# AASH: ASYMMETRY-AWARE SCHEDULER FOR HYPERVISORS

by

Vahid Kazempour

B.Sc., Amirkabir University of Technology (Tehran Polytechnic), 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science


SIMON FRASER UNIVERSITY
Fall 2009

# APPROVAL

**Name:** Vahid Kazempour

**Degree:** Master of Science

**Title of Thesis:** AASH: Asymmetry-Aware Scheduler for Hypervisors

**Examining Committee:** Dr. Diana Cukierman
Chair

---

Dr. Alexandra Fedorova,
Assistant Professor, Computing Science
Simon Fraser University
Senior Supervisor

---

Dr. Richard Vaughan,
Assistant Professor, Computing Science
Simon Fraser University
Supervisor

---

Dr. Sathish Gopalakrishnan, Assistant Professor,
Electrical and Computer Engineering
University of British Columbia
Examiner

**Date Approved:** August 7th, 2009

# Declaration of Partial Copyright Licence

# Abstract

Asymmetric many-core processors (AMPs) consist of cores varying in size, frequency, power consumption and performance, all exposing the same instruction-set architecture. Since this architecture exploits both powerful fast cores and simple slow cores, it can oer a potential speedup that is much greater than symmetric architecture. This work for the rst time implements simple changes to the hypervisor scheduler, required to make it asymmetry-aware, and evaluates the benets and overheads of these asymmetry-aware mechanisms.

Our results indicate the signicant performance improvements, 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 guests expectations. Other benets from asymmetry awareness are fairer sharing of computing resources among VMs and more stable execution time.

*This thesis is dedicated to my wife Neda, who always supported me. It is also dedicated to my parents, who always beleived in me and taught me that even the largest task can be accomplished if it is done one step at a time.*

*"The wisest men follow their own direction."*

*— Euripides*

# Acknowledgments

I am heartily thankful to my supervisor Dr. Fedorova, who has been the ideal thesis supervisor. Her sage advice, insightful criticisms, and patient encouragement aided the writing of this thesis in innumerable ways. I would also like to thank Mr. A. Kamali and other Synar lab members for their support and invaluable help.

# Contents

# List of Figures

# Chapter 1

# Introduction

Asymmetric multicore processors (AMP) consist of cores exposing the same instruction-set architecture (ISA) but delivering different performance [2] [13]. The cores of an AMP system typically differ in clock frequency, power consumption, and possibly other microarchitectural features. A typical asymmetric processor could consist of several large *fast* cores (high clock frequency, complex out-of-order pipeline, high power consumption) and a large number of small *slow* cores (low clock frequency, simple pipeline, low power consumption).

Unlike symmetric multicore processors (SMP), AMP systems better cater to a diversity of workloads, and by doing so, they potentially deliver a better performance per watt and per area than SMPs [12] [13] [17]. For example, on an AMP system, highly CPU-bound applications with a lot of available instruction level parallelism (ILP) could be mapped to fast cores, while memory-bound applications that frequently stall the processor due to a high rate of memory requests could be mapped to slow cores. This mapping maximizes performance per watt, because applications are mapped to cores according to their needs: high-ILP applications that benefit from the features of complex cores run on fast cores, while memory-bound applications are mapped to slow cores due to the notion that the speedup they experience on fast cores relative to slow cores is disproportionately smaller than the additional power that the fast cores consume. Furthermore, fast cores on AMP systems could be used to accelerate sequential phases of parallel applications, effectively mitigating Amdahl's law [2] [12].

The potential of AMP systems can only be realized by means of proper scheduling support [4] [6] [15] [14] [20]. A process scheduler must ensure that the fast cores are allocated to those processes or threads that can use them most effectively (e.g., high-ILP threads or

threads executing sequential phases of parallel applications). Furthermore, it must ensure that scarce fast cores are shared equally among the threads.

Asymmetry-aware schedulers for operating systems have been proposed in recent work [6] [14] [20]. This work has shown that an asymmetry-aware scheduler can deliver as much as 38% improvement in comparison with an asymmetry-agnostic scheduler. These prior schedulers typically measure the runtime characteristics of the threads and then map threads to fast and slow cores based on these characteristics.

While these operating system-level solutions are very effective in the operating systems that run on bare hardware, they would not work in virtualized environments because existing hypervisor schedulers are not asymmetry-aware. As a result, the hypervisor could negate an asymmetry-aware scheduling policy used in the guest operating system. For example, suppose the guest operating system determines that one of its virtual CPUs is fast (e.g., by probing the CPU speed) and then tries to map the most high-ILP threads on that virtual CPU. However, if the hypervisor then remaps that virtual CPU to the slow physical core, the asymmetry-aware scheduling done in the guest will not be effective. This presents a problem for increasingly popular virtualized environments [16]. To address this problem, hypervisor scheduling algorithms must also be made asymmetry-aware.

In this study, asymmetry support in scheduling algorithms of hypervisors is addressed. To the best of our knowledge, this study is the first to address this problem. The first goal is to provide proper support for asymmetry-aware guests. The second goal is to design scheduling algorithms for asymmetry-unaware guests, since a vast majority of operating systems are still asymmetry-unaware. We design, implement and evaluate the following: an asymmetry-aware scheduler for hypervisors (AASH) that has the following properties:

1. **Support for asymmetry-aware guest operating systems:** We design a mechanism to ensure deterministic mapping of fast virtual cores to fast physical cores. Assuming that each virtual guest is entitled to a limited number of fast-core cycles, all fast-core cycles are given to the fast virtual core as opposed to being spread among all virtual cores.

2. **Equal sharing of fast cores cores among all guests:** Hypervisors typically run multiple guests on the same hardware. An asymmetry-agnostic scheduling algorithm could lead to unequal sharing of scarce fast-core resources among virtual guests. This

could result in unstable performance and complicate accounting. We provide a mechanism that ensures equal sharing of fast cores. This mechanism is useful for asymmetry-aware guests (when multiple such guests are running on the same asymmetric hardware) as well as for asymmetry-unaware guests. In addition to improving fairness, this algorithm significantly improves performance of parallel applications, because it equally accelerates all threads performing the computation, resulting in a more balanced utilization of resources.

3. **Acceleration of sequential phases on fast cores:** AMP systems can be effectively used for mitigating Amdahl's law if sequential phases of parallel applications are accelerated on the fast cores [2] [12]. The proposed scheduler provides this functionality for asymmetry-unaware guests. Assuming that each virtual machine runs a single application or service [10], our scheduler detects sequential phases in a virtual machine by monitoring the number of active virtual CPUs. When the number of active virtual CPUs reduces to one, the single active virtual CPU is mapped to a fast core, so that the virtual machine's sequential phase is accelerated. This method also better serves the asymmetry-aware operating systems when they enter their sequential phases.

4. **Providing coarse-grained prioritization in using fast cores:** We also implement a mechanism that allows prioritizing the usage of fast cores. There are two priority classes: *high* and *low*. A virtual machine in a high-priority class gets preference when fast-core CPU time is allocated. The remaining fast-core CPU time is allotted to other virtual machines. This mechanism can be used for implementing service differentiation policies.

Although the provided mechanisms inspire the design of new interesting Quality of Service policies, discussion of these policies is outside the scope of this work: our current focus is on the mechanisms and their effectiveness.

The proposed asymmetry-aware scheduler is implemented on top of the *Xen* hypervisor [5] and evaluated on a real multicore system that is configured to be asymmetric via Dynamic Voltage and Frequency Scaling (DVFS). DVFS provides the ability to set the cores to run at different frequencies. The effects of asymmetry-aware mechanisms on performance are studied, and the associated overheads are analyzed. It is demonstrated that the proposed asymmetry-aware scheduling mechanisms in the hypervisor delivers the following benefits:

- The scheduler provides fair sharing of scarce fast cores and predictable performance. We demonstrate that with the scheduler applications have much more stable completion times than with the default Xen scheduler.
- We show that asymmetry-aware guests can accomplish up to 16% better performance with our scheduler.
- Virtual machines running parallel applications with sequential phases can experience up to 31% performance improvement with the proposed scheduler relative to an asymmetry-unaware default scheduler.

The benefits of these solution are accomplished by means of periodic migrations of virtual CPUs among fast and slow cores. These migrations, an essential tool of any asymmetry-aware scheduling algorithm, may be quite costly since a migrated thread may lose the cache state accumulated in the previous core and suffer performance degradation. The implementation in this work is carefully crafted to avoid this overhead. The migration related performance degradation is very small with the proposed scheduler. It is negligible for most applications and reaches at most 4% in isolated cases. Demonstrating how to implement asymmetry-aware scheduling with low migration overhead is an important contribution of this work.

# Chapter 2

# Design and Implementation

The proposed scheduler is called *AASH* (*A*symmetry-*A*ware *S*cheduler for *H*ypervisors), and it is implemented on top of the *Credit* scheduler. Credit scheduler, the default scheduler in the Xen hypervisor [5] [9], is a simple credit-based scheduler with a number of characteristics that makes it a suitable choice for symmetric multicore processors (SMPs). This scheduler automatically balances the load between all available cores. Before going idle, CPUs will try to steal work from other CPUs if they can find any. It is a fair scheduler which implies that all virtual machines get the same time slice to run on the CPUs. Credit scheduler works by assigning credits. Virtual machines are assigned to CPUs based on the amount of credit they have.

In the Credit scheduler, queues are local to CPUs and each CPU handles its own round-robin scheduling. In order for a virtual CPU to be scheduled, it needs to be on a physical core's run queue. This run queue is a priority queue. There are three types of elements (virtual CPUs) in this queue: *over* virtual CPUs that have consumed all their credits and can run on the physical CPU if there is extra time, *under* virtual CPUs that are yet to consume their credits, and *boosted* virtual CPUs that have the highest priority. When there is a *boosted* virtual CPU at the head of the run queue, it would cause the currently running virtual CPU to get descheduled (if the currently running virtual CPU is not *boosted* itself). There is a run queue that all virtual CPUs to be scheduled are put into and there is a *current* virtual CPU which is currently running on the physical CPU. The CPU keeps the current virtual CPU running, and when it is done with the current virtual CPU, it puts it back into the run queue and takes another virtual CPU from the head of the run queue. A virtual CPU keeps running on a CPU until it has consumed all its credits or until its time slice is

over. In the first case, the priority of the running virtual CPU changes from *under* to *over* and it will be inserted into the run queue. When inserting a virtual CPU into the run queue, the virtual CPU will be inserted according to its priority. The scheduler needs to work in a round-robin fashion, so every time a virtual CPU is descheduled, it will be inserted after all virtual CPUs with the same priority to preserve fairness. In other words, the descheduled virtual CPU will be inserted before the first virtual CPU with a lower priority.

Normally just *under* and *over* virtual CPUs exist in the run queue. *Boosted* virtual CPUs are a special case of I/O bound virtual CPUs. I/O bound virtual CPUs usually do not consume much CPU time, waking up periodically to process the data and then waiting for I/O again. The concept behind *boosted* virtual CPUs is that this type of virtual CPUs is likely to yield to other virtual CPUs because they are doing I/O. Therefore, it is reasonable to schedule them first. If a *boosted* virtual CPU consumes more than a tick of CPU time, its priority will be changed to *under*. In short, whenever a virtual CPU wakes up (e.g. from I/O), it will have a *boosted* priority. If it consumes too much CPU, its priority will be changed back to *under*.

A scheduling clock tick is a period of time during which the scheduler wakes up and checks followings: (1) the state of the run queue for the presence of a virtual CPU with higher priority, and (2) the current virtual CPU to find out if all its credits are consumed. Time slices are equivalent to three ticks. Therefore, when the scheduler schedules a virtual CPU on a CPU, it will wake up three times during the runtime of the virtual CPU to check the states (e.g. if the tick time is 10ms, the time slice time will be $10 * 3 = 30$ms. A virtual CPU will be run for 30ms at most, and during this 30ms the scheduler wakes up every 10ms to do some local accounting). Tick timers are per core. Every core wakes up independently and checks its own run queue independent of the other cores.

These are the characteristics of the Credit scheduler which we retained in our scheduler. The AASH scheduler inherits the same queues, clock ticks and time slices, and it works based on the same mechanism as the Credit scheduler. Although these mechanisms work well on SMPs, they are not sufficient for supporting AMP systems. The Credit scheduler does not recognize the asymmetry of the underlying hardware and therefore cannot benefit from this asymmetry.

The AASH scheduler addresses asymmetry support in scheduling algorithms of hypervisors. Four main criteria have been considered in designing this algorithm: (1) equal distribution of resources in asymmetric many-core environments, (2) supporting asymmetry-aware

operating systems, (3) mitigating the effects of Amdahl's law by scheduling single-threaded workloads and sequential phases of multi-threaded applications on powerful cores, and (4) providing coarse-grained prioritization. The following sections explain how each of these goals is addressed by the AASH scheduling algorithm.

## 2.1 Equal sharing of fast cores among all guests

As mentioned earlier, the default Xen scheduler is unaware of the asymmetry of the system, and as a result it arbitrarily assigns virtual CPUs to the physical cores. In this case, some virtual machines may not be scheduled on the fast cores during their entire run time. In some situations, where the number of active virtual CPUs is smaller than the number of available physical cores, the default Xen scheduler may schedule no virtual CPU on the fast cores. This results in unpredictable and unfair performance in the system. Besides that, the default scheduler tries to avoid unnecessary migration of virtual CPUs on the physical cores, in order to preserve cache affinity. Consider a hypothetical scenario where each virtual machine is a container for a single application, so all threads in the virtual machine cooperate. In multi-threaded applications these threads may often synchronize with one another, so one thread cannot proceed until all threads reach the synchronization point. If the default scheduler runs one of the virtual CPUs on a fast core, one thread among all threads of the parallel application inside that virtual machine will finish earlier than the others. However, the application as a whole does not necessarily experience proportional speedup, because the mentioned thread cannot go beyond the synchronization points and needs to wait for all other threads to finish and reach the synchronization point.

The AASH scheduler addresses these problems by exposing the asymmetric nature of the system to the hypervisor and sharing fast cores equally among all virtual CPUs. Therefore all virtual CPUs experience the same speedup proportional to the amount of time they were scheduled on a fast core. In the above scenario, even virtual machines with a multi-threaded application of cooperating threads may benefit from running on the fast cores. Remember that this timesharing behavior is in effect when all virtual machines on the system have the same priority. In cases when there are virtual machines with higher priorities, e.g. due to entering a sequential phase, fast cores will not be timeshared between all virtual CPUs, but between those virtual CPUs belonging to the virtual machine of a high priority. These cases will be explained later in this section.

In the AASH scheduler, a system-wide thread will wake up every three ticks to perform global accounting. This thread updates the credits of the virtual CPUs. Credits are a metric to make sure that every virtual CPU gets a specified amount of CPU time. The more credits a virtual CPU has, the more it can run on a physical core. There are two types of credits in our scheduler: *fast* and *slow* credits. Only virtual CPUs that have fast credits can run on the fast cores. The whole accounting system is based on credits. On each accounting period, a number of credits (fixed and determined by the resources of the system) is distributed between all virtual CPUs. Virtual CPUs consume their credits when they are running on a physical core. For example if there is a fast core that needs to be shared between two virtual CPUs, in the first accounting period some fast credits will be assigned to the first virtual CPU so that it can run on the fast core, and in the second accounting period the same amount of fast credits will be assigned to the second virtual CPU so that the second can run for an equal duration on the fast core. Both fast and slow credits are required to make sure that every virtual CPU gets the same amount of CPU time on the physical cores.

In each accounting period, fast and slow credits will be distributed among virtual CPUs. At first, the amount of slow credits is computed. This is equal for all virtual CPUs. To get the number of slow credits to distribute, the scheduler divides the total slow credits present in the system (determined by the number of slow cores) by the total number of active virtual CPUs minus the ones that will be served by fast cores. To reduce the number of migrations fast credits will only go to one virtual CPU per accounting period (i.e. one virtual CPU if we have only one fast core, if we have two fast cores then fast credits will go to two virtual CPUs per accounting period etc.). To make sure that every virtual CPU gets fast credits to run on the fast cores, a list of virtual CPUs is maintained in the *normal queue*. In the accounting period, a virtual CPU will be taken from the top of the normal queue and assigned fast credits. This is a First In First Out (FIFO) queue, therefore, after a virtual CPU gets to run on the fast core, it is inserted at the end of the normal queue. This behaviour is generalized so that if there are multiple fast cores present in the system ($n$), in each accounting period "$n$" virtual CPUs are taken from the normal queue and are assigned fast credits. Note that only "$n$" cores are assigned fast credits, this is necessary to minimize the queuing on the run queues.

### 2.1.1 Migration and work stealing

Migration is the task of moving one virtual CPU from the run queue of one physical core to another. There are two types of migrations:

1. It can be due to load imbalance (work stealing). Work stealing is performed when one CPU has a non-empty run queue while another CPU is idle. Before a CPU goes idle, it first checks the run queue of all other CPUs.

2. To ensure fair sharing of fast cores, virtual CPUs must be migrated to run queues of the fast cores so that they can be scheduled. When they have consumed their fair share of fast core credits, they must be migrated back to the run queue of a slow core so that they can run there until the next time they can run on the fast core.

Physical cores do not steal virtual CPUs from physical cores of a different type. A fast core does not steal work from the run queue of a slow core and vice versa. Instead the credit distribution and the second type of migration (to ensure fair sharing of fast credits) are done so as to preserve load balance. First, the credits are distributed so that a fast core will never go idle before a slow core: in each accounting period $n$ cores are assigned fast credits. There are $n$ fast cores and there are $n$ virtual CPUs with fast credits. Therefore, a fast core will never need to steal a virtual CPU from the run queue of a slow core. However, a fast core may steal from another fast core. Second, when a migration is needed, the scheduler does so in a way to make sure that it will not cause a load imbalance. Before explaining how this is done, it is necessary to understand the second type of migrations.

The second type of migration occurs when a virtual CPU needs to be moved from the run queue of a slow core to the run queue of a fast core (because now it has fast credits) or when it needs to be moved from the run queue of the fast core it was running on to the run queue of a slow core (because it has consumed all its fast credits). On each tick, each physical core examines the currently running virtual CPU for an update in credits. If the current virtual CPU has run out of credits the priority will be changed to *over*. If a virtual CPU running on the fast core has finished its fast credits, that virtual CPU will marked for a migration to a slow core. If a virtual CPU running on a slow core now has some fast credits, it will be marked for migration to a fast core. There is a chance of creating a load imbalance here: if the scheduler just moves the virtual CPU from fast core to any slow core it may cause a load imbalance at the target slow core. Load imbalance is prevented by

moving the virtual CPU from the fast core to the run queue of the slow core which is likely to go idle in the next accounting period (because in the next accounting period the running virtual CPU there would be assigned some fast credits and will be forced to move to the run queue of a fast core). This is easy to detect since we are maintaining a list of virtual CPUs that are going to get fast credits on the next period.

## 2.2 Support for asymmetry-aware guest operating systems

Mapping of virtual CPUs in a asymmetry-aware guest to fast and slow physical cores must be deterministic so that the guest operating system can rely on its knowledge of fast cores. If the guest operating system assumes the first virtual CPU is a fast core but the hypervisor maps the first virtual CPU to a slow core, the guest operating system will fail to benefit from its asymmetry-awareness. To address this issue, AASH scheduler maps a subset of virtual CPUs to fast cores (assume there are "$n$" fast cores in the system and the AASH scheduler maps the first "$n$" virtual CPUs to these physical fast cores) and lets the rest of the virtual CPUs run on the slow cores. At the end of the accounting period, the AASH scheduler scans the head of the *normal queue*. If it finds a virtual CPU belonging to a asymmetry-aware guest that must not be assigned fast credits (because the virtual CPU id is bigger than "$n$"), it swaps that virtual CPU with another virtual CPU from the same virtual machine that can be assigned fast credits. Remember that the virtual CPUs that are in the front of the queue will be assigned fast credits in the next accounting period. In this way, the scheduler makes sure that only the first "$n$" virtual CPUs get fast credits and that the credit distribution is fair and proportionate to the number of active virtual CPUs of each virtual machine.

An example would help understanding how the AASH scheduler deals with asymmetry-aware guests. Assume that there are two fast and six slow cores present on the system and two virtual machines (*VM1* and *VM2*) are running. VM1 runs an asymmetry-aware guest while VM2 runs an asymmetry-agnostic guest. Since there are two physical fast cores, only the first two virtual CPUs of VM1 will be scheduled on the fast cores and the two remaining ones will be scheduled to run on any of the six slow cores. However, the fast cores will be time-shared between all four virtual CPUs of the second virtual machine. AASH scheduler is still fair in this scenario, because it will assign twice as many fast credits to each virtual CPU of the first virtual machine. In other words, each virtual machine gets

fast credits proportionate to the number of virtual CPUs it has. If the virtual machine is not asymmetry-aware, that amount of credits will be distributed between all virtual CPUs; while if the virtual machine is aware of the asymmetry, that amount of credits is distributed between the designated virtual CPUs in a way that only a subset of virtual CPUs run on the fast cores and the asymmetry of the system will be visible in the virtual machine.

## 2.3 Acceleration of sequential phases on fast cores

We have slightly changed the behavior explained above to add support for phase-awareness. Instead of keeping only one list of virtual CPUs, there are two lists. The first list, i.e., the *normal queue* works as explained when the second list, i.e., the *fast queue* is empty. When there are some virtual CPUs in the fast queue, they will be assigned fast credits first. If there are fast credits remaining, that will be distributed between the virtual CPUs in the normal queue. In other words, the fast queue's virtual CPUs have a higher priority than the normal queue's virtual CPUs. A virtual CPU moves from the normal queue to the fast queue when its virtual machine enters a sequential phase. The scheduler detects the sequential phase by monitoring the number of active virtual CPUs of that virtual machine. The reason behind this decision is that when a parallel virtual machine enters a sequential phase[1] it will not use all its virtual CPUs. For example when a virtual machine with eight virtual CPUs is only using one of its virtual CPUs, it means that virtual machine is running a sequential application (or a sequential phase of a parallel application). Therefore, the decision is made by monitoring the number of active virtual CPUs of each virtual machine. Whenever the number of active virtual CPUs of a virtual machine is smaller than the number of the fast cores present in the system, all active virtual CPUs of that virtual machine will be moved to the fast queue. When the virtual machine begins using more virtual CPUs, the virtual CPUs will be moved back to normal queue.

Since the AASH scheduler is capable of detecting the phase changes of parallel applications where unused threads go to sleep during sequential phases, it schedules the sequential components on the fast cores in order to accelerate the sequential part of the application. The AASH scheduler gives a higher priority to the virtual machines that are not parallel. Consider a case where a virtual machine with only one virtual CPU is co-scheduled with

[1]We assume that threads will sleep during sequential phases. We do not address the case where threads spin during sequential phase.

a virtual machine with three virtual CPUs and the physical system only has one fast core. There are three options for sharing the fast core in this scenario. The first option is to assign the fast core to one of the cores of the parallel virtual machine and let the serial virtual machine run on the slow core. It is most probable that this strategy would not improve the performance of the parallel virtual machine. As we mentioned before, parallel applications (which are running on parallel virtual machines) cannot finish earlier than normal if one of their threads is accelerated. Usually all threads in an application are synchronized at one point, and if one of the threads finishes earlier than others, it still needs to wait for the other threads to finish. Therefore, accelerating one thread out of all threads of a parallel application will not result in reducing the runtime of the application. The second option is to run the serial virtual machine on the fast core and run the parallel virtual machine on the slow cores. The parallel virtual machine is already benefitting from the parallelism. By running the serial virtual machine on the fast core, the performance of that virtual machine will be improved proportionally to the speed of the fast core. The third option is to timeshare the fast core between both virtual machines. Since there are four virtual CPUs and only one fast core, all virtual CPUs will run faster on our system. The fast core in our system runs 2X faster than slow cores, therefore, each virtual CPU will run 25% faster than if no fast credits were assigned to it. With the third option, the performance of both virtual machines is improved by 25%; while with the second option, the performance of one virtual machine is improved by 100% and the performance of the other virtual machine is not hurt.

## 2.4 Coarse-grained prioritization in using fast cores

The AASH scheduler supports two levels of priorities. To prioritize a virtual machine, the corresponding virtual CPUs of that virtual machine move from the *normal queue* to the *fast queue*. In this way, fast cores will be given to the prioritized virtual machine first (as well as to the serial virtual machines) and if there are extra fast credits remaining, they will be distributed between the remaining virtual machines.

# Chapter 3

# Evaluation

In this chapter, we evaluate the efficiency of the AASH scheduler. The hardware configuration is described in Section 3.1. In Section 3.2, the overhead of inevitable migrations performed by the AASH scheduler is evaluated. In Section 3.3, the mechanisms for fair sharing is evaluated. This evaluation is mentioned before others, because this mechanism is the basic algorithm on top of which other mechanisms are built. In Section 3.4, the supports for asymmetry-aware operating systems are evaluated. Next in section 3.5, the phase-awareness feature is evaluated. Finally, we evaluate the prioritization mechanism in using fast cores in section 3.6. The results of the AASH scheduler are compared with those of the Credit scheduler (the default Xen scheduler).

## 3.1 Hardware configuration

We chose an AMD Opteron 2350 Barcelona, with two quad-core chips, as our experimental machine. 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. Our system is populated with 8 GB of RAM. We assumed 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 core typically use small 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. [13], supporting only two core types is sufficient for achieving most of the potential of asymmetric designs.

Asymmetry was emulated by setting cores to run at different frequencies using dynamic voltage and frequency scaling (DVFS). Several test configurations were created for different experiments. In some configurations, fewer cores than the total available were used to avoid any performance effects due to cache sharing, and due to our console monitoring applications that were running in administrative domain.

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 Cloud Computing initiative or in other public data centers, such as West Grid. Our evaluation provides insight into potential impact of asymmetry-awareness in hypervisors. We found these benefits to be quite significant.

## 3.2 Evaluating the overhead of migrations

Recall that the AASH scheduler timeshares the fast cores among all 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 (LLC). Cross-memory-domain migrations cause the migrated virtual CPU to lose the state accumulated in the LLC. Rebuilding this state after migration may cause performance degradation.

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 [22] to determine which applications are cache-sensitive and which are not. For

cache sensitive applications, we chose *leslie3d* and *libquantum* from the SPEC CPU2006 benchmark suite and *mcf* from the SPEC CPU2000 benchmark suite. These workloads are sensitive to changes in cache availability and have high cache miss rates. For cache insensitive applications, we chose *calculix*, *namd* and *sjeng* from the SPEC CPU2006 and *sixtrack* from the SPEC CPU2000. These applications have low cache access rates [22], and thus we expect a low sensitivity to the loss of cache investment when a migration is performed.

Each workload was run under AASH scheduler with a 3ms, 10ms and a 25ms scheduling clock ticks and under the default Xen scheduler with a 10ms clock tick. Figure 3.1 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 4%. Cache-insensitive applications are largely unaffected by migrations. We have evaluated the sensitivity of performance to migration frequency, changing the timeslice from 3 to 25 milliseconds, and found that the overhead slightly increases when the timeslice is reduced, and decreases when it is increased.

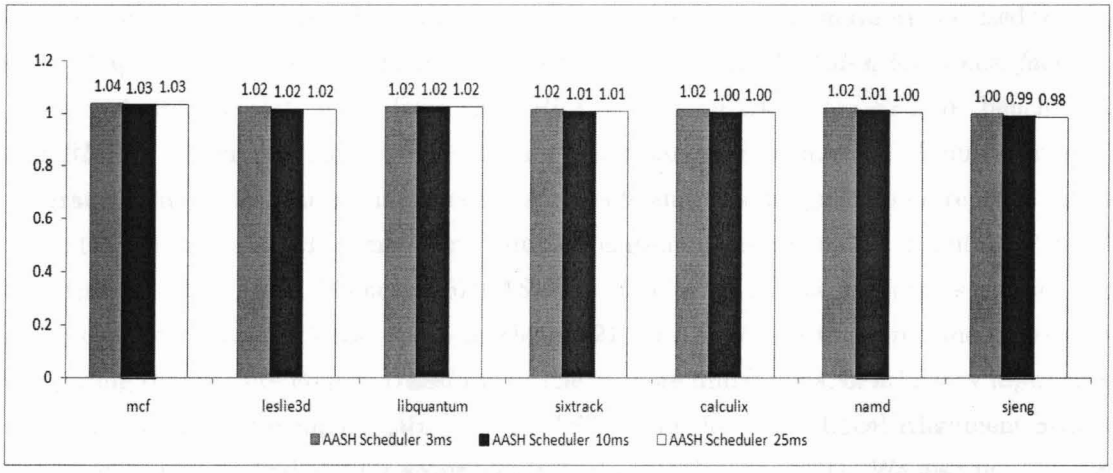



Figure 3.1: Normalized execution times under the AASH scheduler with 3ms, 10ms and 25ms clock ticks relative to the default scheduler

We conclude that migrations do have an effect on performance, albeit a small one. At the same time, migrations are an integral part of any asymmetry-aware scheduling algorithm, as they are the key mechanism used to accomplish various policies of resource sharing. Since the overheads incurred with all migration frequencies, we chose to use 10ms as the

default clock tick in our scheduler to constitute a more fair comparison with the default Xen scheduler that uses a 10ms clock tick and did not investigate the effects of using a larger clock tick on other aspects of performance and fairness.

## 3.3 Evaluating equal sharing capability

In this section, an experimnet is explained to evaluate equal sharing capability of the AASH scheduler, and following resulsts are demonstrated:

1. The AASH scheduler accomplishes equal sharing of fast and slow cores among virtual CPUs, delivering more stable performance time, while the default asymmetry-unaware scheduler is unable to provide these benefits.

2. The AASH scheduler delivers better performance for many applications, because it accomplishes a better utilization of fast cores.

Three different types of workloads were used in this experiment. We used the same sets of sensitive and insensitive benchmarks as in the previous section. Furthermore, we used several parallel applications from the *PARSEC* benchmark suite (*blackscholes*, *bodytrack*, *facesim*, *ferret* and *uidanimate*), *radix* from the *SPLASH* benchmark suite, *FFT-W* benchmark and *BLAST* benchmark suite. *PARSEC* is a benchmark suite composed of multi-threaded programs. The suite focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors [7]. Stanford Parallel Applications for Shared Memory (*SPLASH*) is a suite of parallel programs written for cache coherent shared address space machines [21]. *FFT-W* is a C subroutine library for computing the discrete Fourier transform in one or more dimensions, of arbitrary input size, and of both real and complex data [11]. *BLAST* stands for Basic Local Alignment Search Tool and is used to find similar sequences in biological databases [1]. We also use the *eon* workload from the SPEC CPU2000 benchmark suite.

Two different configurations were used in this experiment. In the first configuration, we used four identical virtual machines. Each virtual machine has one virtual CPU, and the virtual machines were scheduled on the four physical cores (one fast core running at 2 GHz and three slow cores running at 1 GHz). We ran the same benchmark on all virtual machines to simplify the comparison of the completion times for different virtual machines. The completion time of each workload was measured in each virtual machine.

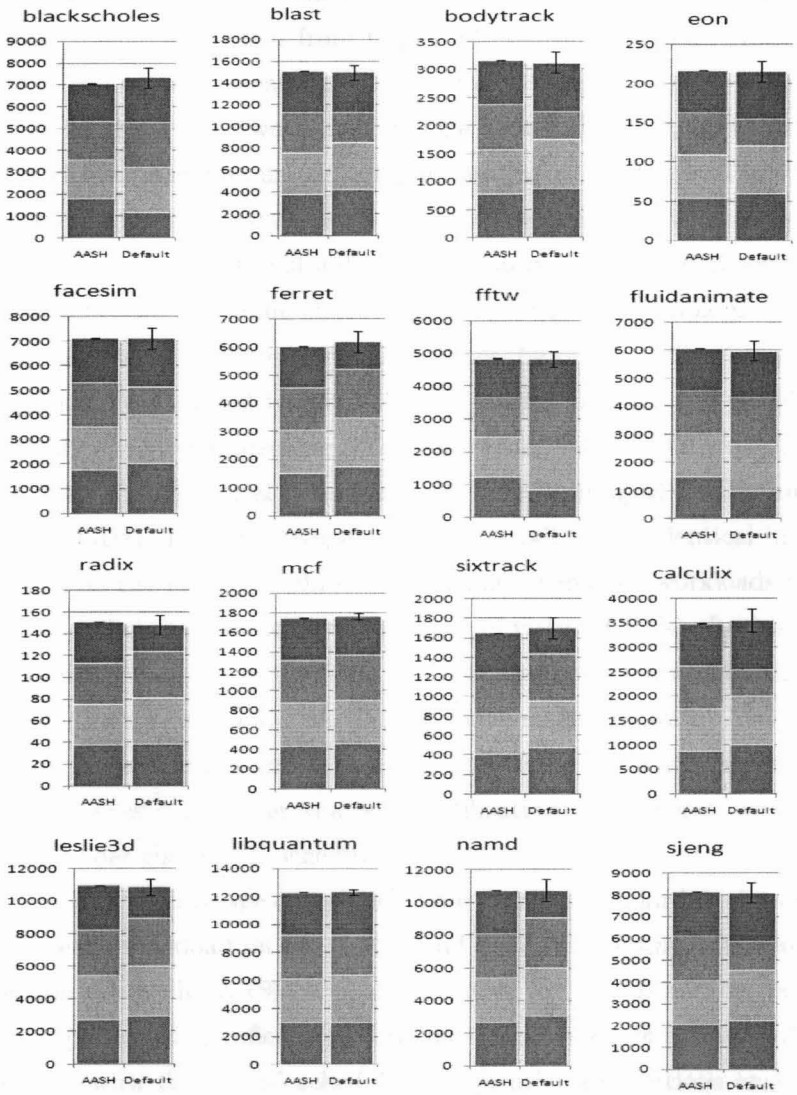



Figure 3.2: Configuration 1: Completion times on four VMs

Figure 3.2 shows the completion time of the benchmarks using the AASH scheduler and the default Xen scheduler. The vertical bar shows the completion time. In Figure 3.2 each bar is made of four parts showing the completion time of each instance of the benchmark in a different virtual machine. The black line at the top of each bar shows the standard deviation of the completion times. As evident from Figure 3.2, the bars for the AASH scheduler are divided into four equal parts, meaning that the four virtual machines (running identical benchmarks) are finishing their work at the same time. However, with the default Xen scheduler, some virtual machines are finishing earlier than the others. This is because the default Xen scheduler is asymmetry-unaware and it may run some VMs on fast cores more than others. We also see that the standard deviation of completion times is much lower under the AASH scheduler, meaning that it delivers a more predictable performance.

The goal of the second configuration was to show the better utilization of fast cores in the AASH scheduler. We wanted to evaluate both single-threaded and parallel workloads, and so we used only one virtual machine with eight virtual CPUs. Our experimental system was configured with eight physical cores (one fast core running at 2 GHz and seven slow cores running at 1 GHz). In case of single-threaded workloads, $n$ identical instances of the workload were run on the virtual machine, and for multi-threaded workloads the number of threads was set to $n$ (where $n$ matches the number of virtual CPUs). In this way, we have at most one running thread on each virtual CPU. Since threads would finish faster on the fast cores, it would be the case under the default (asymmetry-unaware) Xen scheduler that fast cores have no threads to run and they would go idle. In the AASH scheduler fast cores never go idle before slow cores. Due to a better utilization of fast cores, we expected lower completion times under the AASH scheduler.

Figure 3.3 presents the average completion time and the standard deviation of the completion times for each workload on the AASH and default Xen schedulers, and Figure 3.4 shows the speedup under the AASH scheduler relative to Xen. It can be seen that parallel applications experienced significant performance improvements (up to 31% in *FFT-W*) when they are run with the AASH scheduler. Sequential applications experienced small performance overhead due to migrations as explained in Section 3.2. The parallel applications experienced a speedup under the AASH scheduler for two reasons. First, AASH delivers a more balanced utilization of CPU resources. As a result, all virtual CPUs (and

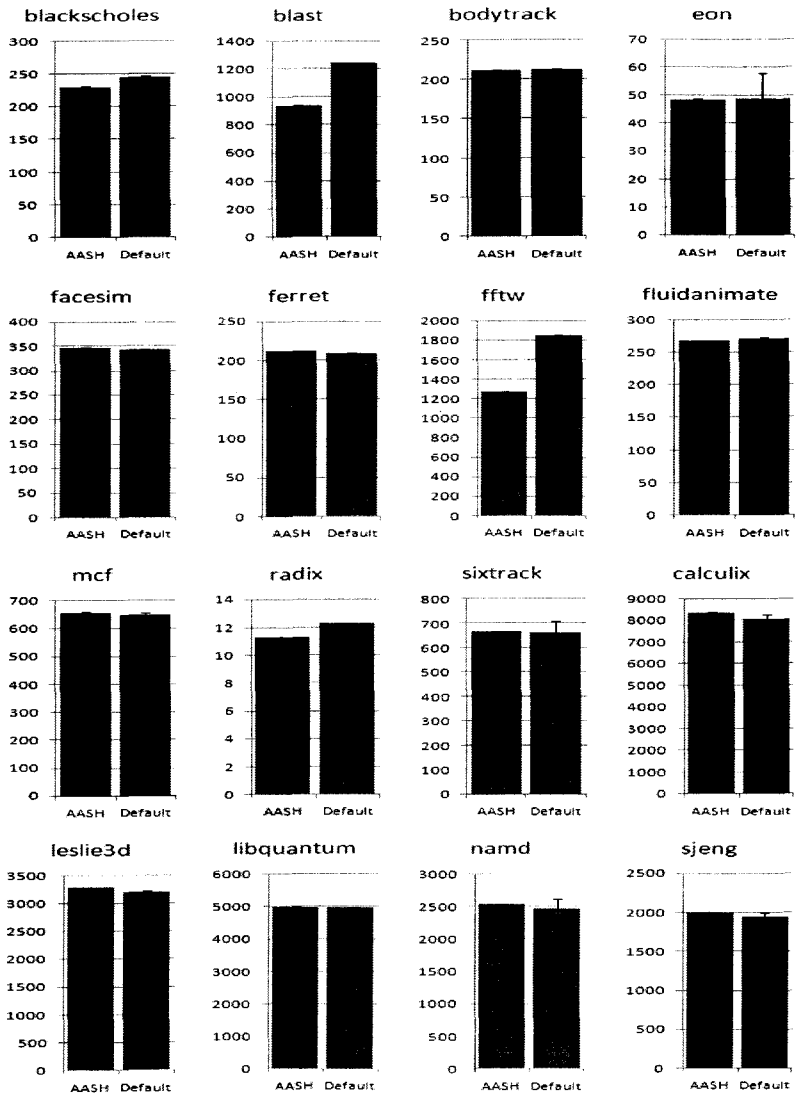



Figure 3.3: Configuration 2: Average Completion Times on one VM

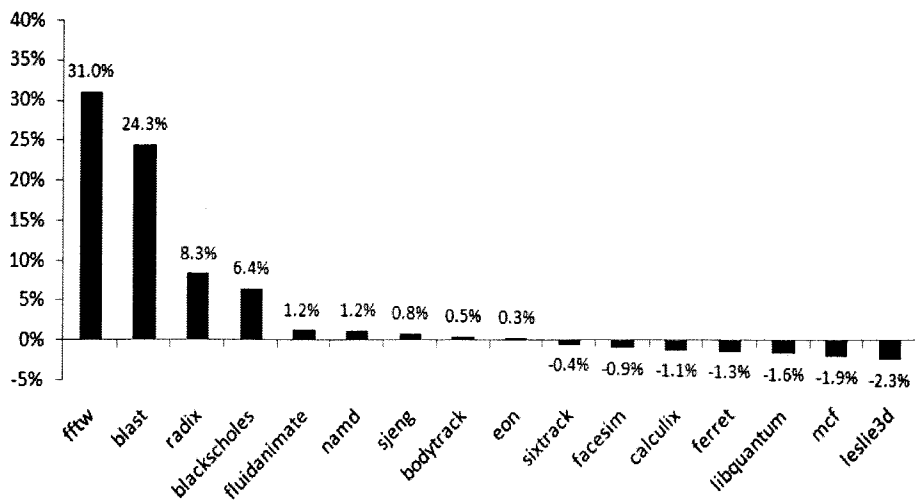



Figure 3.4: Configuration 2: Speedups on the AASH scheduler relative to the default scheduler

thus all threads[1]) are equally accelerated on fast cores. Threads thus advance at an equal pace rather than finishing at different times . The other source of performance improvement comes from the fact that AASH utilizes the fast cores more efficiently for applications with long sequential phases. *BLAST* and *FFT-W* have rather long sequential phases: they spend 44% and 87% respectively running with only a single active thread (and a single active virtual CPU). The AASH scheduler accelerates these sequential phases on the fast cores, because it ensures that the fast cores are always busy as long as there is something to run on them. The asymmetry-unaware default Xen scheduler, on the other hand, may leave the virtual CPU running the sequential phase on the slow core, delivering no acceleration to this bottleneck part of the application. The ability to utilize fast cores more effectively enables AASH to deliver up to 31% performance improvement to parallel applications with large sequential phases. Other parallel applications have smaller sequential phases (or none at all), so they experience smaller (albeit still significant) performance improvements.

Note that in these single-application experiments, acceleration of sequential phases would occur under any asymmetry-aware scheduler that was designed to ensure that the fast cores do not go idle before slow cores. When multiple virtual machines are running, the scheduler

[1]Assuming one thread per virtual CPU and assuming that the guest operating system scheduler does not move threads among virtual CPUs too frequently.

needs to explicitly detect that a particular virtual machine has only a single virtual CPU that is active and give that virtual CPU a priority on a fast core. Our scheduler is equipped with this feature, and we evaluate it in Section 3.5.

## 3.4 Evaluating support for the asymmetry-aware guest OS

In this section, we first present experiments with a single asymmetry-aware guest running on top of the hypervisor equipped with the AASH scheduler. We show that it achieves better performance than when it runs over a hypervisor using the default Xen scheduler. This is due to the fact that our scheduler provides a deterministic mapping of fast virtual CPUs to fast physical cores allowing the guest to implement its asymmetry-aware scheduling policies.

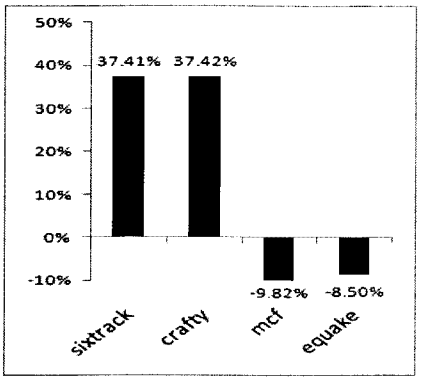
Our scheduler also supports the co-existence of asymmetry-aware guests and legacy guests. We evaluate this feature in the second experiment, by running both types of the guests simultaneously. In this case, comparing the average completion times on the AASH and the default Xen schedulers shows that both guests benefit from running under our scheduler.

The following sub-sections present these two experiments.

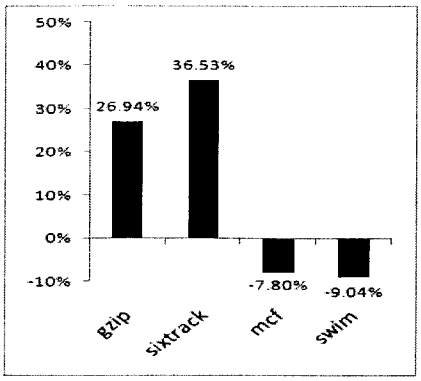
### 3.4.1 Single-VM experiments

To evaluate this feature, we use the same workloads as have been used in [20] to evaluate an asymmetry-aware operating system scheduler. 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*, *equake* and *swim*. The first two benchmarks in each set are CPU-bound and the second two are memory-bound. The experiments in the aforementioned work were run on a system with two fast cores and two slow cores. The asymmetry-aware operating system scheduler ended up mapping the CPU-bound applications to fast cores and the memory bound applications to slow cores[2]. To mimic that scheduling policy in our experiment, we bind the CPU-bound applications to the first two virtual CPUs and the memory-bound applications to the second two virtual CPUs inside the guest operating system. Since the assumption made by the guest is that the first two

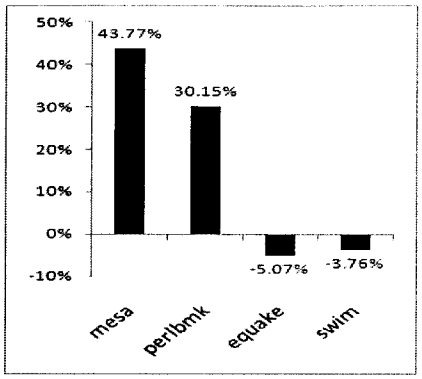
[2]A highly CPU-bound applications with a lot of available instruction level parallelism (ILP) would benefit from running on the fast cores, while memory-bound applications that frequently stall the processor due to high rate of memory requests could be mapped to slow cores.



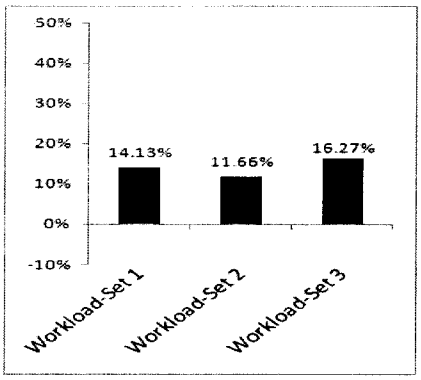

(a) Normalized Speedup of each Benchmark in Set 1 under the AASH scheduler relative to the default scheduler




(b) Normalized speedup of each benchmark in Set 2 under the AASH scheduler relative to the default scheduler




(c) Normalized speedup of each benchmark in Set 3 under the AASH scheduler relative to the default scheduler




(d) Average speedup of each Set of workloads under the AASH scheduler relative to the default scheduler

Figure 3.5: Single-VM Experiment

virtual CPUs are *fast* and the second two virtual CPUs are *slow*, the scheduler needs to respect that mapping to the actual physical cores. The AASH scheduler is equipped with this feature. The default Xen scheduler, on the other hand, is asymmetry-agnostic, and so it performs an arbitrary mapping of virtual to physical cores.

Figure 3.5 shows the results of this experiment. As can be expected, the asymmetry-aware guests performed better under the AASH scheduler. The mean speedup (Figure 3.5(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 (in Figures 3.5(a), 3.5(b) and 3.5(c)) the first two CPU-bound applications in the workload speedup under the AASH

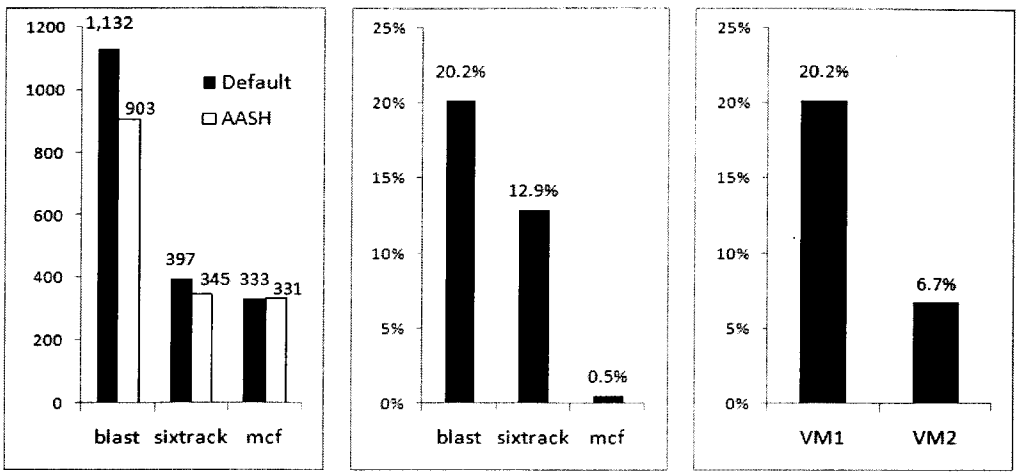
scheduler, while the second two memory-bound applications slowdown. Since the speedup experienced by the CPU-bound applications is greater than the slowdown experienced by the memory-bound applications, the workload as a whole experiences an improvement in performance.

### 3.4.2 Experiments with asymmetry-aware and legacy guests

In this experiment, we used two virtual machines, with each representing a type of guest: a legacy guest and an asymmetry-aware guest. The first virtual machine runs an asymmetry-unaware guest and has six virtual CPUs. It runs a parallel workload that has a significant sequential phase. We ran an instance of *BLAST* with six threads on the first virtual machine. 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-bound application (*mcf*). As in the previous experiment, we simulate asymmetry awareness in the second virtual machine by binding *sixtrack* to the first virtual CPU (the one that is assumed 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.

The AASH scheduler shares the fast core among all virtual machines. Since the second virtual machine is asymmetry-aware, the AASH scheduler assigns fast-core shares to the first virtual CPU in this guest. This results in: (1) equal sharing of scarce fast cores among both guests and (2) providing the deterministic mapping of the fast core for the asymmetry-aware guest.

Figures 3.6(a) and 3.6(b) shows the completion time and speedup for each benchmark under the AASH relative to the default Xen scheduler. *Sixtrack* (the CPU-intensive workload on the asymmetry-aware guest) shows a 13% performance improvement. The mean speedup of the asymmetry-aware guest is 6.7% (Figure 3.6(c)). The asymmetry-unaware guest which runs *BLAST* also shows a 20% speedup. These results demonstrate that both legacy and asymmetry-aware guests benefit from running under the AASH scheduler and show the ability of the AASH scheduler to successfully mix and match different scheduling policies.




(a) Average completion times of each benchmark under both schedulers

(b) Speedup of each benchmark under the AASH scheduler normalized to the default scheduler

(c) Average Speedup of each VM under the AASH scheduler Normalized to the default scheduler

Figure 3.6: Experiments with asymmetry-aware and legacy guests

## 3.5 Evaluating acceleration of sequential phases

In this experiment, we evaluate the mechanism for accelerating sequential phases in the AASH scheduler. Recall from Section 3.3 that when a single application is running, the base AASH scheduler (without the sequential-phase acceleration feature) delivers the acceleration of a sequential phase, because it would always schedule the single active virtual CPU to run on the fast core. In this section, we show that the scheduler also accomplishes acceleration of sequential phases when multiple virtual machines are running.

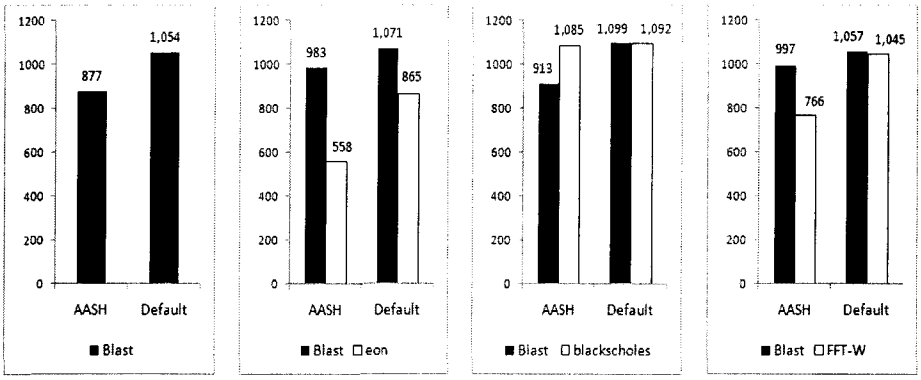
Depending on the number of threads and the size of the sequential phase, four possible combinations of workloads may co-exist in the system:

1. All workloads are completely parallel, and have no or very short sequential phases.
2. Some workloads are single-threaded while others are parallel.
3. Some of the workloads are completely parallel, but others have sequential phases.
4. All workloads have sequential phases.

In the first scenario, since workloads have very small sequential phases, acceleration of these phases on fast cores could not improve the overall performance. However, according to

the results of Section 3.3, the asymmetry-awareness feature of the AASH scheduler provides some improvement in comparison with the default Xen scheduler, due to the fact that the AASH scheduler equally shares the fast cores among all virtual CPUs and speeds up all the threads in parallel applications equally.

However, other scenarios could benefit from acceleration of their sequential phases on the fast cores. The following sub-sections describe the experiments performed with those three multi-VM workloads.




(a) Average completion time of BLAST under both schedulers

(b) Average completion time of BLAST and eon under both schedulers

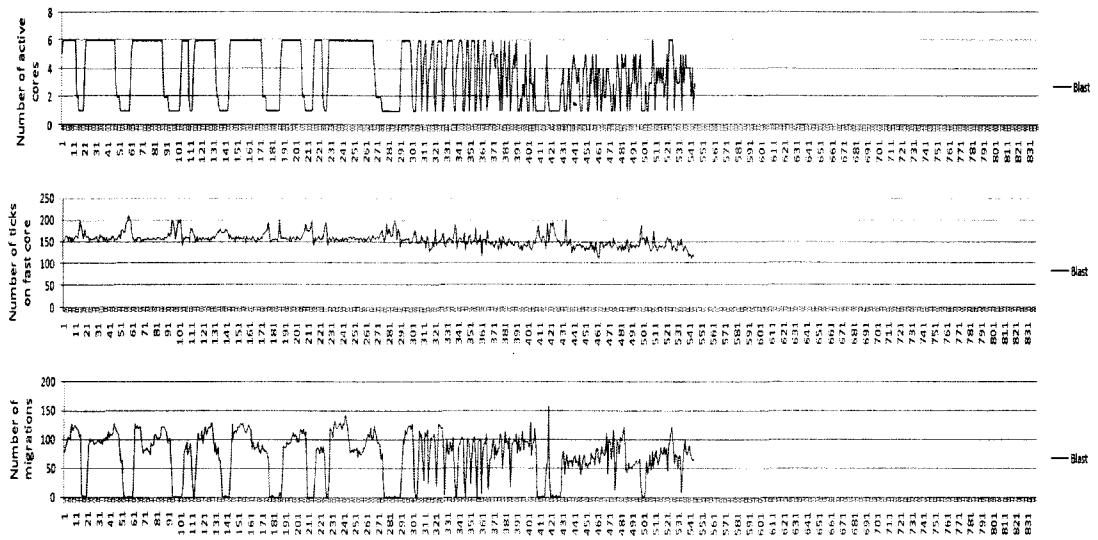
(c) Average completion time of BLAST and blackscholes under both schedulers

(d) Average completion time of BLAST and FFT-W under both schedulers

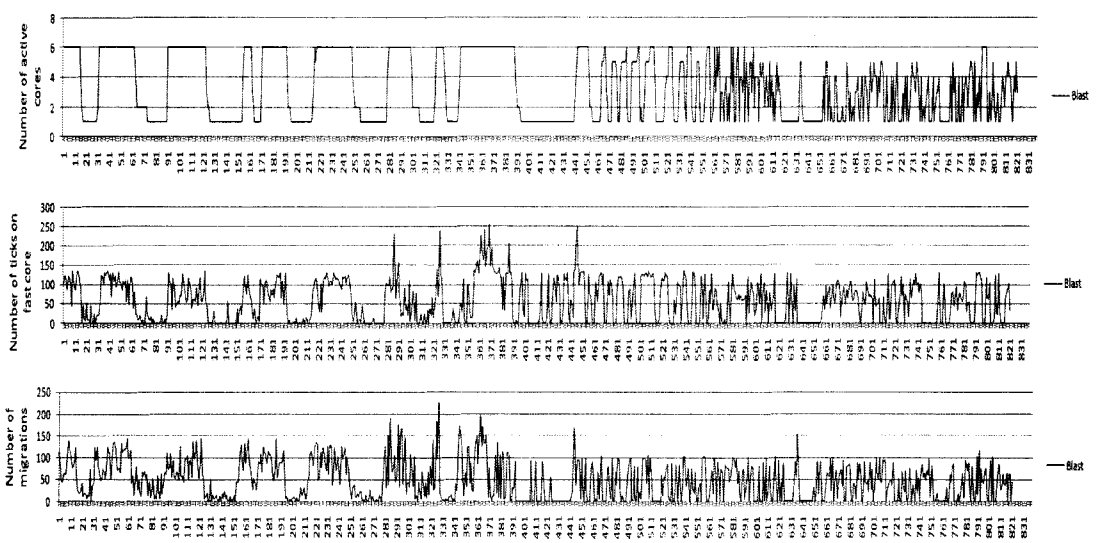
Figure 3.7: Evaluating acceleration of sequential phases

### 3.5.1 Combination of sequential and parallel workloads

In this scenario, the AASH scheduler would assign a higher priority to the virtual machines running single-threaded applications (recall that these virtual machines will have the active virtual CPU count equal to one) and schedule them on the fast cores most of the time. If the number of fast cores exceeds the number of single-threaded applications, the remainder of the fast cores would be fairly shared among the virtual machines running parallel workloads, otherwise the AASH scheduler would leave parallel workloads on slow cores. If any virtual machine running a parallel application enters a sequential phase (i.e., the number of active virtual CPUs reduces to one), that virtual CPU will share the time on the fast core with the virtual machines running single-threaded applications (whose active virtual CPU count never goes above one). As a result, the parallel applications will also experience some




(a) The AASH scheduler




(b) The default Xen scheduler

Figure 3.8: The effects of phase changes in running BLAST

performance improvement relative to the default scheduler.

To evaluate this scenario, we run two virtual machines. The first virtual machine has six virtual CPUs and runs a parallel application (we used *BLAST*). The other virtual machine has a single virtual CPU and runs a sequential application (we used *eon* from SPEC CPU2000). We ran the virtual machines with both the AASH scheduler and the default Xen scheduler on our experimental platform with one fast core running at 2 GHz and seven slow cores running at 1 GHz.

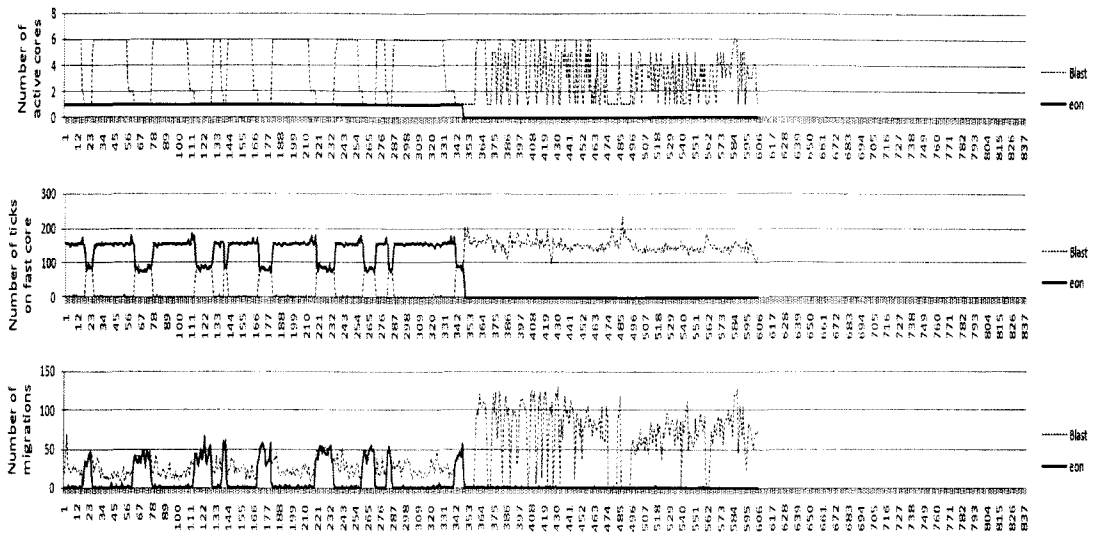
Figure 3.7(b) shows the results. The single-threaded application experienced a 36% speedup and the parallel workload enjoyed an 8% speedup relative to the default Xen scheduler. This shows that both virtual machines benefit from running under our scheduler. While *eon* benefits from running on the fast core most of the time under our scheduler, AASH also accelerates sequential phases of the *BLAST* by co-running it on the fast core.

Figure 3.9 shows the behavior of these two applications on both schedulers. Figure 3.9(a) shows the changes in the number of the active virtual CPUs, the number of clock ticks that each virtual machine spends on fast cores and the number of migrations. As can be seen in this Figure, the AASH scheduler mapped *eon* most of the times to the fast core. In sequential phases of *BLAST*, the AASH scheduler shared the fast core among both virtual machines. It also shows that migrations of *eon* were limited to the sequential phases of *BLAST*, during which it shares fast core with *BLAST*. Figure 3.9(b) on the other hand, shows the behavior of *BLAST* and *eon* under the default Xen scheduler. As expected, the default Xen scheduler shows arbitrary behavior.

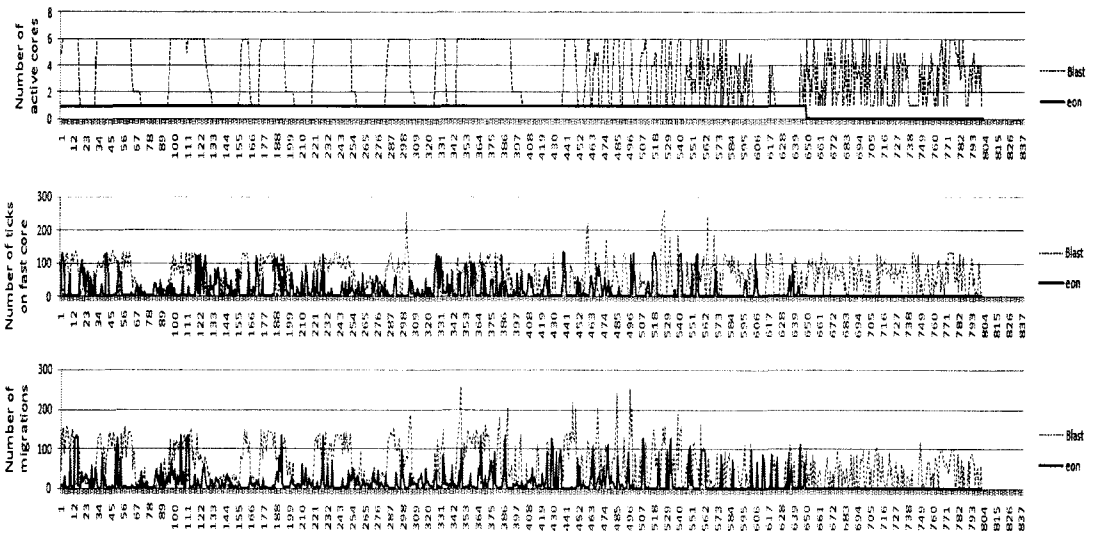
### 3.5.2 Combination of parallel workloads with and without sequential phases

In this scenario, we expect that the AASH scheduler would improve the overall performance due to the fact that the workloads with sequential phases would have those phases accelerated on fast cores, while the complete parallel applications would not experience a significant degradation by loosing their share of the fast cores.

We use two virtual machines, each representing one group of parallel applications. The first virtual machine has six virtual CPUs and runs a parallel workload with significant sequential phases (*BLAST*). The second virtual machine has two virtual CPUs and runs a parallel application with negligible sequential phases (*blackscholes* from the *PARSEC* benchmark suite). We ran virtual machines on our experimental platform (with one fast core running at 2 GHz and seven slow cores running at 1 GHz) with both the AASH and




(a) The AASH scheduler




(b) The default Xen scheduler

Figure 3.9: The effects of phase changes in running BLAST and eon

default Xen schedulers.

The average completion times of both virtual machines are shown in Figure 3.7(c). *BLAST* experienced a 17% speedup; the change in the completion time of *blackscholes* was negligible (less than 1%).

Figure 3.10(a) shows the behavior of these two benchmarks on the AASH scheduler. It can be seen that, whenever *BLAST* enters a sequential phase, the AASH scheduler migrates it to the fast core and stops scheduling *blackscholes* on the fast core, while the rest of the time the two virtual machines share the fast core based on the number of their virtual CPUs. Figure 3.10(b) shows the results of the same test on the default Xen scheduler. As expected, the default Xen scheduler shows arbitrary behavior in phase changes due to the fact that it is unaware of the asymmetry of the system.

### 3.5.3 Parallel workloads with sequential phases

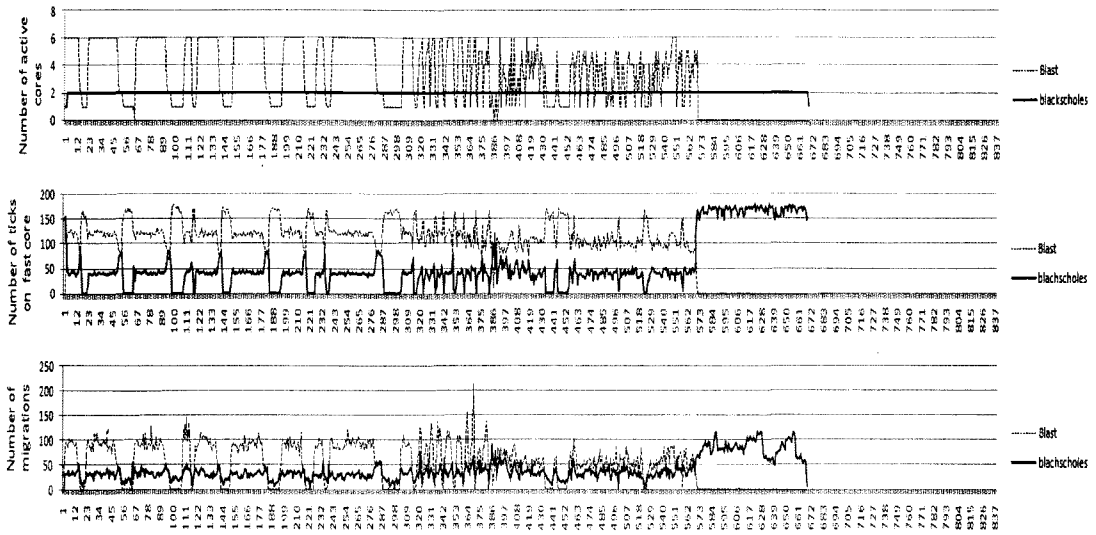
In this scenario, we expect that each workload would experience a lower speedup in comparison with the previous scenario. This is due to the fact that the sequential phases of different workloads may occur at the same time, and so the virtual machines must share the fast cores. As a result, each virtual-machine's sequential phase would be accelerated to a lesser extent than when no sharing of the fast cores occurs.

We ran *BLAST* and *FFT-W* benchmarks on the two virtual machines in this experiment. Recall that both are parallel applications with long sequential phases.

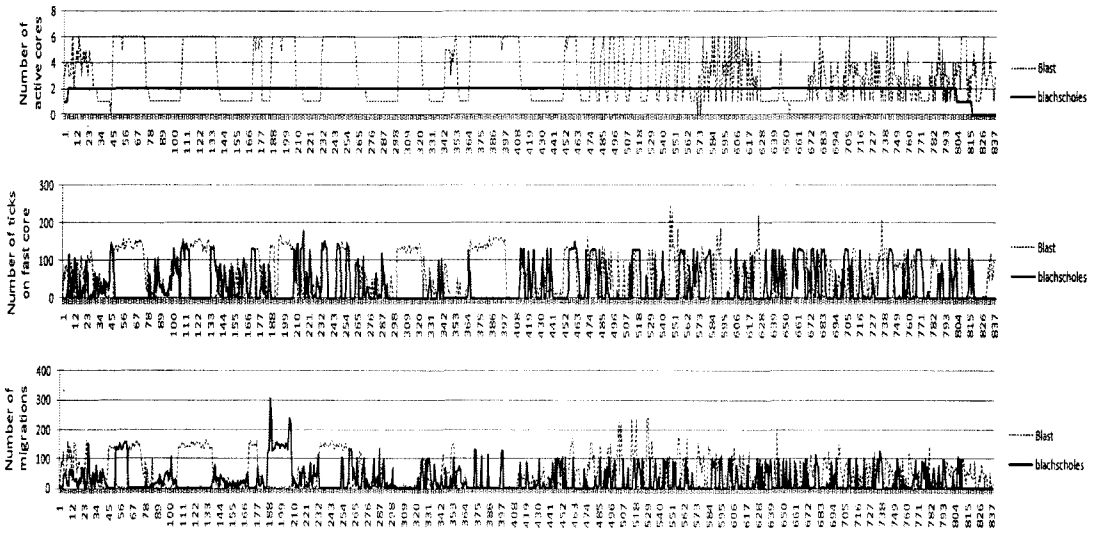
Figure 3.7(d) shows that the workload as a whole achieved a speedup of 16% with the AASH scheduler. *FFT-W* achieved a 27% speedup, and *BLAST* achieved a 6% speedup. *BLAST* achieved a smaller performance improvement than in the experiment of Figure 3.4, because during its sequential phase it had to share the fast core with *FFT-W*. Figure 3.11 shows the behaviour of these two benchmarks under the AASH and the default Xen scheduler. In Figure 3.10(a), it could be seen how *BLAST* and *FFT-W* share the fast core in their sequential phases.

## 3.6 Prioritization experiment

In this experiment we evaluate the mechanism that allows prioritizing the usage of fast cores. Recall that this mechanism provides two priority classes (*high* and *low*). We show that a

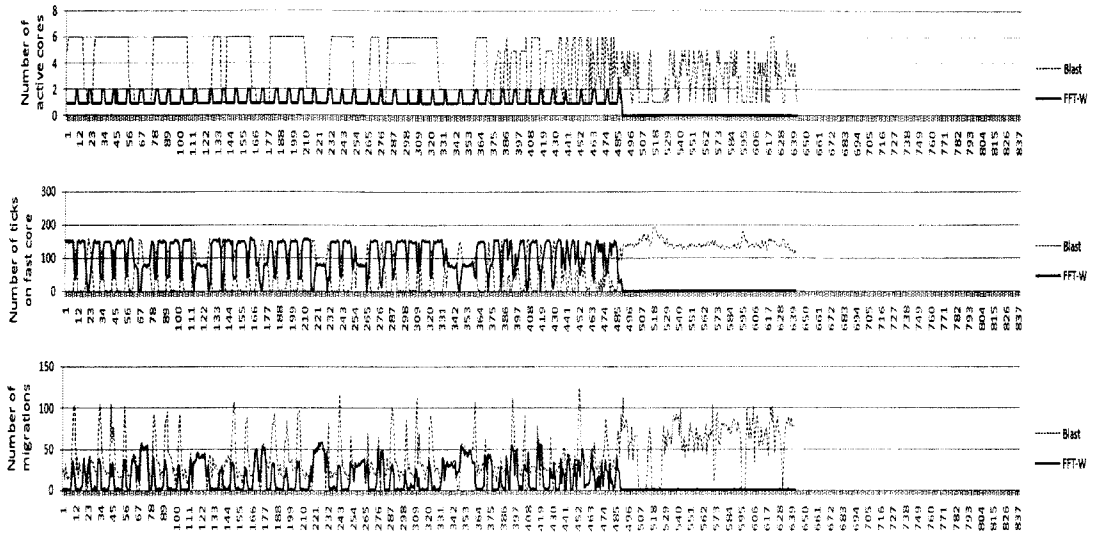



(a) The AASH scheduler

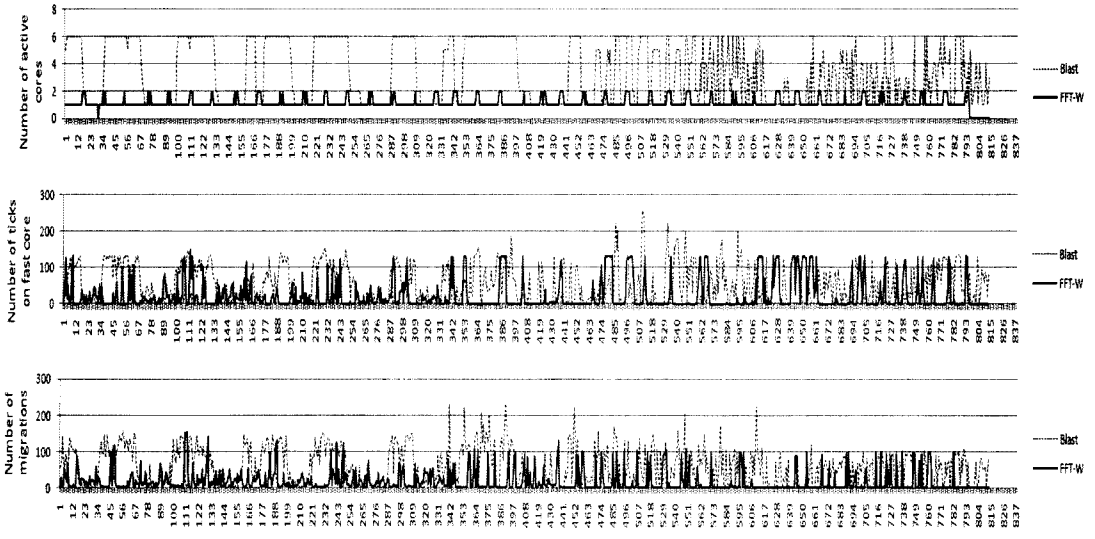



(b) The default Xen scheduler

Figure 3.10: The effects of phase changes in running BLAST and blackscholes



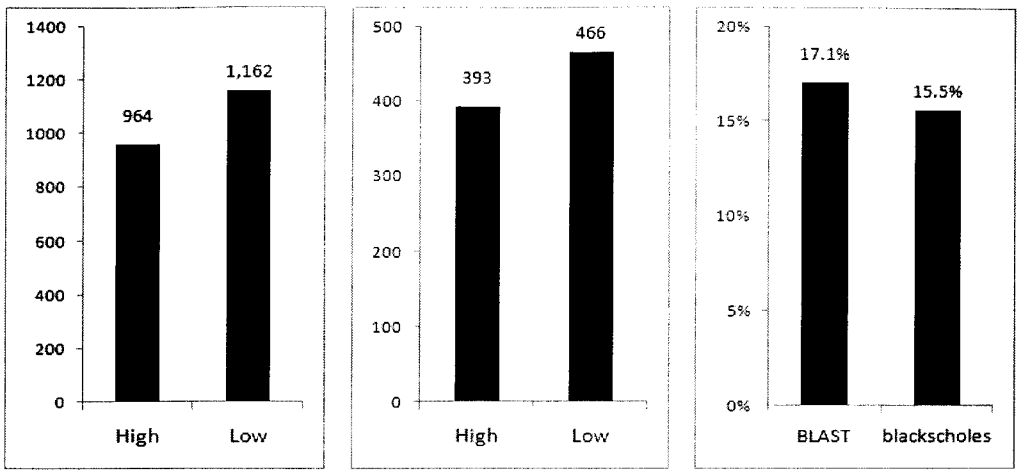

(a) The AASH scheduler




(b) The default Xen scheduler

Figure 3.11: The effects of phase changes in running BLAST and FFT-W

virtual machine in the high-priority class gets preference when the fast-core CPU time is allocated. As a result, it achieves better performance than a low-priority virtual machine.




(a) Average completion times when both VMs running BLAST

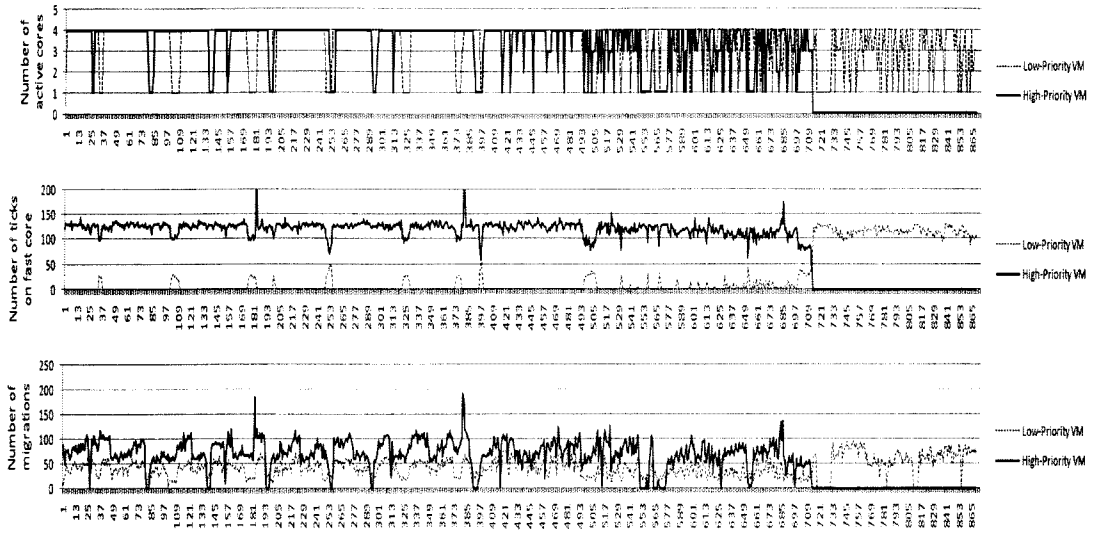
(b) Average Completion times when both VMs running blackscholes

(c) Speedup of high-priority VMs normalized to low-priority VMs

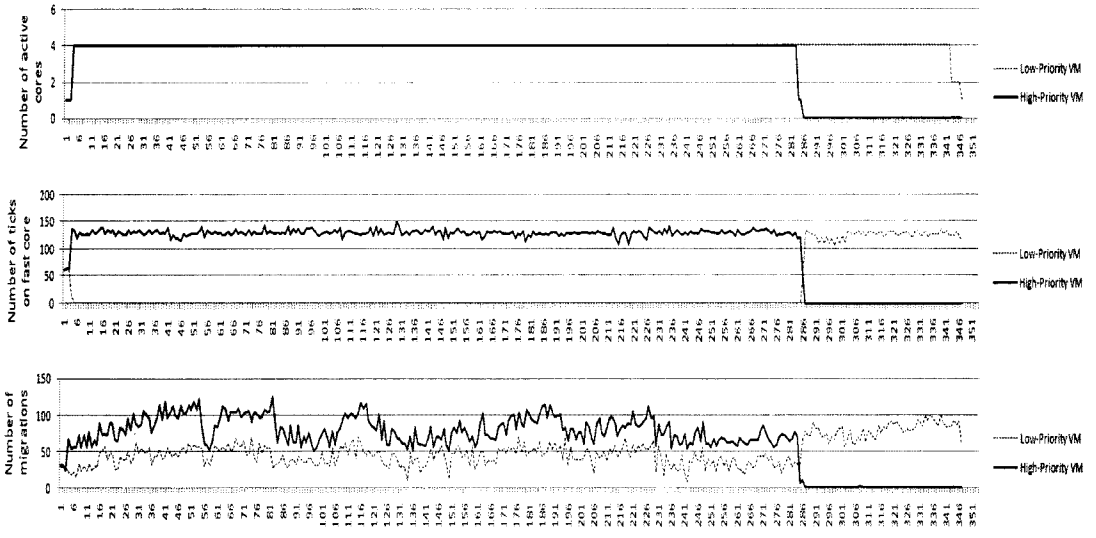
Figure 3.12: Prioritization experiment

To evaluate this experiment, we use two identical virtual machines, each representing one type of priority class (*high* and *low*). Each virtual machine has four virtual CPUs. In order to compare the results of the prioritization, both virtual machines run the same workload (we used *BLAST* and *blackscholes* in this experiment). We run both virtual machines under the AASH scheduler on our experimental platform (with one fast core running at 2 GHz and seven slow cores running at 1 GHz).

Figures 3.12 and 3.13 show the results of this experiment. As can be seen in Figure 3.12(c) the virtual machines with high-priority label get most of the fast-cores' CPU time and improve their performance. Figure 3.13 shows the distribution of the fast core in the system within the virtual machines' life cycle. In both scenarios (*BLAST* and *blackscholes*), fast cores get allocated almost always to the high-priority guest. It can be seen that the low-priority guest gets a share of the fast core's CPU time when it enters a sequential phase (Figure 3.13(a)) or when the high priority guest finishes its work.




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




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

Figure 3.13: Fast core allocation for High and Low priority VMs

# Chapter 4

# Related Work

Most existing platform virtualization software is not designed to take advantage of asymmetric many-core architectures. In study by Nikolopoulos et al. [19], an enhanced version of the Xen hypervisor was proposed and used to leverage application feedback throughout the scheduling process. However, this framework is not asymmetry-aware and thus it can not deliver the benefits of AMP systems to guest operating systems. There has been no other prior work in designing hypervisor support for AMPs. To our knowledge, neither the popular Xen hypervisor [5] [9], nor VMware's recent hypervisor-based ESX server consider the asymmetric nature of the system in scheduling decisions. Our work is the first that provides required scheduling support for delivering the benefits of AMP systems in the hypervisor.

Some studies have been done to show the better performance of AMP systems [13] [17]. However, these works have been done in the context of operating systems. Kumar et al. in [14] presented a core assignment algorithm for maximizing performance on AMP systems. They improved performance by using the diversity of instruction level parallelism (ILP) in a workload on AMP systems. Kumar's algorithm assigned applications with high ILP to fast cores based on the fact that these applications could effectively use the fast cores' resources to extracting ILP. Their algorithm used normalized IPC as a heuristic for core assignment. The study by Becchi and Crowley [6] is another well-known scheduling algorithm in this category. This algorithm also use dynamic performance monitoring to determine the optimal thread-to-core assignment. Shelepov et al. [20] proposed a similar algorithm. However, instead of relying on dynamic monitoring, it exploits compact summaries of applications' runtime properties. All of these algorithms, however, require our asymmetry-aware hypervisor to work correctly in the hypervisor environment. Our scheduler provides the required support

for these schedulers to use their knowledge of asymmetry and improve the performance of their corresponding guests in the hypervisor environment. However, we do not implement the same policies as in these operating system scheduling algorithms, because hypervisors do not have such visibility into applications.

The study by Hill and Marty [12] presented a theoretical model that clearly demonstrated the benefit of AMP systems for mitigating the effects of Amdahl's law. They showed better performance of AMP systems for applications with small sequential regions and concluded that the AMPs never performed worse than SMPs. Annavaram's study [2] demonstrated how the effects of the Amdahl's law can be mitigated on AMP systems experimentally. They showed this by modifying the application code and scheduling threads manually. This manual approach, however, is not scalable to multiple applications and could not fairly share scarce resources among all workloads. It is better to address this issue globally and provide required scheduling support in operating systems and hypervisors in order to deliver the benefit of AMP systems to the applications.

Although the aforementioned studies have clearly demonstrated, both theoretically and experimentally, the benefits of AMP systems for mitigating the effects of Amdahl's law, this issue has not been addressed in either operating systems or hypervisors. Most of the prior works on AMP scheduling algorithms use AMP systems to leverage diversity of the instruction level parallelism in a workload. The studies by Li [15] and Balakrishnan [4] tried to address the goal of keeping the fast cores as busy as possible. This would automatically accelerate the sequential phases in a single application. However they did not consider scenarios with multiple workloads. Our study is the first that uses AMP systems to mitigate the effects of Amdahl's law in hypervisors and it provides required scheduling support for delivering the benefits of AMP systems to applications.

# Chapter 5

# Summary

In this study, we present the AASH scheduler (an asymmetry-aware scheduler) for AMP systems. We also evaluate the implementation of our proposed algorithm on the Xen hypervisor. Our main objective is utilizing the potential of asymmetric many-core systems to improve the performance of the wide range of the applications.

Four main targets are addressed in this study: (1) equal distribution of scarce resources in AMP environment, (2) providing required mechanisms to support asymmetry-aware guest operating systems, (3) accelerating sequential phases on fast cores and mitigating the effects of Amdahl's law and (4) providing coarse-grained prioritization for service differentiation. The results of our experiments show that we achieved our goals and validated our method. Asymmetry-aware guest operating systems performed better under the AASH scheduler (as much as 16% speedup). Parallel applications with large sequential phases show as much as 31% speedup on the AASH scheduler in comparison with the default Xen scheduler. Our results show the fair behavior of the AASH scheduler. We also demonstrate the behavior of guests in different priority classes. Some applications with high sensitivity to their cache states or with small sequential phases experienced negligible performance degradation (up to 3%) due to the inevitable migration in the AASH scheduler. However, by increasing the length of the accounting periods, this overhead could be completely eliminated.

We do not consider user-level policies in using fast and slow cores in this study. This functionality, however, is available through existing features in our scheduler and the Xen hypervisor.

# Bibliography


[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Molecular Biology*, 215(3):403–410, 1990.

[2] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 298–309, Washington, DC, USA, 2005. IEEE Computer Society.

[3] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelik. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 506–517, Washington, DC, USA, 2005. IEEE Computer Society.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

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

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[8] S. Borkar. Thousand core chips: a technology perspective. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 746–749, New York, NY, USA, 2007. ACM.

[9] D. Chisnall. *The definitive guide to the xen hypervisor.* Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[10] A. Fedorova, V. Kumar, V. Kazempour, S. Ray, and P. Alagheband. Cypress: A scheduling infrastructure for a many-core hypervisor. In *Proceedings of the Workshop on Managed Multi-Core Systems (MMCS'08) held in conjunction with the 17th International Symposium on High Performance Distributed Computing (HPDC-17)*, 2008.

[11] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[12] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[13] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Washington, DC, USA, 2003. IEEE Computer Society.

[14] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. *SIGARCH Comput. Archit. News*, 32(2):64–75, 2004.

[15] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.

[16] D. A. Menascé. Virtualization: Concepts, applications, and performance modeling. In *Int. CMG Conference*, pages 407–414, 2005.

[17] 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.

[18] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. *IEEE Workload Characterization Symposium*, 0:182–188, 2006.

[19] 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.

[20] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. Hass: a scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.*, 43(2):66–75, 2009.

[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.

[22] Y. Xie and G. H. Loh. Dynamic classification of program memory behaviors in cmps. In *In the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects. (CMP-MSI, held in conjunction with ISCA-35)*, 2008.