Virtualizing Performance Asymmetric Multi-core Systems

Youngjin Kwon, Changdae Kim, Seungryoul Maeng, and Jaehyuk Huh
Computer Science Department, KAIST
{yjkwon and cdkim}@calab.kaist.ac.kr, {maeng and jhhuh}@kaist.ac.kr

ABSTRACT

Performance-asymmetric multi-cores consist of heterogeneous cores, which support the same ISA, but have different computing capabilities. To maximize the throughput of asymmetric multi-core systems, operating systems are responsible for scheduling threads to different types of cores. However, system virtualization poses a challenge for such asymmetric multi-cores, since virtualization hides the physical heterogeneity from guest operating systems. In this paper, we explore the design space of hypervisor schedulers for asymmetric multi-cores, which do not require asymmetry-awareness from guest operating systems. The proposed scheduler characterizes the efficiency of each virtual core, and map the virtual core to the most area-efficient physical core. In addition to the overall system throughput, we consider two important aspects of virtualizing asymmetric multi-cores: performance fairness among virtual machines and performance scalability for changing availability of fast and slow cores.

We have implemented an asymmetry-aware scheduler in the open-source Xen hypervisor. Using applications with various characteristics, we evaluate how effectively the proposed scheduler can improve system throughput without asymmetry-aware operating systems. The modified scheduler improves the performance of the Xen credit scheduler by as much as 40% on a 12-core system with four fast and eight slow cores. The results show that even the VMs scheduled to slow cores have relatively low performance degradations, and the scheduler provides scalable performance with increasing fast core counts.

Categories and Subject Descriptors

C.1.2 [**Processor Architecture**]: Multiple Data Stream Architectures (Multiprocessors); D.4.1 [**Operating Systems**]: Process Management

General Terms

Design, Performance

ISCA'11, June 4–8, 2011, San Jose, California, USA.

Keywords

virtualization, asymmetric multi-core, scheduling, fairness

1. INTRODUCTION

Performance-asymmetric multicores consist of heterogeneous cores, which support a single ISA, but have different computing capabilities. As power consumption has become a critical constraint for multi-core designs, such performance-asymmetric multicores have emerged as a power-efficient alternative of conventional homogeneous multi-core architectures [8, 9, 13]. Each core type has different characteristics in its performance, area, and power consumption. Large fast cores deliver higher performance but consume more power than small slow cores. Depending on how efficiently an application exploits the different computing capabilities of core types, the best core type, which can maximize performance per area or watt, must be used for the application. To achieve the improved area/power efficiency of asymmetric multi-cores, operating systems must find the characteristics of threads and schedule them properly to the different types of cores [16, 15, 7].

However, many current servers do not run traditional operating systems directly upon physical systems. Instead, the physical systems are virtualized, and an operating system and applications are contained in each virtual machine. Virtualization provides an illusion of multiple virtual machines in a physical machine, allowing the consolidation of otherwise under-utilized systems. A software layer called virtual machine monitor (VMM), or hypervisor, multiplexes virtual CPUs (vCPUs) to physical cores. Such virtualization has become widely applied to servers to reduce their operating costs. Furthermore, cloud computing is getting popular by enabling elastic computing resources, which eliminate the need for over-provisioning to meet the peak demand. Virtualization has been used as the basic platform of cloud computing for its flexibility to deploy and destroy virtual systems rapidly to meet changing demands. Furthermore, its support for isolated virtual systems allows multiple tenants to share a single physical system.

Virtualization poses a challenge to use heterogeneous cores efficiently. It hides the physical asymmetry of cores, and guest operating systems may schedule threads as if they are using homogeneous cores. A possible solution for the problem is to export the physical asymmetry to the guest OSes. Kazempour et al proposed an asymmetry-aware scheduler for hypervisors, which requires guest OSes to be responsible for mapping threads to different types of cores efficiently [6]. With the hypervisor scheduler, a guest VM requests a com-

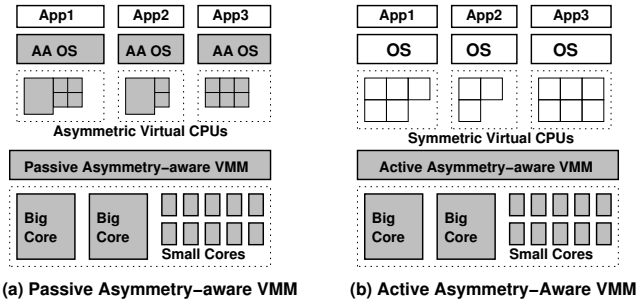

Figure 1: Virtualization of performance asymmetric multi-core systems

bination of fast and slow virtual CPUs to the hypervisor. Such *passive asymmetry-aware hypervisors* support only the fair allocation of fast cores to guest VMs.

However, delegating the responsibility of supporting asymmetric cores to the guest OSes cannot fully utilize the potential throughput of asymmetric multi-cores. Virtual machines may have various characteristics in their use of fast or slow cores, but each guest operating system must make scheduling decisions without global knowledge about other guest VMs. Only the hypervisor can identify the relative differences in core efficiency among virtual machines, and properly schedule them on different types of cores. Furthermore, such a passive asymmetry-aware hypervisor requires modified operating systems for the hypervisor, which are not always available.

To truly virtualize physical resources, the hypervisor must hide the physical asymmetry of underlying cores, and create an illusion of symmetric cores for virtual machines. Guest OSes must be able to use virtual resources without knowing the physical constraints. *Active asymmetry-aware hypervisors* should find the best mapping between virtual and physical cores to maximize the throughput of a given asymmetric multi-core system. Figure 1 describes the difference between passive and active asymmetry-aware hypervisors. The shaded parts are asymmetry-aware components of the systems. In passive asymmetry-aware hypervisors, a virtual machine must use an asymmetry-aware guest OS, and virtual asymmetric CPUs are created for the VM. In active asymmetry-aware hypervisors, a VM creates only virtual symmetric CPUs.

Maximizing system throughput with a given combination of asymmetric cores, is not the only role of active asymmetry-aware hypervisors. Beside the overall throughput, the hypervisors must consider two important aspects of virtualized asymmetric multi-core systems: *fairness* and *scalability*. As multiple virtual machines may share a physical system, the performance interference among VMs must be minimized. Such performance isolation becomes more important in virtualized clouds, since cloud providers must guarantee the Service Level Agreement (SLA). However, there is an inherent conflict between performance and fairness in asymmetric multi-cores. To maximize the performance, only VMs with high fast core efficiencies must monopolize fast cores, hurting fairness among VMs. In this paper, we show that system managers must be able to find a balance between fairness and performance, and adjust the hypervisor policy to enable multiple levels of fairness guarantee. The second aspect to consider for asymmetric multi-cores is performance scalability with the available fast cores. In virtualized server farms, there may exist heterogeneous systems with various combinations of fast and slow cores. Furthermore, depending on the co-tenants sharing a system, the available fast core resource may vary. Virtual machines must be able to scale their performance with the available fast cores to maximize the system throughput. The role of hypervisor is to help the performance of VMs scale with the available fast cores, even if the guest OSes are not aware of the asymmetry of cores.

In this paper, we explore the design space of active asymmetry-aware hypervisor schedulers. The proposed scheduler characterizes the behavior of each virtual core without any input from guest OSes or applications. It maps virtual cores to physical cores to maximize the throughput of asymmetric cores, with only the information collected online at the hypervisor. Furthermore, the schedulers can support multiple levels of fairness guarantee and performance scalability with various ratios of fast to slow cores. To the best of our knowledge, this is one of the first work to design a hypervisor scheduler, which maps virtual machines to asymmetric cores without any support from guest OSes.

We have implemented an asymmetric-aware scheduler in the open-source Xen hypervisor, modifying the default credit scheduler. Using various mixes of single-threaded and parallel applications in addition to I/O-intensive applications, we have evaluated how effectively the proposed scheduler can achieve throughput improvement without asymmetry-aware guest OSes. We use the support for dynamic voltage and frequency scaling (DVFS) to configure a 12-core processor to an asymmetric multi-core. In this paper, we use only two types of cores, fast and slow cores, although this work can be applied to multi-level asymmetric cores. Using combinations of various workloads, the proposed scheduler can improve the throughput by as much as 40% compared to the default Xen scheduler, and by as much as 28% compared to a fair scheduler. Also, the proposed scheduler provides scalable performance with increasing fast core resources.

The rest of the paper is organized as follows. Section 2 presents factors to consider for designing hypervisor schedulers that support asymmetric multi-cores. Section 3 describes the details of modification we apply to the Xen hypervisor. Section 4 presents experimental results. Section 5 discusses the prior work on asymmetric multicores and asymmetry-aware scheduling by operating systems. Section 6 concludes the paper.

2. VIRTUALIZING ASYMMETRIC MULTI-CORES

To achieve the benefits of virtualization with asymmetric cores, an active asymmetry-aware hypervisor must satisfy the following requirements. i) Firstly, it must maximize the system throughput without requiring asymmetry-awareness from guest OSes. The primary goal of asymmetric multi-cores is to improve the area/power efficiency of multi-cores by using the best type of cores for each thread or virtual CPU. The hypervisor scheduler must find the best mapping between virtual CPUs and physical cores to maximize the throughput. ii) Secondly, the hypervisor must consider the fairness requirement among co-tenants, which may vary by types of cloud systems. A private cloud with loose fairness requirement, may want to improve only the overall system

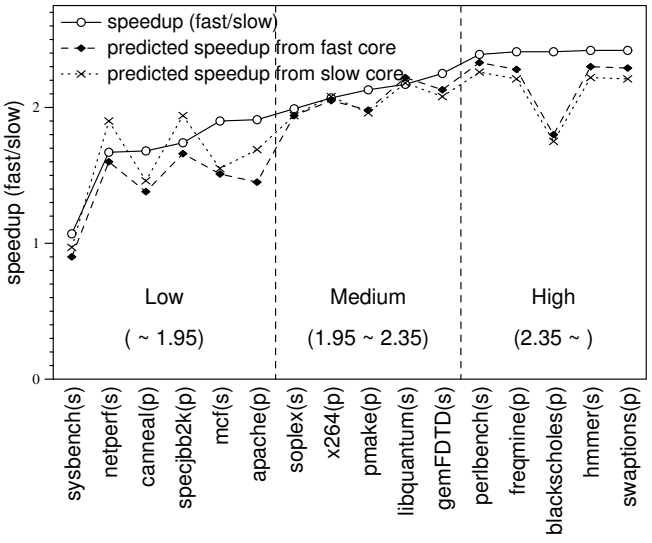

Figure 2: Measured and predicted speedups of virtual machines

throughput, even with compromised fairness. Public clouds, which must satisfy the SLA, may impose strict fairness restriction at the cost of performance. The hypervisor must support such multiple levels of fairness requirements. iii) Finally, VMs can be placed in systems with different fast and slow core configurations in large scale cloud systems. Also, the available fast core resources can be affected by other VMs sharing the system. As the availability of fast cores is not fixed, the performance of a VM must be scalable to the available fast core resources. Otherwise, even if a VM uses more fast cores, the performance may not improve proportionally.

In this section, we discuss how to maximize the system throughput by characterizing VMs, and how to balance the trade-offs between performance and fairness in asymmetric multi-cores. We also discuss the role of a hypervisor to support scalable performance with available fast cores.

2.1 Maximizing System Throughput

An active asymmetry-aware hypervisor must characterize the behaviors of virtual machines and find ones with high speedups of using fast cores. *Fast core efficiency* is defined as the speedup of using fast cores over the same number of slow cores. Virtual machines with higher fast core efficiencies must use fast cores to maximize the overall throughput. To characterize how efficiently a VM uses fast core resources, we consider two factors: *IPC* and *utilization*.

The first factor to consider for active asymmetry-aware scheduling is how effectively a virtual CPU uses each core type, when virtual machines can use the core resource continuously. Several prior studies for native operating systems use various metrics to capture the efficiency. The most obvious but accurate way is to run an application both on a fast core and then a slow core [9]. However, in this paper, to estimate the efficiency with little overhead and to respond to changing phases quickly, we directly use instruction per cycle (IPC) as an approximate representation of fast core efficiency. As the IPC of an application is higher, the application tends to exhibit a higher speedup of using fast cores than slow cores.

The second factor to consider for the scheduler is how important computing capability is for the overall performance of applications. For compute-intensive workloads, which can always run on cores without any I/O activities, the performance benefit from fast cores will be larger than that with I/O-intensive workloads which spend most of the time on waiting the completion of I/O requests. We define utilization as the portion of CPU execution cycles over the total execution cycles. The hypervisor measures utilization by monitoring clock cycles used by a virtual CPU for each time interval.

We combine two factors, IPC and utilization, to represent the fast core efficiency of a virtual machine. In our benchmark applications, the multiplication of the two factors shows the best prediction accuracy, and thus in the rest of the paper, we use the metric to predict fast core efficiency:

$$fast\ core\ efficiency\ score = IPC\ \times\ utilization$$

Figure 2 shows the correlation between the actual measured speedups and the predicted speedups estimated from fast cores and slow cores. The predicted speedups are derived from the estimated fast core efficiencies with scaling. The maximum speedup is 2.375, which is the clock frequency ratio between the fast and slow cores used for the experiments. The detailed configurations are shown in Section 4.1. The fast core efficiency should be predicted when a VM is running either on fast cores or on slow cores, and thus both estimations should provide consistent results. For scheduling decisions, only a relative order in the fast core efficiencies of virtual machines matters.

The fast core efficiency prediction based on IPCs may not always be a perfect metric. For example, if long latency instructions with cross-dependency dominate the execution of an application, the prediction based on IPC exhibits a low fast core efficiency, but such application will have a high measured speedup. However, considering the limitation of the available performance monitoring counters, in general, our results show a relatively high correlation between the predicted and actual speedups. Exploring the architectural support for better fast core efficiency metrics will be our future work.

The high correlation shows that active asymmetry-aware hypervisors can effectively find the most area-efficient core type for a VM by estimating IPCs and utilizations, which the hypervisor can observe without any guest OS support.

2.2 Supporting Multiple Levels of Fairness

In asymmetric multi-cores, there is a fundamental trade-off between allocating resources fairly and maximizing system throughput. To allocate resources fairly, fast cores must be equally shared by all threads (or virtual CPUs in virtualized systems). On the other hand, to maximize the system throughput of asymmetric multi-cores, threads with high fast core efficiency scores, must monopolize the fast cores. This trade-off should be considered carefully, as requirements for fairness may vary over different cloud systems.

Traditionally, fair CPU scheduling assigns an equal amount of CPU time for each thread. However, this classic definition cannot be applied to asymmetric multi-core systems as each core type can have a different computing capability. A possible way to make the scheduler fair is to give different weights on the CPU time depending on the types of cores [11]. For instance, if a fast core is two times faster than a slow core

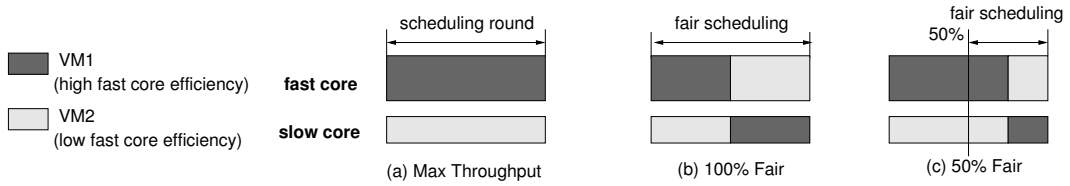

Figure 3: Throughput-maximizing scheduling, 100% fair scheduling, and 50% fair scheduling

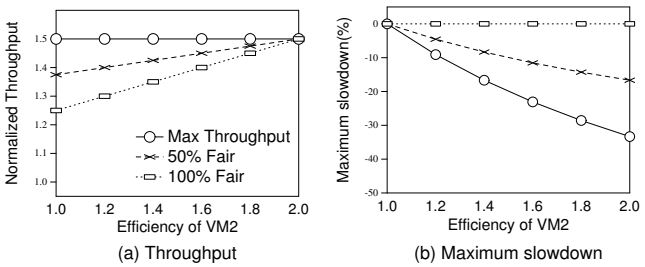

Figure 4: The effect of fairness policies: throughput and maximum slowdown

on average, a thread running on the slow core must receive a twice longer scheduling slot than a thread running on the fast core. However, the speedups of using the fast core over the slow core vary by the characteristics of workloads, which makes finding a fixed weight ratio difficult. Instead, we define an alternative fair scheduling method for asymmetric multi-cores. The fair scheduler, called `amp-fair` guarantees that each virtual CPU has the same share of fast core and slow core cycles. With `amp-fair`, each active virtual CPU (vCPU) always receives an even share of fast and slow core cycles, regardless of the fast core efficiency.

However, such a strict fair scheduling policy, which ignores fast core efficiencies, cannot provide any performance benefit of asymmetric cores, and the strict fairness may not be required for practical cloud systems. To accommodate different fairness requirements, we define a partially fair scheduler, which can distribute a certain portion of fast core cycles evenly, and assign the rest of fast core cycles by fast core efficiencies. `amp-R%-fair` scheduler evenly distributes R% of fast core cycles to all virtual CPUs, and assigns the rest of fast core cycles to vCPUs with high fast core efficiency scores.

Figure 3 illustrates the three scheduling policies by different levels of fairness support. The throughput-maximizing scheduler ((a) max throughput) considers only the maximization of overall throughput, and assigns the entire fast core shots to VM1 with the higher efficiency. The completely fair scheduler ((b) 100% fair) evenly distributes the entire fast core cycles to the two VMs. A 50% fair scheduler ((c) 50% fair) evenly distributes only 50% of fast core cycles to the two VMs, but the rest 50% is assigned to VM1 with the higher efficiency.

Using the `amp-fair` scheduler as the perfect fair scheduler, we evaluate the fairness of asymmetry-aware hypervisors by the performance degradation of each VM compared to the same VM running with the `amp-fair` scheduler. For the same set of VMs, the performance of each VM with a scheduler is compared to that of the same VM with the `amp-fair` scheduler. Our metric for fairness is *the maximum slowdown* among VMs against the `amp-fair` scheduler, which must be minimized for fairness.

With a simple analytic model, Figure 4 illustrates the trade-offs between the overall throughput and fairness among VMs. For the figure, a processor with two cores has a fast core and a slow core, and the maximum speedup of using the fast core over the slow core is 2. There are two VMs, and the first VM has the fast core efficiency of 2.0, which is the maximum fast core efficiency in the processor. X-axis represents the fast core efficiency of the second VM, ranging from 1.0 to 2.0. We assume that the fast core efficiencies do not change during the execution, and the system uses perfect schedulers, which can predict the efficiencies perfectly and schedule VMs accurately without any overhead. The figure shows two graphs, the left one for the combined throughput of both VMs normalized to that with two slow cores, and the right one for the maximum slowdown compared to the 100% fair scheduler.

The max throughput policy will always schedule the first VM to the fast core, and the second VM to the slow core, as the first VM always has the higher efficiency than the second VM. The max throughput policy provides the maximum throughput from the two cores. With the policy, the combined throughput is always 1.5 even with various fast core efficiencies of the second VM, as the first VM is always running on the fast core with a speedup of two, and the second VM is always running on the slow core with a speedup of one. With the 100% fair policy, if the efficiency of the second VM is close to 2.0, the throughput becomes close to the maximum throughput, as the two high efficiency VMs are sharing the fast core almost equally. However, if the efficiency of the second VM is close to 1.0, the 100% fair policy has a 16.7% throughput reduction compared to the maximum throughput.

For the maximum slowdown, as the second VM is always running on the slow core more often than the first VM, the performance drop of the second VM against the 100% fair policy becomes the maximum slowdown. If the efficiency of the second VM is close to 2.0, the maximum slowdown with the max throughput policy is almost 33%. The second VM even with a very high efficiency is always scheduled to the slow core, and thus has a high performance degradation from the 100% fair policy. However, if the efficiency is close to 1.0 for the second VM, even the unfair max throughput policy provides the maximum slowdown close to zero. With the efficiency of 1 for the second VM, even if the fair scheduler assigns part of the fast core shots to the second VM, the VM cannot improve the performance.

If the fast core efficiencies of two VMs are very high, close to the maximum, the 100% fair policy has the same overall throughput as the max throughput policy, and supports the 100% fairness between two VMs. On the other hand, if a high efficiency VM is running with a low efficiency VM, the

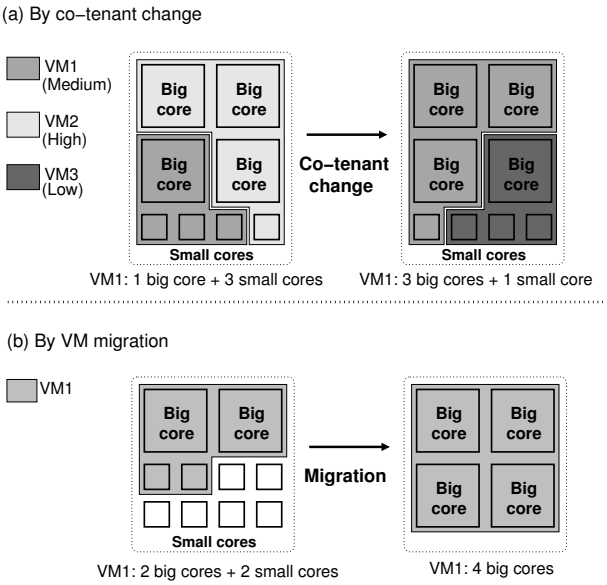

Figure 5: Changes of available fast cores

max throughput policy can improve the overall throughput, while minimizing the maximum slowdown close to the 100% fair policy. The fairness among VMs can be maintained even with the max throughput policy, when the VMs have widely different fast core efficiencies. This observation may lead to an adaptive scheduler, which adjusts the percentage of fair scheduling based on the efficiency difference among VMs. The adaptive scheduler may make a compromised decision between fairness and throughput dynamically. Exploring the design space of such schedulers will be our future work.

2.3 Performance Scalability

In virtualized cloud systems, a virtual machine may receive a different amount of fast core resources for two reasons. Figure 5 illustrates the two cases. Firstly, a VM shares the asymmetric cores with other co-tenants with different fast core efficiencies. Depending on the relative efficiency order among co-tenants, the VM can receive a different share of fast core cycles. In the top part of the figure, VM1 uses only one big core and three small cores, but after co-tenants change, it receives three big cores and one small core. Secondly, a VM may migrate to other physical systems, which have different processor configurations. The processors in cloud systems may have different numbers of fast and slow cores or different core architectures. In the bottom part of the figure, VM1 uses two big cores and two small cores in a system, but after it migrates to another system, it uses four big cores. In both cases, the available fast core resources may change dynamically during the lifetime of a virtual machine. Active asymmetry-aware hypervisors must support scalable performance with fast cores, as the availability of fast cores may change. Such performance scalability is also important to achieve power-scalability of asymmetric cores. As more power is consumed by large cores, the performance must also improve proportionally.

Performance scalability represents whether adding more fast cores can improve the overall performance in asymmetric multi-cores. However, as discussed in Balakrishnan et al [1], the performance scalability of applications and operating systems may vary by the application behaviors and operating system designs. Figure 6 presents the performance scalability of our benchmark applications with varying numbers of fast cores on a 12-core system. Y-axis shows the speedups over the configuration with all 12 slow cores, with increasing numbers of fast cores. For example, the `f8/s4` configuration has eight fast cores and four slow cores. For each run, a single VM is created for the application in the system. Not to include the effect of hypervisor scheduling, the hypervisor fixes 12 virtual CPUs to respective physical cores. The guest operating system, which is unaware of asymmetry, schedules threads on the fixed virtual CPUs.

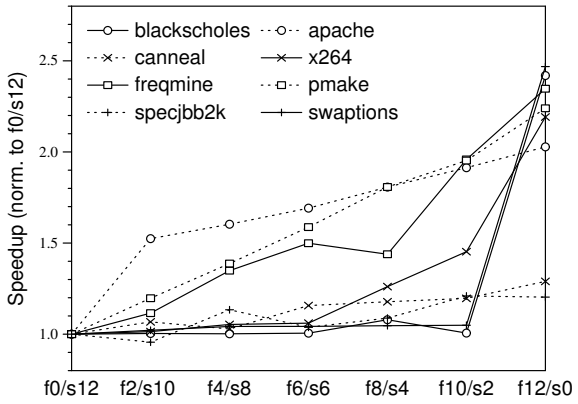

Figure 6: Scalability with fast cores

As shown in the figure, the performance of certain applications does not scale with the number of fast cores. Applications in dotted lines are either the ones which scale well with higher fast core ratios or have low speedups even with 12 fast cores (f12/s0). Applications in solid lines have high speedups with 12 fast cores, but can achieve the high speedups only when all the cores are fast cores. For those applications with solid lines, the guest OS is not using the fast cores effectively. The guest OS may allow fast cores to become idle when slow cores are busy. It may allow some threads to use only fast cores or slow cores, causing unbalanced loads. For applications with the tight coupling of thread executions, such as scientific workloads with static thread allocation, asymmetric multi-cores may not provide performance scalability [10].

Active asymmetry-aware hypervisors must provide the scalability of VMs even if the guest operating systems are not aware of asymmetry. In this paper, we will show what cause the scalability problem shown in Figure 6, and how the hypervisors can improve the scalability by proper fast core scheduling policies.

3. IMPLEMENTATION

In this section, we describe the implementation details of our scheduler for asymmetric multi-cores. We modified the open source Xen hypervisor (version 3.4.1) for our scheduler implementation [2]. We first present a brief description of the default credit scheduler in the Xen hypervisor. The credit scheduler supports fair scheduling among virtual CPUs for traditional homogeneous multi-cores. In the rest of this section, we present the mechanism to estimate fast core efficiency, and the scheduling schemes based on the estimation.

3.1 Xen Credit Scheduler

In the Xen hypervisor, the default scheduler is a credit scheduler [4]. The scheduler distributes credits to each virtual CPU (vCPU) at every accounting period (30ms). At every tick period (10ms), if a vCPU uses a physical core, the scheduler reduces the credits of the vCPU. A credit is a guaranteed time slot for which a vCPU can run until the next accounting period. If a vCPU has remaining credits, its priority is `under`. Otherwise the priority of the vCPU is `over`. To guarantee fairness, the credit scheduler tries to run vCPUs of priority `under` first, which have remaining time slots for the current accounting period. Not to waste unused CPU cycles (work-conserving property), a vCPU of priority `over` can be scheduled if there exists no other vCPUs of priority `under`. However, if an `under` priority vCPU wakes up from sleep state, it preempts the currently running vCPU with a priority boost up (`boost` priority).

For each physical core, there is a run queue which is sorted by priority (boost, under, over). If a run queue of a core runs out of vCPUs of priority `under`, it will try to steal under priority vCPUs from other run queues. Therefore, the scheduler maintains the priority globally for all the cores.

3.2 Estimating Fast Core Efficiency

The proposed scheduler determines fast core scheduling by looking up the efficiency order among all the running virtual machines. To estimate IPC and utilization, the hypervisor monitors hardware performance counters, and measures the retired instructions and processor cycles during a time interval for each virtual CPU. Based on the measurement, the scheduler periodically updates scheduling decisions. A scheduling interval is a periodic time interval to initiate a new round of sampling and scheduling, and a sampling window, which occurs within a scheduling interval, is the time for which the scheduler collects sample data. We empirically configure the sampling window to 2 seconds and the scheduling interval to 2.5 seconds. For every scheduling interval, performance counters are monitored for 2 seconds, and the next new scheduling round is prepared for the rest 0.5 seconds. Although we evaluated shorter sampling periods and scheduling intervals, they did not show meaningful differences in system throughput. Although shorter intervals may be necessary for some applications with fast changing behaviors, our benchmark applications did not require such short intervals.

We use a simple scheme to prevent drastic changes of IPCs from affecting scheduling decisions. At the end of every sampling period, the scheduler checks the difference between the prior IPC and the IPC measured in the current sampling window. If the difference is greater than a threshold, the efficiency for the next round of scheduling is not updated with the newly measured IPC. The new IPC replaces the current one, only when the newly measured IPC is within a threshold from the current one, or the IPC changes beyond the threshold are stable for two consecutive scheduling intervals. We empirically set the threshold to 15%. Our method for estimating utilization and efficiency does not need to modify guest operating systems or applications. The overheads of measuring IPC and utilization are negligible, with less than 0.05% slowdown for our applications.

3.3 Scheduling Asymmetric Multi-cores

Using the relative orders of the efficiency scores, the scheduler assigns VMs to fast or slow cores. We explore two possible policies to set fast core efficiency scores for virtual machines: scoring by VM (`amp-vm`) and scoring by vCPU (`amp-vcpu`). With the `amp-vm` policy, all the vCPUs in a virtual machine have the same fast core efficiency, and with the `amp-vcpu`, each vCPU even in the same VM, may receive a different one by per-vCPU efficiency. For a VM with a single vCPU, both policies have the same effect.

The first policy, `amp-vm`, is to assign a single fast core efficiency for each virtual machine, which is calculated by averaging the fast core efficiency scores of all the active virtual CPUs of the VM. If a virtual CPU is in sleep state, the virtual CPU is not included for calculating the score of the VM for the scheduling interval. The rationale behind this policy is that threads in many parallel workloads exhibit similar behaviors, and each VM commonly runs a single application. If users need to run different applications, they usually create separate VMs. Furthermore, the guest operating system running on a virtual machine also changes the mapping between application threads and virtual CPUs by thread scheduling. By averaging the fast core efficiencies of all the active vCPUs, the efficiency of a VM can avoid unnecessary fluctuations, which may cause unnecessary scheduling changes. If the virtual CPUs of a VM are assigned to a mix of fast and slow cores, each vCPU in the VM will be scheduled to fast and slow cores in a round-robin manner, having the same share of fast core slots. This feature is important to support performance scalability for asymmetric multi-cores.

The second policy called `amp-vcpu` is to use per-vCPU efficiency for scheduling. In the policy, the hypervisor does not consider the virtual machine each vCPU belongs to. This policy may lead to a better throughput than `amp-vm`, if a virtual machine has threads with different fast core efficiencies. However, for this policy to be effective, heterogeneous behaviors of vCPUs must be stable for a few scheduling intervals. However, during the intervals, guest operating systems may change the mapping between application threads and virtual CPUs. If the thread schedule changes inside the VM, the prediction at the hypervisor may be incorrect with `amp-vcpu`.

For `amp-vm` and `amp-vcpu` policies, the fast cores are scheduled by the efficiency scores of vCPUs, from the highest one to the lowest one. If all the fast cores are scheduled, the rest of vCPUs use slow cores. It never allows the fast cores to become idle, when the slow cores have active vCPUs to run (fast core conservative property). For `amp-R%-fair` policies, the scheduler distributes a part of fast core credits evenly to all the vCPUs. For each scheduling round of the Xen credit scheduler, R% of the total fast core credits are distributed evenly to all the vCPUs. Each vCPU is entitled to use a fast core for the equal amount of time slots. After all vCPUs consume the assigned fast core credits, which are R% of the total fast core credits, the remaining fast core credits of the round are distributed to vCPUs by the fast core efficiency order, in the same way as `amp-vm` and `amp-vcpu`.

4. EVALUATION

4.1 Methodology

The target system has a 12-core AMD Opteron 6168 processor. The processor consists of two dies, and each die has six cores sharing a 6MB L3 cache. Each core has separate 64KB instruction and data L1 caches, and a 512KB pri-

VM	Mix 1	Mix 2	Mix 3	Mix 4	VM	Mix 1	Mix 2	Mix 3	Mix 4
Single-vCPU, non-overcommitted					Multi-vCPU, non-overcommitted				
	4H-4M-4L	2H-6M-4L	6H-2M-4L	8M-4L		1H-1M-1L	2H-1M	1M-2L	1H-1L
VM 1	hmmer	hmmer	hmmer	gemFDTD	VM 1	freqmine(4)	freqmine(4)	pmake(4)	freqmine(6)
VM 2	hmmer	perlbench	hmmer	gemFDTD	VM 2	pmake(4)	swaptions(4)	apache(4)	canneal(6)
VM 3	perlbench	gemFDTD	hmmer	libquantum	VM 3	canneal(4)	x264(4)	canneal(4)	
VM 4	perlbench	gemFDTD	perlbench	libquantum	Multi-vCPU, overcommitted				
VM 5	gemFDTD	soplex	perlbench	soplex		1H-3M-2L	2H-2M-2L	2H-1M-1L	4H-2L
VM 6	gemFDTD	soplex	perlbench	soplex	VM 1	freqmine(4)	freqmine(4)	freqmine(6)	freqmine(4)
VM 7	soplex	libquantum	soplex	soplex	VM 2	x264(4)	swaptions(4)	swaptions(6)	freqmine(4)
VM 8	soplex	libquantum	soplex	soplex	VM 3	x264(4)	x264(4)	x264(6)	swaptions(4)
VM 9	mcf	mcf	mcf	mcf	VM 4	pmake(4)	pmake(4)	canneal(6)	swaptions(4)
VM 10	mcf	mcf	mcf	mcf	VM 5	canneal(4)	canneal(4)		canneal(4)
VM 11	sysbench	sysbench	sysbench	sysbench	VM 6	apache(4)	apache(4)		apache(4)
VM 12	sysbench	sysbench	sysbench	sysbench					

Table 1: Benchmark configurations

vate L2 cache. To emulate performance asymmetric multi-cores with current homogeneous multi-cores, we use dynamic voltage and frequency scaling (DVFS) to change clock frequencies of cores. DVFS has been a standard technique to emulate asymmetric cores with real hardware systems in prior work [6, 10, 14, 15, 16]. We configure cores into two core types, fast and slow cores which have 2.375:1 clock frequency ratio. The fast core type is configured to 1.9 GHz clock frequency, and the slow core type is to 0.8GHz clock frequency. As the target system has a single socket for the twelve cores, it does not have any NUMA effect, and thus scheduling threads across the shared L3 caches has relatively low overheads. We modified the Xen hypervisor (version 3.4.1) with the support for asymmetry-aware scheduling. We added `amp-fair`, `amp-vm`, `amp-vcpu`, and `amp-R%-fair` policies in addition to the mechanism to estimate fast core efficiency with hardware performance monitoring counters. For each virtual machine, para-virtualized linux-2.6.31.13 kernel was used as a guest operating system. The guest operating system was not modified for asymmetry-awareness.

We use applications with various characteristics, representing I/O-intensive workload (sysbench), server applications (apache and specjbb2k), single-threaded applications (SPECCPU2006), and parallel applications (pmake and PARSEC [3]). For pmake, SPEC CPU 2006 and PARSEC applications, execution times are used as the performance metric. For server and I/O workloads, the transaction throughput is used as the metric. For SPECCPU 2006 applications, we use the `ref` input set, and for PARSEC, we use the `native` input set.

We classified the applications into three classes, low (L), medium (M), and high (H) speedup applications. Table 1 presents various combinations of applications using the classification. The numbers in parentheses are the number of vCPUs used for the applications. We set up three execution scenarios for virtualized systems. The first case consists of VMs with a single vCPU, and the number of the total vCPUs in guest VMs is equal to the number of physical cores (`single-vCPU, non-overcommitted`). The second case consists of VMs with multiple vCPUs per VM, and vCPUs are also non-overcommitted (`multi-vCPU, non-overcommitted`). For these two non-overcommitted cases, the total number of vCPUs used by guest VMs is 12. The last case consists of VMs with multiple vCPUs per VM, but the number of vCPUs used by guest VMs is twice of the physical cores (`multi-vCPU, overcommitted`).

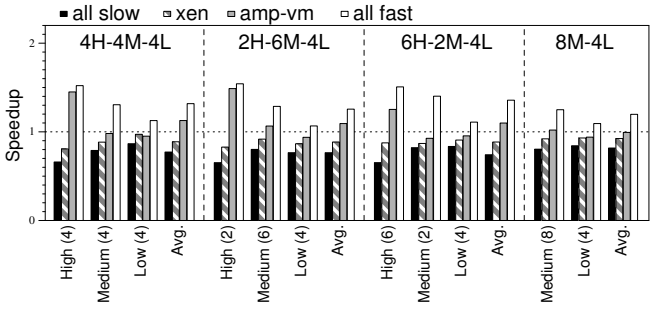

Figure 7: Single-vCPU, non-overcommitted: speedups over the fair scheduler (4-fast/8-slow)

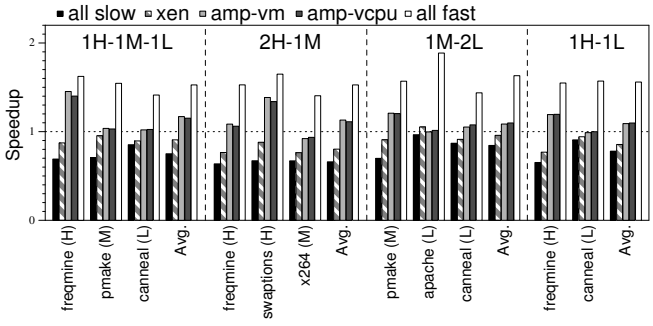

Figure 8: Multi-vCPU, non-overcommitted: speedups over the fair scheduler (4-fast/8-slow)

4.2 Performance

In this section, we evaluate the performance of the proposed scheduler with the `amp-vm` and `amp-vcpu` policies. As the performance lower and upper bounds, we also show the performance of all slow and fast core configurations. Figures 7, 8, 10 present the speedups of various configurations normalized to the performance with the `amp-fair` scheduler. Out of the twelve available cores in the target system, four cores are configured to fast cores, and eight cores are to slow cores.

Figure 7 presents the speedups with various schedulers in the single-vCPU and non-overcommitted runs. Y-axis shows the speedups over the fair scheduler, with the horizontal dotted line representing the fair scheduler. For each mix, we show the average speedups for high, medium, and low

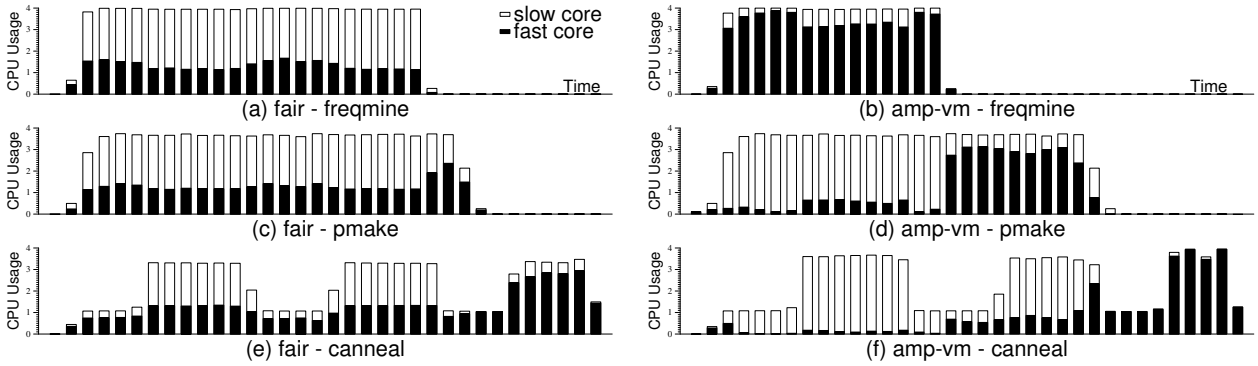

Figure 9: Breakdown of fast and slow core usage

efficiency VMs respectively, in addition to the average for the mix.

In the results, the high efficiency VMs in the first two mixes (4H-4M-4L and 2H-6M-4L) show the speedups close to those with the all-fast-core configuration. For the two mixes, 4 and 2 virtual CPUs of the high efficiency VMs can always be scheduled to fast cores. In the third mix (6H-2M-4L), only four vCPUs of the high efficiency VMs can use fast cores, reducing the average speedup for the high efficiency VMs. As expected, the performance of low efficiency VMs is slightly lower than that of the fair scheduler, since they do not receive fast core slots as much as the fair scheduler. The last mix (8M-4L) presents the case when asymmetry-aware schedulers cannot improve the performance, since all the VMs have low fast core efficiencies. As expected, the performance gain by asymmetry-aware scheduling is minor (less than 1%). However, even with the all-fast-core configuration, the average speedup is only 17%, due to their low fast core efficiencies. Even for the mix, the performance with `amp-vm` is as high as that with the `amp-fair` scheduler.

The Xen credit scheduler shows lower performance than that of the fair scheduler. The Xen default scheduler, unaware of asymmetry, may assign fast cores completely to low efficiency VMs, or may even make fast cores idle, when slow cores are busy. The performance of the Xen scheduler compared to `amp-fair` improves in the overcommitted runs shown in Figure 10, since all cores are kept busy in the overcommitted runs for most of the time and thus fast cores rarely become idle.

Figure 8 presents the speedups with the multiple-vCPUs and non-overcommitted runs. The overall performance trend is similar to that of the single-vCPU runs. For the first two mixes, the high efficiency VMs (freqmine in mix 1, and swaptions in mix 2) achieve significant speedups, by 40-45% over the fair scheduler. For mix-3 (1M-2L), pmake has a modest speedup of 20%, since the fast core efficiency is not very high for the VM. For 2H-1M mix, there are two high speedup applications, but only one of the two applications is selected for fast cores. In this mix, the higher efficiency application (swaptions) acquires fast cores over the other one (freqmine).

To show the scheduling differences between `amp-fair` and `amp-vm`, Figure 9 shows the breakdowns of CPU usage in the 1H-1M-1L mix during the runtime. Y-axis is the total CPU usage of a VM, which is up to 4 cores, and X-axis shows the timeline. Three graphs on the left show the results with the fair scheduler and the three graphs on the right with

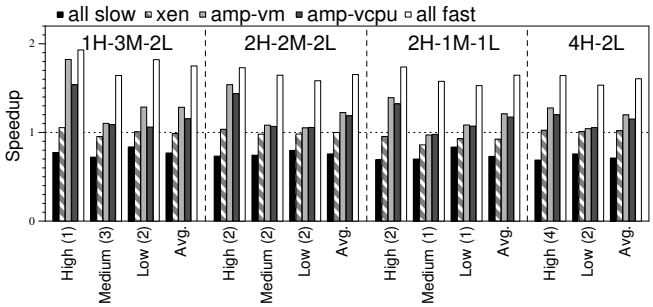

Figure 10: Multi-vCPU, overcommitted: speedups over the fair scheduler (4-fast/8-slow)

the `amp-vm` scheduler. With `amp-fair`, all of the co-running VMs use a nearly equal amount of fast core slots over time. Canneal often uses one fast core, during the serial execution phases. Note that we ran canneal three times to have co-running VMs finish with relatively similar execution times, so canneal shows such serial phases three times. In contrast to `amp-fair`, the `amp-vm` policy clearly shows that the higher efficiency application monopolizes fast cores and the lower efficiency ones continuously run on slow cores. However, even with `amp-vm`, pmake uses fast cores occasionally, either when the available parallelism of freqmine drops under four threads or when the efficiency of pmake increases temporarily higher than that of freqmine.

Figure 10 presents the speedups with the multi-vCPU and overcommitted runs. When vCPUs are overcommitted, the asymmetry-aware scheduler can achieve better performance improvement over the fair scheduler, compared to that in the non-overcommitted runs (10% on average). It shows that the importance of asymmetry-aware schedulers increases when there are more vCPUs to schedule than the number of physical cores. Such overcommitment of vCPUs is common in highly consolidated virtualized systems. In the first mix (1H-3M-2L), the high efficiency VM shows performance improvement nearly close to the all-fast-core configuration. Interestingly, the low efficiency workloads also show some speedups since they can use more fast core cycles than `amp-fair`, as the high and medium efficiency VMs finish earlier than they do with the fair scheduler.

Except for the 1H-3M-2L mix, `amp-vm` and `amp-vcpu` show nearly the same performance. The `amp-vcpu` policy shows a significant performance drop compared to `amp-vm` for 1H-3M-2L. This is because the `amp-vcpu` policy may result in a large number of vCPU migrations across the last level

Single-vCPU		Multi-vCPUs, non-overcommitted			Multi-vCPUs, overcommitted		
workload	amp	workload	amp-vm	amp-vcpu	workload	amp-vm	amp-vcpu
4H-4M-4L	7.38%	1H-1M-1L	0.0%	0.0%	1H-3M-2L	1.23%	4.19%
2H-6M-4L	10.0%	2H-1M	7.84%	6.43%	2H-2M-2L	0.0%	0.0%
6H-2M-4L	7.75%	1M-2L	0.27%	0.0%	4H-2L	0.0%	0.0%
8M-4L	13.89%	1H-1M	0.06%	1.21%	2H-1M-1L	2.85%	2.19%

Table 2: Fairness: the maximum slowdown against the fair scheduler

caches, as our target system has two L3 caches on the processor. With the mix, the total number of migrations across the last level caches in `amp-vcpu` is 5.6 times higher than that in `amp-vm`. With the other three mixes, the vCPU migration rates across the last level caches with `amp-vcpu` is only 1.1-1.2 times higher than those with `amp-vm`. Although `amp-vcpu` may be able to distinguish the fast core efficiencies of different vCPUs of the same VM, it may unnecessarily increase the vCPU migrations to different physical cores, often across the last level caches.

The performance results indicate that an active asymmetry-aware hypervisor can trace the fast core efficiencies of VMs only by monitoring performance counters for each VM or vCPU. Without any support from guest OSes, the hypervisor can achieve the potential performance improvement from a given asymmetric multi-core processor.

4.3 Fairness

The fair scheduler attempts to allocate fast and slow cores equally for all virtual CPUs, regardless of the behaviors of virtual machines. In this section, we evaluate how much fairness the active asymmetry-aware hypervisor can support compared to the fair scheduler. Table 2 presents the maximum slowdown against the fair scheduler for our applications. For each mix, the performance of each VM is compared to that of the same VM with the fair scheduler. The table shows the performance drop for the VM with the highest degradation. As shown in the table, the proposed `amp-vm` and `amp-vcpu` schedulers do not break the fairness among VMs significantly. For the multi-vCPU mixes, two mixes show 4.2% and 7.8% maximum slowdowns. Except for the two mixes, the maximum slowdowns are less than 3%. The maximum slowdowns are much higher in the single-vCPU workloads than in the multi-vCPU mixes. The worst degradation is 13.9% for the 8M-4L mix. As discussed in Section 2.2, `amp-vm` schedulers increase the maximum slowdown, when there are more high efficiency VMs than the fast cores can run.

The `amp-vm` and `amp-vcpu` policies assign fast cores to high efficiency VMs, not considering any fairness among VMs. Even with such unfair policies, the performance degradation against the fair scheduler is relatively low for two reasons. Firstly, virtual machines scheduled to slow cores have low efficiencies anyway. Therefore, even if the fair scheduler allocates some share of fast core cycles to the low efficiency VMs, the performance gain is low. Secondly, even low efficiency VMs use some fast core cycles during the execution because some virtual CPUs of the high efficiency VMs often may not be ready for execution.

Although in general, the maximum slowdowns are modest for most of our mixes, a more strict fair scheduling can be required to further reduce the maximum slowdowns. For such cases, we evaluate the amp-R%-fair scheduler to reduce the maximum slowdowns. Table 3 presents the maximum slowdowns with the amp-R%-fair scheduler for four mixes which showed the worst degradations with `amp-vm`. By increasing the portion of fast cores assigned with fair scheduling from 0% (`amp-vm`) to 80% (`amp-80%-fair`), the maximum slowdown decreases significantly for all four mixes.

	Single-vCPU		Multi-vCPUs non-over	Mult-vCPUs over
workload	4H-4M-4L	8M-4L	2H-1M	2H-1M-1L
amp-vm	7.38%	13.89%	7.84%	2.85%
20%-fair	6.12%	11.93%	6.82%	2.03%
40%-fair	4.82%	10.4%	6.79%	2.59%
60%-fair	4.16%	9.36%	4.97%	1.41%
80%-fair	2.81%	6.19%	3.2%	0.92%

Table 3: Fairness: the maximum slowdown with partially fair policies

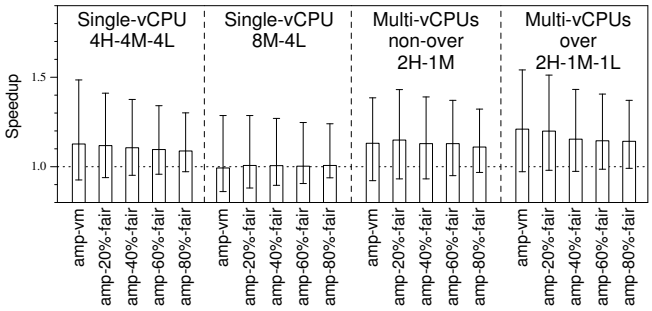

Figure 11: Average, maximum, and minimum speedups with partially fair policies

Figure 11 presents performance trends with different fair scheduling slots. The performance is normalized to that of `amp-fair`. Each bar is the average speedup with a scheduling policy, and the error range for each bar indicates the difference between the maximum speedup and slowdown. The figure clearly shows the performance and fairness trade-off. As the portion of fair scheduling increases, the average performance of each mix decreases slightly, with the maximum speedups decreasing faster than the average speedups. At the same time, the maximum slowdown decreases, getting close to the complete fair scheduler.

In summary, even without consideration on fairness in `amp-vm` and `amp-vcpu`, the maximum slowdowns are modest except for a few cases. However, if a cloud provider needs a strict fairness policy, the hypervisor can support partially fair scheduling at the cost of the overall throughput. As discussed in Section 2.2, the optimal percentage of fair scheduling is dependent upon the difference in the fast core efficiencies of co-running VMs. In this paper, we evaluated the effect of static fair scheduling ratios, but the dynamic adjustment of fair scheduling ratios to meet the fairness requirement will be our future work.

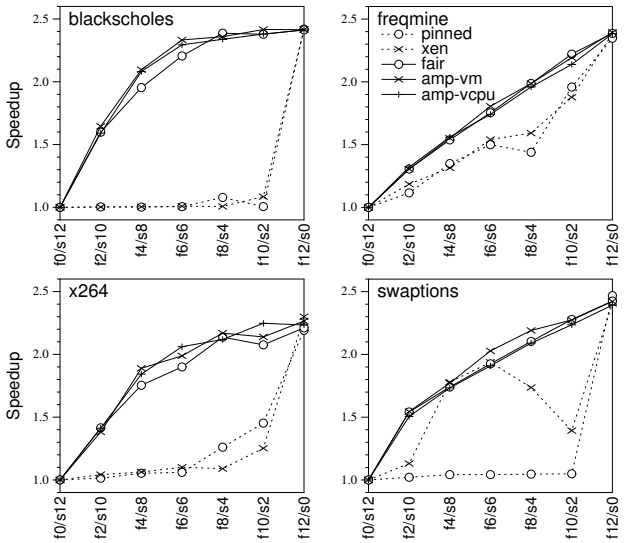

Figure 12: Performance scalability: single virtual machine

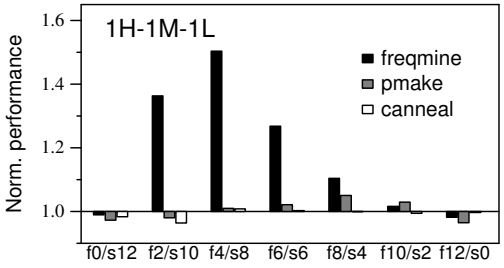

Figure 14: Performance normalized to amp-fair with increasing fast core counts

4.4 Scalability

As discussed in Section 3.3, for certain parallel applications, asymmetric multi-cores may not provide scalable performance, if fast and slow cores are mixed. In this section, we investigate whether making the hypervisor asymmetry-aware, can mitigate the scalability problem. Among the parallel applications in our benchmarks, we evaluate the scalability of our scheduler with the four applications which showed low scalability in Section 3.3: blackscholes, freqmine, x264, and swpations.

Figure 12 presents the speedups of increasing fast core counts in the 12-core system. For these results, only one guest VM is used in the system. The speedups are normalized to the all-slow-core configuration for each scheduler and application. As discussed in Section 3.3, in the `pinned` configuration, the hypervisor fixes the mapping between vCPUs and physical cores. The scheduling is completely dependent upon the guest OS, which is not aware of asymmetric cores. We also measure the scalability of the Xen credit scheduler. The scalability of the credit scheduler is as low as the `pinned` configuration. In both cases, neither the guest OS nor the hypervisor is aware of asymmetry.

The fair scheduler and two asymmetry-aware schedulers provide much better scalability than the credit scheduler and the `pinned` configuration. The three schedulers with high scalability maintain two rules in scheduling vCPUs to asymmetric cores. Firstly, the fast cores never become idle when the slow cores have active vCPUs to run. Secondly, each vCPU in the same VM gets an equal share of fast core cycles. By allocating fast and slow cores evenly for vCPUs in the same VM, even if a thread in a slow core lags behind other threads, the thread will be able to catch up when it receives some share of fast core cycles later. `amp-vm` and `amp-vcpu` show slightly better scalability than `amp-fair`, most notably in blackscholes. `amp-vcpu` shows a similar behavior to `amp-vm`, since the behaviors of all vCPUs in a VM are similar in the four applications.

We also evaluate the scalability with asymmetric multi-cores with multiple virtual machines running in a system. For the scalability evaluation, we use four application mixes from the multi-vCPU and non-overcommitted runs. Figure 13 presents the scalability with the `pinned` and `amp-vm` configurations. For both `pinned` and `amp-vm` configurations, the highest efficiency VM starts to improve its performance first when the number of fast cores is small. As the number of fast cores exceeds six, the second highest efficiency VM also shows a significant speedup. Compared to the `pinned` configuration, `amp-vm` provides much higher scalability with multiple virtual machines. As the number of fast cores increases from the `f0/s12` configuration, `amp-vm` picks the high efficiency VM first, and schedules it on the fast cores.

Figure 14 presents the speedup of `amp-vm`, compared to `amp-fair` with increasing numbers of fast cores. For each fast and slow core combination, the performance of a VM with `amp-vm` is normalized to that with `amp-fair` for the combination. When the processor has all slow or fast cores (f0/s12 or f12/s0), two policies have almost the same performance. However, with two, four, and six fast cores, freqmine with the highest efficiency takes advantage of the fast cores. As more fast cores are available, the difference between `amp-fair` and `amp-vm` decreases rapidly, as even the fair scheduler can schedule fast cores for most of the vCPUs.

The results in this section indicate it is important to guarantee performance scalability as well as high system throughput and fairness. Without the support for scalability, a VM, even if it may run a subset of its vCPUs on fast cores, may not exploit the computing capability of the fast cores. In virtualized cloud systems with multiple VMs sharing a physical system, the available fast core share for a VM may change constantly. In such cases, the support for scalability is essential not to waste expensive fast cores.

5. RELATED WORK

Performance asymmetric multi-cores were proposed to improve the power and area efficiency of multi-cores. Kumar et al. show that asymmetric multi-cores can be a power-efficient alternative to traditional symmetric multicores, and evaluate the potential power reduction using single-threaded applications [8]. Their extended study with multi-programmed workloads shows that asymmetric multi-cores can provide higher performance than symmetric multi-cores with the same area [9]. Balakrishnan et al. evaluate the impact of asymmetric multicores with various commercial workloads. They investigate performance variability and scalability problems with asymmetric multi-cores. They show that some applications need kernel support or application restructuring to reduce variability and improve scalability [1]. In this paper, we look into how virtualization can improve

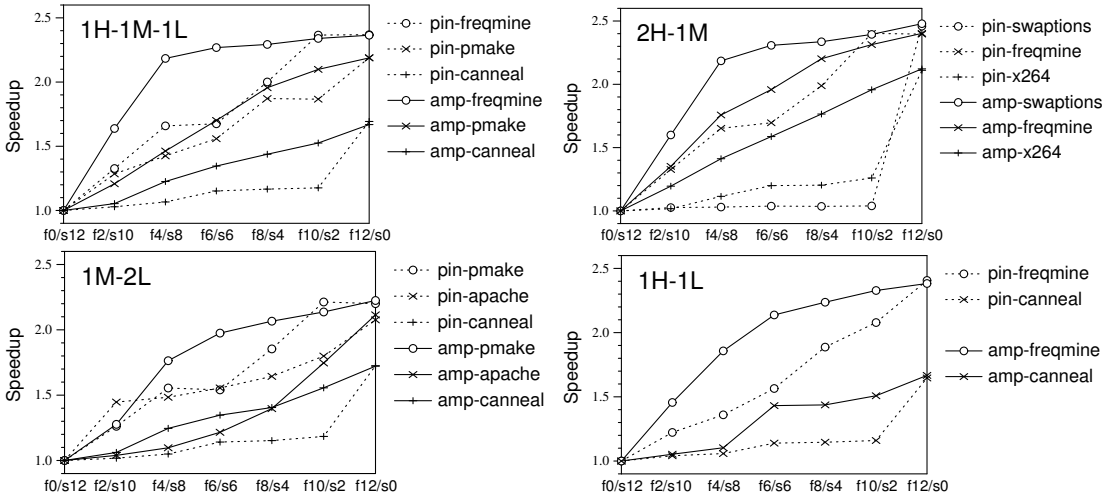

Figure 13: Performance scalability: multiple virtual machines

scalability without using asymmetry-aware operating systems.

The improved area efficiency of asymmetric multi-cores leads to the improvement of power efficiency, as simpler small cores usually consume much less power than more complex big cores. However, asymmetric multi-cores can achieve the power efficiency only when the operating systems can assign threads to the best type of cores, maximizing performance per watt for each core. To select threads for fast cores, Kumar et al. use a speedup of using a fast core compared to a slow core [9]. To calculate the speedup metric, their method requires running a thread both on fast and slow cores. HASS, instead of using a direct speedup by measuring IPCs on fast and slow cores, uses offline information which represents the characteristics of applications [16]. For each application, a signature which shows the cache behavior of the application is created by off-line measurement. With the signature associated with each application, the operating system, which knows the cache size of the system, can estimate the fast core speedup for the application. Bias scheduling categorizes threads into either small core bias or big core bias, and schedules threads based on the estimated bias [7]. The bias represents how much the performance of each thread is dependent upon core capability. To estimate the bias of a thread, they use last-level cache misses and front-end stalls as metrics of fast core speedup.

Another aspect of using asymmetric multi-cores is to schedule threads by their available parallelism. Highly parallel applications may use many small cores more efficiently than serial applications. Fedorava et al. discuss parallelism-awareness for scheduling asymmetric cores [5]. They proposed scheduling policies to allocate fast core cycles to applications with less parallelism to finish serial phases quickly. Saez et al. proposed a new metric which combines the fast core speedup metric for single-threaded applications and the parallelism awareness [15]. However, parallelism-awareness may have a different effect on hypervisors, compared to native operating systems. For on-demand computing clouds, users may pay for each virtual core. If the parallelism awareness is added to the hypervisor scheduler, expensive virtual machines with many virtual cores will be scheduled to slow cores, if there are less expensive virtual machines with fewer virtual cores running on the same system. To avoid such anomaly, we do not use parallelism awareness for this work.

Li et al. explore the design space of fair schedulers for asymmetric multi-cores [11]. Their design focuses on improving the fairness of schedulers and also considers the non-uniformity in the memory hierarchy to estimate the cost of re-mapping between fast and slow cores. Age-based scheduling improves throughput for parallel applications with unbalanced loads among threads [10]. It allocates fast core cycles to the threads that become performance bottlenecks. Li et al. extend single-ISA asymmetry to overlapping-ISA support [12]. Cores share the basic ISA, but each core type may have specialized units to support distinct instructions. They show how operating systems can support such asymmetric cores efficiently. Kazempour et al. proposed an asymmetry-aware scheduler for hypervisors, which focuses on the fair use of fast cores among competing virtual machines [6]. The hypervisor does not attempt to measure the fast core efficiency of virtual cores, but guest OSes must make scheduling decisions to use asymmetric cores efficiently. Guest VMs allocate fast and slow cores, and the role of the hypervisor is limited to supplying fast core cycles fairly to all virtual machines which request fast cores.

6. CONCLUSIONS

In this paper, we proposed a hypervisor scheduler which can schedule virtual machines in asymmetric multi-cores without any support from guest operating systems. Using a simple approximate method to estimate speedups based on instruction throughput and CPU utilization, the active asymmetry-aware hypervisor can identify the characteristics of virtual machines, and assign fast cores to VMs which can maximize the throughput. The fairness with the scheduler is close to that of the asymmetry-aware fair scheduler, when the VMs sharing a system exhibit wide differences in their fast core efficiencies. We also showed that without proper supports from operating systems and hypervisors, certain applications cannot achieve performance scalability with fast cores when both fast and slow cores are used for the applications. The asymmetry-aware hypervisor can mitigate the

scalability problem by preventing fast cores from becoming idle before slow cores, and by allowing virtual CPUs from the same VM to use both fast and slow cores.

We believe that with proper supports for asymmetric multi-cores in hypervisors, it is feasible to hide physical asymmetry from the guest operating systems. The hypervisor can use fast and slow cores efficiently by identifying the characteristics of virtual machines. Each virtual machine does not need to be aware of asymmetry, and it can use virtual homogeneous CPUs, which are not restricted by the underlying physical architecture. This paper shows that virtualization can make the adoption of asymmetric multi-cores less costly as the virtualization layer can hide the machine-dependent physical asymmetry. Users can use conventional operating systems without any modification for asymmetric multi-cores. Properly designed hypervisors can extract the benefits of asymmetric multi-cores, without changing already complex operating systems.

Acknowledgments

This research was partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0025511). This work was also partly supported by the IT R&D Program of MKE/ KEIT (2011-KI002090, Development of Technology Base for Trustworthy Computing).

7. REFERENCES

[1] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, pages 506–517, Washington, DC, USA, 2005. IEEE Computer Society.

[2] 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 *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, New York, NY, USA, 2003. ACM.

[3] 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 (PACT)*, October 2008.

[4] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, 2007.

[5] A. Fedorova, J. C. Saez, D. Shelepov, and M. Prieto. Maximizing power efficiency with asymmetric multicore systems. *Commun. ACM*, 52(12):48–57, 2009.

[6] V. Kazempour, A. Kamali, and A. Fedorova. AASH: an asymmetry-aware scheduler for hypervisors. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 85–96, New York, NY, USA, 2010. ACM.

[7] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, pages 125–138, New York, NY, USA, 2010. ACM.

[8] 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 *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 81, Washington, DC, USA, 2003. IEEE Computer Society.

[9] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, page 64, Washington, DC, USA, 2004. IEEE Computer Society.

[10] N. B. Lakshminarayana, J. Lee, and H. Kim. Age based scheduling for asymmetric multiprocessors. In *Proceedings of the 2009 ACM/IEEE Conference on Supercomputing (SC)*, pages 1–12, New York, NY, USA, 2009. ACM.

[11] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC)*, pages 1–11, New York, NY, USA, 2007. ACM.

[12] T. Li, P. Brett, R. Knauerhase, D. A. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-isa heterogeneous multi-core architectures. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[13] J. C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro*, 28(3):26–41, 2008.

[14] J. C. Saez, A. Fedorova, M. Prieto, and H. Vegas. Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors. In *Proceedings of the 7th ACM International Conference on Computing Frontiers (CF)*, pages 31–40, New York, NY, USA, 2010. ACM.

[15] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, pages 139–152, New York, NY, USA, 2010. ACM.

[16] 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.