AASH: An Asymmetry-Aware Scheduler for Hypervisors

Vahid Kazempour Ali Kamali Alexandra Fedorova
Simon Fraser University, Vancouver, Canada
{vahid_kazempour, ali_kamali, fedorova}@sfu.ca

Abstract

Asymmetric multicore processors (AMP) consist of cores exposing the same instruction-set architecture (ISA) but varying in size, frequency, power consumption and performance. AMPs were shown to be more power efficient than conventional symmetric multicore processors, and it is therefore likely that future multicore systems will include cores of different types. AMPs derive their efficiency from core specialization: instruction streams can be assigned to run on the cores best suited to their demands for architectural resources. System efficiency is improved as a result. To perform effective matching of threads to cores, the thread scheduler must be asymmetry-aware; and while asymmetry-aware schedulers for operating systems are a well studied topic, asymmetry-awareness in hypervisors has not been addressed. A hypervisor must be asymmetry-aware to enable proper functioning of asymmetry-aware guest operating systems; otherwise they will be ineffective in virtual environments. Furthermore, a hypervisor must ensure that asymmetric cores are shared among multiple guests in a fair fashion or in accordance with their priorities.

This work for the first time implements simple changes to the hypervisor scheduler, required to make it asymmetry-aware, and evaluates the benefits and overheads of these asymmetry-aware mechanisms. Our evaluation was performed using an open source hypervisor Xen on a real multicore system where asymmetry was emulated via CPU frequency scaling. We compared the asymmetry-aware hypervisor to default Xen. Our results indicate that asymmetry support can be implemented with low overheads, and resulting performance improvements can be significant, reaching up to 36% in our experiments. Most performance improvements are derived from the fact that an asymmetry-aware hypervisor ensures that the fast cores do not go idle before slow cores and from the fact that it maps virtual cores to physical cores for asymmetry-aware guests according to the guest's expectations. Other benefits from asymmetry awareness are fairer sharing of computing resources among VMs and more stable execution times.

Categories and Subject Descriptors D.4.1 [*Process Management*]: Scheduling

General Terms Algorithms, Design, Experimentation, Management, Measurement.

Keywords multicore processors, asymmetric, heterogeneous, scheduling algorithms, hypervisor, virtual machine monitor

VEE'10, March 17–19, 2010, Pittsburgh, Pennsylvania, USA.

1. Introduction

Asymmetric multicore processors (AMP) consist of cores exposing the same instruction-set architecture (ISA) but delivering different performance [1, 11]. The cores of an AMP system differ in clock frequency, power consumption, and possibly other microarchitectural features. A typical asymmetric processor would consist of several *fast* cores (large area, high clock frequency, complex out-of-order pipeline, high power consumption) and a large number of *slow* cores (small area, low clock frequency, simple pipeline, low power consumption).

Compared to symmetric multicore processors (SMP), AMPs better cater to diversity of the workload (in terms of demand for architectural resources), and in doing so they can deliver better performance per watt and per area [7, 11, 14]. For example, fast cores are best suited for CPU-intensive applications that can efficiently utilize these cores' "expensive" features, such as the superscalar pipeline and the multiplicity of functional units. Slow cores, on the other hand, can be dedicated to applications that use the CPU inefficiently, for example as a result of frequent stalls on memory requests: these memory-intensive applications can run on slow cores without significant performance loss relative to fast cores, but consuming much less energy [11]. This *specialization* of computing resources, which is typically aided by an asymmetry-aware thread scheduler [3, 12, 19], promises to make AMP systems significantly more energy efficient than SMP systems (one study reported up to 60% energy savings [11]). For this reason they are an extremely attractive platform for data centers, where energy consumption is a crucial concern [9, 18].

Unfortunately, virtualization software, which is typically used in data centers to provide safe multiplexing of hardware resources, is not asymmetry-aware. This prevents using AMP systems in an effective way. To ensure that asymmetric hardware is well utilized, the system must match each thread with the right type of core: e.g., memory-intensive threads with slow cores and compute-intensive threads with fast cores. This can be accomplished by an asymmetry-aware thread scheduler in the guest operating system [17, 3, 12, 19], where properties of individual threads can be monitored more easily than at the hypervisor level. However, if the hypervisor is not asymmetry-aware it can thwart the efforts of the asymmetry-aware guest OS scheduler, for instance if it consistently maps the virtual CPU (vCPU) that the guest believes to be "fast" to a physical core that is actually slow.

An equally important goal is supporting fair sharing of fast cores. This is relevant for both asymmetry-aware guests as well as for asymmetry-unaware (legacy) guests. As the number of cores on a single chip increases, scenarios where multiple VMs run on the same hardware will be more common. In these cases, providing equal sharing of resources with varying energy costs and performance characteristics will be essential for delivering stable performance and fair accounting of CPU utilization.

Complementary to fair sharing, is the ability to give a particular VM (or VMs) a higher priority in using scarce and "expensive" fast cores: in scenarios where one virtual machine is more "important" than others a hypervisor should provide this flexibility. Finally, when performance is the top concern, an asymmetry-aware hypervisor must ensure that fast cores do not go idle before slow cores. We found that this feature alone can lead to significant performance gains.

The goal of our work is the design, implementation and evaluation of asymmetry awareness in a virtual machine hypervisor. We are not aware of previous attempts to address this subject. In conducting our study we were interested in answering the following question: *How to implement efficient mechanisms for asymmetry awareness in hypervisors and to what extent do they affect performance and efficiency of virtual machines running atop asymmetric systems?* To that end, we implemented the following asymmetry-aware features in the Xen hypervisor: fair sharing of fast cores among all vCPUs in the system; support for asymmetry aware guests; a mechanism for controlling priority of VMs in using fast cores; a mechanism ensuring that fast cores never go idle before slow cores. These mechanisms enable better performance relative to an asymmetry-unaware hypervisor and avoid the undesirable consequences of asymmetry unawareness.

Our mechanisms for asymmetry distribute CPU cycles on fast and slow cores among vCPUs according to the fairness policy, priorities, and other considerations. This is accomplished by periodic migrations of vCPUs among physical cores of different types. To avoid performance loss, these migrations must be infrequent. Cross-core migrations cause the vCPUs to lose the cache state accumulated on the old core, and the performance penalty due to this loss will be especially large if the abandoned state belongs to a last-level cache, such as the L2 or L3 cache. The migrations in our system are performed not more frequently than once every 30 milliseconds per vCPU (for non-I/O-bound workloads), and so the overheads from migration are small.

We evaluated our system AASH – an Asymmetry Aware Scheduler for Hypervisors implemented in Xen – on real multicore hardware where asymmetry was emulated by setting the cores to run at different clock frequencies. As the baseline for comparison we used the default Xen hypervisor, whose scheduler is not asymmetry-aware. For evaluation we used primarily scientific applications, focusing on those that perform little I/O, since these applications are especially sensitive to optimizations related to allocation of CPU resources. Furthermore, scientific applications are increasingly executed in data centers via so-called "cloud" services [6, 20] or other initiatives, such as West Grid. Our evaluation provides insight into potential impact of asymmetry-awareness in hypervisors. We found these benefits to be quite significant. We observed performance improvements (of up to 36%) relative to an asymmetry-unaware hypervisor, reduced variance in execution time, and a fairer distribution of computing resources among VMs. At the same time, performance degradation from asymmetry support was small, never exceeding 3%. We conclude that (1) neglecting asymmetry awareness in hypervisors can lead to measurable performance sacrifices for both single-VM and multi-VM workloads, regardless of whether the guest is asymmetry-aware or not; (2) asymmetry awareness can be implemented via simple and effective mechanisms, and thus the cost of supporting it is low.

The rest of this paper is organized as follows. Section 2 describes the components of asymmetry-aware aware support available in AASH. Section 3 describes the implementation of AASH in Xen. Section 4 presents the experimental results. Section 5 discusses related work. Section 6 summarizes our work and presents conclusions.

2. Features of AASH

We assume a system with two core types: fast and slow. Fast cores are typically characterized by a large area, high clock frequency, complex superscalar out-of-order pipeline and high power consumption. Slow cores typically use less area, have a lower clock frequency and a relatively simple pipeline, and consume a lot less power. The reason for assuming only two core types is that this structure is mostly likely to be adopted in future AMP systems. According to a study by Kumar et al. [11], supporting only two core types is sufficient for achieving most of the potential of asymmetric designs.

A scheduler running on asymmetric hardware must provide several features unique to this type of systems. If the goal of the user is to maximize the system's overall efficiency, the scheduler must ensure that the cores of different types are allotted to threads that use these cores most efficiently. If fairness is desired, the cores of different types should be shared among the running entities equally. Finally, if some VMs are more "important" than others, the scheduler should ensure that they receive a higher share of premium fast cores. Our scheduler AASH includes the mechanisms addressing these goals.

Asymmetry-aware thread schedulers in operating systems typically monitor individual threads to determine their relative benefit, or efficiency, in running on cores of different types [3, 12, 19]. We believe that these thread-aware algorithms belong to the operating system rather than a hypervisor. Implementing them in a hypervisor requires monitoring thread context switches within the guest OS (in order to learn properties of individual threads), and is thus cumbersome. Furthermore, if the OS is already performing asymmetry-aware scheduling there is no need to replicate this work in the hypervisor. As a result, we decided that instead of copying asymmetry-aware OS algorithms in the hypervisor, the hypervisor should provide the support for asymmetry-aware guests necessary for them to carry out their policies.

In the rest of this section we describe the mechanisms in AASH that serve to support asymmetry. There are three key mechanisms: (1) support for fair sharing of fast physical cores (and by extension of slow cores) among virtual CPUs, (2) support for asymmetry-aware guest operating systems, and (3) support for priorities in allocating fast-core cycles among vCPUs.

2.1 Fair sharing of physical fast cores among virtual CPUs

Hypervisors typically run multiple guests on the same hardware. Fair and predictable sharing of hardware resources is crucial in this environment. First of all, it simplifies accounting when CPU cycles have unequal "cost" depending on if they were used on a fast or on a slow core. Second, lack of predictable distribution of CPU time on fast and slow cores leads to unpredictable and unstable performance [2]. To aid in resolution of these problems, AASH ensures that the fast physical cores are shared equally among all virtual CPUs. In our system, a VM's share of fast-core cycles is proportional to the number of vCPUs that this VM contains, and each vCPU within a given VM gets the same share of fast-core cycles as other vCPUs in the system. An alternative way to share fast-core cycles is to divide them evenly among VMs rather than vCPUs, and this change can be easily made in our scheduler. In the current implementation, however, we decided to divide the fast-core cycles among vCPUs, since in that case the fast-core cycles are given to a VM in proportion of the computing resources it demands (its number of vCPUs).

In addition to improving fairness, fair sharing of fast cores improves performance of certain parallel applications, because it equally accelerates all virtual CPUs. As a result, resources are distributed among the threads in a more balanced fashion.

In addition to fairly sharing fast cores, AASH ensures that fast cores do not go idle before slow cores.

2.2 Support for asymmetry-aware guest operating systems

AASH ensures deterministic mapping of virtual CPUs of a particular type to physical cores of the same type. In a virtualized system running multiple guests, each guest is allotted a fair share of cycles on fast physical cores. These fast-core cycles are allocated to the vCPU that the guest considers to be "fast", and the excess, if any, is allocated to the vCPUs considered "slow". If multiple "fast" vCPUs are competing for physical fast cores, and if the number of "fast" vCPUs exceeds the number of fast physical cores, the fast-core cycles will be equally shared among those vCPUs. In that case, a "fast" virtual CPU cannot be mapped to the fast physical core 100% of the time to ensure that other competing vCPUs receive a fair share of their cycles on fast cores, and as a result will be mapped for part of the time to a slow physical core.

Another useful part of support for asymmetry-aware guests would be the capability enabling the guests to discover the types of underlying physical cores. This can be done by virtualizing access to model specific registers (MSR), which are typically used to discover the features of CPUs. Implementation of this mechanism can be performed more comprehensively when actual AMP systems become available, and so we defer this task for future work.

2.3 Support for priorities in using fast cores

In virtual environments where multiple VMs share the same hardware some VMs may be considered more "important" than others, and so AASH allows giving to these VMs a higher priority in allocation of fast-core cycles. We designed two priority mechanisms. The first one is a *coarse priority system* where VMs can be classified either as high-priority or low-priority. In that case, high-priority VMs will be assigned fast-core cycles first, and if several high-priority VMs are present, they will share these cycles equally. The remaining fast-core cycles, if any, will be allotted to other virtual machines.

The second mechanism is where a VM with a low number of active vCPUs receives a higher priority in running on fast cores than VMs with a large number of active vCPUs. This is relevant for scenarios where a VM is used as a container for a single application, as is commonly the case in virtual environments. In this case, a low number of active vCPUs indicates that the VM is running code with a low degree of parallelism, or sequential code[1]. Such VMs are given priority on fast cores, because this can deliver a greater benefit to the application's overall performance. Consider scheduling on asymmetric system a VM running a parallel application. Only a small subset of threads of this parallel application can scheduled on fast cores at any given moment, because the number of fast cores is usually small relative to the total number of cores in the system. The application as a whole, therefore, receives only a very small performance improvement. On the other hand, for applications with only one or a handful of threads, running on a fast core as opposed to a slow core typically results in performance improvement proportional to the speed of the fast core relative to the slow core. For parallel applications with sequential phases, this mechanism will *accelerate* the sequential phase on the fast core, reducing the cost of the serial bottleneck [7, 1].

In accelerating VMs with a low number of active vCPU on fast cores, the system assumes that the application's phases of low parallelism will be exposed to the hypervisor. This would occur if unused application threads are blocked, causing the vCPUs where they were running to go idle. Idle vCPUs are visible to the hypervisor, so it can react when the number of active threads, and thus active vCPUs, decreases below a threshold. If an application is designed such that unused threads busy-wait on the CPU as opposed to releasing it, this method, which relies on detecting the change in the number of active virtual CPUs, will not detect phases of low parallelism. In that case an asymmetry-aware guest OS that interacts with an application runtime environment is required [16].

Another way to implement this prioritization scheme is to calculate the fast core cycles for each VMs in inverse proportion to the number of that VM's active vCPUs. We found, however, that this approach would require excessively frequent redistribution of fast-core cycles if the number of active vCPUs changes frequently. Instead, it is better to use an active vCPU threshold, where VMs with an active vCPU count under the threshold are considered to have low parallelism and thus get a high priority on fast cores, while the VMs with an active vCPU count above the threshold are considered highly parallel and thus get a low priority in running fast cores.

The described mechanisms can be used on their own, depending on the goals of the user, or simultaneously. In our experimental implementation we have chosen to combine all three mechanisms. In this case, any high-priority VMs share the fast cores equally. The remaining fast-core cycles are shared equally among all vCPUs with the exception of asymmetry-aware guests whose fast vCPUs receive all the fast-core cycles allotted to that VM. While we chose a particular way to combine the mechanisms, we do not advocate this particular approach to be the only correct way. We evaluate each mechanism separately to enable understanding the merits of the individual mechanisms and also show how they work in tandem.

3. Design and implementation

In implementing the AASH algorithm we pursued two goals: simplicity and low overhead. Overhead may occur when virtual CPUs are migrated between fast and slow physical cores, especially when migrations cross memory-domain boundaries. The term memory domain is used to refer to a group of cores sharing a last-level cache (LLC). When a vCPU is migrated to a new memory domain it loses the cache state accumulated in its old LLC. This can cause performance degradation. To avoid this overhead, we ensured that cross-core migrations are not very frequent. In our implementation they occur, on average, not more often than once per 30 millisecond accounting period per vCPU. As a result, as we show in the experimental section, performance overheads are small. Another potential source of overhead is due to non-uniform memory access (NUMA) [13]. We have not investigated effects of NUMA in this project, leaving this for future work.

To implement AASH we extended the default Credit Scheduler in Xen [5]. We begin with providing pertinent background about the Xen credit scheduler and then explain how we extended it to support asymmetry.

3.1 The Xen Credit Scheduler

To accomplish fair sharing of physical cores among vCPUs Xen relies on a system of credits. Credits are distributed among vCPUs on each 30 millisecond accounting period and the physical CPU time granted to a vCPU depends on the number of credits assigned to this vCPU. The total number of credits in the system depends on the number of physical cores. Each physical core is worth 300 credits for a 30ms accounting period. Credits are distributed among vCPUs according to the cap and weight of the VM to which these vCPUs belong; cap controls the maximum CPU time that a VM can receive during the accounting period, and weight determines that VM's proportion of CPU resources. In our system we assume

[1] We assume that unused threads of a parallel application block, rather than busy-wait on a CPU. In that case, a VM that has entered a phase of low parallelism will have idle vCPUs, which will be visible to the hypervisor.

that the cap is unlimited and that the weight of a VM is proportional to the number of vCPUs with which this VM was configured, but supporting other weights would not be difficult.

As the vCPU runs on a physical core, its credits are decremented by 100 units on each 10 millisecond scheduling clock tick. All vCPUs are organized in a priority runqueue. On expiration of a vCPU's timeslice (or when a vCPU goes idle or blocks) the scheduler decides which vCPU to run next according to its priority. A vCPU that has a positive credit balance has a priority of *under*; a vCPU with a zero or a negative balance has a priority of *over*[2]. A vCPU with a priority of *under* will be chosen to run before any vCPU with a priority of *over*.

3.2 Equal sharing of asymmetric cores

We describe this mechanism first, because it underlies other asymmetry-aware mechanisms in our scheduler. To support equal sharing of asymmetric cores we introduce two types of credits: *fast* credits and *slow* credits. Fast credits entitle the vCPU to run on a fast core. Slow credits entitle it to run on slow cores.

Fast and slow credits are distributed on each accounting period. The amounts of fast and slow credits available on each period are proportional to the number of fast and slow physical cores. Not all vCPUs get fast credits on each accounting period; therefore not all of them run on fast cores in a given period. A *fast queue*, described below, is maintained to ensure equal sharing of fast credits.

All vCPUs entitled for fast credits are placed in the fast queue. On each accounting period, the scheduler selects *nFastCores* vCPUs from the top of the fast queue; *nFastCores* corresponds to the number of fast cores in the system. The selected vCPUs are assigned fast credits for the following accounting period: during that period they will be mapped to fast cores. The remaining vCPUs are assigned slow credits: during the following accounting period they will be mapped to slow cores. At the end of the credit distribution, *nFastCores* vCPUs will be moved from the tail of the queue to the head[3]. These vCPUs, which did not receive fast-core credits during the current period, will get them during the next period. The vCPUs that did get fast credits during the current period, will drift back down the queue until moved to the head again. This mechanism ensures fair sharing of cores of both types and also ensures that fast-to-slow core migrations are not overly frequent, since fast-core credits are awarded only once per accounting period.

If a vCPU blocks while running on the fast physical core, that core will try to steal work from other cores – that is, it will run a vCPU that is waiting in the queue of another core, preferably a fast core.

A vCPU's credits are decremented as it runs on a particular type of core. Once the number of credits reaches zero, the credits are deemed expired. A vCPU whose fast credits have expired will be marked as a candidate for migration to a slow core. To avoid load imbalance, the scheduler will find a vCPU running on a slow core that is about to receive fast-core credits (i.e., a vCPU at the top of the fast queue), and mark it as a candidate for migration to a fast core. The actual migration happens right before the physical core begins running the new vCPU.

Our mechanism for fairly sharing asymmetric core is efficient, as shown in the evaluation section, because it requires relatively few cross-core migrations. A vCPU that is assigned fast credits gets the right to use a fast core for the entire 30ms timeslice; only upon the expiration of the timeslice may it get migrated to a slow core. With such relatively infrequent migrations the loss of last-level cache state becomes amortized, since a 30ms period is large enough to allow the vCPU to refill its cached data and run with the warm cache for most of the timeslice [8]. Although applications with very large cache working sets do see some performance degradation due to migrations, the overhead never exceeds 4%, because the frequency of migrations is low.

3.3 Support for asymmetry-aware guest operating systems

Mapping of virtual CPUs in a asymmetry-aware guest must be deterministic so that the guest operating system can effectively apply its policies. To this end, AASH maps to fast cores only those vCPUs that are considered "fast" by the asymmetry-aware guest. In this work we assume that vCPUs with IDs up to *nFastCores* are "fast", but any other mapping can be supported in our algorithm[4]. "Fast" vCPUs will be assigned to run on fast physical cores as often as possible, depending on the competition for fast cores, while the remaining vCPUs will always run on slow cores, unless idle fast cores are available or unless a fast core steals them from a slow core as part of routine load balancing. Another alternative would be to never allow the mapping of "slow" vCPUs to fast physical cores, but for the sake of optimizing performance we chose to allow such a mapping if there are idle fast cores.

To support this mapping policy, we modify the process of updating the fast queue as follows. Remember that once the scheduler is finished distributing fast-core credits, it moves the vCPUs from the tail of the queue to the head. If a "slow" vCPU is encountered, it is left at the tail of the queue. Note that if a "slow" vCPU is encountered at the head of the queue during the distribution of fast credits, it will not be skipped, since with our mechanism where eligible vCPUs are moved from the tail to the head, this can only occur when the number of vCPUs eligible for running on fast cores exceeds the number of fast cores. In this case we choose to run "slow" vCPUs on fast cores to maximize performance. If the goal is to save power, it might be wiser to leave fast cores idle so that they could be brought into a more economical low-power state. Considering power saving policies for AMP systems, however, is outside the scope of this work.

If there are other VMs in the system, either asymmetry-aware or asymmetry-unaware, fast credits will be shared among all vCPUs that are entitled to use fast cores. If the scheduler is unable to assign fast credits to a "fast" vCPU of an asymmetry-aware guest on a particular timeslice, as a result of competition for fast cores from other VMs, the "fast" vCPU will be instead assigned slow-core credits.

3.4 Support for priorities

To support priorities we introduce another queue: *a high-priority fast queue*. Virtual CPUs that have an elevated priority for using fast cores are placed on that high-priority queue. When fast credits are distributed, the vCPUs in that queue will be given the credits first. Credits are distributed among high-priority VMs in a fair fashion, in the same way as for the "normal" fast queue. The remaining fast credits, if any, will be distributed among the remaining eligible vCPUs.

A VM is considered to have a high priority for using fast cores in two cases: (1) when it has been explicitly designated as a high-priority VM by a system administrator; (2) if the number of active vCPUs in the corresponding VM reduces below a pre-defined threshold, assuming that the feature for prioritizing VMs with a low number of virtual CPUs is enabled. This threshold can be set to equal the number of fast cores, and it was set to "one" in our experiments. If the number of active vCPUs increases, the vCPUs of that VM are moved again to the normal fast queue.

[2] There is another priority boosted, but it is not relevant to this discussion.

[3] We chose to move vCPUs from tail to head, rather than from head to tail, to simplify the implementation of support for asymmetry-aware guests.

[4] As mentioned earlier, development of mechanisms for the discovery of fast cores by a guest OS is outside the scope of this work.

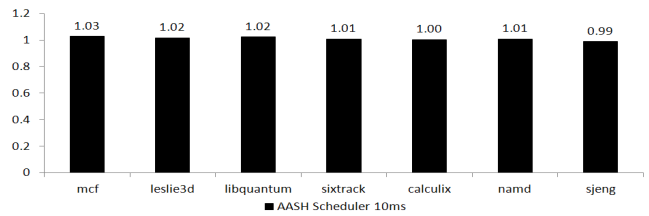


Figure 1. Completion time under AASH normalized to that under default Xen. Low numbers are better.

4. Evaluation

In this section we evaluate the AASH scheduler by comparing it to the default Xen scheduler. In Section 4.1 we describe the experimental platform and benchmarks. In Section 4.2 we evaluate the overhead. In Section 4.3 we evaluate the mechanism for fair sharing of fast cores. This mechanism is evaluated before others, because it forms the basis on top of which other mechanisms are built. We evaluate the support for asymmetry-aware guests in Section 4.4 and the two prioritization mechanisms in Sections 4.5 and 4.6 respectively.

4.1 Experimental Platform and Benchmarks

We use an AMD Opteron 2350 (Barcelona) system, with two quad-core CPUs and 8GB of RAM. Cores on the same chip share a 2 MB L3 cache and each core has a private 512 KB L2 cache, a 64 KB instruction cache and a 64 KB data cache. Asymmetry was emulated by setting cores to run at different frequencies using dynamic voltage and frequency scaling (DVFS). Fast cores are emulated by running the core at the highest available frequency: 2GHz. Slow cores are emulated by running the core at the lowest available frequency: 1GHz. We vary the number of cores and the ratio of fast to slow cores according to the experiment.

Virtualization is often used for services like Amazon EC2 Elastic Compute Cloud, and these services are becoming increasingly popular in the HPC scientific community [6, 20]. In our experiments we use primarily scientific applications that perform little I/O, since the effect of CPU optimizations is particularly evident on applications that spend a large majority of their time on CPU. The applications are drawn from SPEC CPU2000, SPEC CPU2006, PARSEC [4] and SPLASH benchmarks suites, *BLAST* (a series of sequence alignment codes used in bioinformatics), and *FFT-W* (a parallel implementation of a fast Fourier transform). We repeat each experiment at least three times and report the average execution time or speedup.

In our experiments the number of virtual CPUs matches the number of physical CPUs; using a configuration where the number of virtual CPUs exceeds the number of physical ones is only reasonable for I/O-bound applications, but the applications we used perform little to no I/O. If a workload with a high CPU utilization is run on a system where the number of vCPUs exceeds the number of physical cores, it may suffer from a high context switching overhead, without the added benefit in performance. Although workloads with more vCPUs than physical cores were not used, we do not see a reason why our results would not extend to such scenarios.

4.2 Evaluating the overhead of migrations

Recall that the AASH scheduler shares the fast cores among virtual CPUs and therefore causes more migrations than the default Xen scheduler. Migration of virtual CPUs among physical cores of different types may be costly if the cores are located in different memory hierarchy domains; by memory hierarchy domain we mean a group of cores sharing a last-level cache. Cross-memory-domain migrations cause the migrated vCPU to lose the state accumulated in the LLC. Rebuilding this state after migration may cause performance degradation [8].

In order to evaluate the overhead of migrations, all cores were configured as slow (1GHz), but the AASH scheduler still deems the system asymmetric (with one fast core) so it performs its regular migrations. This experimental setup allows us to bring out the overhead associated with AASH's migrations, while eliminating any performance improvements from asymmetry-aware scheduling policy. We compare the completion time of the applications running under the AASH scheduler to that under the default Xen scheduler; any additional latency under AASH is due to migration overhead.

Migration overhead could manifest differently for applications with different memory access patterns. Cache-sensitive applications (those with a large cache footprint and a high cache access rate) could be sensitive to frequent migrations. Cache-insensitive applications could be indifferent to additional migrations. We used the classification scheme similar to that in [21] to determine which applications are cache-sensitive and which are not. As cache-sensitive we identified *leslie3d* and *libquantum* from the SPEC CPU2006 benchmark suite and *mcf* from the SPEC CPU2000 benchmark suite. As cache-insensitive we identified *calculix*, *namd* and *sjeng* from the SPEC CPU2006 and *sixtrack* from the SPEC CPU2000.

An experiment consists of running one virtual machine with eight virtual CPUs executing eight copies of the same benchmark. We measure the completion time of all instances of the benchmark and report the average under AASH normalized to that under Xen.

Figure 1 shows the results (low bars are good). There is a negligible performance degradation with the AASH scheduler for cache-sensitive applications. For the most cache-sensitive application *mcf* the overhead reaches 3%. Cache-insensitive applications are largely unaffected by migrations. We have evaluated the sensitivity of performance to migration frequency, changing the timeslice from 3 to 100 milliseconds, and found that the overhead slightly increases when the timeslice is reduced, and decreases when it is increased.

4.3 Evaluating the equal sharing capability

For experiments in this section we use both single-threaded and parallel applications. The single-threaded applications are *leslie3d*, *libquantum*, *calculix*, *namd* and *sjeng* from the SPEC CPU2006 suite, *mcf*, *sixtrack* and *eon* from the SPEC 2000 suite. The parallel applications are *blackscholes*, *bodytrack*, *facesim*, *ferret* and *fluidanimate* from PARSEC, *radix* from the SPLASH suite, *FFT-W* and *BLAST*.

In the first set of experiments we used a machine with four physical cores: one fast and three slow. In each experiment we ran four virtual machines, each configured with one virtual CPU and running one application. Each application was run with one thread, and four identical applications were run in an experiment. We repeat the experiment for each application in the list. Comparing completion times of applications across the four VMs under AASH and under the default Xen scheduler enables us to evaluate the impact that equal sharing of the fast core has on performance of applications.

Figure 2 shows the results. The vertical bars show the completion times of the four VMs stacked one on top of the other. If the stacked components have equal height, this means that the completion times of the four VMs are roughly equal, which is what we would like to see if the fast core is shared equally. The black line at the top of each bar shows the standard deviation of completion times across the VMs. AASH delivers more equal completion times, because it shares the fast core equally among them. The default Xen scheduler, on the other hand, may deliver disparate per-

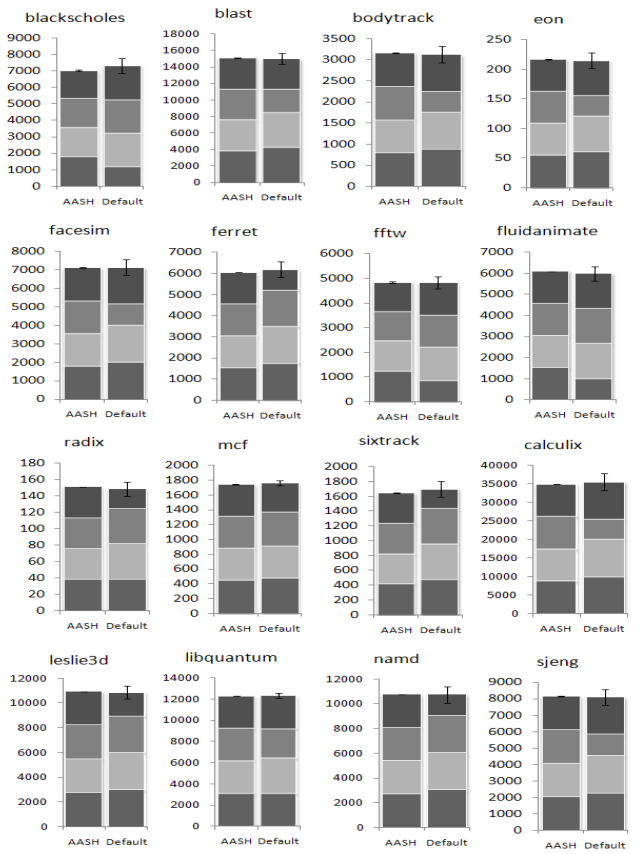


Figure 2. Completion times in seconds for four single-vCPU VMs running concurrently

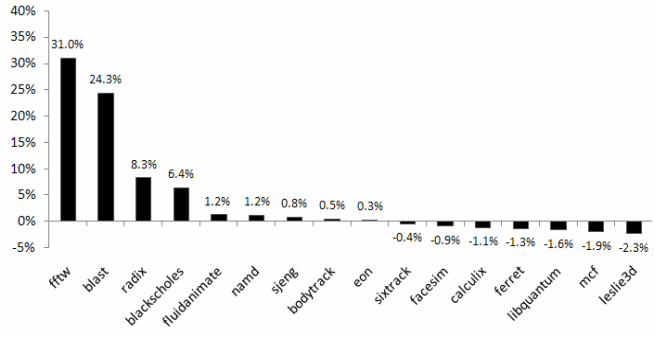


Figure 3. Speedup under AASH relative to default Xen in an experiment with a single VM and eight vCPUs

formance for these identical VMs, because it may arbitrarily favour one VM over another in terms of the fraction of time spent running on the fast core. Standard deviation of completion times across the VMs is also significantly higher under the default Xen scheduler. Examining the raw completion times, we see that they are roughly the same under AASH as under the default Xen scheduler, suggesting that AASH delivers more stable performance and fairer sharing of fast cores without compromising performance.

In the course of our experiments we also observed that in some cases performance is improved due to the fact that AASH ensures that fast cores never go idle before slow cores. For applications with unbalanced load, where some CPUs are occasionally left idle, this can improve performance. We demonstrate this effect in the following experiment. We use a physical machine with eight cores, one of which was fast and the rest were slow. In each experiment we run a single VM with eight virtual CPUs. We choose one application to run inside a VM; in case the application is parallel we launch it with eight threads, in case the application is sequential we run eight copies of it. We measure the performance of each application under AASH and under the default Xen scheduler, and in Figure 3 we report the speedup under AASH relative to default Xen. Here and in subsequent experiments, speedup percent is computed using the following formula: $(1 - \frac{T_{AASH}}{T_{default}}) * 100$, where T_{AASH} and $T_{default}$ are completion times under the AASH and default schedulers respectively.

We observe that some parallel applications (*FFT-W, BLAST, radix and blackscholes*) experience significant speedup under the AASH scheduler. There are two reasons for this. First, because of equal sharing of fast cores among vCPUs, AASH equally accelerates the computation running on all cores as opposed to accelerating one vCPU while letting others trail behind. In applications where threads synchronize, ensuring equal acceleration for all threads may reduce the completion time.

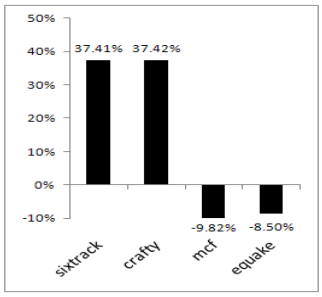
Another reason for improved performance is better utilization of fast cores under AASH. Consider, for example, *FFT-W* that reaches the speedup of 31%. Although *FFT-W* is a parallel application it spends a large amount of time (87%) running with only a single thread. When only a single thread is active, the guest operating system reports idle vCPUs to Xen. As a result, Xen scheduler removes idle vCPUs from its scheduling queue and assigns processor cycles only to the active ones. The asymmetry-unaware default scheduler may map the active vCPU to either a fast or a slow core, because it cannot tell them apart. The asymmetry-aware AASH, however, will always map the active vCPUs to fast cores first. As a result, the virtual CPU will run faster than under the default Xen scheduler. This brings performance improvements to applications that often leave some of the cores idle. For parallel applications like *BLAST* and *FFT-W*, which reduce the number of active threads during a sequential phase also causing the reduction in the active vCPU count, this amounts to the acceleration of their sequential phases on fast cores. Potential performance benefits of this effect were discussed in literature [1, 7]. When only one VM is running, sequential phases of the enclosed application are accelerated automatically. When multiple VMs are running the scheduler has to be designed specifically to detect and accelerate low-parallelism phases of the VM. The said capability of AASH will be evaluated in Section 4.5.

4.4 Evaluating support for asymmetry-aware guests

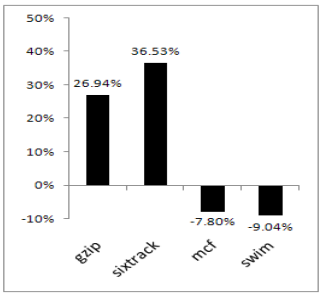
In this section, we first present experiments with a single asymmetry-aware guest running under the AASH scheduler. We show that it achieves better performance than under the default Xen scheduler, because AASH provides deterministic mapping of "fast" vCPUs to fast physical cores. AASH also supports the co-existence of asymmetry-aware guests and legacy guests. We evaluate this property in the second experiment, by running both types of guests simultaneously. In this case, comparing performance under AASH and under the default Xen schedulers shows that both guests benefit from an asymmetry-aware scheduler. The following sub-sections present these two experiments.

4.4.1 Single-VM experiments

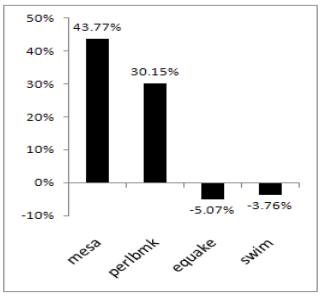
We use the same workloads as have been used in a previous study [19] to evaluate an asymmetry-aware operating system scheduler called HASS. In that work, the following workloads made up of the SPEC CPU2000 benchmarks were used: (1) *sixtrack*, *crafty*, *mcf* and *equake*, (2) *gzip*, *sixtrack*, *mcf* and *swim* and (3) *mesa*, *perlbmk*,



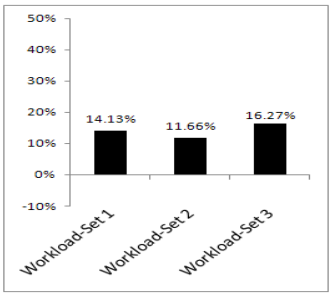
(a) Normalized speedup of Set 1 benchmarks under AASH relative to default Xen



(b) Normalized speedup of Set 2 benchmarks under AASH scheduler relative to default Xen



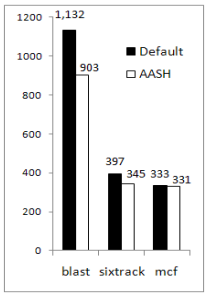
(c) Normalized speedup of Set 3 benchmarks under AASH relative to default Xen



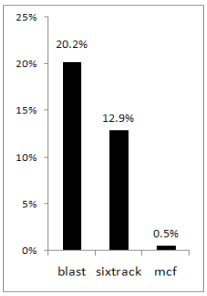
(d) Average speedup of workload under AASH relative to default Xen

Figure 4. Support for asymmetry-aware guests: a single-VM Experiment

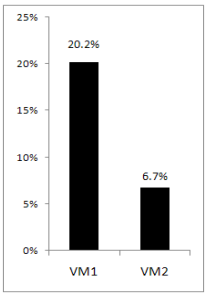
equake and *swim*. The first two benchmarks in each set are CPU-intensive, meaning that they effectively utilize the CPU pipeline, rarely stalling it, and the second two are memory-intensive, meaning that they frequently stall the CPU pipeline as a result of issuing a large number of memory requests. The experiments in the aforementioned work were run on a system with two fast cores and two slow cores. The HASS scheduler mapped the CPU-intensive applications to fast cores and the memory-intensive applications to slow cores, since this kind of mapping is known to maximize efficiency of AMP systems [11]. The authors of the HASS paper reported performance improvements of up to 12% from using this policy on an asymmetric platform similar to ours. To replicate the same experiment in our study and to mimic the HASS scheduling policy we use a single VM that runs one of the three aforementioned workloads. We bind the CPU-intensive applications to the first two virtual CPUs and the memory-intensive applications to the second two virtual CPUs inside the guest operating system. Since the guest assumes that the first two virtual CPUs are fast and the second two virtual CPUs are slow, the scheduler needs to respect that mapping to the actual physical cores. This is what the AASH scheduler does. (We assume that there is a mechanism in place that enables the guest to inform the hypervisor which vCPUs it considers fast; implementation of this mechanism is beyond the scope of the paper.) The default Xen scheduler, on the other hand, is asymmetry-unaware, and so it performs mapping of virtual to physical cores arbitrarily, disrupting the asymmetry-aware policies in the guest OS.



(a) Average benchmark completion times under both schedulers



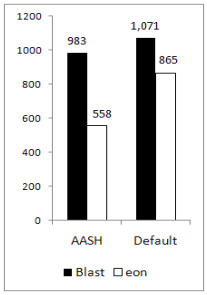
(b) Speedup under AASH normalized to default Xen



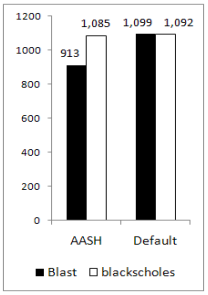
(c) Average speedup of each VM under AASH normalized to default Xen

Figure 5. Experiments with asymmetry-aware and legacy guests

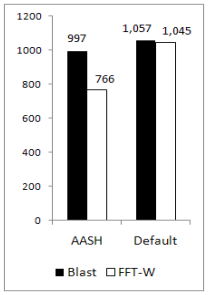
Figure 4 shows the results of this experiment. As can be expected, the asymmetry-aware guests performed better under the AASH scheduler. The mean speedup (Figure 4(d)) was as much as 16.27% for the *mesa*, *perlbmk*, *equake* and *swim* workload and reached 11% and 14% for the other workloads. For all workloads we see that the first two CPU-intensive applications in the workload speed up under the AASH scheduler, while the second two memory-intensive applications slow down (in Figures 4(a), 4(b) and 4(c)) . This is the expected behaviour, and the results agree with those reported in the original HASS paper, since the asymmetry-aware OS scheduler runs CPU-intensive applications on fast cores, relegating memory-intensive applications to slow cores. Since the speedup experienced by the CPU-intensive applications is greater than the slow-down experienced by the memory-intensive applications, the workload as a whole experiences an improvement in performance. The performance gains are possible only with an asymmetry-aware hypervisor. An asymmetry-unaware hypervisor sacrifices 12-16% in performance.



(a) Average completion time in seconds of BLAST and eon.



(b) Average completion time in seconds of BLAST and blackscholes

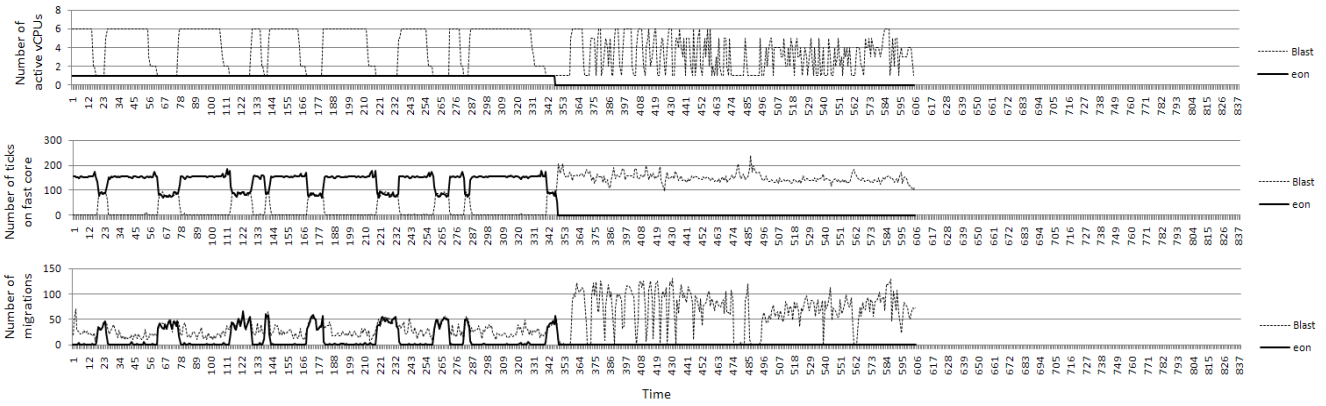


(c) Average completion time in seconds of BLAST and FFT-W

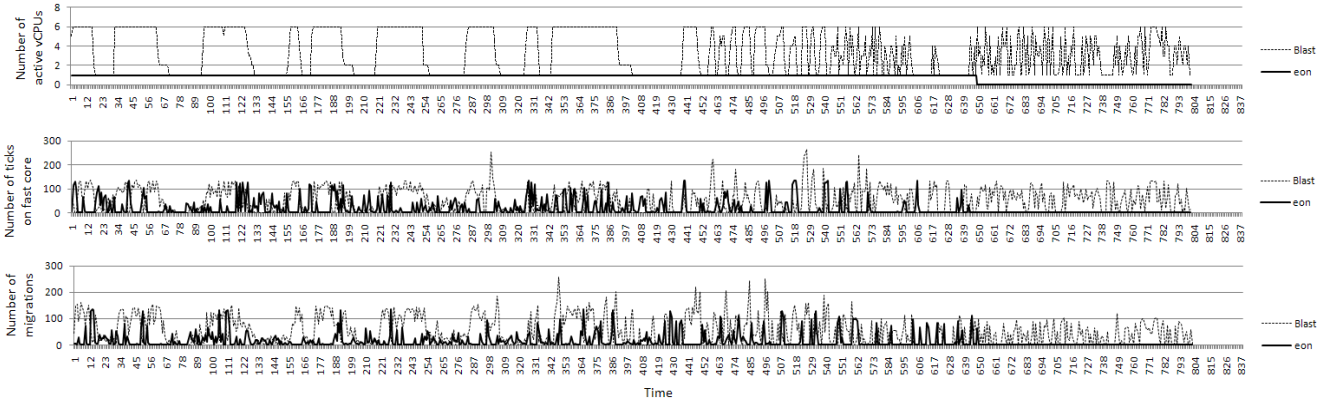
Figure 6. Prioritizing single-CPU VMs

4.4.2 Experiments with an asymmetry-aware and a legacy guest

In this experiment, we used two virtual machines, one running an asymmetry-unaware (legacy) guest and the other running an asymmetry-aware guest. The legacy guest has six virtual CPUs and runs *BLAST*, which is configured to use six threads. (*BLAST* is a parallel application with frequently occurring sequential phases.) The second virtual machine runs an asymmetry-aware guest and has two virtual CPUs (it assumes the first virtual CPU is fast). It runs one instance of a CPU-intensive application (*sixtrack*) and another instance of a memory-intensive application (*mcf*). As in the previous experiment, we simulate asymmetry awareness in the second virtual machine by binding *sixtrack* to the first virtual CPU



(a) The AASH scheduler



(b) The default Xen scheduler

Figure 7. Time series data on the dynamic count of active vCPUs, usage of fast and slow cores, and the number of migrations for the *BLAST/eon* workload. Time on x-axis is represented in terms of statistics-gathering intervals; each interval is roughly equal to 1-2 seconds.

(the one that is assumed by the guest to be fast) and *mcf* to the second virtual CPU. We ran virtual machines on our experimental platform with one fast core running at 2 GHz and seven slow cores running at 1 GHz.

AASH shares the fast core among all virtual machines. Since the second virtual machine is asymmetry-aware, the AASH scheduler assigns all its fast-core cycles to the vCPU deemed "fast" by this guest. This results in: (1) sharing of fast cores among both guests in proportion to the number of vCPUs in each guest, and (2) deterministic mapping of the "fast" vCPUs to fast physical cores for the asymmetry-aware guest.

Figures 5(a) and 5(b) show the completion time and speedup for each benchmark under AASH relative to the default Xen scheduler. *Sixtrack*, the CPU-intensive application running in the asymmetry-aware VM, shows a 13% performance improvement. The mean speedup of the asymmetry-aware guest is 6.7% (Figure 5(c)). The asymmetry-unaware guest, which runs *BLAST*, also shows a 20% speedup. We note that the speedup for *sixtrack* is smaller than in the experiment of Figure 4(b), because in *sixtrack* shares the fast physical core with *BLAST*, unlike the earlier experiment where it had the fast core to itself. Here, the benefit of using a fast core is equally distributed among both asymmetry-aware and asymmetry-unaware guests.

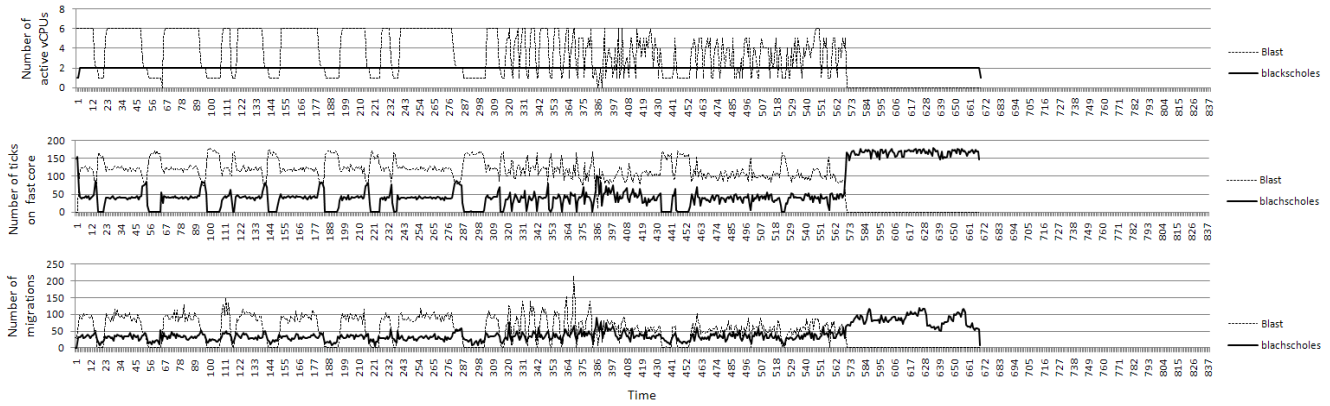
4.5 Prioritizing VMs running code low in parallelism

In this experiment we evaluate the mechanism for giving a higher priority for running on fast cores to VMs deemed as running code low in parallelism. Remember that AASH uses the number of active vCPUs as a heuristic for the degree of parallelism. In our system a low-parallelism threshold is set to one: that is, a VM is considered low in parallelism when its active vCPU count reduces to one. Assuming the VM runs a single application, this amounts to acceleration of sequential phases of this application on a fast core. Earlier we demonstrated that sequential phases are automatically accelerated when there is only one VM running on the system, because that VM's single active vCPU will be always mapped to a fast core. In this section, we evaluate the scheduler's ability to accelerate low-parallelism VMs when multiple VMs are present.

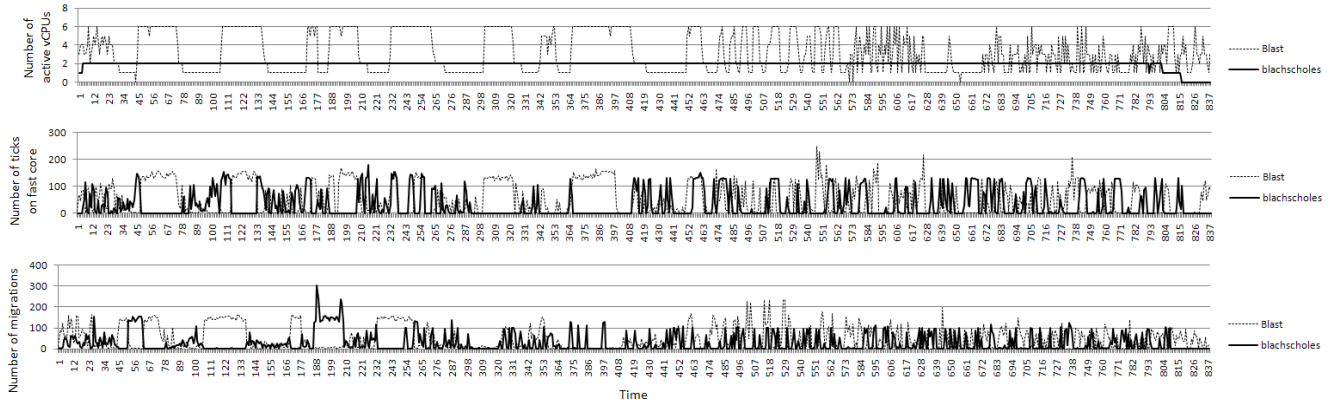
Remember that the feature giving a higher priority for low-parallelism VMs can be selectively turned on in AASH. When it is on, the scheduler does not equally share fast cores among all vCPUs, but preferentially allocated fast-core cycles to VMs whose active vCPU count is below the low-parallelism threshold. The remaining fast-core cycles are shared among other eligible vCPUs.

We identified several workload mixes that were interesting for evaluating this feature:

1. A combination of a sequential workload and a parallel workload with sequential phases.



(a) The AASH scheduler



(b) The default Xen scheduler

Figure 8. Time series data on the dynamic count of active vCPUs, usage of fast and slow cores, and the number of migrations for the *BLAST/blackscholes* workload. Time on x-axis is represented in terms of statistics-gathering intervals; each interval is roughly equal to 1-2 seconds.

2. A combination of a highly parallel workload and a parallel workload with sequential phases.
3. A combination of two parallel workloads, both of which have sequential phases.

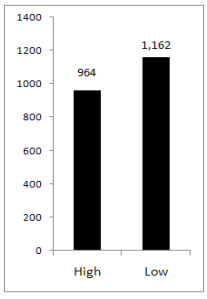
The following sub-sections describe the experiments performed with those three workloads. In all cases with use a system with eight physical cores where one core is fast and others are slow.

4.5.1 Sequential workload and a parallel workload with sequential phases

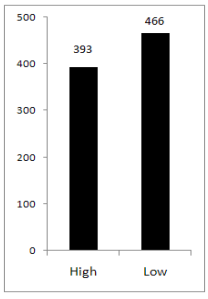
In this scenario, the AASH scheduler would preferentially allocate fast-core cycles to the VM running the sequential applications, assuming this VM will always have only one active vCPU. If the other VM reduces its active vCPU count to one, the fast-core cycles will be equally shared among the vCPUs of the two VMs.

To evaluate this scenario, we run two virtual machines. The first virtual machine has six virtual CPUs and runs *BLAST* – a parallel applications with sequential phases. The other virtual machine has a single virtual CPU and runs a sequential application *eon* from SPEC CPU2000.

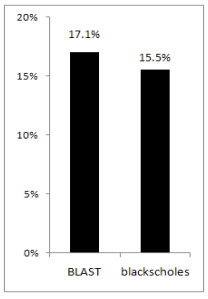
Figure 6(a) shows the results. The single-threaded application experienced a 36% speedup and the parallel application enjoyed an 8% speedup under AASH relative to the default Xen scheduler.



(a) Average completion time of VMs running BLAST



(b) Average completion time of VMs running blackscholes

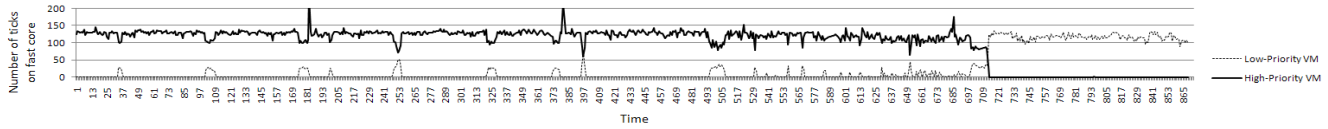


(c) Speedup of the high-priority VM relative to the low-priority VM

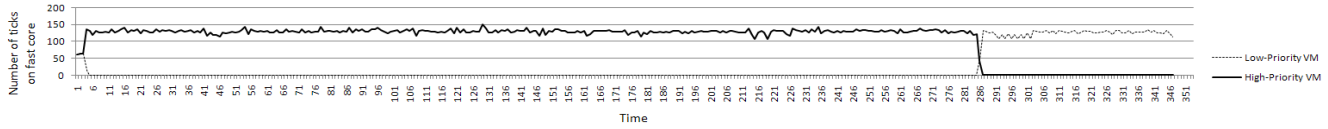
Figure 9. Prioritization experiment

While *eon* benefits from running on the fast core most of the time under the AASH scheduler, *BLAST* also enjoys the acceleration of its sequential phases.

Figure 7 shows the behavior of these two applications under both schedulers over time. Figure 7(a) shows the changes in the



(a) Fast core allocation behavior (both VMs running BLAST)



(b) Fast core allocation behavior (both VMs running blackscholes)

Figure 10. Fast core allocation for High and Low priority VMs

number of the active virtual CPUs, the number of clock ticks that each virtual machine spends on fast cores and the number of migrations under AASH; Figure 7(b) shows the same statistics for Xen. As can be seen in the middle portion of Figure 7(a), *eon* spends most of its time on the fast core. In sequential phases of the *BLAST* workload, the AASH scheduler shares the fast core among both virtual machines. The figures also show that migrations of *eon* were limited to its sequential phases during those periods where it shared the fast core with *BLAST*. Figure 7(b) on the other hand, shows that the behavior of the benchmarks under the default Xen scheduler is largely arbitrary.

4.5.2 A highly parallel workload and a parallel workload with sequential phases

We use two virtual machines. The first one has six virtual CPUs and runs a parallel workload with frequently occurring sequential phases (*BLAST*). The second one has two virtual CPUs and runs a highly parallel application (*blackscholes*). The average completion time of both application is shown in Figure 6(b). *BLAST* experienced a 17% speedup under the AASH scheduler; performance of *blackscholes* was largely unaffected.

Figure 8(a) shows the behavior of these two benchmarks under the AASH scheduler. It can be seen that whenever *BLAST* enters a sequential phase and the underlying VMs active vCPU count reduces to one, the AASH scheduler migrates the vCPU to the fast core and stops giving any time on the fast core to *blackscholes*. When both VMs have the vCPU count exceeding one they share the fast core proportionally to the number of their active vCPUs. Figure 8(b) shows the behaviour exhibited by the default Xen scheduler during this experiment. Migration patterns and allocation of virtual CPUs is rather arbitrary.

4.5.3 Parallel workloads with sequential phases

In this experiment, we run *BLAST* and *FFT-W* benchmarks on the two virtual machines. The VM running *BLAST* uses six virtual CPUs and the VM running *FFT-W* uses two. Both of these parallel applications have long sequential phases.

Figure 6(c) shows that the workload as a whole achieved a speedup of 16% with the AASH scheduler. *FFT-W* achieved a 27% speedup, and *BLAST* achieved a 6% speedup. *BLAST* achieved a smaller performance improvement than in the experiment of Figure 3, because during its sequential phase it had to share the fast core with *FFT-W*.

The results presented in Figures 7 and 8 of this section demonstrate that the AASH scheduler reacts sufficiently quickly to the change in the active vCPU count and begins allocating fast-core cycles to VMs whose active vCPU count reduces to one, or stops allocating them if the vCPU count increases beyond one and there is another single-CPU VM running. This suggests that the choice of distributing fast-core cycles only every 30ms for the sake of minimizing migration overhead does not impede the scheduler's ability to share fast cores equally among the competing vCPUs.

4.6 Coarse-grained priorities

In this experiment we evaluate the mechanism in AASH that enables it to preferentially grant fast-core cycles to high-priority VMs before low-priority VMs. Recall that AASH supports two priority classes, high and low. We show that a high-priority VM receives a higher share of fast-core cycles and achieves better performance than a low-priority VM.

To enable performance comparisons across VMs, we use two VMs running identical applications, but one VM has a high priority, while another VM has a low priority. Each virtual machine has four virtual CPUs. We run two experiments, each using a different workload: in the first experiments the two VMs run *BLAST* and in the second the two VMs run *blackscholes*.

Figures 9 and 10 show the results of this experiment. As can be seen in Figure 9(c) the high-priority VMs achieve better performance than low-priority VMs. Figure 10 shows the distribution of fast core cycles among VMs over time. In both scenarios fast cores are used most of the time by the high-priority guest. In Figure 10(a) we see that a low-priority guest is occasionally assigned to run on fast cores: this happens when the vCPU count of that guest reduces to one and the priority of that VM rises. In this case, both guests share the fast core.

5. Related work

Existing virtualization systems are not asymmetry aware. VT-ASOS [15] is an enhanced version of the Xen hypervisor that leverages application feedback when deciding how to allocate machine resources. In principle, this system is well positioned to support asymmetry, but the existing implementation is not asymmetry-aware and, in particular, it does not provide proper support for asymmetry-aware guests.

Most work related to AMP systems was done in the domain of operating systems. We classify the asymmetry-aware algorithms for operating systems according to three categories. In the first category are the algorithms that determined the best threads to schedule on cores of different types via continuous monitoring of performance and its analysis [3, 12, 17, 10] or via analytical models relying on static information about applications [19]. In the second category are the algorithms that accelerated sequential phases of parallel applications on fast cores [17]. In the third category are the

algorithms that either ensured fair sharing of the fast core [2] or placed a higher load on fast cores than on slow cores [13].

The algorithms in the first category required information about individual threads in the workload in order to pick for running on fast cores those threads that would use these cores most efficiently. Since the hypervisor is not aware of context switches among individual threads, it is difficult to implement the same algorithms in the hypervisor. Therefore, we believe that this type of asymmetry support belongs to the guest OS, and asymmetry awareness in the hypervisor should be limited to providing proper support for asymmetry-aware guests, fair sharing of different types of cores, and prioritization in using more "expensive" cores.

We now briefly describe the algorithms falling into the first category. An algorithm designed by Kumar et al. [12] is representative of algorithms in this category. The best threads to run on fast cores were determined via direct measurement of performance: each thread had to be run on both fast and slow cores, its performance (in terms of instructions per cycle) was measured, and the ratio or performance on the fast core relative to a slow core was used to guide the thread assignment. Threads with the highest fast-to-slow performance ratios were selected to run on fast cores. An algorithm designed by Becchi and Crowley [3] used a similar method to select the most efficient threads. Unfortunately, these algorithms were not implemented in a real operating system and when a later study attempted a real implementation [19], it was found that the need to run each thread on each core type created load imbalance and caused performance degradation. To address this problem, Shelepov et al. proposed an algorithm that relied on the architectural signature of an application (a statically obtained summary of its architectural properties) to find the best candidates for fast cores [19]. Saez et al. further improved on that work by estimating an application's speedup on the fast core dynamically, using data obtained via hardware counters, but without requiring to run each thread on cores of all types [17]. Koufaty et al. proposed a similar mechanism [10]. Similarly to other approaches, however, these algorithms required detailed knowledge about the application, and would thus be cumbersome to implement in the hypervisor.

In the second category are algorithms proposed by Saez et al. [17] and by Annavaram et al. [1]. These algorithms accelerated sequential phases of parallel applications on the fast core. In giving a higher priority in running on fast cores for low-parallelism VMs, AASH borrows ideas from these algorithms.

In the third category are the schedulers designed by Li et al. [13] and Balakrishnan et al. [2]. Li's scheduler ensured that fast cores run more threads than slow cores. The idea is that a core's load should be proportional to its speed, so threads running on fast cores compute more quickly, but receive less CPU time. We do not pursue the same strategy; instead we allow fast cores to be used by vCPUs either equally or in accordance with a priority scheme. This way fast cores end up executing more instructions than slow cores, and this benefit is distributed among all vCPUs, either equally or in accordance with a priority policy. While Li's strategy implicitly assumes that all threads benefit from running on the fast core to the same degree, an assumption that does not always hold [11], our strategy does not make the same assumption and instead gives each vCPU (modulo its priority) a chance to benefit from running on a fast core. Balakrishnan's scheduler ensured that the fast cores do not go idle before slow cores; our scheduler has a similar feature. What these two schedulers did not provide, however, is a true round-robin sharing of fast and slow cores. Our scheduler provides this feature while imposing only negligible performance overhead.

All in all, although there are similarities between asymmetry-aware schedulers in operating systems and our asymmetry-aware scheduler in the hypervisor, there are several key differences that distinguish our work from previous studies. Our scheduler is implemented in the hypervisor, and not in the operating system. Unlike OS schedulers, it provides support for asymmetry-aware guests. We design and implement round-robin sharing of cores of different types using a unique mechanism, new to our work, which enables fair sharing while incurring little overhead. Most importantly, we evaluate asymmetry-aware support in the context of virtual systems, where the benefits and overheads may be fundamentally different than in non-virtualized environments.

6. Conclusions

We presented AASH, an asymmetry-aware scheduler for hypervisors. To the best of our knowledge, this is the first implementation of asymmetry support in a VM hypervisor. Using AASH we were able for the first time to evaluate the impact of asymmetry awareness in the hypervisor's scheduler, both in terms of performance and overhead.

Our conclusion is that support for AMP systems can be implemented in modern hypervisors relatively efficiently and that it results in measurable performance benefits for most workloads (up to 36% in our experiments), and with only a small performance loss for some.

7. Acknowledgements

This research is supported by Canadian National Science and Engineering Research Council (NSERC) Strategic Project Grants program and Sun Microsystems.

References

[1] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law through EPI Throttling. In *Proc. of ISCA'05*, pages 298–309, 2005.

[2] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. *SIGARCH CAN*, 33(2):506–517, 2005.

[3] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proc. of Computing Frontiers '06*, pages 29–40, 2006.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. of PACT'08*, October 2008.

[5] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU Schedulers in Xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, 2007.

[6] C. Evangelinos and C. N. Hill. Cloud Computing for Parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. In *Proc. Cloud Computing and Its Applications*, 2008.

[7] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, 2008.

[8] V. Kazempour, A. Fedorova, and P. Alagheband. Performance Implications of Cache Affinity on Multicore Processors. In *Proc. of Euro-Par '08*, pages 151–161, 2008.

[9] J. Koomey. Estimating Total Power Consumption by Servers in the US and the World. Technical report, Lawrence Berkeley National Laboratory, February 2007.

[10] D. Koufaty, D. Reddy, and S. Hahn. Bias Scheduling in Heterogeneous Multicore Architectures. In *Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys 2010)*, 2010.

[11] R. Kumar, K. I. Farkas, and N. Jouppi et al. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proc. of MICRO 36*, 2003.

[12] R. Kumar, D. M. Tullsen, and P. Ranganathan et al. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proc. of ISCA '04*.

[13] T. Li, D. Baumberger, and D. A. Koufaty et al. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In *Proc. of SC '07*, pages 1–11.

[14] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguade. Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors. *IEEE Comput. Archit. Lett.*, 5(1):4, 2006.

[15] D. S. Nikolopoulos, G. Back, J. Tripathi, and M. Curtis-Maury. VT-ASOS: Holistic System Software Customization for Many Cores. pages 1–5, April 2008.

[16] J. Saez, A. Fedorova, M. Prieto, and H. Vegas. Operating System Support for Mitigating Software Scalability Bottlenecks on Asymmetric Multicore Processors. In *Proceedings of the ACM International Conference on Computing Frontiers (CF)*, 2010.

[17] J. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A Comprehensive Scheduler for Asymmetric Multicore Processors. In *Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys) 2010*, 2010.

[18] A. Shah and N. Krishnan. Optimization of Global Data Center Thermal Management Workload for Minimal Environmental and Economic Burden. *Components and Packaging Technologies, IEEE Transactions on*, 31(1):39–45, March 2008.

[19] D. Shelepov, J. C. Saez, S. Jeffery, A. Fedorova, Z. P. Huang, N. Perez, V. Kumar, and S. Blagodurov. HASS: A Scheduler for Heterogeneous Multicore Systems. *ACM Operating System Review*, 43(2), 2009.

[20] T. Sterling and D. Stark. A High-Performance Computing Forecast: Partly Cloudy. *Computing in Science and Engineering*, 11(4):42–49, 2009.

[21] Y. Xie and G. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *Proc. of CMP-MSI*, 2008.