ACCESS: Smart Scheduling for Asymmetric Cache CMPs

Xiaowei Jiang[†], Asit Mishra[‡], Li Zhao[†], Ravishankar Iyer[†], Zhen Fang[†], Sadagopan Srinivasan[†], Srihari Makineni[†], Paul Brett[†], Chita R. Das[‡]

[†]Intel Labs
Intel Corporation
{xiaowei.jiang, li.zhao, ravishankar.iyer}@intel.com

[‡]Dept. of Computer Science and Engineering
The Pennsylvania State University
{amishra, das}@cse.psu.edu

Abstract

In current Chip-multiprocessors (CMPs), a significant portion of the die is consumed by the last-level cache. Until recently, the balance of cache and core space has been primarily guided by the needs of single applications. However, as multiple applications or virtual machines (VMs) are consolidated on such a platform, researchers have observed that not all VMs or applications require significant amount of cache space. In order to take advantage of this phenomenon, we explore the use of asymmetric last-level caches in a CMP platform. While asymmetric cache CMPs provide the benefit of reduced power and area, it is important to build in hardware/software support to appropriately schedule applications on to cores with suitable cache capacity. In this paper, we address this problem with our ACCESS architecture comprising of: (a) asymmetric caches across a group of cores, (b) hardware support that enables prediction of cache performance on the different sized caches and (c) OS scheduler support to make use of the prediction capability and appropriately schedule applications on to core with suitable cache capacity. Measurements on a working prototype using SPEC2006 benchmarks show that our ACCESS architecture can effectively schedule jobs in an asymmetric cache CMP and provide 23% performance improvement compared to a naive scheduler, and is 97% close to an oracle scheduler in making schedules.

1 Introduction

Today's chip-multiprocessors (CMPs) are already integrating multiple general-purpose cores on the die. With every successive generation, chip manufacturers are integrating more and more cores. For such a scaling trend to continue, it is important to find ways to reduce area and power. Last-level cache space is a significant fraction (40% or more) of the processor die area [4, 26]. Reducing last-level cache space on the die can help improve area and power significantly [31].

Here, we identify the fact that workloads are also changing rapidly. Nowadays, CMP platforms are employed to run multiple virtual machines (VMs) or applications. Not all VMs or applications require similar caching capacity. Rather, many of the workloads have either a small working set or streaming behavior, which prevents them from making use of a larger cache capacity. On the other hand, applications or VMs that need a large cache space would be affected if they are allocated with smaller caches. To tackle this dilemma, it is natural to enforce shared caches with cache partitioning feature (*Virtual Asymmetry*) or private caches with different sizes (*Physical Asymmetry*) [4]. Comparing to virtually asymmetric caches, physically asymmetric cache uniquely offers power/performance efficiency, convenient chip power management and eliminates the hardware/software complexity incurred by cache partitioning. Hence, we explore the use of physically asymmetric last-level caches with the goal of reducing area and power consumption, while maintaining competitive performance.

One key challenge with an asymmetric cache CMP design is that Operating System (OS) scheduler is unaware of the asymmetry in cache space across the cores. As a result, a naive scheduler may end up scheduling applications that require a large cache on a core that is connected to a small cache. In this paper, we explore the hardware and software required to perform asymmetry-aware scheduling. Our proposed solution is called ACCESS, an Asymmetric Cache CMP Enhanced with Smart Scheduling support. The key questions that we try to address are the following: (1) What are the benefits of building an asymmetric cache CMP if applications can be scheduled to the most suitable cores? (2) What hardware support would be advantageous to help OS/hypervisors schedule effectively on the asymmetric cache CMP? (3) What OS scheduler changes are needed to take advantage of hardware support to achieve the best performance?

We address these questions by designing and implementing a scheduler and conduct an in-depth measurement-based evaluation with many workloads. Overall, the contributions of this paper are the following:

- We investigate the performance implications of physically asymmetric cache CMPs based on the observation that there are many applications which are cache insensitive and hence, scheduling these applications on cores attached to smaller caches improves performance per watt.
- We describe the hardware support needed in ACCESS for performance prediction of threads on a CMP with asymmetric caches. This hardware support is intended to help OS and VMM schedulers to efficiently schedule on the asymmetric cache CMP.
- We propose asymmetry-aware OS scheduler optimizations to take advantage of the above hardware support and achieve scheduling with minimal O(1) overhead. With a detailed measurement-based evaluation on a 4-core Xeon 5160 chip with asymmetric last-level caches, we show that our scheduler based contention-mitigating technique can achieve on average 20% speedup comparing to the Linux CFS scheduler and its scheduling accuracy is >97% of an asymmetric-cache aware oracle scheduler. We also conclude that the highest impact of contention-aware scheduling technique is not in improving performance as a whole but in minimizing the contention of the shared last-level asymmetric cache.

The rest of this paper is organized as follows. Section 2 motivates a case for asymmetric cache architectures for general purpose CMP. Section 3 introduces our proposed asymmetric caches architecture and presents the asymmetric cache-aware OS scheduler design. Section 4 depicts the experiment setup while Section 5 presents the results and analysis. Section 6 discusses related work in this area and Section 7 concludes the paper.

2 A Case for Asymmetric Caches in CMPs

In order to motivate the asymmetric caches, we present a simple cache size sensitivity study in Figure 1. Figure 1(a) shows the performance (in terms of CPI) degradation of 12 SPEC2006 [30] applications measured in Intel Xeon 5160 processor when the cache capacity is progressively decreased. We find that as we progressively decrease the last-level cache size from 4MB to 512KB, the performance degradation for many benchmarks is not appreciable. Overall, out of the 12 benchmarks analyzed, 7 showed less than 10% performance degradation while the last-level cache size is reduced from 4 MB to 512 KB. Clearly, these benchmarks can sustain their performance even with a smaller

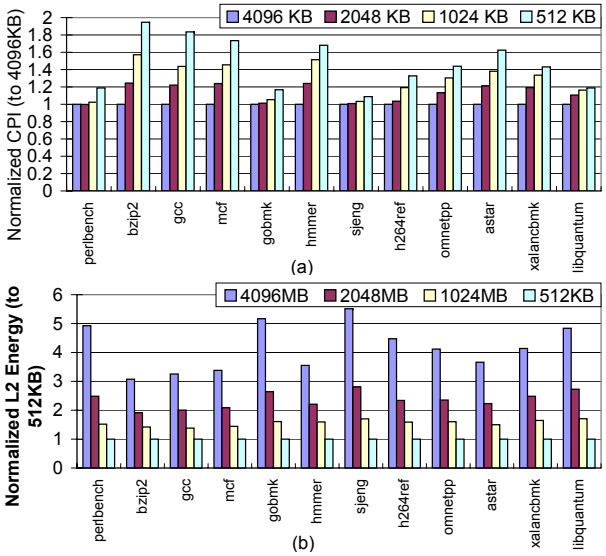

Figure 1. (a) CPI and (b) Energy consumption of SPEC 2006 workloads for different last-level cache sizes. Machine parameters are presented in Section 4.

cache size and providing a larger cache for these application leads to cache underutilization.

There are two primary reasons behind this phenomenon: (1) The working-set size (WSS) of the benchmark is small, in which case the benchmark does not benefit from cache space more than its WSS, and (2) The benchmark is streaming in which case the caching reuse is very less, implying that the cache capacity does not contribute to performance. In our analysis, 7 out of the 12 SPEC2006 benchmarks are cache insensitive (5 are streaming and 2 have a WSS less than 256 KB) and performance of the remaining benchmarks is a critical function of cache capacity. Thus, when a number of these benchmarks are consolidated on a multi-core platform, it makes sense to support asymmetric caches from performance/energy perspective.

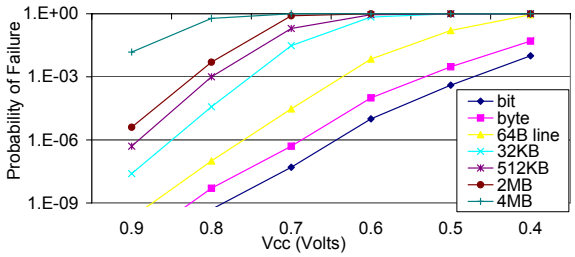

Figure 2. Pfail vs. VCC of SRAM arrays

The second motivation for using asymmetric caches is that it can help improve Energy per Instruction (EPI) due to two factors. First, power-gating a large cache when it is idle eliminates its power consumption. This improves EPI of the system in lightly loaded cases. The energy savings

with a smaller cache is quantified in Figure 1(b), which shows the total energy consumption for each application. We used Cacti 6.5 [26] for modeling the cache access energy and used on-chip counters for measuring the cache activity. The energy numbers are normalized to the case of a 512KB cache. We find that, as the cache size increases, the corresponding energy consumption increases significantly. Overall, this analysis shows that, applications that have a smaller WSS or whose performance does not improve with increase in cache size, can benefit with a smaller cache (i.e. the performance per Joule would be significantly better).

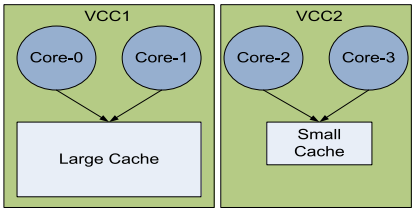

Figure 3. Layout of asymmetric LLC.

The second and a less obvious reason behind asymmetric caches helping improve EPI lies in the fact that a smaller cache enables a lower operating voltage for the power plane that it belongs to. Reducing supply voltage (Vcc) is arguably the most effective way to reduce the power consumption of a microprocessor, since dynamic power is a quadratic function of Vcc and leakage is an exponential function of Vcc. However, there is a minimum supply voltage, Vcc_min, below which the circuit no longer works reliably. The SRAM arrays typically have higher Vcc_mins than other blocks on a chip such as register files and computation logic. The primary reason is that with bit errors randomly occurring across the chip, the high transistor count in an SRAM array corresponds to high probability of failures. Thus, the nominal Vcc_min of the cache actually determines the Vcc_min of the whole processor. Figure 2 shows how probability of failure (Pfail) increases as the SRAM size of interest increases at any given Vcc. (Data obtained from [31].) For example, assuming that Pfail of 1E-03 is needed for an acceptable manufacturing yield, then a processor with a 512KB cache and one with a 4MB cache would require a Vcc of 0.8V and 1V (projected) respectively under this process. The above results form the basis of our asymmetric cache CMP proposal, where we propose that CMPs should be designed in such a way that applications do not benefit from a larger cache is scheduled on a core with a smaller cache. Specifically, Figure 3 shows the platform that we envision with asymmetric caches. In general there can be multiple CPUs sharing a last level cache that is not the same in size. Additionally, a completely heterogeneous CMP would have different types of cores as well. In this work, to quantify the benefits solely due to asymmetric cache, we only consider heterogeneity in cache capacity at the last-level.

3 ACCESS Architecture: Hardware and Software Support

We now provide an overview of our ACCESS architecture (Section 3.1) , and then discuss the details on the hardware support (Section 3.2) and OS support (Section 3.3) required for achieving high performance on this architecture.

3.1 ACCESS Architecture Overview

Figure 4 provides a basic overview of the ACCESS architecture. It shows the asymmetric cache CMP platform with multiple cores, where a group of cores share a last-level cache (LLC). The LLC across these groups of cores is asymmetric in size. In this paper, we primarily focus on two sizes (referred to as big and small). When running an OS on such an architecture, a naive scheduler that is not asymmetry-aware will have significant challenges in performance because it may run applications with significant cache requirements on cores attached to small LLC or vice-versa. In order to address this problem, our ACCESS architecture provides hardware support for predicting the cache performance of a task on each of the asymmetric cache sizes supported in the platform. In other words, we propose an ACCESS Prediction Engine (APE) in the last-level cache subsystem as shown in Figure 4 that is designed to achieve the following functionality:

- For any running task (on any core connected to any cache), APE monitors the behavior of the running task and provides an estimate of the performance of that task on each of the other cache types. For example, when a threadX is scheduled on a core attached to a big cache, ideally, APE can provide an estimate of the cache performance of that task if it was to run on a core connected to a small cache.
- From an OS perspective, APE is exposed as a performance monitoring capability. When the OS deschedules a task, it should be able to read the APE data to determine the expected performance of that task on the core with the other cache type.

The goal of our ACCESS architecture is to let the OS make use of the APE data and perform smart scheduling of applications. In other words, the OS must not have to sample the tasks across all of the core/cache types, but just needs to run a task on the core where it is initially spawned, read performance data from the APE engine, and determine using smart scheduling policies (as described later). For experimentation, our targeted system is a 4 core platform, with 2 cores sharing a bigger LLC (4M) and the other 2 cores sharing a smaller LLC (512KB). The goal of our OS scheduler is to (1) utilize the asymmetric cache to improve throughput and reduce power and, (2) minimize

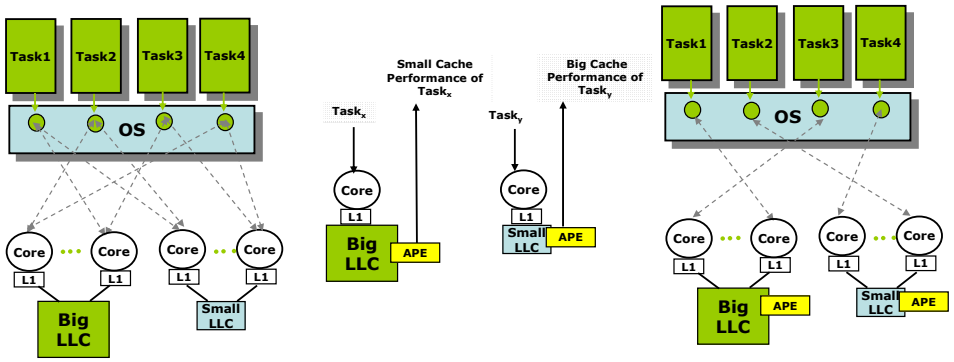

Figure 4. ACCESS Architecture: Overview, Hardware Support and OS Support.

shared cache contention. We compare the smart scheduler to an asymmetry-unaware Linux 2.6.32 completely fair scheduler (CFS) [2] and an asymmetry-aware Oracle scheduler. Figure 5 shows the performance of the default Linux scheduler as well as the Oracle scheduler on a asymmetric cache CMP with no hardware prediction engine. Since the Linux scheduler has no knowledge of asymmetric cache platform, it blindly schedules applications on caches leading to sub-optimal performance. The Oracle scheduler on the other hand has knowledge appropriate mapping of applications to caches that would benefit them maximally. This helps the Oracle scheduler achieve 20% better performance over the Linux scheduler. Our proposed scheme tries to perform as close as possible to such an Oracle scheduler. For this, we require APE hardware support for predicting the performance of a thread on the asymmetric caches. We next detail the APE solution and the smart scheduling algorithm.

3.2 ACCESS Prediction Engine

To predict both the single-thread performance and the contention in a cache other than the one on which a thread is currently running, we use shadow tags array [11, 27]. A shadow tag is functionally similar to a regular cache structure with the exception that it does not have a corresponding data array. In our case, we use it to estimate the performance of an application if it were to be running alone on a bigger cache and a smaller cache. To measure the performance of an application running alone on a 4MB and 512KB cache, we have a shadow tag that emulates a 4MB cache and another shadow tag emulating a 512KB cache. Both the LLC and these two shadow tags receive the same memory access stream and the shadow tags enable us to know the single thread performance on a 4MB and 512KB cache.

Figure 6 shows the schematic diagram of 2 cores and their corresponding shadow tags. In this schematic, the LLC is connected to two cores and for an application running on each core. We have two shadow tags - one for measuring the performance of the application on the 4MB LLC and the other for 512KB LLC.

Use of shadow tags introduces area and energy overhead. The number of shadow tags required is equal to the product of the number of cores and the number of unique cache sizes that we are interested. For instance, since we have two cores sharing a LLC and thus, we require 4 shadow tags per LLC (8 shadow tags in total). To minimize the area overhead due to the introduction of shadow tags, we do set-sampling and only maintain one set for every 256 sets in the actual cache [32, 15]. Hence, the number of sets in the shadow tags is 16 for 4MB 16-way cache (the 4MB LLC has 4K sets), and 16 for 512KB 2-way cache. Thus the additional storage overhead for 4MB shadow tag is 512B and 512KB shadow tag is 64B. One primary justification for sampled shadow tags to do prediction as opposed to a complete shadow tag is that the accesses to the cache are usually uniformly distributed. We verified that the error rate of a set-sampled shadow tag is within 2% of a complete shadow tag.

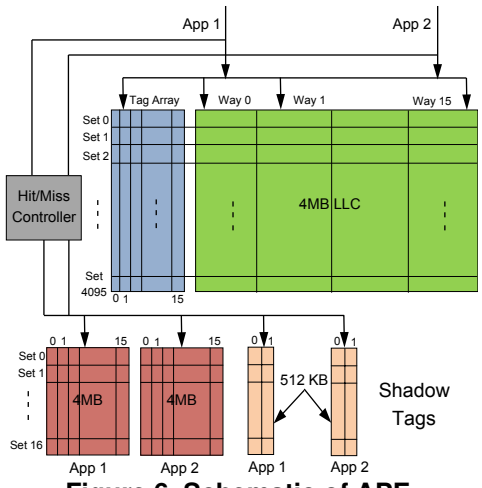

Figure 6. Schematic of APE.

The shadow tags simulate the performance of the applications only during the training phase of our algorithm (described later in Section 3.3) which is 1% of benchmarks total run-time. Thus, the shadow tags are only accessed during this training phase, and because of set-sampling the number of accesses to shadow tags is 1.56% (the shadow

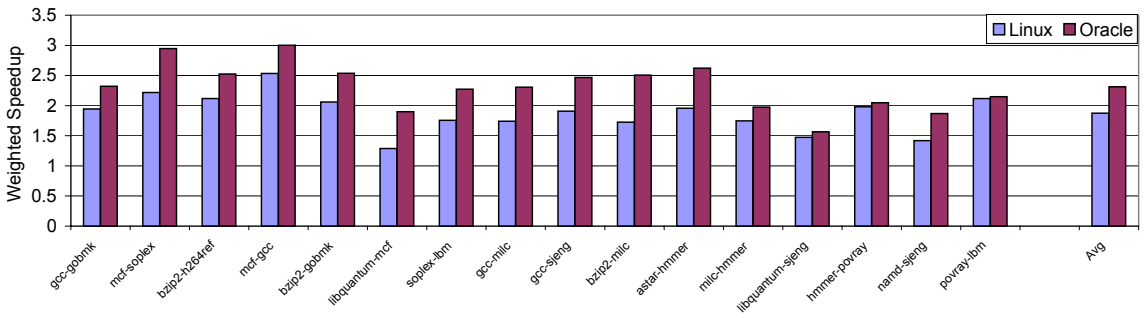

Figure 5. Speedup of applications using the default Linux scheduler and an Oracle scheduler.

tag has 1 set per 256 sets * 4 shadow tags per LLC) of the total accesses during the training phase. To facilitate this, the cache controller only forwards requests to the shadow tags if they would be hits in the shadow tag (i.e. the shadow tag has the particular sampled sets).

3.3 OS Scheduler Support for ACCESS

The OS scheduler is a critical piece of system software that manages processor resource allocation to software threads. The OS scheduler designs are typically centered on fairness [21, 20] rather than threads' performance. In this work, we propose a novel OS scheduler design that seeks to improve overall system performance on the ACCESS architecture by allocating the cache resources to threads wisely. We call this scheduler design as ACS, an Asymmetric Cache Scheduler. One way to improve overall thread performance using the OS scheduler is to use training based approach. In this model, all possible thread-to-core/cache mappings (i.e. schedules) are evaluated and the best candidate is identified after exhaustive search [18, 19]. This incurs significant overhead. Our scheduler, however, uses minimal training by studying an arbitrary schedule and deriving the best schedule as follows.

To simplify our discussion, we center our explanation on the architecture shown in Figure 3. However, our scheduler design can be easily extended to architectures with more number of caches. For a given threads schedule, let

$$MPI_{sum} = \sum_{Threads\_on\_L} MPI_{Thread_i} + \sum_{Threads\_on\_S} MPI_{Thread_i} \quad (1)$$

where *MPI* is the Misses Per Instruction of a thread. MPI_{sum} denotes the aggregated *MPI* of all threads currently running in the system. In practice, we found that the threads schedule that has the smallest MPI_{sum} always leads to the best overall performance.

In the training phase, our scheduler looks at an arbitrary schedule (threads are trained on cores where they are spawned initially) and obtains the performance statistics of each thread under this specific schedule. Performance statistics are selectively picked using hardware performance counters along with shadow tags. As we discussed in Section 3.2, shadow tags enable us to get the cache miss numbers of a given thread when it is running alone on a large or a small cache. Consequently, for each thread i, our scheduler maintains $\langle MPI_{threadi\_L}, MPI_{threadi\_S}\rangle$, where $MPI_{threadi\_L}$ ($MPI_{threadi\_S}$) is the *MPI* of thread i running alone on the large (small) cache. Now the question is how best to utilize such information obtained from an arbitrary schedule to arrive at an optimal scheduling policy. We separate our discussion into two cases, where the last level asymmetric caches are either shared or private.

Private Cache Case. In this scenario, the last level cache (either large or small) is exclusively used by one thread at any given time. The obtained $\langle MPI_{threadi\_L}, MPI_{threadi\_S}\rangle$ from one schedule can be directly used for a different schedule, since it remains constant across different schedules (program phase changes are handled separately in our scheduler). We can calculate MPI_{sum} of each possible schedule based on all $\langle MPI_{threadi\_L}, MPI_{threadi\_S}\rangle$ and pick the schedule that yields the lowest MPI_{sum}. In a two threads (T_1 and T_2) example, the following two schedules are compared.

1. T_1 on large cache, T_2 on small cache:

   $MPI_{sum} = MPI_{T1\_L} + MPI_{T2\_S}$;

2. T1 on small cache, T2 on large cache:

   $MPI_{sum} = MPI_{T1\_S} + MPI_{T2\_L}$.

Shared Cache Case. This scenario (e.g. in Figure 3, 2 cores share a cache) is more complex due to the presence of cache contention [5, 16]. $\langle MPI_{threadi\_L}, MPI_{threadi\_S}\rangle$ obtained from one schedule cannot be directly applied to other schedules like in the private cache case, because *MPI* of a thread will change when it is co-scheduled with another thread on the same cache. Moreover, the degree of such change may vary with respect to the thread co-scheduled.

Before discussing how our ACS scheduler handles shared cache case, let us first introduce the Power Law of Cache Misses [28, 9]. Mathematically, it states that if MR_0 is the miss rate of a thread for a baseline cache size C_0, the miss rate (*MR*) for a new cache size *C* can be expressed as

$$MR = MR_0(\frac{C}{C_0})^{-\alpha} \quad (2)$$

where α is a measure of how sensitive the thread is to changes in cache size. α is dependent on both the thread and the underlying core architecture. However, for the same thread running on the same core architectures (such as in Figure 3), α can be viewed as a constant. Note that *MPI* can be expressed as

$$\begin{aligned} MPI &= \frac{L2\_miss\_num}{instruction\_count} = \frac{MR \times L2\_access\_num}{instruction\_count} \\ &= MR \times \frac{L1\_miss\_num}{instruction\_count} \end{aligned} \tag{3}$$

where $\frac{L1\_miss\_num}{instruction\_count}$ is identical for a given thread on both the small and large caches. Plug 3 into 2, we can conclude that Power Law of Cache also holds for *MPI*, that is

$$MPI = MPI_0(\frac{C}{C_0})^{-\alpha} \tag{4}$$

Recall that we obtain $\langle MPI_{threadi\_L}, MPI_{threadi\_S}\rangle$ for each thread. Thus, α for thread i can be derived as

$$\alpha = -log_{\frac{C_{L\$}}{C_{S\$}}} \frac{MPI_{threadi\_L\$}}{MPI_{threadi\_S\$}} \tag{5}$$

Given 4 and 5 together, if we know how much cache capacity will be taken away (the rest capacity can be expressed as cache occupancy) for thread i when it is shares the cache with another thread j, we can then derive the corresponding $\langle MPI_{threadi\_j\_L}, MPI_{threadi\_j\_S}\rangle$.

When a cache miss on thread i occurs, a new block of thread i is brought into the shared cache, replacing an existing cache block of either thread i itself, or a block that belongs thread j. Only when the later case happens, the cache occupancy of thread i changes. Essentially,

$$Occupancy_{thread_{i\_j}} = \frac{miss\_num_i \times Prob_{i\_replace\_j}}{miss\_num_i \times Prob_{i\_replace\_j} + miss\_num_j \times Prob_{j\_replace\_i}} \tag{6}$$

Where *miss_num* of a thread is the number of last level cache misses when the thread runs alone on the cache. In practice, we found that 6 can be simply estimated as

$$Occupancy_{thread_{i\_j}} = \frac{miss\_num_i}{miss\_numi + miss\_numj} \tag{7}$$

In a shared cache scenario, our ACS scheduler maintains the misses, α and *MPI* for each thread . With such numbers, the adjusted *MPI* of when threads share the cache can be derived. For example, to calculate the new *MPI* of thread i when it shares cache with another thread j on the large cache, we first calculate $Occupancy_{threadi\_j}$ using 7, then the adjusted *MPI* can be expressed as

$$MPI_{threadi\_j} = MPI_{threadi\_L\$} \times Occupancy_{threadi\_j}{}^{-\alpha} \tag{8}$$

Using the adjusted MPIs, the scheduler can determine the best schedule that yields the lowest MPI_{sum}.

O(1) ACS Scheduler. In the training phase of a thread, all necessary performance statistics are read out and stored along with each thread's `task_struct`. Once the training phase is over, the best schedule is computed and each thread is mapped to its best location for the sake of overall performance. The thread then enters a long execution phase.

Recall that our ACS scheduler aims to improve system performance by figuring out the best schedule that leads to the smallest MPI_{sum}. Although it avoids unnecessary training attempts, without careful design, the scheduler can incur significant computation overhead due to the excessive amount of potential schedules that need to be computed and compared.

The naive bar in Figure 7(a) depicts the computation overhead (in cycles) of computing and sorting out all possible schedules ($O(n^2)$ complexity) for various numbers of threads. For 16 threads, 2ms (on a 3GHz processor) is needed to calculate the best schedule. This is 50% of a typical OS scheduler time quantum and is clearly not acceptable. Another concern with such a naive scheduler design is that whenever a new thread finishes its training phase and needs to be scheduled, a re-shuffle of entire thread-to-cache mapping might be needed, since the new best schedule may change significantly with respect to the old best schedule. This is problematic in that it may entail many thread migrations. For instance, to migrate a thread from the 512KB cache to the 4MB cache, both the 4MB cache and the TLBs (e.g. 128 entries with 4KB page size) on the corresponding processor core have to be refilled (we ignore L1 caches and other processor resources in this example). Assuming a sustainable memory bandwidth at 12.8 GB/s, ideally this refill will last (4MB+128*4KB)/12.8GB/s=0.35ms. This again contributes towards roughly 9% of a typical 4ms schedule time quantum. Such overhead becomes compounded with unbounded number of migrations and will significantly affect threads' execution time.

To mitigate this schedule computation overhead, we propose a novel ACS scheduler that has a constant O(1) computation complexity. Different than the naive scheduler that has completely no memory of current system state (i.e. mapping of applications to cores/caches) and picks the best schedule by computing from scratch, our O(1) ACS scheduler always selects the best system state based on current system state. Three cases of thread events are handled respectively when (1) a thread arrives (i.e. it finishes training and needs to be mapped to a proper cache/core), (2) a thread exits or (3) a program phase change occurs. Conceptually, phase change can alternatively be seen as an old thread exits from the system and a new thread arrives. We center our discussion on how our scheduler handles thread arrival. The scheduler operation on thread exiting is handled similarly.

When a thread arrives, the scheduler decides where to map the thread by comparing the following six cases (keep in mind that number of threads on each core/cache needs to be roughly the same to maintain fairness in terms of CPU

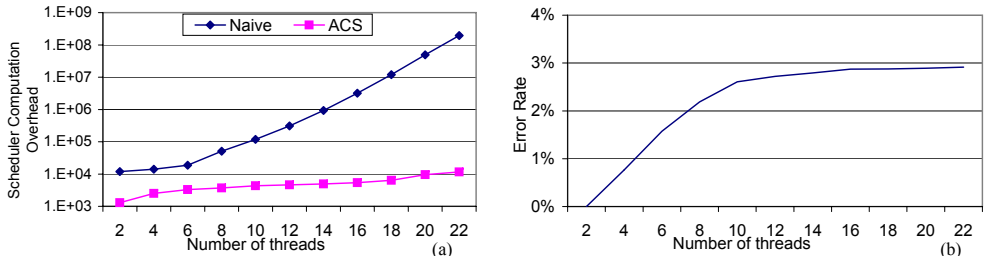

Figure 7. a. Scheduler Computation Overhead. b. ACS Accuracy in Computing Best Schedule

time a thread can get):

(1) The new thread is scheduled on the large cache;

(2) The new thread is scheduled on the large cache, and a candidate thread from the large cache is migrated to the small cache;

(3) The new thread is scheduled on the large cache, and a candidate thread from the large cache is swapped with another candidate thread from the small cache;

(4) The new thread is scheduled on the small cache;

(5) The new thread is scheduled on the small cache and a candidate thread from the smaller cache is migrated to the large cache;

(6) The new thread is scheduled on the small cache and a candidate thread from the large cache is swapped with another candidate thread from the small cache.

The scheduler then computes MPI_{sum} for each case and picks the best schedule that yields the smallest MPI_{sum}. The best candidate is updated by comparing the difference between the best schedule and the second-best schedule. In other words, to arrive at the second best schedule, which threads need to be migrated correspondingly. Consequently, when a new thread arrives, the ACS scheduler always determines a best mapping to core/cache based on the current best schedule (case 1 and 4) and second-best schedule (case 2, 3, 5 and 6). Meanwhile, the number of potential migrations is also limited (3 migrations at most). To compute the best mapping of a thread, a constant amount of computation is needed, which explains the nature of O(1) complexity of our ACS scheduler.

The computational overhead of our scheduler is shown in (ACS bar) of Figure 7(a). It computes the best schedule at magnitudes faster than the naive scheduler. Since our ACS scheduler will not perform compute and sort for all possible schedules when a thread arrives, it may not always pick the theoretical best schedule. Figure 7(b) depicts its accuracy in generating best schedules. The error rate denotes the ranking of the schedule generated by our O(1) ACS scheduler within all the sorted possible schedules. Results in Figure 7(b) demonstrated that the schedule generated by ACS scheduler is always close to (top 97%) or actually is the best schedule.

A walk-through example: To better understand the functioning of our scheduler, we present a walk-through example in Figure 8. To keep this example simple, we depict a case where only 2 threads (T_1 and T_2) are running in the system at steady state and a new thread (T_3) arrives. With the system at steady state, T_1 is scheduled on small cache and T_2 on large cache (since this minimizes MPI_{sum}). When T_3 arrives, the scheduler computes the six possible cases and chooses the cases where MPI_{sum} would be smallest. In this case, case-1 is the best schedule and hence T_3 is scheduled on large cache with no extra thread migrations. To update the candidate threads on both caches, the scheduler compares the best case with the second-best case (i.e. case-1 with case-5 in this example) and computes the minimum thread migrations required to reach the second-best case from the best-case. In this example, the second best case has T_1, T_2 on large cache and T_3 on smaller cache; best case has T_2, T_3 on large cache and T_1 on smaller cache, thus swapping T_1 and T_3 in case-1 would make it case-5. Hence, candidate threads on large and small caches are T_3 and T_1 respectively.

4 Experimental Platform

We implemented our scheduler in Red Hat Enterprise Linux release 5.2 with Linux 2.6.32 kernel version.

Fast thread migration methodology. In a traditional Linux kernel, thread migration technique is designed specifically for load-balancing purposes (may also be invoked when a thread's CPU affinity is changed). Before a core is about to get idle, a `SoftIRQ` is raised by scheduler to invoke the `load_balancer`. When the `SoftIRQ` is handled, a migration request is initiated. A per-core migration thread is then waken up to handle the migration request. The migration thread is responsible for moving the thread to the destination core's run queue. While this technique is suitable for load-balancing, due to the use of `SoftIRQ` and an extra migration thread, it incurs significant performance overhead when naively applied to asymmetric cache/core domain as such platforms typically targets for performance using migration techniques. The overhead of migration thread based technique cannot be easily amortized on such platforms. To deal with it, our ACS scheduler deploys a

Thread T1 and T2's MPI statistics at time t0

Thread Name	MPI on Large $	MPI on Small $
T1	0.40	0.50
T2	0.45	0.90

System state at time t0

Thread on Large $	Thread on Small $	Candidate on Large $	Candidate on Small $	MPI on Large $	MPI on Small $	Sum of MPIs
T2	T1	T2	T1	0.45	0.50	0.95

Thread T3's MPI statistics when it arrives at time t1

Thread Name	MPI on Large $	MPI on Small $
T3	0.60	0.75

Scheduler computation at time t1

Case	MPIL	MPIS	SUM
1	1.05	0.50	1.55
2	0.60	1.40	2.00
3	1.00	0.90	1.90
4	0.45	1.25	1.70
5	0.85	0.75	1.60
6	0.40	1.65	2.05

System state after time t1

Thread on Large $	Thread on Small $	Candidate on Large $	Candidate on Small $	MPI on Large $	MPI on Small $	Sum of MPIs
T2,T3	T1	T3	T1	1.05	0.50	1.55

Figure 8. A Walk-through Example

flexible and a fast-track thread migration technique. We modified the `scheduler_tick` function (invoked during each timer tick) such that it reads out a thread's performance statistics during the thread's training phase, and determines if a migration is needed. When a migration is needed, our new scheduler function will context switch out the current thread, pick up the next runnable candidate and migrate the current thread to a proper destination core on the fast-path in the scheduler code.

Table 1. IPC of benchmarks with a 4MB and 512KB LLC; S=cache sensitive, I=cache insensitive.

Name	4MB	512KB	Type
astar(ast)	0.752604	0.438637	S
bzip2(bzp)	1.335801	0.793942	S
gcc(gcc)	0.606141	0.407053	S
gobmk(gbk)	0.907877	0.787272	S
h264ref(264)	1.297192	1.086523	S
mcf(mcf)	0.216344	0.125643	S
omnetpp(omt)	0.354652	0.249655	S
perlbench(pbh)	1.174421	0.708547	S
soplex(sop)	0.632873	0.380203	S
sphix(spx)	0.919752	0.567493	S
hmmer(hmr)	1.048936	1.03221	I
lbm(lbm)	0.188145	0.16843	I
libquantum(lbq)	0.872071	0.873831	I
milc(mlc)	0.378481	0.37839	I
namd(nmd)	1.557044	1.500439	I
povray(pry)	1.071142	0.983982	I
sjeng(sjg)	1.021994	0.935973	I

Real-Machine Platform. Our target platform is dual-socketed Intel Core 2 processors [1]. Each socket contains 4 cores, with each core operating at 3GHz frequency, having a private 32KB L1 cache and an L2 cache shared by 2 cores. The L2 cache sizes are 4MB and 512KB respectively in each socket. The corresponding architectural schematic is shown in Figure 3.

To avoid interference to our experiment results, we run our experiments on socket-1 and leave socket-0 up for handling interrupts and running OS daemons. Since there is no shadow tag hardware available in the system, we profile the MPIs of each application on the 4MB and 512KB caches offline, and pass such profiles into our ACS scheduler at the end of each training phase of a thread. The profiling is performed for 100ms at each 10s interval such that the access to shadow tags and performance counter would take on average 1% of thread execution time. To simulate the effect of shadow tag with set sampling, we apply an average of 2% random errors to the profiled data.

Application Benchmarks. We used all 17 C/C++ applications from the SPEC CPU 2006 [30] suite for our experiments. Table 1 shows the measurement based IPC of the benchmarks on our experimental platform (we use acronym for each benchmark in order to clearly represent our result). Based on the IPC changes from a 4MB to 512KB LLC, we categorized the applications into sensitive (*S*) and insensitive/streaming (*I*) applications. For our experiments with 2 and 4 thread cases, we construct workloads that have various combinations of *S* and *I* type benchmarks. For 2 thread case, we create a total amount of 70 *S0I2*, *S1I1* and *S2I0* workloads, where the numbers in *SxIy* denote the number of benchmarks from each type; for 4 thread case, we create a total amount of 30 *S0I4*, *S1I3*, *S2I2*, *S3I1* and *S4I0* workloads. We start all benchmarks in each workload simultaneously and measure our results from the beginning

until the fastest benchmark completes.

Simulation Environment. To compare the performance and power effectiveness of our asymmetric cache architecture with regular symmetric cache architectures, we use Simics full system simulator [25] with its default cache model (*g_cache_ooo*) and processor model (*micro_arch_x86*) to build an asymmetric cache platform with a 4MB and a 512KB cache (Figure 3), as well as a symmetric cache platform that has an identical overall cache size (i.e. (4MB+512KB)/2=2.25MB for each cache). We evaluated single thread (all 17 benchmarks) and 2-thread cases (we create 21 *S0I2, S1I1* and *S2I0* combinations), respectively. Since not all cores are fully utilized in this comparison, we power down one idle core on each cache and chop off half of the cache sizes. Hence the experimented cache sizes on the asymmetric cache are 2MB and 256KB respective, and caches on the symmetric caches are 1.125MB. For this set of experiments, we launch all benchmarks in a workload simultaneously and measure for 5 billion instructions.

5 Evaluation Results

In this section, we present the evaluation results of the proposed APE architecture and ACS scheduler. We first show the performance and power effectiveness of asymmetric cache architectures by comparing against an size-equivalent traditional symmetric cache architecture, and then show the performance effectiveness of our ACS scheduler by comparing against existing Linux default scheduler.

5.1 Effectiveness of Asymmetric Cache Architecture

To compare the effectiveness of asymmetric cache architecture against symmetric cache architecture, the major metrics we focus on are weighted speedup for performance, energy delay product (EDP) and energy delay2 product (ED^2P) for both performance and power considerations. We define $WS = \sum_{i=1}^{N} \frac{IPC_i}{i\_alone\_512KB}$ to be the summation of threads' speedup obtained when running on the given architecture compared to the case where it is running alone on a 512KB cache. In a single thread case, the OS scheduler decides on which cache/core to schedule the application and switch the other one to a deep power saving state. Compared to symmetric caches, better performance is naturally obtained on the asymmetric cache by scheduling a thread on the large cache. Hence we do not present our data on single thread performance comparison. Rather, we present their obtained EDP and ED^2P respectively.

Figure 9 shows the EDP of individual benchmarks when scheduled on a symmetric (*Symmetric* bars) and an asymmetric cache (*Asymmetric* bars). On an asymmetric cache,

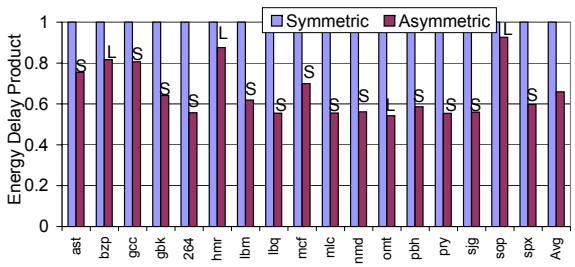

Figure 9. Energy Delay Product (EDP) of benchmarks with symmetric and asymmetric cache organization.

a wise OS scheduler can decide where to schedule an application to achieve better EDP, either on the small cache or on the large cache, while there is no such flexibility for EDP optimization on a size-equivalent symmetric cache. To highlight this artifact, in Figure 9, we show that the EDPs of all applications are better on an asymmetric cache than on a symmetric cache, with an average of 34% EDP reduction in asymmetric cache. We plot on which cache the better EDP is obtained using the *L/S* annotation in Figure 9. We find that, 13 out of the 17 benchmarks achieve better EDP on a small cache, while the 4 rest benchmarks achieve better EDP on the large cache. This indicates the importance of cache selection flexibility provided by the asymmetric cache. Note that for the case of asymmetric cache, with the given schedule plotted in Figure 9, we measured an average of 3.6% increase in L2 cache miss numbers, which incurs a negligible increase in DRAM power [3].

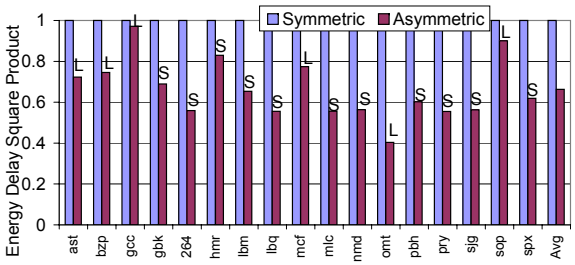

Figure 10. Energy Delay^2 Product (ED^2P) of benchmarks with symmetric and asymmetric cache organization.

To give more weight to application performance, we analyzed the ED^2P of the applications. Figure 10 shows the results of this analysis. We find that asymmetric cache also achieves much better ED^2P than symmetric cache in general, with an average ED^2P reduction of 33% than symmetric cache. 11 out of the benchmarks arrive at better ED^2P by executing on the small cache, while the rest 6 benchmarks have better ED^2P by running on the large cache. This in turn again demonstrates the effectiveness of asymmetric cache,

that is, it provides a fertile ground for the OS scheduler to optimize the system performance or EDP. In contrast, traditional symmetric cache design lacks such flexibility.

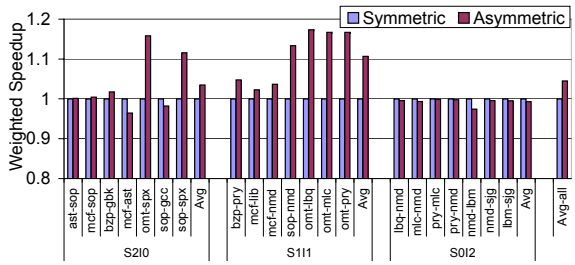

Figure 11. Weighted speedup of 2-application workloads on symmetric/asymmetric cache.

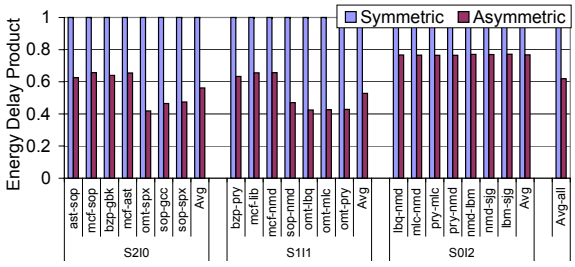

Figure 12. EDP of 2-application workloads on symmetric/asymmetric cache.

Figure 11 shows the weighted speedup of workloads that consist of 2 benchmarks on the symmetric cache case and the asymmetric cache case, while Figure 12 depicts the EDP for the same set of workloads on both architectures. As we stated earlier, the weighted speedups are calculated as the cumulative speedups of applications with a baseline case of 512KB cache. The results obtained in the asymmetric cache assume an oracle OS scheduler that is always able to pick the best schedule.

As we can see from Figure 11, 12 out of the 21 representative workloads achieve significant speedups, with an average speedup of 8.8%; for the rest 9 workloads a negligible (1.1% on average) slowdown is incurred. For the *S2I0* workloads, both benchmarks are performance-sensitive to the given cache capacity. With the asymmetric cache architecture, one of the benchmarks would be able to achieve performance improvement since it can be scheduled on the large cache, while the other will be slowed down since it is scheduled on the small cache. Because the OS scheduler is able to manage the thread to cache mapping wisely based on the degree of cache sensitivity, for *S2I0* workloads, an average of 3.5% speedup is obtained over symmetric cache. Moreover, such speedup is obtained with an optimized EDP (an average of 44% EDP reduction as shown in Figure 12). For the *S1I1* workloads, one benchmark is sensitive to the cache while the other is not. Hence a wise OS scheduler would always pick the cache-sensitive benchmark to run on the large cache and leave the insensitive one on the small cache. Thus, all 7 workloads in this group can benefit, with an average of 10.7% speedup against symmetric cache. Again, the speedup is obtained at an average EDP reduction of 47%. Finally, for the *S0I2* workloads, both benchmarks are cache insensitive. Scheduling either benchmark on either cache will have a noticeable performance impact. As a result, for these workloads, an average of 0.7% slowdown is incurred. However, EDP is also reduced by 23% for *S0I2* benchmarks as compared to symmetric cache case.

To conclude the comparison between symmetric cache and asymmetric cache, asymmetric caches inherently offer the flexibility for OS schedulers to optimize thread performance and power, while traditional symmetric caches lack such flexibility.

5.2 Effectiveness of Asymmetric Cache Scheduler (ACS)

To evaluate our ACS scheduler on the asymmetric cache platform, we use the weighted speedup metric to compare its performance with a traditional Linux default scheduler. Again the speedups of threads are normalized to the case when it runs alone on a 512KB cache. We first present results of workloads that consist of 2 benchmarks (2T), and then analyze results of workloads that consist of 4 benchmarks that run together. Note that for 2T case, each thread uses the cache exclusively as in a private cache case, while in the 4T case, 2 threads contend for a single cache space.

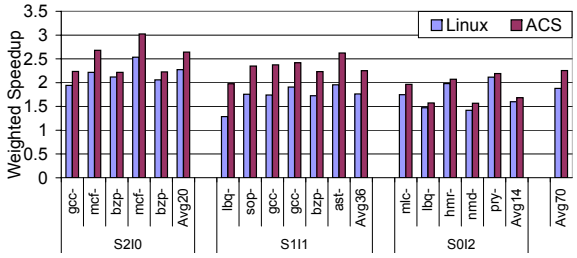

Figure 13. Weighted speedup of applications with ACS (2T without cache contention).

Figure 13 plots the weighted speedup of various workloads that consist of 2 benchmarks running on the asymmetric cache architecture, using either the Linux default scheduler (*Linux* bars), or our ACS scheduler (*ACS* bars). Note that we experimented with 20 *S2I0* workloads, 36 *S1I1* workloads and 14 *S0I2* workloads respectively. Due to space constrains, we show results of a few workloads in Figure 13 and depicts the average for each group. We run our experiments for ten times and plot the average in this figure.

Figure 13 shows that our ACS scheduler outperforms default Linux scheduler in general for all workloads evaluated, with an average speedup of 20%. In fact, with ACS scheduler we observed performance improvement for all 70 workloads evaluated. The reason is that, given such an architecture, it is important for the OS scheduler to have knowledge on the underlying architecture, and information of each thread like how they will behave on each cache. The Linux scheduler lacks such capabilities and makes thread schedules blindly. In contrast, our ACS scheduler takes into account thread performance statistics at runtime and makes a best schedule based on the statistics. The results shown in Figure 13 also demonstrates the effectiveness of minimizing overall MPI of threads in helping find out the best performing schedule.

For the *S2I0*, *S1I1* and *S0I2* groups of workloads, the average speedups are 16%, 28% and 5% respectively. The higher speedup obtained by *S1I1* group of workloads indicates that it becomes increasingly important for the OS scheduler to make a right decision when the workloads have distinct performance characteristics. Note that the oracle scheduler that uses brute-force approach to attempt all possible schedules (assuming with zero overheads) obtains an average speedup of 23% (Figure 5), while our ACS obtained an average speedup of 20%. This indicates that ACS scheduler may not always pick the best performing schedule (the error rate is less than 3% as shown in Figure 7(b)). However, in reality an oracle scheduler cannot be implemented at no cost. The amount of speedup it obtained will not amortize the attempt overhead it made.

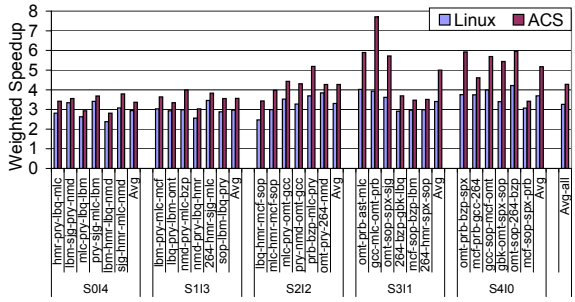

Figure 14. Weighted speedup of applications with ACS (4T with cache contention).

Figure 14 illustrates the weighted speedup of various workloads running on asymmetric cache platform with the Linux scheduler (Linux) or the ACS scheduler (ACS). In this regard, we constructed 6 workloads for each of the 5 evaluated groups. In this set of experiments, 2 benchmarks are scheduled on 2 cores that share the same cache. Therefore cache contention takes place. As depicted in Figure 14, our ACS scheduler again outperforms the Linux default scheduler in general, with an average speedup of 31%. Combining the results from Figure 13 and 14 together, we observe on average 23% performance improvement for 2T and 4T cases on the asymmetric cache architecture. This again demonstrates the effectiveness of our ACS scheduler. It also demonstrates the effectiveness of our proposed MPI under-contention model in helping find out the best performing schedule. We also exhaustively searched for the best thread schedule for each workload, and found that using the model we proposed, ACS scheduler is able to find the best or second-best schedule in 97% cases. Another observation made from Figure 14 is that, a much higher performance speedup (31% vs. 20%) is obtained in 4T case than the 2T case (Figure 13). This indicates that with the increase in the number of threads, as well as the presence of cache contention, it becomes more important for the OS scheduler to be aware of the underlying architecture and thread performance characteristics to make a best performing schedule.

6 Related Work

To our best knowledge, no prior work has proposed an OS scheduler for an asymmetric cache CMP. In this section, we discuss the most closely related prior work in the areas of scheduling in heterogeneous cores and in cache management, as well as proposals on asymmetric caches.

Scheduling in Heterogeneous Cores. There are many prior proposals that posit to leverage the core asymmetry for performance and power improvement through scheduling. Li et al. [21] proposed an OS support for heterogeneous architectures in which cores have asymmetric performance and overlapping, but non-identical instruction sets. Similarly, Saez et al. [29] proposed an asymmetric-core scheduler design. The scheduler identifies ILP and TLP threads and schedules them on fast cores and slow cores, respectively. Kumar et al. [18, 19] proposed to use exhaustive training approach for determining thread assignments on asymmetric multicore processors. Our proposal differs from these studies in that we leverage the benefits of an asymmetric cache through OS support. We propose to use run-time information of workloads using hardware support and then schedule the threads on cores so as to maximize performance.

Cache Management. Researchers [13, 14, 24, 28, 33] have observed that cache, prefetching, memory controller, and memory bus contention account for a high percentage of performance degradation that threads running on multicore processors experience. Numerous other proposals exist in literature [5, 11, 12, 23, 22, 27] that propose cache management schemes to improve performance by reducing shared cache contention, minimizing miss-rates, or to improve energy of the system as a whole. As many of such schemes require an offline profiling being available, our scheme proposes a novel empirical model to predict the cache contention effect and improve performance

and energy in an asymmetric cache CMP. There have also been prior proposals [6, 7, 8, 17] on leveraging OS scheduler to mitigate cache contention impact. These proposals observed that by evenly distributing workloads that exhibit high and low cache miss rates to different caches, the amount of cache contention can be reduced. The premise is that the degree of miss rate of an application will not change much when co-scheduled with other applications. However, this is not always true across a broader range of workloads. Outlier applications (such as streaming workloads) may significantly trash the cache and change the degree of miss rate of co-scheduled workloads.

Asymmetric and Heterogeneous Caches. Apparao et al. [4] analyzed the implications of asymmetric caches for server consolidated environments. Hu et al. [10] proposed an asymmetric structure for set associative cache where the size of each way can be different. Our proposal, however, investigate the implications of cache size asymmetry from performance and energy perspective leveraging OS scheduler support.

7 Conclusions

In this paper, we made a case for asymmetric cache CMP design and then propose a novel scheduling scheme that is asymmetric cache aware. With a detailed measurement-based evaluation on a Xeon 5160 chip with asymmetric last-level caches, we show that our scheduler based contention-mitigating technique can achieve throughput that is within 3% of an asymmetric-cache aware oracle scheduler. We also conclude that the highest impact of contention-aware scheduling technique is not in improving performance of a workload as a whole but in minimizing the contention of the shared last-level asymmetric cache. We believe our design is simple and complements asymmetric platforms comprising of heterogeneous cores.

References

[1] Intel Xeon 5160. *http://www.intel.com*, 2009.

[2] Linux kernel archives. *http://www.kernel.org/*, 2010.

[3] Micron Technology. *http://www.micron.com*, 2010.

[4] P. Apparao et al. Implications of Cache Asymmetry on Server Consolidation Performance. *in IISWC*, 2008.

[5] D. Chandra et al. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. *in the Proc. of International Symposium on High Performance Computer Architecture (HPCA)*, 2005.

[6] A. Fedorova et al. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. *in the Proc. of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[7] A. Fedorova et al. Managing Contention for Shared Resources on Multicore Processors. *Communications of the ACM*, 2008.

[8] D. Guo et al. Performance Characterization and Cache-Aware Core Scheduling in a Virtualized Multi-Core Server under 10GbE. *in IISWC'09*, 2009.

[9] A. Hartstein et al. On the Nature of Cache Miss Behavior: Is It $\sqrt{(2)}$? *In The Journal of Instruction-Level Parallelism*, 2008.

[10] Z. Hu et al. Improving Power Efficiency with an Asymmetric Set-Associative Cache. *Workshop on Memory Performance Issues*, 2001.

[11] J. Jeong et al. Cost-sensitive cache replacement algorithms. *in Proc. of the 9th International Symposium on High Performance Computer Architecture (HPCA)*, 2003.

[12] X. Jiang et al. Architecture Support for Improving Bulk Memory Copying and Initialization Performance. *In the Proc. of the 18th Intl Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[13] X. Jiang et al. CHOP: Adaptive Filter-based DRAM Caching for CMP Server Platforms. *in the Proc. of the 16th International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[14] X. Jiang et al. CHOP: Integrating DRAM Caches for CMP Server Platforms. *in IEEE Micro Top Picks 2010*, 2010.

[15] R. Kessler et al. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 1994.

[16] S. Kim et al. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. *in PACT*, 2004.

[17] R. Knauerhase et al. Using OS Observations to Improve Performance in Multi-core Systems. *in IEEE Micro*, 2008.

[18] R. Kumar et al. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. *in the 36th MICRO,*, 2003.

[19] R. Kumar et al. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. *in the Proc. of the 31st International Symposium on Computer Architecture (ISCA)*, 2004.

[20] T. Li et al. Efficient and Scalable Multiprocessor Fair Scheduling Using DistributedWeighted Round-Robin. *in PPoPP*, 2009.

[21] T. Li et al. Operating System Support for Overlapping-ISA Heterogeneous Multi-core Architectures. *in the Proc. of the 16th International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[22] G. Liao et al. A New IP Lookup Cache for High Performance IP Routers. *in 47th DAC*, 2010.

[23] G. Liao et al. A New TCB Cache to Efficiently Manage TCP Sessions for Web Servers. *in the 6th ANCS*, 2010.

[24] F. Liu et al. Understanding How Off-chip Memory Bandwidth Partitioning in Chip-Multiprocessors Affects System Performance. *in Proc. of the 16th International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[25] P. Magnusson et al. Simics: A Full System Simulation Platform. *Computer, 35(2):50.58*, 2002.

[26] N. Muralimanohar et al. Optimizing NUCA organizations and wiring Alternatives for large caches with CACTI 6.0. *in MICRO*, 2007.

[27] M. Qureshi et al. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. *in the 39th MICRO*, 2006.

[28] B. Rogers et al. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. *In Proc. of Intl. Symposium on Computer Architecture (ISCA)*, 2009.

[29] J. Saez et al. A Comprehensive Scheduler for Asymmetric Multicore Systems. *in EuroSys*, 2010.

[30] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks. *http://www.spec.org/cpu2006/*, 2006.

[31] C. Wilkerson et al. Trading Off Cache Capacity for Reliability to Enable Low Voltage Operation. *in Proc. of International Symposium on Computer Architecture (ISCA)*, 2008.

[32] L. Zhao et al. CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms. *in PACT*, 2007.

[33] S. Zhuravlev et al. Addressing Shared Resource Contention in Multicore Processors via Scheduling. *in the 15th ASPLOS*, 2010.