca

Zhi Feng Huang[1]
zfh@sfu.ca

Sergey Blagodurov[1]
sba70@cs.sfu.ca

Viren Kumar[1]
vka4@cs.sfu.ca

[1] Simon Fraser University
8888 University Drive
Burnaby, BC, Canada

[2] Complutense University of Madrid
Ciudad Universitaria – 28040,
Madrid, Spain

[3] University of Waterloo
200 University Avenue West
Waterloo, ON, Canada

Abstract

Future heterogeneous single-ISA multicore processors will have an edge in potential performance per watt over comparable homogeneous processors. To fully tap into that potential, the OS scheduler needs to be heterogeneity-aware, so it can match jobs to cores according to characteristics of both. We propose a Heterogeneity-Aware Signature-Supported scheduling algorithm that does the matching using per-thread architectural signatures, which are compact summaries of threads' architectural properties collected offline. The resulting algorithm does not rely on dynamic profiling, and is comparatively simple and scalable. We implemented HASS in OpenSolaris, and achieved average workload speedups of up to 13%, matching best static assignment, achievable only by an oracle. We have also implemented a dynamic IPC-driven algorithm proposed earlier that relies on online profiling. We found that the complexity, load imbalance and associated performance degradation resulting from dynamic profiling are significant challenges to using this algorithm successfully. As a result it failed to deliver expected performance gains and to outperform HASS.

Categories and Subject Descriptors D.4.1 [**Operating Systems**]: Process Management – scheduling.

General Terms Algorithms, Management, Performance, Design.

Keywords heterogeneous, multicore, scheduling, asymmetric, architectural signatures.

1. Introduction

Single-ISA heterogeneous multicore processors, also known as *asymmetric* single-ISA (ASISA) [18], consist of cores exposing the same ISA, but delivering different performance. These cores differ in clock frequency, power consumption, and possibly in cache size and other microarchitectural features. Asymmetry may be built in by design [14][15], or may occur due to process variation [13] or explicit clock frequency scaling. Given a diverse workload, an ASISA system can deliver more performance per watt than a homogeneous system, because threads can be matched to cores according to the relative benefit that they derive from running on different core types. For example, in an ASISA system with several fast and powerful cores (high clock speed, ILP-oriented optimizations) and several simple and slow cores, memory-bound threads 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 they consume. Power and area efficiencies of ASISA systems have been demonstrated in numerous studies [3][14][15][16][18]. In addition, asymmetric systems allow superior performance for mixed workloads of sequential and parallel applications [10].

Efficiency of ASISA systems is maximized when workloads are *matched* with cores according to the properties of the workload and features of the core. This matching is typically done by a *heterogeneity-aware scheduling algorithm* in the operating system (het.-aware from now on for brevity). In this paper we describe a new het.-aware scheduling algorithm 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 memory-boundedness, available ILP, sensitivity to variations in clock speed and other parameters. The common property of these parameters is that they can all be relatively easily and quickly interpreted by the scheduler to determine how well a given application "matches" a given core. The signatures are generated offline and are presented to the scheduler as a single unit with the application binary, perhaps by being embedded into the binary itself. The scheduler then matches jobs with cores based on these signatures.

Unlike HASS, previously proposed het.-aware algorithms determined the best matching of threads to cores via online performance monitoring [3][15], which determined relative speedup of each thread on different core types. As the number of cores (and core types) on the chip increases [1][5], the overhead of performance monitoring grows and it becomes less practical as a means of determining optimal assignment. Our scheme does not use online monitoring and thus removes the overhead associated with it, in exchange sacrificing some accuracy.

Static nature of HASS imposes some limitations on its structure and functionality, and so it is important to investigate their impact. First of all, in the current implementation there is only one signature per application. While this scheme can be extended to multithreaded applications with relative ease, it is more difficult to accommodate for different input sets, which can sometimes cause significant changes in application behaviour and therefore optimal thread-to-core mappings. We investigated the effect of varying program inputs on HASS's accuracy and determined that while some inaccuracies are unavoidable, the overall impact on accuracy is small. Another limitation of HASS is that it requires cooperation from the application development side for generation of architectural signatures. This, however, can be achieved without significant involvement from the developer. For example, signature generation can be done in conjuction with compilation with minimal additional manual effort. Thus this aspect is not a significant hindrance if improved system performance is achieved in return. A final limitation is that HASS does not account for phase changes, since the architectural signature persists for the lifetime of a program. While this means that HASS cannot outperform the best static (oracle) assignment, it also eliminates the need for dynamic performance monitoring, which, as we discovered, causes load imbalance and thus hurts performance.

The advantages of HASS, on the other hand, are (1) better scalability, (2) simpler implementation (both due to the absence of dynamic monitoring in the scheduler), and (3) support for short-lived threads: threads that would otherwise spend the whole or the majority of their lifetime in the suboptimal "performance monitoring" stage.

HASS deliberately trades accuracy for simplicity and efficiency compared to algorithms based on online profiling. In order to fully evaluate whether this trade-off is worth making, we compare HASS to a het.-aware *IPC-driven* algorithm that uses online profiling, proposed by Becchi et al. [3]. We found that while the IPC-driven algorithm usually improves thread assignments, it causes load imbalance due to its need to frequently migrate threads from one core type to another for performance measurements. This causes performance overhead, often negating the benefits of improved thread assignments.

We implemented both algorithms, HASS and IPC-driven, which had not previously been implemented, in the OpenSolaris operating system. We evaluated the algorithms on two real multicore platforms made asymmetric via CPU frequency scaling. We found that both HASS and IPC-driven improve performance by 7-13% for a diverse workload, but that the IPC-driven algorithm suffered significant instability in workloads other than those with few phases.

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 HASS scheduler, and Section 5 provides the evaluation. Section 6 describes the IPC-driven algorithm, provides the evaluation, and compares it with HASS. Section 7 summarizes and outlines plans for future work.

2. Background and Related Work

The problem of scheduling on ASISA multicore processors is relatively new. Two of the most well-known scheduling algorithms proposed are by Becchi et al. [3] and Kumar et al. [15]. 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. Those 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 core type per thread. In addition, Kumar proposed an algorithm that tries to determine a globally optimal assignment by sampling performance of thread groups rather than making local thread-swapping decisions.

Both these approaches promise significantly better performance than naïve heterogeneous-agnostic policies with any kind of heterogeneous workload, but they are both difficult to scale to many cores. There are several barriers to scalability.

Firstly, in their unmodified versions they both support only two distinct core types. Secondly, their reliance on profiling means that demand for different core types will be unequal. In particular, the larger the ratio of slow cores to fast 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 predetermined performance profiles, it sacrifices accuracy, but at the same time avoids scalability barriers related to sampling and has a much simpler implementation.

Teodorescu and Torrellas [22] 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.

Balakrishnan et al. [2] implemented a simple het.-aware scheduler in Linux that ensures that fast cores never go idle before slow cores. While this scheduler mitigates the effects of performance asymmetry, it is not meant to improve efficiency. Mogul et al. [18] 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 pre-generate architectural signatures), but it only applies to workloads dominated by system calls.

Li et al. [16] designed a het.-aware algorithm for Linux, AMPS, that makes sure that the load on each core is proportional to its power and that fast cores are never under-utilized. HASS also ends up loading faster cores more than slow cores (as we explain later), but HASS makes this decision based on the properties of threads, while AMPS does not, so AMPS may run a memory-bound thread on a fast core and lose efficiency.

3. Architectural Signatures

Signatures Overview

An architectural signature is a summary of the architectural characteristics of an application. HASS relies on the ability to estimate potential performance of a thread on a core given the load and characteristics of that core. To that end, the signature must enable it to predict threads' relative performance on different cores. In this work we focus on systems where cores differ in clock frequency (as on the evaluation platforms used in this paper) and in cache size, parameters expected to play a prominent role in future ASISA systems. Construction of signatures that take into account the pipeline characteristics is left for future work.

To predict performance variations due to clock frequency, we must consider the application's degree of memory-boundedness [9]. An application with a high rate of memory accesses is likely to stall the core often, so clock frequency will not have a significant effect on performance. Memory-boundedness can be captured by an application's *reuse-distance profile.* A reuse distance indicates the number of intervening memory accesses between consecutive accesses to the same memory location, and the reuse distance profile is the distribution of reuse distances. From a reuse-distance profile we can accurately estimate the application's last-level cache miss rates for any cache configuration [4][11][19][21]. *These estimated miss rates are the contents of the signature.* From them we approximate performance on cores with different frequencies and cache sizes. Before explaining how this is done, we describe the process of constructing the signatures.

Constructing Signatures

An important property of signatures is their microarchitecture-independence: 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 the application binary itself. For evaluation covered in this paper we have not implemented the binary embedding scheme, and have instead opted for hard-coding 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 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" behaviour during this signature run. If it is impossible to do 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 [17], along with a custom Pin extension, MICA [12]. A more detailed account can be found in our previous work [19]. Once the profile is collected, we estimate (also offline) cache misses for a limited set of realistic cache configurations. These estimations, collected in a matrix, comprise the architectural signature. We support 11 different 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 below is an example signature for the benchmark *179.art* from SPEC CPU2000 suite:

	cache size						
set-assoc.	**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

Each integer represents the expected number of misses per 4096 instructions (the number 4096 was selected to speed up calculations at scheduling time). Columns for sizes 16K to 128K are omitted, because the values there are in this case exactly the same as for 256K (427 misses).

Using Signatures for Scheduling

At runtime the architectural signature is used to estimate a thread's performance on each type of core present. To accomplish this, we calculate a hypothetical completion time of some constant number of instructions. Two separate parts 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. We have used a cost of 1.5 cycles per instruction. Clock speed is also factored in.

We approximate stall time by the number of cycles used for servicing last-level cache misses that ended up going into main memory. Although this is a coarse approximation, it gives reasonable accuracy, because memory access time dominates other stalls. To estimate this, we need memory access latency (discoverable by the OS) and the miss rate that we obtain by looking up the signature. Note that since we are assuming 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. The exact method is described in the next section.

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 or even completely disappear when shared caches were present. In our case, in addition to a shared L3 cache, each core had a large exclusive L2 cache. The situation could be even worse if the L2 cache were shared. Modelling 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 are currently investigating a solution.

To summarize, we predict performance of different threads on different cores based on threads' caching behaviour and cores' cache size and frequency. This allows threads to distinguish cores by their relative desirability. While we have not been able to test the accuracy of this relative performance estimation for cores that differ in cache sizes, we have been able to test it on cores that differ in frequency by using Dynamic Voltage and Frequency scaling (DVFS) facilities. DVFS allows the operating system to control the clock speed of the CPU. Figure 1 shows how well real performance ratios match predicted ratios for some of our test configurations (described in 5.1). As evident, the estimation method is successful in separating memory-bound threads (which

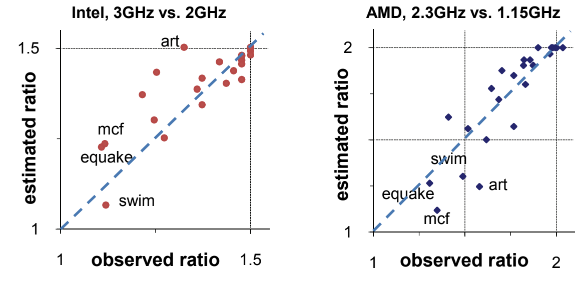

Figure 1. Signature-estimated performance ratios vs. observed ratios. Some outliers labeled. Perfectly accurate estimations would have all points on the diagonal line.

are less sensitive to changes in frequency and therefore concentrated towards lower left) from CPU-bound threads (upper right), but is less precise in characterizing memory-boundedness.

We would like to emphasize that the process of performance estimation occurs at scheduling time, and not offline. Nevertheless, we have presented it in this section, because it is closely connected with the format of the signature.

4. HASS Scheduler

HASS stands for Heterogeneity-Aware Signature-Supported. Besides those properties, the scheduling algorithm also emphasizes scalability, because future ASISA processors may, in fact, be many-core. Scalability mainly manifests in two aspects: the lack of global locks, and local greedy decision-making on the part of threads. As we describe the algorithm, we will point out particular features that ensure scalability.

We begin the description of the algorithm by introducing a few key abstractions. The first abstraction is a *processor class*. A processor class has a unique combination of features (described inside the class) such as clock frequency, cache hierarchy, or different execution cores. If two cores belong to the same class, they must be identical, and a system must have at least two classes to be heterogeneous. We assume that the class configuration of the system is static. Since classes may contain a very large number of cores on future many-core systems, class-based load balancing and accounting may become tedious. To manage this problem, processors are also grouped together in CPU partitions. Each partition must belong to exactly one processor class, but each class may contain one or more partitions. Each core must belong to exactly one partition, and a partition is the widest locking scope during normal operation of the scheduler. This allows managing a large number of cores in a scalable way. Each partition keeps a counter of *runnable* threads (threads either currently running or ready to be run), which is updated in real time. This counter is the primary partition-wide contention point, as it has to be fully synchronized.

When threads enter the system, they iterate through all existing processor classes and estimate their performance using signatures (the is method described in Section 3.3) according to the attributes of that class (there needs to be only one such estimation per class). These values can be calculated once and used until the thread exits. Note that these *base* ratings represent the expected performance, were the thread able to get a core of that class for exclusive use.

To assign itself to a partition the thread goes through the list of all partitions and estimates its performance in that partition using the base ratings and the current number of runnable threads per core in the partition (the thread assumes that CPU time will be shared equally within the partition). After that, the thread selects the partition with the highest expected performance and assigns itself 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 when large processor classes need to be partitioned. Assignment 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 is called a *refresh*. By having the refresh period tied to CPU rather than wallclock time, we avoid increasing the absolute number of refreshes as the load factor grows.

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, when there is an idle core somewhere in the system. To prevent such occurrences, it is forbidden to move to fully loaded or overloaded partitions when some partitions are less than fully loaded (*underloaded*).

The greedy approach has a potential problem where threads may become locked in a suboptimal assignment and further optimization can only be accomplished by cooperative action between two threads (swapping) rather than by a greedy decision of any one thread. There is a mechanism to do that, and it is called *optimistic assignment*. It can be selected by the thread instead of regular assignment during a refresh, if it fails to find a good target partition. The initiator has to find a partner in the target partition and swap with it. The initiator can only trigger the switch if it confirms that the swap will actually increase its own, as well as overall system performance. This is done by comparing the base performance ratings of the initiator and 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, it is critical to forgo exhaustive search and instead use randomized search with a limited amount of probing.

One special case where optimistic rebinding is especially important is when the aforementioned partition underload protection mechanism kicks in. Remember that during these situations regular assignment to any partitions except those that are underloaded is forbidden. However, it is permissible to rebind *optimistically* to any partition, even those that are not underloaded. This is because swapping threads cannot create a load imbalance worse than that which already exists. Without this relaxation, any balanced configuration that had the number of runnable threads in the system equal to the number of cores would be immutable, even if it were suboptimal.

The partition scheme allows the scheduler to avoid global locks 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.

We chose Solaris as the platform to implement the scheduling algorithm due to its powerful profiling framework DTrace [6]. We used build 86 of OpenSolaris 2008.05. In implementing HASS we reused much of the Solaris code for CPU partitions.

We have not implemented the "multiple partitions per processor class" scheme, because our testing platforms did not have enough cores to motivate this apparatus. Finally, due to time constraints, we have simplified optimistic assignment target search. Only one partition was probed for a partner, the partition with the highest expected performance.

5. Evaluation of HASS Algorithm

Evaluation Platform and Methodology

We used two machines for our experiments. One was an Intel Xeon X5365 server with four dual-core packages. A pair of cores on a package shared a 4MB L2 cache. Another was an AMD Opteron 8356 with four quad-core chips. Cores on the same chip shared a 2MB L3 victim cache (512KB L2 caches were private). Both systems were running versions of OpenSolaris 2008.05 build 86 for the x86 platform.

Table 1. Test configurations.

Name	Partitions / cores	Other information
Intel-2,2	*Part. 1:* (2@2GHz), *Part. 2:* (2@3GHz)	exclusive L2$ per core
AMD-2,2	*Part. 1:* (2@1.15GHz), *Part. 2:* (2@2.3GHz)	exclusive L3$ per core, NUMA among partitions
AMD-12,4	*Part. 1:* (12@1.15GHz), *Part. 2:* (4@2.3GHz)	fast cores share one L3$, slow cores share three L3$, NUMA among slow cores
AMD-4,4,2,2	*Part. 1:* (4@1.15GHz), *Part. 2:* (4@1.4GHz), *Part. 3:* (2@2GHz), *Part. 4:* (2@2.3GHz)	one L3$ per partition, NUMA among partitions

Note that although reuse-distance profiles for our test applications 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 created heterogeneity by setting cores to run at different speeds using DVFS. We created several test configurations, and in each configuration we had a number of partitions, each with its own frequency. The test configurations used in our tests are summarized in Table 1. In some configurations we used fewer cores than the total available (AMD-2,2, Intel-2,2) in order to avoid any performance effects due to cache sharing (to avoid this we had to use at most one core per chip). Conversely, configurations where most of the cores are used, AMD-12,4 and AMD-4,4,2,2, are subject to cache interference effects (each partition shared an L3 cache). The AMD machine has NUMA, and half the cores experienced more costly memory accesses on average due to asymmetric memory topology. We configured our platform such that fast cores experience *higher* latency. Since we prefer to assign memory-bound threads to slower cores, it is there that most of the memory traffic originates. Therefore it is more efficient to place slower cores closer to memory.

The benchmarks that we used for evaluation were from the SPEC CPU2000 suite. We have used two categories of workloads for most of our tests. The first category is *highly heterogeneous* (*HH*), and consists of a pair of highly CPU-bound benchmarks and a pair of memory-bound benchmarks. In this category we used three base workloads: (1) sixtrack, crafty, mcf and equake, (2) gzip, sixtrack, mcf and swim, and (3) mesa, perlbmk, equake and swim. In each of these workloads, the first two benchmarks are CPU-bound with virtually any cache size (and thus are good candidates for faster cores), and the second pair is memory-bound. The base workload was replicated in experiments where we wanted to run more than four threads per system. For instance, if we wanted to run sixteen threads, we would run four copies of each benchmark in the base workload.

The second category of workloads is *moderately heterogeneous* (*MH*). It consists of the following base workloads: (1) vortex, twolf, art and fma3d, (2) gap, parser, applu and vpr, (3) apsi, ammp, lucas and mgrid, and (4) bzip2, wupwise, gcc and art. Here the benchmarks represent the whole spectrum of memory-boundedness, with less extreme differences between the benchmarks. The behaviour of some benchmarks (art, for example) is highly dependent on cache availability, so their properties vary among different test configurations.

Due to space limitations, for the rest of the paper we focus only on the first workload from each category, and just summarize the other results.

We have also performed tests with a homogeneous workload LH (four instances of wupwise), where HASS, as expected, did not deliver any performance improvements since all benchmarks have similar signatures. Again, due to space limitations we do not analyze LH results in detail.

For a given test we launch a predetermined number of benchmarks, and as individual copies terminate, they are immediately restarted by a script. Thus we keep the workload constant and measure average completion time of every type of benchmark (for each test there were at least three completion time value samples, depending on the benchmark).

Our original goal was to compare completion times achieved with HASS to completion times achieved with the native Solaris scheduler, but we found that completion times under the native scheduler were highly variable (standard deviation was as high as 23% of the mean in some cases) and thus not suitable for comparison. This is due to the fact that the native scheduler is not het.-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 from one run to another. Achieving a low standard deviation is not possible in these conditions.

Instead we compare HASS completion times with two composite metrics. The first is the *default* metric. It is calculated by taking a weighted average of benchmark completion times when they were bound to a particular type of core, while the overall system load was as specified in the corresponding experiment. This metric gives us the expected completion time of a benchmark if it randomly binds to a core at the start and never switches. It is a good approximation of how the default scheduler operates, because if possible, it tries to keep the thread on its original core to maintain cache affinity. At the same time, this metric is too pessimistic for sustained loads, because as threads running on faster cores retire more often, faster cores will be available for assignment more often. To compensate, we also show a second *ideal round robin* (*ideal-RR*) metric. It is calculated by combining the 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. It illustrates a hypothetical scheduler that is perfectly fair and suffers no additional penalties or overhead compared to the default scheduler. A round-robin scheduler can be an approximation of such a perfect scheduler, but it suffers from additional non-trivial delays from numerous migrations, synchronization overheads and cold cache effects. 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 no better than the ideal-RR metric.

For performance comparison we report completion times normalized to the default metric for each benchmark and the geometric average for all benchmarks of that workload.

Performance Analysis

First we explored the behaviour of the algorithm when there is one runnable thread per core. The results are reported in Figures 2 and 3 (recall that we only illustrate the first workload in HH and MH). As can be expected, the scheduler performed especially well with the HH workload, where the average speedup was as much as 13% on AMD-2,2 (Figure 2a). This result is within 0.5% of the speedup on the *best static assignment*. A static assignment is a partition mapping for the workload that is decided at the beginning of execution and never changed thereafter. This assignment is obtained by testing all possible ones and picking the one with the best performance. To understand why HASS successfully matches the best static performance, we ran the experiments again and traced the execution with DTrace. Table 2

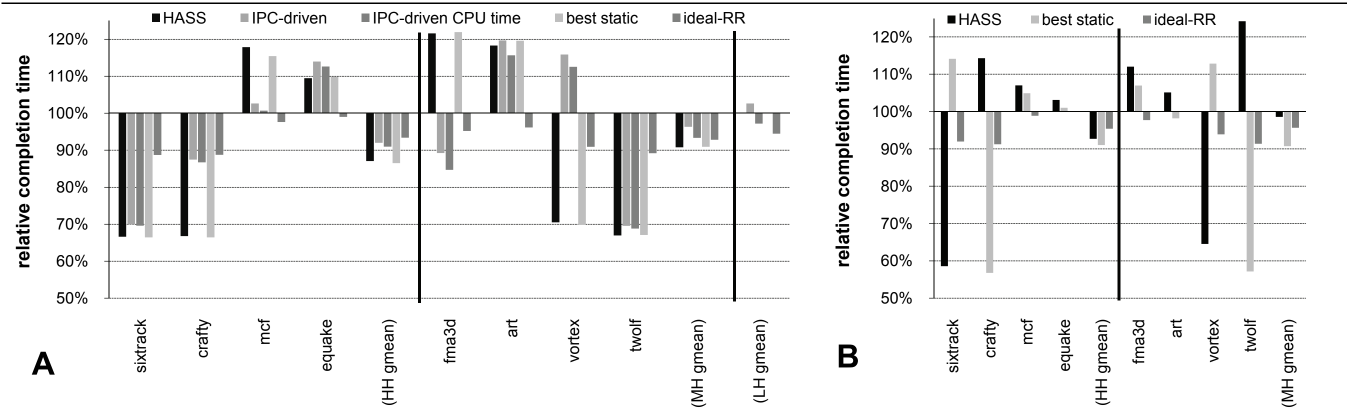

Figure 2. Average benchmark completion times relative to the default metric (results for IPC-driven are explained in 6.2): (a) AMD-2,2 – base workload, and (b) AMD-12,4 – base workload multiplied by 4 (16 benchmarks in total). Bars above 100% represent slowdown, and below 100% – speedup.

shows how HASS assigns threads to cores, and this assignment actually corresponds to the best static assignment.

On other two-partition setups results were also positive for the HH workload, although on Intel-2,2 (graph not shown), the boost was smaller (7% average speedup vs. default; 4.5% speedup vs. ideal-RR) due to a smaller difference between minimum and maximum frequencies on this system. AMD-12,4 too had a smaller speedup, but this time due to only a quarter of cores being fast, rather than half. Because HASS considered sixtrack to be the most CPU-bound benchmark according to its signature, it picked four copies of sixtrack to run on faster cores, while crafty was left to run on slower cores (Figure 2b).

MH workloads were more difficult to optimize. Since application behaviour was less extreme, the signatures were less distinctive, and there was more room for error in determining the best candidate to be placed on a faster or slower core. As Figure 2a shows, HASS was still able to correctly identify two memory-bound applications on AMD-2,2 (fma3d and art) and thus get a 9.2% speedup vs. default or 2% vs. ideal-RR, matching the best static assignment. With Intel-2,2, most of the benchmarks were actually fairly close together in memory-boundedness, so overall HASS lost 1% to ideal-RR, still gaining 2.5% over default. MH fared worse on AMD-12,4, where HASS identified vortex as the best candidate for faster cores, when, in fact, twolf was. Twolf was the best candidate, because by running in its own partition it (as well as other memory-bound benchmarks) could have had better cache performance due to less contention, but HASS could not detect this, because it does not account for cache sharing. As a result, performance slid below ideal-RR in both configurations (by 2.9%), although still retaining some margin over default.

HASS had trouble getting any speedup on AMD-4,4,2,2 (Figure 3). There were two reasons for this. Firstly, our simplified implementation of optimistic assignment did not optimize thread mappings among the three slower partitions (Table 2 shows that mcf spent more time in fast partitions than crafty, which was suboptimal). Secondly, cache sharing again disrupted the optimality of thread assignments, this time even in the workload HH (mcf being the victim of undue cache contention this time). Combined, these factors caused the performance to be no better than even the default metric, let alone the best static performance, which we do not report due to a large search space for such an assignment. This scenario showcases that awareness of shared caches is critical in configurations where they are present.

As to the remaining workloads from categories HH and MH, the average speedups were very similar to what we have shown: 12% and 7% compared to default for HH and MH workloads respectively on AMD-2,2; 7.7% and 3% on Intel-2,2; 9.5% and 4.9% on AMD-12,4; and –1% and 0.6% on AMD-4,4,2,2.

In the next round of experiments we tested the algorithm in overloaded conditions. At any given moment we had five times as

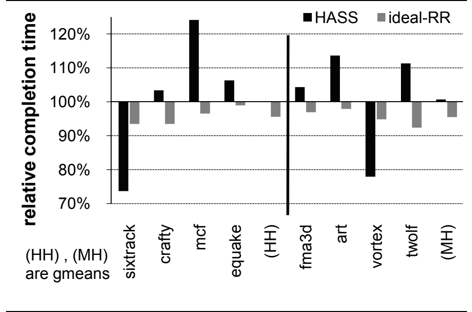

Figure 3. Average completion times on AMD-4,4,2,2, relative to the default metric. Base workload multiplied by 3 (12 in total).

Table 2. HASS thread to partition assignments.

	% of time spent in different partitions (ordered from slow to fast)					
	Intel-2,2	AMD-2,2	AMD-12,4	AMD-4,4,2,2	AMD-2,2 overloaded	AMD-4,4,2,2 overloaded
sixtrack	0.9 / 99	6.9 / 93	1.2 / 98	13 / 5.8 / 13 / 67	0 / 100	0 / 0 / 44 / 56
crafty	1.8 / 98	8.5 / 91	99 / 1.0	60 / 17 / 21 / 1.9	0 / 100	0 / 39 / 36 / 25
mcf	98 / 1.5	95 / 4.4	99 / 0.2	0.1 / 8.5 / 65 / 26	100 / 0	99 / 1.2 / 0 / 0
equake	98 / 1.1	88 / 12	99 / 0.5	40 / 36 / 22 / 0.5	90 / 10	14 / 86 / 0.2 / 0.1
fma3d	94 / 5.5	93 / 7.0	98 / 2.0	43 / 28 / 26 / 2.5	77 / 23	8.0 / 66 / 26 / 0.2
art	90 / 9.1	89 / 11	99 / 0.2	23 / 70 / 6.3 / 0.3	100 / 0	99 / 0.7 / 0.2 / 0.2
vortex	7.5 / 92	9.7 / 90	2.9 / 97	18 / 10 / 7.1 / 64	0.1 / 100	0 / 0 / 16 / 84
twolf	6.6 / 93	8.2 / 91	99 / 0.6	49 / 24 / 25 / 0.8	0 / 100	7.4 / 65 / 28 / 0.2

Table 3. HASS overheads. Values in parentheses are maxima observed for any one thread.

	Partition assignment logic (% of CPU time)		# of partition migrations per thread per minute of CPU time	
	HH	MH	HH	MH
AMD-2,2	0.020	0.021	0.50 (1.36)	0.45 (0.97)
AMD-12,4	0.011	0.011	0.29 (1.03)	1.78 (4.69)
AMD-4,4,2,2	0.013	0.014	1.12 (2.48)	8.19 (12.3)
AMD-2,2 ovrld	0.026	0.026	0.43 (0.91)	0.17 (0.84)
AMD-4,4,2,2 ovrld	0.019	0.028	0.77 (2.31)	0.84 (1.81)

many threads running as there were cores available. Unfortunately, the native Solaris load balancer and scheduler were not successful in sharing the CPU fairly among so many competing threads, and so the completion times were inconsistent in both the native and HASS schedulers (which use these default mechanisms for intra-partition scheduling). Nevertheless, we were able to evaluate optimality of partition assignments (Table 2), which was in fact reasonable: CPU-bound benchmarks were spending more time on faster cores and faster partitions received higher loads.

In summary, the results demonstrate that HASS is able to differentiate among benchmarks with different architectural properties and assign them to the "right" types of cores, especially when the workload is highly heterogeneous. Moreover, HASS is highly unlikely to perform worse than the default metric even for homogeneous workloads, where performance improvements are difficult to obtain. The biggest performance issue we found was due to HASS being unaware of shared caches, a problem that we expect to address in future work. We defer discussion regarding the lack of phase awareness in HASS until later in the paper, when we have had a chance to evaluate the IPC-driven algorithm.

HASS is unfair by design – it assigns to fast cores those jobs that experience the most speedup on those cores, and thus may not be appropriate in situations when memory-bound threads must be run at a higher priority. HASS can be extended to be more priority-aware by introducing priorities into its swapping and assignment mechanisms. For example, a higher priority thread would be able to switch to a more optimal partition regardless of the overall system throughput. This would work well only with fixed priorities (as opposed to those that change regularly), and potentially compromise throughput, because high-priority threads would be allowed to run in the fast partition even if they are not sensitive to the CPU speed. We have not implemented this improved technique, and have instead opted for a simpler algorithm. In scenarios where improving the overall system efficiency is the primary goal, the basic version of our algorithm is able to deliver improved performance per watt by increasing the overall throughput of the workload.

Scalability Analysis

Scalability was one of the main emphases in HASS' design, and so we evaluated how its overhead scales as setup complexity grows. The most superficial overhead measurement, the difference between a thread's wallclock completion and CPU times, was comparable to that of the native scheduler (between 0.1% and 0.3% depending on the configuration). We did not consider overloaded tests, because there we cannot distinguish between legitimate runqueue waiting time and overhead. HASS overheads were slightly higher in some configurations with workload MH, due to more inter-partition migrations, which can sometimes leave cores idle for short periods of time.

We then focused on a part more specific to our algorithm: the time spent executing HASS partition assignment logic (regular and optimistic). This is the part that we would expect to grow as setup complexity increases. The results, however, showed that at 16 cores the overhead was insignificant (Table 3). The maximum overhead observed for any single thread was 0.06% of its CPU time (not shown). Overall, the range of values indicates that partition assignment is not a likely bottleneck, at least as we move into medium-scale setups.

Lastly, we have traced rates of inter-partition thread migration. This metric reflects how quickly the workload stabilizes once a disruption occurs, and whether there is unnecessary thrashing (also Table 3). The numbers indicate that overall the algorithm is not prone to thrashing, except in cases where the workload is not very heterogeneous, but the effect was still relatively benign in our case. We postulate that this thrashing may grow as the complexity of the setup grows, indicating a possible area of improvement: a mechanism to throttle some migrations.

Sensitivity to Varying Inputs

One potential area of concern for HASS is a scenario where an application is run with an input different than the one assumed by its signature (recall that there is only one signature per application). Since benchmarks in SPEC CPU2000 are available with different inputs, we were able to investigate this effect. Ideally, we want the performance estimated with a signature for a particular input to be about the same as the performance estimated with signatures for all other inputs. To see whether this is so, we compared estimated performance ratios using signatures obtained on different inputs of the same program. We constructed signatures for all train and ref inputs of 23 SPEC benchmarks (eon, galgel, and facerec had compilation issues and we did not

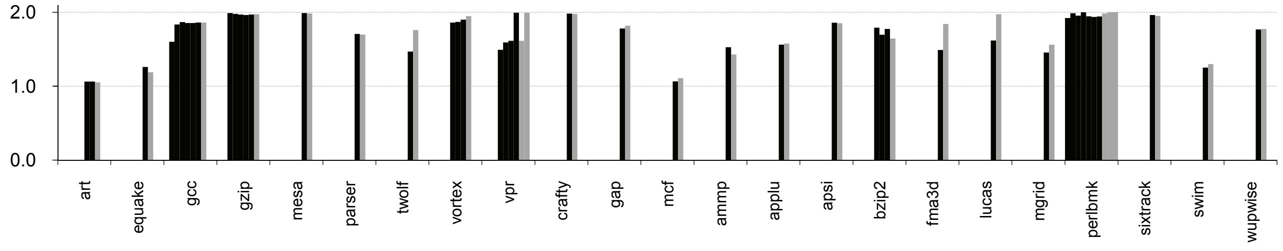

Figure 4. Estimated performance ratios for AMD-12,4 given a distinct signature for each input. Black bars represent ref sets, and gray bars represent train sets.

use them for the paper). We estimated performance with each signature on our configurations, and then computed performance ratios between fast and slow cores (recall that such ratios are used by HASS for thread assignments). As an illustration we present the results for the AMD-12,4 configuration (1.15GHz vs. 2.3 GHz, cache size 512KB) in Figure 4. Each set of bars shows ratio estimates for different inputs of the same application. Each estimate is done with its own signature. Ideally, all bars for a given application should be the same; then we know that a single signature can be used regardless of input. While it is evident that the estimated ratio varies depending on the input, the effect is not significant in all benchmarks. In other words, cache miss rates (on which we base our signatures) do not always vary significantly between inputs. Thus HASS is likely to retain at least some accuracy in the presence of varying inputs.

There has been other research related to the problem of miss rate estimation for various inputs given a profile for only one input [8], but we have not attempted to incorporate these approaches into the current version.

Extension to Multithreaded Applications

Although the signature-based framework 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, or in cases where a thread switches between various heterogeneous tasks (as in newly emerging parallel environments), the signature could be generated per task and communicated to the OS upon the beginning of each new task. To minimize the overhead, the signature could be propagated to the OS only if it is sufficiently different from the previous signature associated with that thread. While support for multithreaded applications would require changes to the signature framework, 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.

6. IPC-driven Algorithm

We wanted to compare HASS side by side against a phase-aware dynamic algorithm based on online profiling of threads. Our goal was to evaluate the benefits of phase awareness and drawbacks (if any) of online monitoring. We were not aware of any implementation of such an algorithm in a real system, and therefore decided to implement such an algorithm ourselves. We chose the IPC-driven scheduler proposed previously by Becchi [3], an algorithm that combined good results, applicability to general purpose systems and specification completeness. We created the first implementation of the IPC-driven algorithm in OpenSolaris (in the original work [3] the scheduler was simulated) and report our evaluation results in this section.

The Algorithm

The IPC-driven algorithm assumes two types of cores ("fast") and ("slow"). The assignment is done based on IPC ratios, which determine the relative benefit of running a thread on a particular core type. A thread with a high IPC ratio between fast and slow cores is expected to benefit from the fast core. The scheduler periodically samples threads' IPC on both cores types and compares the smallest IPC ratio on the fast core with the highest IPC ratio on the slow one. If the latter exceeds the former, the corresponding threads are swapped, so that the thread with the higher IPC ratio is set to run on the fast core. Even though the original algorithm assumed only two types of cores, our implementation is a generalization for `n` different types. As in HASS, cores are organized into partitions according to their types. The scheduler measures IPC ratios in all partitions relative to the current partition.

Whenever a program enters a new IPC phase, the IPC ratios relative to all partitions are re-measured. This is done via `forced migrations` where a thread is switched to run in every partition other than its current one for a period of time called `refresh_period`.

IPC phase changes, which are independent of the type of core [20], are detected through sudden changes in the moving average of IPC that exceed a certain `ipc_threshold`.

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 in a *pinned* state. A thread whose IPC ratio is in the process of being updated is *refreshing*.

Performance of the IPC-driven algorithm is sensitive to the settings of the aforementioned parameters (`refresh_period`, `swap_inactivity` period, etc.). 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 milliseconds, `ipc_threshold` to 10%, and `swap_inactivity` period to 1.5 seconds.

Evaluation

We use the same experimental conditions as in Section 5, but due to space limitations we refrain from presenting the entire set of our experiments, highlighting only those that best illustrate our findings about the algorithm.

Figure 2a shows the results for three workloads: HH {`sixtrack, crafty, mcf, equake`}, MH {`vortex, twolf, art, fma3d`} and LH {`wupwise, wupwise, wupwise, wupwise`} on AMD-2,2. In addition to reporting completion time, we also show user CPU time. The difference between the two is overhead: system time plus runqueue wait time.

For the HH workload, we note that IPC-driven improves performance over default and fair by 8% and 1.5% respectively. Contrary to our expectations, however, IPC-driven achieves a smaller speedup than both HASS and the best static assignment. We expected it to improve over them, since it is phase aware. The reason that it did not is as follows.

The best static assignment would map the two frequency-sensitive applications `sixtrack` and `crafty` to the two fast cores, and the two memory-bound applications `mcf` and `equake` to the two slow cores. IPC-driven, on the other hand, maps `mcf` to the fast core roughly 51% of the time, pushing `crafty` to run on the slow core during this time. Although `mcf` does have some high-IPC phases when it makes sense to map it to the fast core (see Figure 5), those phases last only 25% of the time, not 51%. So 26% of the time `mcf` is not being mapped to the right core, which hurts performance.

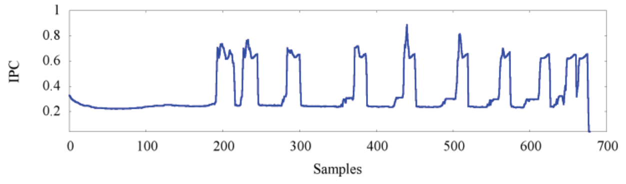

Figure 5. IPC over time for `mcf` on Intel Quad Xeon at 2.0 GHz.

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 (towards low 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 IPCs measured on fast and slow cores during the *same* program phase. When this is not the case, the IPC ratio is inaccurate, and in this particular example it is much higher than it should be. In fact, we saw ratios as high as 2.2 and 2.5, which are not realistic, because the ratio of high and low frequencies in our configurations is at most 2. Since the ratio is so high, the algorithm decides that mcf derives significant benefit from running on the fast core and pins it there, when in fact it would be more optimal to pin it to the slow core.

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 partitions, and hence the number of IPCs, is large (recall that the ratio has to be computed for each partition). We tried to solve this problem by increasing the ipc_threshold, but this did not work, because no single threshold worked well for all workloads. The reason why this problem did not occur in the original evaluation of the IPC-driven algorithm [3] is that IPC refreshing was not simulated realistically. IPCs used to compute ratios were obtained from offline IPC traces, so, in contrast with real systems, IPCs always corresponded to the same program phase.

Turning again to Figure 2a, we note that in all workloads the algorithm experiences significant overhead, causing LH workload to perform worse than default, even though it beats it in terms of user CPU cycles. The overhead is caused by forced migrations. When a thread is migrated to a new core to refresh its IPC, the threads pinned to that core experience longer waiting times. One solution would be to swap the refreshing thread with one of the pinned threads. We did not pursue this solution after considering its complexity and potential degradation of cache affinity.

Migration overhead is particularly large for MH and LH workloads due to very frequent migrations caused by frequent phase changes, every 500-600 milliseconds, in fma3d (MH) and wupwise (LH). Increasing ipc_threshold and swap_inactivity period alleviates this problem, but at the expense of making the algorithm less phase-aware.

Migration overhead was not detected in the original paper on the IPC-driven algorithm, perhaps because runqueue contention was modelled differently than in a real scheduler. The paper did not provide sufficient detail about this part of the simulation.

Summary

Our real-world implementation of the IPC-driven algorithm revealed that IPC ratios were often inaccurate due to the unstable nature of phase changes, and that the overhead of forced migrations in highly phased workloads was prohibitively high. In workloads with infrequent and smooth phase changes the IPC-driven algorithm performs better. One such "friendly" workload consisting of {sixtrack, crafty, equake, equake} experiences less than 1% overhead and its performance is only 2% worse than the best static assignment. At the same time, with weakly phased workloads, IPC-driven does not benefit from phase awareness, which renders unjustified the implementation complexity associated with this feature. Although previous work on IPC-driven and similar het.-aware algorithms demonstrated through simulation that phase-aware dynamic assignment significantly outperforms the best static assignment [3][14], we were not able to achieve the same results on a real system due to the aforementioned difficulties. Solving these problems would require major changes to the dynamic algorithm that we evaluated, which was outside the scope of our work.

The difficulties that we encountered while trying to implement a dynamic phase-aware algorithm raise a question whether the benefits of a phase-aware algorithm would be worth the effort expended in overcoming the difficulties associated with its implementation. While we do not have our own data to reason about this trade-off, data presented by Becchi et al.[3] and Kumar et al.[15] can facilitate this analysis. The data in the Becchi's paper suggests that the largest portion of the speedup effected by the IPC-driven algorithm comes from the best static assignment (approximated by our algorithm) and from the fact that the fast core is never kept idle (also ensured by our algorithm). Beyond that, the actual IPC-driven assignment offers little additional performance improvement. Kumar's data concurs with this finding: it shows that a large portion of the speedup on heterogeneous systems can be obtained with the best static assignment, and additional gains from phase-aware dynamic assignments are smaller on average.

This suggests that the static strategy adopted in our approach could be the right trade-off between performance gains on heterogeneous multicore systems and complexity of the system software required to effect these gains.

That said, there certainly might be workloads and applications where phase-awareness is crucial. In those cases, our static approach could perhaps be extended to be more phase-aware without having to detect phases online. For example, phase detection could be done offline (using a profiler or with the assistance of a programmer), a signature generated for each phase, and markers inserted in the executable at the points of phase change. Then during runtime as the program changes phases, the correct signature would be provided to the OS. While this mechanism complicates the signature framework, its implementation would be relatively straightforward. Evaluating the feasibility of this approach is in our plans for future work.

7. Conclusions and Future Work

We have described a new het.-aware scheduler, HASS. Its novelty is in using offline-generated architectural signatures for determining thread assignments on ASISA multicore processors. Using signatures removes a need for dynamic profiling, which causes load imbalance and results in unstable performance estimates on highly phased workloads. HASS achieves performance comparable to the best static assignment and outperforms default and fair heterogeneity-agnostic assignments most of the time. In cases where HASS mispredicts relative performance, it usually still improves over default in our experiments. Our analysis shows that the error rate of the signature-based performance prediction is small, so we expect HASS to perform well overall. Two potential limitations of HASS, the inability of the signature to adjust for different program inputs and lack of consideration for phase changes, do not appear to be significant roadblocks. Our analysis of the effect of input variation on the accuracy of signature-based performance shows that it is not too detrimental and should not negate the overall benefits of the algorithm.

Lack of phase awareness in HASS does not prevent it from improving performance, because a static signature captures the overall trends in the behaviour of the application. Thus HASS is able to achieve performance comparable to the best (oracle) static assignment. Contrary to our expectations, we found that lack of phase awareness in HASS has worked to its advantage, because it saved it from many problems linked to phase changes that were revealed by our implementation of IPC-driven algorithm.

As to the IPC-driven algorithm, we initially expected that it would outperform HASS on highly phased workloads, but were surprised to find that phase awareness caused so many problems. While we do not claim that phase awareness is problematic in scheduling algorithms in general, it was problematic in the IPC-driven algorithm. This was due to the necessity to compute a stable IPC ratio, which was impossible because of inherently unstable nature of phases, and because phase changes triggered devastating migrations that hurt performance. Our experience with the IPC-driven algorithm underscored the importance of validating simulated results on real systems.

Another potential improvement is to make HASS account for other types of heterogeneity (e.g., differences in the pipeline architecture) and adapt it to support multithreaded applications. These important problems, as well as support for architectures with cache sharing and alternative methods of phase awareness are in our plans for future work.

References

[1] K. Asanovic et al. *The Landscape of Parallel Computing Research: A View from Berkeley.* UC Berkeley Technical Report UCB/EECS-2006-183, 2006.

[2] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. *The Impact of Performance Asymmetry in Emerging Multicore Architectures*. In Proceedings of the 32nd Annual International Symposium on Computer Architecture (Madison, Wisconsin USA, June 04–08, 2005). ISCA '05. IEEE Computer Society, Washington, DC, USA, 506–517.

[3] M. Becchi and P. Crowley. *Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures*. In Proceedings of the 3rd Conference on Computing Frontiers (Ischia, Italy, May 02–05, 2006). Computing Frontiers '06. ACM, New York, NY, USA, 29–40.

[4] E. Berg and E. Hargersten. *StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis*. In Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (Austin, Texas, USA, March 10–12, 2004). ISPASS '04. IEEE Computer Society, Washington, DC, USA, 20–27.

[5] S. Borkar. *Thousand Core Chips—A Technology Perspective*. In Proceedings of the 44th Annual Conference on Design Automation (San Diego, California, USA, June 04–08, 2007). DAC '07. ACM, New York, NY, USA, 746–749.

[6] B. Cantrill, M. Shapiro, and A. Levinthal. *Dynamic Instrumentation of Production Systems*. In Proceedings of the USENIX Annual Technical Conference (Boston, MA, USA, June 27–July 02, 2004). USENIX '04. USENIX Association, Berkeley, CA, USA, 2.

[7] D. Chandra, F. Guo, S. Kim, and Y. Solihin. *Predicting Inter-Thread Cache Contention on a Multi-Processor Architecture*. In Proceedings of the 11th International Symposium on High-Performance Computer Architecture (San Francisco, California, USA, February 12–16, 2005). HPCA '05. IEEE Computer Society, Washington, DC, USA, 340–351.

[8] C. Ding, Y. Zhong. *Predicting Whole-program Locality through Reuse Distance Analysis*. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (San Diego, California, USA, June 09–11, 2003). PLDI '03. ACM, New York, NY, USA, 245–257.

[9] V. Freeh et al. *Analyzing the Energy-Time Trade-Off in High-Performance Computing Applications*. IEEE Transactions on Parallel and Distributed Systems, 18, 6 (June 2007). IEEE Press, Piscataway, NJ, USA, 835–848.

[10] M. Hill and M. Marty. *Amdahl's Law in the Multicore Era*. IEEE Computer, 41, 7 (July 2008). IEEE Computer Society Press, Los Alamitos, CA, USA, 33–38.

[11] M. Hill and A. Smith. *Evaluating Associativity in CPU Caches*. IEEE Transactions on Computers, 38, 12 (December 1989). IEEE Computer Society, Washington, DC, USA, 1612–1630.

[12] K. Hoste and L. Eeckhout. *Microarchitecture-Independent Workload Characterization*. IEEE Micro, 27(3), 2007. IEEE Computer Society Press, Los Alamitos, CA, USA, 63–72.

[13] E. Humenay, D. Tarjan, and K. Skadron. *Impact of Process Variations on Multicore Performance Symmetry*. In Proceedings of the Conference on Design, Automation and Test in Europe (Nice, France, April 16–20, 2007). DATE '07. EDA Consortium, San Jose, CA, USA, 1653–1658.

[14] R. Kumar et al. *Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction*. In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (San Diego, California, USA, December 03–05, 2003). MICRO '03. IEEE Computer Society, Washington, DC, USA, 81.

[15] R. Kumar et al. *Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance*. In Proceedings of the 31st Annual International Symposium on Computer Architecture (München, Germany, June 19–23, 2004). ISCA '04. IEEE Computer Society, Washington, DC, USA, 64.

[16] T. Li, D. Baumberger, D. A. Koufaty, and Scott Hahn. *Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures*. In Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (Reno, Nevada, USA, November 10–16, 2007). SC '07. ACM, New York, NY, USA, No. 53.

[17] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, K. Hazelwood. *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation*. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (Chicago, Illinois, USA, June 11–15, 2005). PLDI '05. ACM, New York, NY, USA, 190–200.

[18] J. Mogul et al. *Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems*. IEEE Micro, 28, 3 (May 2008). IEEE Computer Society Press, Los Alamitos, CA, USA, 26–41.

[19] D. Shelepov and A. Fedorova. *Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures*. In Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with the 35th International Symposium on Computer Architecture (Beijing, China, June 21–25, 2008). WIOSCA '08.

[20] T. Sherwood, S. Sair, and B. Calder. *Phase Tracking and Prediction*. In Proceedings of the 30th Annual International Symposium on Computer Architecture (San Diego, California, USA, June 09–11, 2003). ISCA '03. ACM, New York, NY, USA, 336–349.

[21] A. Smith. *A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory*. IEEE Transactions on Software Engineering, 4, 2 (March 1978). IEEE Press, Piscataway, NJ, USA, 121–130.

[22] R. Teodorescu and J. Torrellas. *Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors*. In Proceedings of the 35th International Symposium on Computer Architecture (Beijing, China, June 21–25, 2008). ISCA '08. IEEE Computer Society, Washington, DC, USA, 363–374.