Symbiotic Scheduling for Shared Caches in Multi-Core Systems Using Memory Footprint Signature

Mrinmoy Ghosh
mrinmoy.ghosh@arm.com
Corporate R&D
ARM, Inc., Austin, TX 78735

Ripal Nathuji
ripal.nathuji@microsoft.com
Microsoft Research
Redmond, WA 98052

Min Lee
min.lee@gatech.edu
College of Computing
Georgia Tech, GA 30332

Karsten Schwan
schwan@cc.gatech.edu
College of Computing
Georgia Tech, Atlanta, GA 30332

Hsien-Hsin S. Lee
leehs@gatech.edu
School of Electrical & Computer Engineering
Georgia Tech, Atlanta, GA 30332

ABSTRACT

As the trend of more cores sharing common resources on a single die and more systems crammed into enterprise computing space continue, optimizing the economies of scale for a given compute capacity is becoming more critical. One major challenge in performance scalability is the growing L2 cache contention caused by multiple contexts running on a multi-core processor either natively or under a virtual machine environment. Currently, an OS, at best, relies on history-based affinity information to dispatch a process or thread onto a particular processor core. Unfortunately, this simple method can easily lead to destructive performance effect due to conflicts in common resources, thereby slowing down all processes.

To ameliorate the allocation/management policy of a shared cache on a multi-core, in this paper, we propose Bloom filter signatures, a low-complexity architectural support to allow an OS or a Virtual Machine Monitor to infer cache footprint characteristics and interference of applications, and then perform job scheduling based on symbiosis. Our scheme integrates hardware-level counting Bloom filters in caches to efficiently summarize cache usage behavior on a per-core, per-process or per-VM basis. We then proposed and studied three resource allocation algorithms to determine the optimal process-to-core mapping to minimize interference in the L2. We executed applications using allocation generated by our new process-to-core mapping algorithms on an Intel Core 2 Duo machine and showed an averaged 22% (up to 54%) improvement when applications run natively, and an averaged 9.5% improvement (up to 26%) when running inside VMs.

Keywords

Scheduling, Symbiosis, Shared Caches, Multi-Core, Bloom Filter, Virtualization, Fairness

1. INTRODUCTION

The ever increasing demands for performance and manageability in modern enterprise computing systems has directly affected innovation in both computer architecture and system design. On one end, modern computing platforms provide multi-core processors to improve performance by exploiting thread-level parallelism. On the other hand, it has become a requirement for enterprise systems to provide flexible and efficient use of resources via software management. Industry has responded to this need with the integrated hardware/software solutions (e.g., Xen or VMware) for virtualization [2, 23], which allows applications to obtain benefits such as fault and performance isolation [2, 20].

To optimize performance, prior work considered cache interference of different processes due to affinity effect [33] and thrashing in shared cache machines [10]. These studies showed significant performance implications when execution instances are scheduled onto processors oblivious of cache contention effects. In this paper, we quantify such destructive caching effects on both native multi-core machine and the virtualized execution where multiple virtual CPUs (vcpus) are mapped to physical cores sharing a cache. We highlight two issues preventing an optimized process-to-core allocation decision. First, there is the issue of determining cache usage characteristics of workloads. As our experiments showed, using online profiling mechanisms, *e.g.*, event-based performance counters do not always reflect the cache footprint of workloads. In response, we propose per-core counting Bloom filters to enable online profiling of cache usage behavior in the hardware. The second issue is to determine the interactions between workloads based on our Bloom filter profile. We employ simple hardware support to comparing *Bloom filter signatures* of workloads. This information is then provided to software-based allocation algorithms. These policies facilitate the efficient resource management among applications to maximize system performance. The algorithms can be implemented in the OS scheduler or as a user-level monitoring process. In the case of virtual machines, these algorithms execute in the control domain, Domain zero (Dom0). For VMs, these policies determine the virtual-to-physical resource mappings to improve performance while providing fairness across workloads. Our evaluation— by gathering Bloom filter signature, doing resource allocation algorithms on Simics, and using the resource allocation results in real execution, showed great performance benefits.

The remainder of this paper is organized as follows. Section 2 motivates hardware support for multi-core resource allocation. We then outline our system design in Section 3. Our evaluation methodology is described in Section 4. Section 5 analyzes our results. We discuss related work in Section 6 and Section 7 concludes.

2. MOTIVATION

2.1 Resource Allocation in Multi-Core

Workloads usually compete for common resources in a multi-core processor. As a result, the job of the OS dispatching workloads to physical cores is very critical to achieve the best possible performance. In addition, a future OS also needs to consider power management, assess formation of hot-spots, avert the risk of thermal runaway, and guarantee QoS for given workloads. This problem is particularly interesting in the case of providing support for virtualization for scalable enterprise solutions.

In virtualized systems, workloads execute using the set of virtual resources defined to make up an underlying *virtual platform*. It is the job of the virtualization layer, then, to map these virtual resources to physical resources at runtime. Management policies can employ

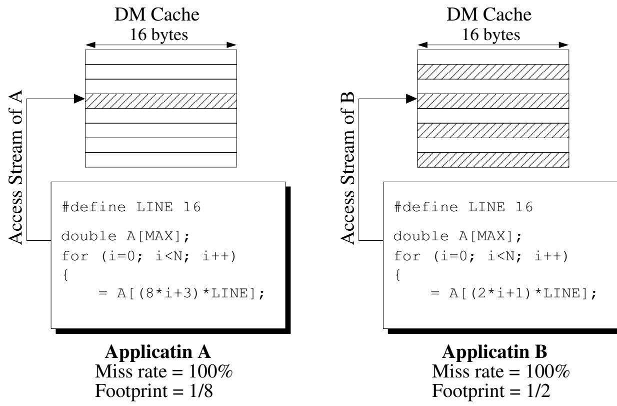


Figure 1: Different Cache Footprints with the Same Miss Rate

intelligent decision-making schemes to perform allocation that improves systems. In this paper, we particularly consider improving the allocation decisions when mapping physical resources like a shared cache to virtual CPUs that run guest VMs. An important goal of resource allocation is to determine allocation that maximizes performance by minimizing negative caching effects described next.

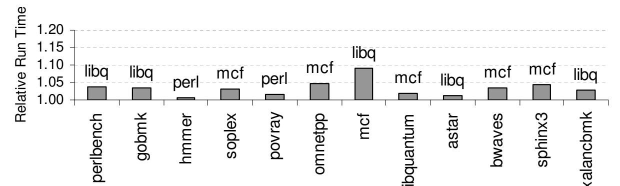
2.2 Cache Working Set and Footprint

A large body of work addressed cache affinity scheduling in shared memory multiprocessors. Devakonda *et al.* [6] showed the effect of cache affinity scheduling and stated that affinity scheduling works well only for certain applications. This observation is substantiated by Salehi [30]. Also, the authors in [38] stated that cache affinity scheduling is not very effective over simple scheduling policies.

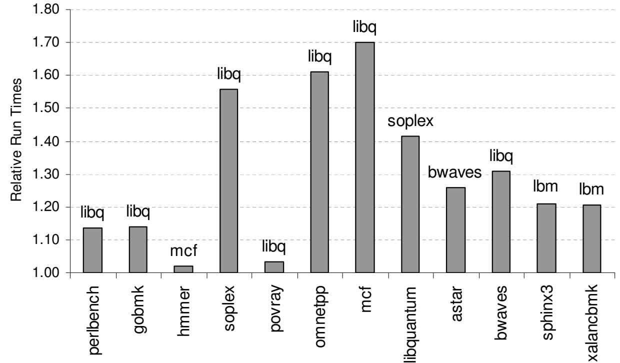
One conclusion from these prior work is that when multiple applications running on different processors sharing an L2, the co-scheduling of incompatible applications may affect the overall performance due to destructive effect on each other's cache state. In response, there was research to aid scheduling decisions using cache miss rates [9]. However, predicting incompatibility of active applications cannot be accurately done using performance metering approaches such as counting the number of cache misses. This is because cache miss data do not provide sufficient information about the coverage, distribution, or cache footprint of an application. We illustrate this with a simple example in Figure 1, which shows two conjured access patterns in an 8-set direct mapped cache. Application A accessing different cache lines that fall into the same cache set has a 100% miss rate. However, the *footprint* of A is just one-eighth of the entire cache. In contrast, application B also exhibits a strided access pattern with 100% miss rate. However, due to its smaller stride, application B will occupy a much larger footprint, *i.e.*, half of the cache, contributing a great detrimental effect to other applications sharing the same cache.

To further demonstrate the fact that miss rates do not signify the cache working set size, we use the full-system simulator Simics and explore the correlation between the working set size and event-based performance counters monitoring like cache misses, TLB misses, and page faults. The workload used for this experiment is aim9_disk from the AIM IX Independent Resource Benchmark [1]. Figure 2(a) shows the L2 cache working set calculated at every tick (4ms). The cache working set size is defined as the number of unique cache lines touched in the 4 ms interval. Figure 2(b) shows the measured the number of L2 misses over the same time interval of every 4ms. The figure clearly demonstrates that the benchmark exhibits four phases of execution. In contrast, only two periodic phases are visible in the working set in Figure 2(a). Clearly, there is no direct correlation between cache working set and cache misses (based on performance counters) for this benchmark. Other metrics such as TLB misses or page faults have similar problems.

The finding from the above analysis is that performance counter based approaches from prior studies [11] do not accurately characterize cache working set and are therefore, not a suitable basis for making resource allocation decisions. Determination of the cache working set of an application is essential to determine its effect on other applications sharing the same cache. In the following subsection, we quantify the effects of applications sharing in memory.



(a) On a P4 Xeon System (Run on a Single P4)



(b) On an Intel Core 2 Duo System

Figure 3: Worst-case Performance Disturbance

2.3 Quantifying Application Incompatibility

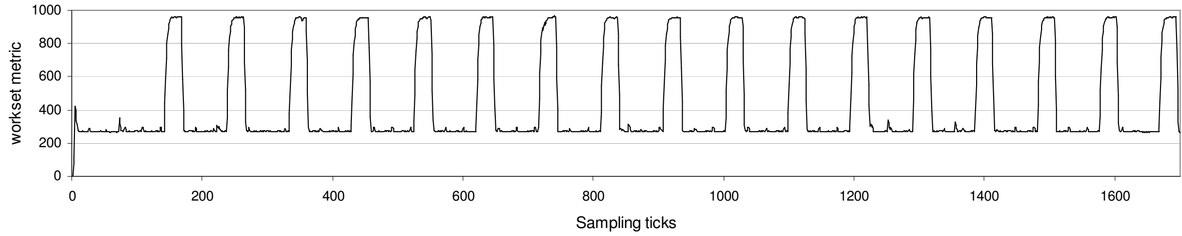
We used 12 SPEC2006 benchmark programs to quantify the effects of application incompatibility when sharing a cache. They were chosen to exhibit a good mix of compute-intensive and memory-intensive behavior for demonstrating the mutual effect to their respective performance. All possible pairs of the 12 SPEC2006 benchmark programs were run on two different real systems to be discussed as follows. For the graphs to be presented, each bar represents the worst-case "user time" of a benchmark when running with another benchmark relative to the "user time" if the benchmark was executed standalone. All programs were run to completion.

2.3.1 P4 Xeon SMP System

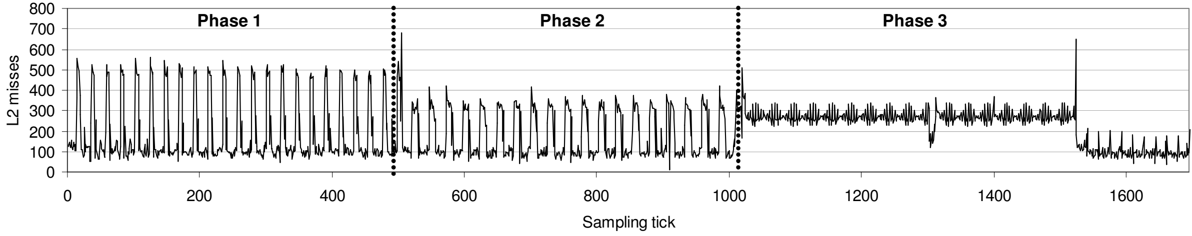
We performed our first experiments on an Intel Xeon SMP with two P4 3.0GHz processors, each containing its own private 2MB 8-way L2. We confined the paired processes to run on one single processor to quantify the negative effects due to their interference in the L2. The results are illustrated in Figure 3(a). The reported runtimes are averaged over three independent runs for each benchmark pair. We can see that the maximum performance degradation is less than 10%. When processes are constrained to run on the same processor, the primary cause of performance degradation is cache warm-up due to context switches. Given the low frequency of context switch occurrences, the performance is not significantly affected.

2.3.2 Shared Cache in Intel Dual-Core

For a shared cache architecture, we ran our experiments on an 2.34 GHz Intel Core 2 Duo system with a 4MB 16-way *shared L2*. We scheduled two processes on different cores sharing the same L2. As shown in Figure 3(b), even though the L2 is twice larger than the P4, the relative performance degradation is much more severe. We found that the maximum degradation is 67% for the mcf paired with libquantum. These results show that the performance is more sensitive to process allocation in a multi-core sharing the cache, with a potential performance improvement of as much as 67%. This actual

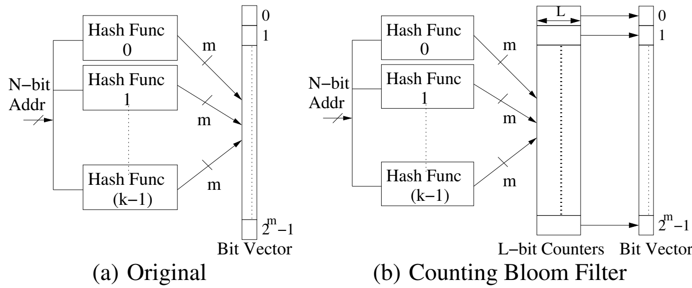


(a) Working Set Size (# of Unique Cache Lines Accessed)



(b) L2 Misses

Figure 2: Correlation between Working Set Size and L2 Misses (aim9_disk)



(a) Original (b) Counting Bloom Filter

Figure 4: Bloom Filters

measurement motivated the need for a better core allocation policy for applications based on their cache footprints. Ideally, each application's cache footprint should be recorded and compared to the footprint of others for reaching an optimal process-to-core mapping.

However, maintaining cache footprints of individual processes over time is prohibitively expensive in terms of both space and computational complexity. Therefore, to better assess the dynamic cache resource utilization of an application, new hardware structures will be needed to efficiently maintain a signature of cache accesses for that particular application. Also required is a simple method for inspecting different signatures to determine their compatibility. We expect that an OS that uses a metric of determining compatibility of applications for co-scheduling and resource allocation will achieve significant performance improvement. Toward this, we will describe a novel infrastructure capable of monitoring cache resource utilization with a software layer that uses this information to intelligently allocate processes. The architectural extension involves the use of a modified Counting Bloom Filter (CBF), a low-cost data structure known for its high efficiency in maintaining signatures of large data sets.

2.4 Counting Bloom Filters

A Bloom filter provides a low-cost structure to efficiently test if an element is present in the set. Figure 4(a) shows a generic Bloom filter in which a given N-bit address is hashed into k hash values using k different hash functions. The output of each hash function is an m-bit index value that indexes the Bloom filter's bitvector of 2^m elements. Here, m is much smaller than N. Each element of the Bloom filter bitvector contains only one bit that can be set. Initially, the Bloom filter bit vector is cleared to zero. Whenever an N-bit address is observed, it is hashed to the bitvector and the corresponding indexed bit values are set to one.

When a query is to be made whether a given N-bit address has been observed before, the N-bit address is hashed using the same hash functions and the bits are read from the locations indexed by the m-bit hash values. If at least one of the bit values is 0, this means that this address has definitely not been seen before. This is called a *true miss*. If all of the bit values are 1, then the address may have been observed but the filter cannot guarantee it. In the case when an address was never observed but the filter indicates 1, it is a *false hit*. As the number of hash functions increases, the Bloom filter bitvector will be polluted much faster. On the other hand, the probability of finding a zero on a query also increases if more hash functions are used.

The major drawback of the original Bloom filter is that the filter can be polluted rapidly and filled up with all 1's as it does not have deletion capability. To address this shortcoming, the Counting Bloom Filter (CBF) [8] was proposed to allow deleting entries from the filter. As Figure 4(b) shows, the CBF reduces the number of false hits by introducing counters. In the CBF, when a new address is entered to the Bloom filter, each m-bit hash index addresses to a specific counter in an L-bit counter array.[1] Then, the counter is incremented by one. Similarly, when a new address is observed for deletion from the Bloom filter, each m-bit hash index addresses to a counter, and then the counter is decremented by one. If more than one hash index addresses to the same location for a given address, the counter is incremented or decremented only once. If the counter is zero, it is a true miss. Otherwise, the outcome is inconclusive.

From the description of the CBF, we can see that it is a simple, low overhead structure that can keep a signature of addresses present in a cache. This enables the hardware to keep track of applications and "Bloom Filter" their signatures for the cache. The signatures can be used for two purposes. Firstly, they provide information about the footprint of an application in the cache. In Figure 5, we show an example using the same aim9_disk benchmark in Figure 2(a). Secondly, they also provide the extent of interference between an application and other applications. This information can be efficiently used by the OS to guide resource allocation. The next section details our mechanism to support the OS in making resource allocation decisions.

We demonstrate that counting Bloom filters effectively monitors

[1] L must be wide enough to prevent saturation.

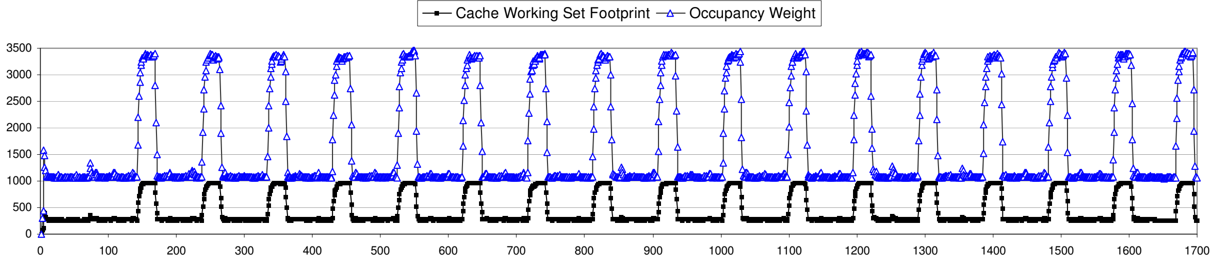


Figure 5: Bloom Filter's Occupancy Weight versus Cache Working Set Footprint (aim9_disk)

the cache footprint in Figure 5 using the same benchmark in Figure 2(a). We define *occupancy weight* to be the number of ones in the bit vector of a counting Bloom filter. As can easily be seen, the occupancy weight follows the cache footprint size more closely.

3. SYSTEM DESIGN

3.1 Multi-Core with Bloom Filter Signature

This section describes the CBF-based infrastructure that enables the OS to efficiently schedule applications in a shared-L2 multi-core. Figure 6(a) illustrates a quad-core processor sharing an L2.

First, we modify the CBF structure explained in Section 2.4 by splitting it into one counter array and multiple bitvector arrays to enable application-level monitoring of cache footprint. In essence, we de-associate the Bloom filter bitvector from its counters and associate one bitvector with each core. We call this bitvector the core filter (CF). The CF is responsible for monitoring the L2 footprint for the core to which it is assigned. The counters are exactly identical to the CBF counters and maintain complete information about the state of the L2. Another simplification to the CBF is that we only use one hash function to hash to our Bloom filters. The reason for this decision is the limited size of the Bloom filters. Using multiple hash functions will saturate the bit-vectors faster. Also, using multiple hash functions incur a larger hardware overhead.

For each L2 miss, the corresponding counter in the counter array indexed by the address hash is incremented. Along with counter's increment, the corresponding index of the CF of the core from which the miss originated is also set to 1. As such, the CF is only responsible for tracking memory requests originated from the core to which it was attached. The line replacement in the L2 causes the corresponding hashed counter to be decremented. If the counter becomes zero after decrementing, all CFs are accessed and their bits corresponding to the decremented counter index are set to zero. The CF has a 1-to-1 mapping with the cache working set of a particular process except barring two exceptions. First, if there are hash collisions in the counter, the CF only counts one entry, an artifact due to aliasing that will underestimate the working set size. Second, when a counter becomes zero, the corresponding bit of all the CFs are reset to zero. This is inaccurate because the line that caused the CF to be one in the first place may have had been replaced long before, but the counter becomes zero only after all the addresses mapped to it are replaced.

Despite the minor inaccuracy, this special arrangement of the CBF enables efficient tracking of cache accesses on a per-core basis. However, the objective of the CBF extensions is to track them on a per-application basis. To enable this, we need an additional bitvector we call the Last Filter (LF) for each core. The LF keeps a snapshot copy of the CF whenever a context switch takes place. The topmost bitvector in Figure 6(b) shows such a snapshot when App1 is being swapped out of Core3 by App2. This state information kept just before the new application or VM accesses the cache helps identify exactly how much of the cache resources have been consumed. Therefore, whenever an application is context-switched out of its core, a *difference* between the CF and LF of that core provides a signature of the cache working set of the application. We call this the Running Bit Vector (RBV) as illustrated in Figure 6(b). As shown, it was calculated by taking the inverse value of $CF \rightarrow LF$ (implication logic). In other words, $RBV = \neg(\neg CF \vee LF)$. Counting the number of ones in the RBV is a metric of the cache's *occupancy weight* for this application. Further, to get an idea of the extent to which the application is interfering with the others, a bitwise XOR of the RBV with all the CFs is performed. We define *symbiosis* to be the sum total of the number of ones in the bit vector obtained by XORing the CF and the RBV. A high symbiosis value indicates low interference. A low value either means higher interference, or that both vectors have a very low occupancy. We show two examples in Figure 6(b) where App1/Core0 generates higher symbiosis(=6) than App1/Core1 (=1). We will show how to use symbiosis to perform core scheduling in Section 3.3.2 and 3.3.3.

The occupancy and symbiosis with other cores are kept with the application as part of its context. The OS can use these metrics to allocate a process to minimize L2 interference.

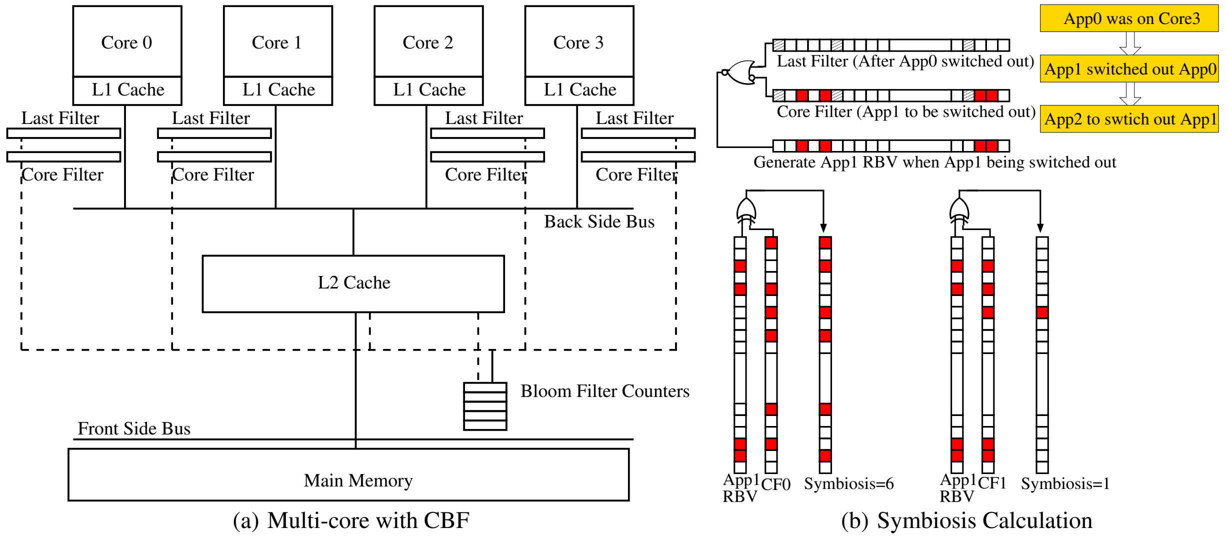
The infrastructure needed to support VMs is exactly the same as the one just described. The only difference is that for the VMs, the RBV will be computed on a per-VM basis instead of a per-application basis. Similarly, the *occupancy weight* and *symbiosis* data structures will be maintained on a VM granularity. Every time the hypervisor decides to do a context switch of a VM, it computes the RBV of the VM from the CF and LF. From the RBV, the hypervisor computes the *occupancy weight* and *symbiosis* of the VM. The hardware infrastructure in this case will interact with the hypervisor instead of the OS.

3.2 Software Support

The software components of our resource allocation system are distributed between the OS and a user-level monitoring process. The OS is responsible for interacting with the hardware described above. In particular, for each application, the OS keeps a simple data structure consisting of $(2 + N)$ entries, where N is the number of physical cores. The first entry of the structure keeps the core ID of the last physical core running the application. The second entry is the *occupancy weight* while the remaining N entries store the *symbiosis* with other cores. Whenever an application is context switched out from a core, the data structure associated with it is updated.

While the OS handles the symbiosis and occupancy via hardware support, the actual resource allocation decisions are made in a monitoring user-level process. An allocation policy running in this monitoring application utilizes the system call interface to periodically query the OS for updated information regarding executed applications. To be detailed later, this information can then be incorporated into different algorithms to obtain an updated allocation decision that is then passed to the OS using existing system call interface. The user-level process is only responsible for setting affinity bits of processes, such that processes are allocated to specific cores. However, since the OS is responsible for handling context switches within the core, processes will not suffer from issues like starvation. Also, if the number of processes is less than the number of cores, our resource allocation algorithms will work exactly like a cache affinity based algorithm.

The software for resource allocation in VMs is very similar. In the case of VMs, our resource allocation system is distributed between the hypervisor and the Xen management or control domain, Dom0.



(a) Multi-core with CBF (b) Symbiosis Calculation

Figure 6: Multi-Core Enhanced with Signature for Symbiotic Scheduling

Virtualization solutions such as Xen utilize this privileged domain to execute management and control components to provide extensibility while maintaining a thin hypervisor [2]. The hypervisor is responsible for interacting with the hardware functionality described above. The per-VM data structure maintained by the hypervisor is exactly the same as the per-application one. Whenever the vcpu of a VM is removed from execution on a core, the data structure associated with the VM is updated.

In the case of VMs, the actual resource allocation decisions are made in Dom0. An allocation policy running in this domain utilizes a hyper-call interface to periodically query the hypervisor for updated information regarding executing VMs. This information can then be incorporated into different algorithms to obtain an updated allocation decision that is then passed along to the hypervisor using existing control interface.

3.3 Resource Allocation Algorithms

The objective of the resource allocation algorithm is to map processes to cores. As explained in Section 2.3.2, the allocation is done in such a way that processes that adversely affect each others' performance should be scheduled to the same core. The rationale is that the processes assigned to the same core will not execute simultaneously, mitigating their resource conflicts that could reduce each others' performance. The quantitative effect in our execution on a real Intel Core Duo machine has been illustrated in Section 2.3.2. We now describe three algorithms developed for this purpose.

3.3.1 Weight Sorting Algorithm

The sorting algorithm uses a simple mechanism to detect the L2 occupancy of each process. The only metric used here is the sum of the bits inside the Running Bit Vector (RBV) which gives a reasonable idea of the cache footprint of a process. Upon every context switch, the RBV is reduced to one weight as described in Section 3.1.

The user-mode algorithm collects the weights of all the processes to be scheduled. Then it sorts the processes according to their weights. If the number of cores is *N* and the number of processes to be allocated is *P*, the group size is $\lceil \frac{P}{N} \rceil$, the number of processes to be scheduled on one core. After sorting the weights, it simply forms groups of processes in their sorted order. For processes in the same group, the same *affinity bits* used by the OS for process scheduling will be assigned. In other words, the OS will schedule the processes of the same group onto the same core. The rationale behind it is that a process having a larger weight is more likely to affect performance of others. When herding them into the same group, they will be scheduled to run on the same core and will not be executed at the same time, minimizing the cache contention.

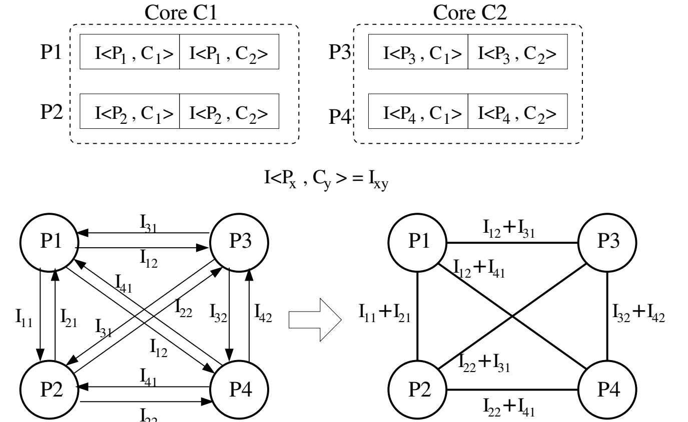
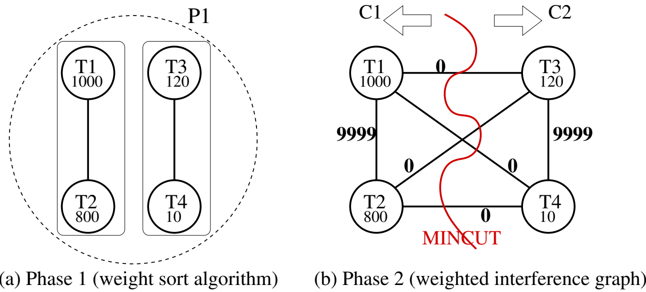


Figure 7: Forming an Interference Graph

3.3.2 Interference Graph Algorithm

The Interference Graph algorithm is explained with the example illustrated in Figure 7. Assume we have a dual-core machine with cores C1 and C2. There are 4 processes (P1 through P4) to be scheduled. We first define an *interference metric* to be the *reciprocal of symbiosis* for constructing an interference graph. We also define the notation $I{<}P_x, C_y{>}$ to represent the degree of Interference of *process x* with *core y*. The top of Figure 7 shows the interference metrics of each process when the algorithm is invoked. For example, considering process P3— its interference metrics $I{<}P_3, C_1{>}$ and $I{<}P_3, C_2{>}$ are obtained by counting the 1's in the bitvector obtained after XORing the Core Filters of core C1 and C2 (self-core) with the Running Bit Vector of P3 when P3 was context switched out of C2. An example was shown previously at the bottom of Figure 6(b)

Using the interference metric, the interference graph is constructed as illustrated in the lower-left corner of the figure. Each process is a node in the graph. The weight assigned to a directed edge connecting two processes is based on its interference metric. The directed edge $P1 \rightarrow P3$ has a weight I_{12}, because it is the interference of process P1 (running on C1) with core C2. We assume that a process has equal interference with all processes of a different core, since it is difficult to know which process was executing in each core when the interference data is taken. This gives us the directed graph shown in the figure. The directed graph is then consolidated into an undirected graph by adding the weights of the two unidirectional edges connecting any two nodes. This consolidated graph gives an approximate idea of



(a) Phase 1 (weight sort algorithm) (b) Phase 2 (weighted interference graph)

Figure 8: Allocation for Multi-threaded Applications

the interference for each process with other processes running in the system.

Thus, the objective of this algorithm in this example is to partition the graph into two groups such that the weights of edges *within* a group is maximized. Maximizing weights of edges within a group (*i.e.*, intra-group interference) ensures that the processes will be allocated to the same core and thus will not run together to affect each others' performance. A converse of this problem is to partition the graph into equal groups such that the weights of edges between the groups (or inter-group interference) are minimized. This problem is better known as the MIN-CUT problem. Although a generic solution to the MIN-CUT problem is NP-hard, several fast approximation algorithms exist to get to a certain percentage of the optimal solution. We use the SDP solver to solve the problem.

This algorithm provides a good solution for the case where there are 2 cores. It can easily be extended to machines with more cores by hierarchically using the MIN-CUT algorithm. For example, if we have four cores, we first divide into two groups using MIN-CUT and then apply MIN-CUT to each group.

3.3.3 Weighted Interference Graph Algorithm

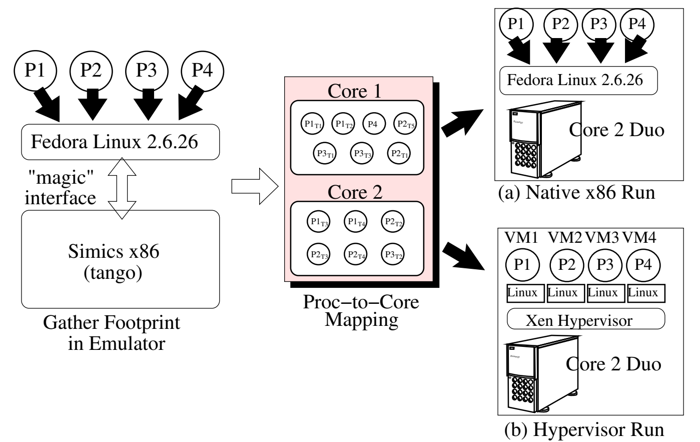
The interference graph algorithm had one impediment. As explained earlier, a low symbiosis (or high interference metric) means either high interference or low occupancy. That is, if the *weight* of the bitvectors whose symbiosis being calculated is small, then the interference metric will come out high, but that does not necessarily imply that the two vectors really interfere heavily. Therefore, we came up with a *weighted interference metric* that incorporates the weight of the vector whose interference is being calculated. Whenever we compute the interference between a node and a core, we simply multiply the result with the weight of the node. Therefore, the weight of the edge connecting nodes P_1 and P_3 will be $W_{P1} \cdot I_{12} + W_{P3} \cdot I_{31}$, where W_{P1} and W_{P3} are the *occupancy weights* of nodes P1 and P3 calculated by counting the number of 1's in their respective RBVs as defined in Section 3.1.

Using this metric will ensure that if a node has a small weight, it will lead to a low interference metric, making the algorithm perform more effectively. We show in Section 5.2 that this is indeed the case.

3.3.4 Resource Allocation for Multi-threaded Apps

The algorithms just described must be adapted properly for multi-threaded applications. To enable this, we must collect occupancy weights and interference metric statistics at the per-thread granularity. For multi-threaded applications, threads of the same application often share data intensely. Therefore, if we compute the interference metrics among threads of the same process, we will obtain a very high interference value. This is misleading, as in this case the threads are actually sharing data rather than contending space against each other. To overcome this shortcoming, we adapt our algorithms to perform resource allocation in two phases. Note that, this adaption for threads will be applied together with the process-level allocation discussed in the previous sections.

In the first phase, we consider multi-threaded processes in isolation and perform resource allocation for threads of each multi-threaded process. Since the interference metric will be inappropriate for threads of the same process, we use the *occupancy weight sorting algorithm* described in Section 3.3.1 to decide for a given process which threads will be allocated to the same core. This is shown in Figure 8(a) where the occupancy weight is shown under the thread ID in each node.



(a) Native x86 Run

(b) Hypervisor Run

Figure 9: Evaluation Methodology

After this, we begin the second phase of the resource allocation algorithm by forming the interference graph. In this phase, we perform the weighted interference graph algorithm at the thread granularity. We use the results of phase one by setting the edge weights of threads allocated on the same core to a very large value. Similarly, we set the edge weights of threads that will be on different cores to zero. This is shown in Figure 8(b). Such edge weight adjustment will ensure that the MIN-CUT of the graph will always place threads that have very large edge weights onto the same core and threads that have zero weights to different cores. What we did not show is that in addition to thread level allocation, we will follow our prior weighted interference graph algorithm described in Section 3.3.3 to set all other edge weights between processes to complete the entire resource allocation.

4. EVALUATION METHODOLOGY

We conduct our experiments in two phases as shown in Figure 9. The first phase emulates our CBF infrastructure and gather statistics for the OS scheduler. The second phase is the actual execution on commercial machines.

4.1 Gathering Footprint

In this phase, we use Simics to emulate an x86-based virtual machine called *tango*. A Fedora Core Linux (kernel version 2.6.26.33) OS is run on top of *tango*. All simulations are done with groups of four processes running on top of the Fedora Core Linux.

The hardware-software interface was implemented using Simics *magic* instructions. The Linux 2.6.16.33 kernel was modified to incorporate calls to the special magic instruction during context switches of targeted applications. The targeted applications for which this scheduling is being done are specified by the user. We use Linux's `proc` interface to let the kernel know the PIDs of targeted processes. When a *magic* instruction is executed, it passes the control to the Simics magic handler which contains our enhancement to pass Bloom Filter Signatures. The Bloom Filter Signatures infrastructure was implemented inside the Simics g-cache module. The cache models exactly the same configuration of our target machine Intel Core 2 Duo, *i.e.*, 16-way 4MB L2. This experimental approach enables us to apply the decisions, based on Simics simulator implemented with our footprint scheme, on actual machines to gather measurement results. A kernel module reads these signatures and keeps them as a part of the process context. The Linux kernel was also modified to implement a syscall to make this signature data available to user-mode programs.

The job of resource allocation is done by a user-level application. It involves setting affinity bits of processes, so that the scheduler can assign the process to a particular core. The algorithms were explained

Table 1: Example of Experiments

Benchmarks	AB & CD	AC & BD	AD & BC
povray (A)	125	126	125
gobmk (B)	107	107	99
libquantum (C)	124	123	111
hmmer (D)	104	105	104

in Section 3.3. The goal of this phase is to provide a process-to-core mapping for a certain set of processes for performance optimization. This mapping will be used at runtime where we use these decisions to guide process scheduling on real machines. The emulation phase ran for 2 billion instructions after fast-forwarding 5 billion instructions. The resource allocator was invoked every 100ms of simulated time. The allocation picked by the simulated allocator majority of the times is considered to be chosen schedule for the given mix of benchmarks.

4.2 Real Machine Execution

We perform two sets of experiments. The first involved running sets of four benchmarks simultaneously on a Fedora Core Linux OS on top of an Intel Core 2 Duo 2.6 GHz machine with a 4MB shared cache. The benchmarks were run simultaneously and restarted accordingly until the longest of the four benchmarks completed. For the second set of experiments we used the open source industry standard virtualization software Xen running on the same Core 2 Duo machine. Four VMs were configured on the Xen hypervisor. Each VM ran Fedora Core Linux and one benchmark. This set of VMs were run till the longest running benchmark completed. The other three benchmarks were restarted accordingly.

We report the maximum and average performance improvement. We explain how we arrive at these two numbers using an example. Let us choose four benchmarks (povray, gobmk, libquantum and hmmer) from our pool. We run all possible mappings of these four on a Core 2 Duo machine and record their user run-time to completion. For this example, the user run times in seconds for all possible process-to-core mappings are listed in Table 1.

There are only three possible mappings for 4 processes running on a dual-core as shown in the table. Once we obtain this table, we examine our results collected in the emulation phase and find that during our simulation the mapping (AD and BC) was preferred by our resource allocation algorithm for majority of the emulation times. Our results show that benchmarks gobmk and libquantum have significant performance improvement for the chosen schedule while no schedule has any significant effect on the runtimes of benchmarks povray and hmmer. We note for this set of benchmarks libquantum has a performance improvement of 11%. We run libquantum with all possible combination of benchmarks from our pool of benchmarks and report the maximum performance improvement over all possible combinations. Similarly we also report the average performance improvement over all possible benchmark combinations involving libquantum.

Our pool of benchmarks consists of 12 SPEC 2006 programs. They were chosen to have a diverse mix. To observe the performance effect of the benchmarks on each other, they were run to completion in mixes of 4 for all the three possible allocations on a dual-core machine. The benchmarks were run till the longest running benchmark completed.

For the VM experiments we used the same 12 SPEC programs and encapsulated them in a VM. Four VMs were configured on the Xen hypervisor. Each VM ran Fedora Linux and one benchmark from the pool. Note that, due to limitations and scalability issues of executing virtualization solutions such as Xen in Simics, our first phase was performed using process-based encapsulation of workloads instead of VMs. Our execution results (mapping of VMs to cores), though, map directly to scenarios where workload processes are executed in independent VMs as opposed to a single OS. Using this approach, the hypervisor functionality described in Section 3 is incorporated into the host kernel via modifications to the OS and an associated module. The control domain is mapped to a user space application which communicates to underlying components using system calls as opposed to the hypercalls that would exist in an actual virtualized implementation.

For the multi-threaded applications we use *PARSEC* suite [3]. We run all possible combinations of four benchmarks from this suite. For the multi-threaded benchmarks, each application has four threads. We measure the user time to completion of the enclosing process to report performance improvements.

5. RESULTS AND ANALYSIS

5.1 Performance Improvement

5.1.1 Intel Core 2 Duo Execution

Figure 10 reports the maximum performance benefit obtained per benchmark application using our best resource allocation algorithm. The results were obtained by running the benchmark suite in groups of four, and the reported results on the left bars are the maximum performance benefit obtained by an allocation by our weighted interference graph algorithm over the worse-case performance for that group of four. All reported results were obtained from running the mix of applications on a real Intel Core 2 Duo system. Our technique shows significant improvement for those heavily exercising the L2 cache (e.g., mcf). The maximum improvement is 54% for mcf followed by 49% for omnetpp. Another noteworthy point is that cache contention does not affect performance for two types of applications. The first type is compute-bound such as povray which does not depend much on the L2. The second type is bandwidth-bound such as hmmer, a sequence profile searching package. It frequently accesses a protein database and shows low locality yet high memory traffic. Figure 10 also shows the average performance gain using the weighted interference graph algorithm obtained for each benchmark across all the mixes of benchmarks.

5.1.2 Virtual Machine Execution on Xen

Figure 11 reports the maximum and average performance benefit obtained per benchmark running inside a Xen hypervisor using our weighted interference graph algorithm. It shows that the maximum and average performance improvements are lower than if they were running on a native machine. For example, the maximum performance improvement for mcf is 26 % when running inside hypervisors in contrast to 54% on a native machine. One reason for the reduction is the virtualization overhead. Despite the lowered performance improvement, the relative trend of improvements remains much the same. It implies that even though the applications are encapsulated inside VMs, the negative caching effect among them still maintain similar impact on each other's performance.

5.1.3 Multi-Threaded Workload on Intel Core 2 Duo

Figure 12 shows the maximum and average performance improvement for multi-threaded *PARSEC*. Similar to single threaded benchmark, the multi-threaded benchmarks also show reasonably good performance improvement. The maximum performance improvement (10.1%) is observed in ferret. The performance improvements for the multi-threaded workloads seem more modest than their single threaded counterparts as the memory working set (footprint) of *SPEC 2006* is known to be much larger than the *PARSEC*; the latter was focused more on compute-bound applications.

5.2 Comparison of Three Resource Allocation Algorithms

Figure 13 shows the relative performance improvement of a few representative benchmark mix for three proposed algorithms. Interestingly, the weight sorting algorithm, despite its simplicity, gave the best results in certain cases. This indicates that the cache footprint is a very good metric for predicting performance effect on processes/VMs sharing the L2. We can also observe that the weighted interference

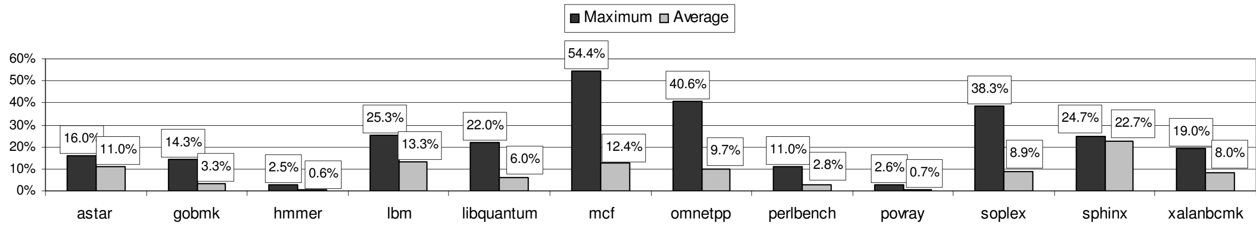


Figure 10: Maximum Performance Improvement for Each Benchmark (Native run on Intel Core 2 Duo)

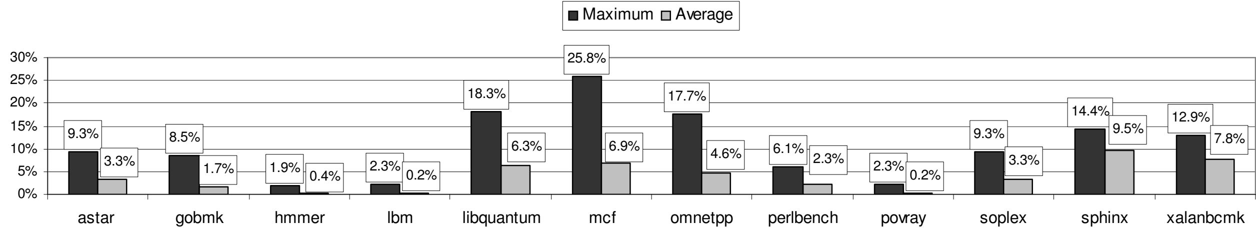


Figure 11: Maximum Performance Improvement for Each Benchmark (Run in Xen Hypervisor)

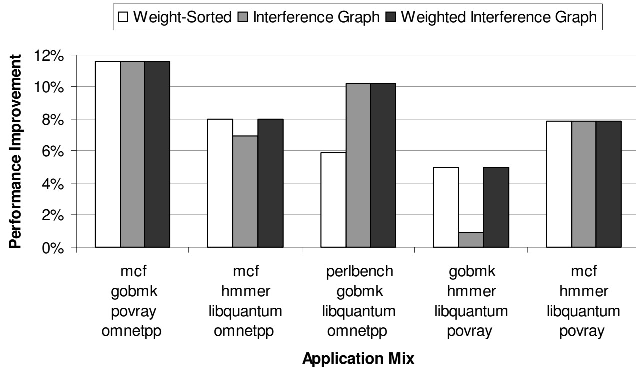


Figure 13: Resource Allocation Algorithms

graph mechanism achieved as good or better performance among all. This result is not surprising as the weighted algorithm considers both the symbiosis and the occupancy weights.

5.3 Comparison of Different Hash Functions

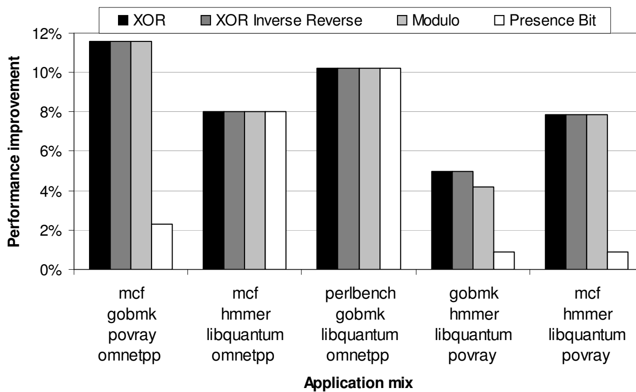


Figure 14: Comparing Different Hash Functions

One important design criterion in our proposed architecture is choosing a suitable hash function for the Bloom filters. Aside from minimizing collisions, a low-complexity hash function is highly desirable for practical hardware implementation. Figure 14 shows a few representative mix of benchmarks showing the relative performance improvements for different hash functions. We tested four hash functions for our evaluation:

- XOR: The block address is divided into equal chunks of hash index wide which are bitwise-XORed to obtain the hash index.
- XOR Inverse Reverse: This is same as XOR except that the obtained index from XOR is bitwise inverted and reversed.
- Modulo: This is a simple modulo operation of the block address with the Bloom filter size.
- Presence bits: It has a one-to-one mapping with the cache lines being sampled. More explanation is described later.

We found that the first three hash functions perform identically for almost all the mixes. The exception is the mix with gobmk, hmmer, libquantum and povray where modulo performs slightly worse. In general, XOR-based hash suffices in terms of performance and hardware cost.

As explained, the presence bits have a one-to-one mapping with the cache lines being sampled. Therefore, instead of maintaining a bloom filter whose number of entries is bigger than the number of cache lines we are sampling, a presence bit vector has exactly the same number of bits as the number of cache lines it is sampling. So a presence bit vector contains an exact per-core footprint of the cache. The results are shown in the rightmost bar of Figure 14. We found that the presence bit vectors have no effect on scheduling decisions. All the results shown are default schedules with which the processes began execution. In the case of perlbench, gobmk, libquantum, and omnetpp mix and the mcf, hmmer, libquantum, and omnetpp the default schedule coincidentally is the best possible schedule. The reason why presence bit vectors do not work is because they get saturated quite often for processes that heavily use the cache. A saturated presence bit vector conveys little information. This is exactly the same reason why we chose not to use multiple hash functions which will set multiple bits in the bit vector for a single address entering the cache. This will saturate the Bloom filter and render the technique ineffective just as in the case of presence bit vectors. Multiple hash functions would have been effective if we did not have a strict hardware budget on the number of Bloom filter entries.

We need to mention here that since our technique is at a cache line granularity, page granularity changes in the system like page remapping is unlikely to affect its effectiveness. However, there may be slight changes in the interference metric due to remapping. Also, since our technique uses a user-level process to set affinity bits and as-

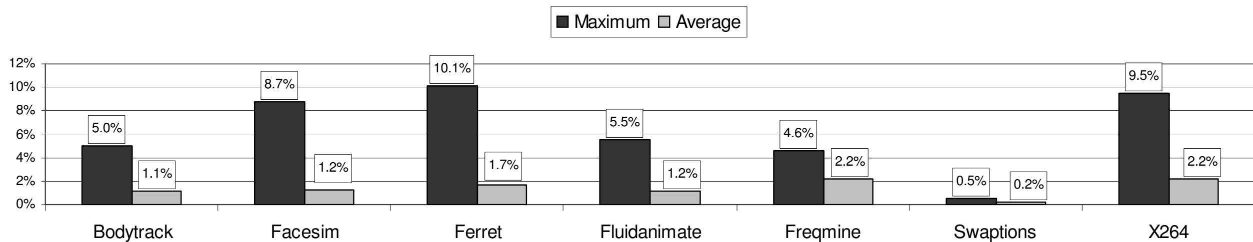


Figure 12: Maximum Performance Improvement for Multi-threaded PARSEC (Native run on Intel Core 2 Duo)

signs processes to processor cores, it will not affect Linux's scheduling algorithm of using distributed queues for a multicore system.

5.4 Implementation Overheads

We consider two different aspects of the overheads for our technique. The first is the software overhead that involves book-keeping of interference data with the process/VM context. This part of the overhead is trivial since the amount of data needed to be maintained as a process context is just a set of three 32-bit numbers. Also there is the overhead of computation of the appropriate schedule by the algorithm. Since the graphs created by the scheduling algorithms have tens of nodes, the overhead of creating such graphs is hundreds of instructions. Also the graph algorithm will be in the order of hundreds of instructions. Since the algorithm is only invoked once every 100 ms in a GHz processor, this performance overhead for a fast heuristic algorithm is also negligible.

Another aspect of the overhead is computing the weight and interference metrics for every context switch, which are done using parallel bitwise XOR gates and will not take more than a few nanoseconds. We also need to consider the communication overhead of transferring the current RBV's between cores during a context switch. Since the number of bytes in an RBV is 1KB, the communication overhead for a dual-core machine per context switch is two transfers of 1KB data for roughly once every 2-3 billion cycles.

Another aspect of overhead is the hardware. This overhead involves the hardware cost of maintaining the Counters, Core Filters, and Last Filters. The number of entries in the counter array, LFs and CFs were chosen to be equal to the number of cache lines. Assuming a 64-byte line and an *N-core* machine and 3-bit counters, the overhead of the LF and CF is given by $(2 * N + 3)/(64 + 18) * 100\%$. For a dual-core machine it is 8.5% of the cache size, which would be inordinately large. We, therefore, consider a widely used technique of sampling data sets for keeping signatures. Sampling of data sets is a widely used technique for cache profiling [27]. We performed 25% sampling of data sets and also compared the results to unsampled system. We found that the correctness of our algorithm is not affected by the sampling for the benchmarks we considered.[2] Thus our total overhead can be boiled down to only about 2.13% of the L2 size.

6. RELATED WORK

A slew of prior work have been focused on the impact of L2 cache partitioning [4, 12, 13, 15, 16, 18, 26, 28, 29, 35, 39]. A notable one similar to ours is *CacheScout* [39] which tags cache lines with software guided monitoring ID to track interference. The approach is similar to using presence bits, which we showed in Figure 14, given little performance benefit. Comparing their simulated speedup results reported in [39] against our actual measurement results for four same benchmark programs, the speedup for three of them (art, crafty, swim) showed two to four times improvement. By the way, all the L2 partitioning schemes still suffered from the problem that they need to change the interface of the normal caching. For example, the replacement policy needs to be modified in [16, 18, 28, 35] while dedicated sets were adaptively given to processors in [34] to improve L2 sharing.

Other cache partitioning techniques require software layers to explicitly specify the application requirements for cache capacity and bandwidth, and provide hardware mechanisms to uphold guarantees [24, 25]. Settle *et al.* [31] uses an activity-factor as a metric to schedule threads in an SMT platform. They require to rewrite the OS scheduler while our scheme is OS-agnostic. Another technique for scheduling was proposed by Snavely *et al.* [32]. They compute the diversity, balance, and resource conflicts between competing possible schedules in SMT and performs symbiotic scheduling, very different from our memory footprint signature. Li *et al.* [21] studied the OS support for fair-sharing on heterogeneous cores with asymmetric ISAs. Kandemir *et al.* [17] applied code restructuring in compilers to mitigate inter-core conflict misses. Herdrich *et al.* [14] communicated task priority to a rate controller for estimating cache and memory bandwidth needed by applications. In [5, 36], shared cache partitioning is done at the OS level by page allocation. In contrast, we did not change normal cache mechanisms nor require specified requirements for virtual resources. Our less invasive scheme simplifies the implementation.

Researchers also examined metrics like cache affinity for resource allocation [11, 10, 33, 37]. Some [19] leveraged the OS to capture misses per cycle and cache accesses on a per-thread basis for scheduling. However, these techniques are less effective without dynamic hardware's involvement.

Dhodapkar *et al.* [7] presented algorithms that dynamically collect and analyze working set to configure the I-cache. Though the manner of collecting *Working Set Signatures* is similar, they did it for phase change detection. Liu *et al.* [22] and Zhuravlev *et al.* [40] analyzed resource sharing issues in non-cache system resources such as memory bandwidth, prefetch hardware, etc. Different from ours, miss rates were used in the scheduling algorithms by [40].

7. CONCLUSIONS

Demands for performance and manageability and technological constraints have led to two trends: multi-core architectures and support for virtualization. Meanwhile, resource sharing and contention on these emerging platforms can degrade performance if the resource allocation policy is ignorant. In this paper, we proposed architectural support to mitigate the interference caused by the L2 cache contention. We also proposed and studied three resource allocation algorithms to minimize interference and find the symbiosis among processes and threads. By integrating Bloom Filter Signature collected on a per-core, per-process, or per-VM basis, the OS can perform job scheduling based on their dynamic, mutual symbiosis to minimize negative performance side effect when incompatible processes run on multiple cores sharing a cache.

Unlike prior studies that completely relied on simulation results and defined their own fairness metrics, we substantiated our performance results by executing our new process-core mapping on an Intel Core 2 Duo machine and reported the runtime speedup. Our experiments were done in two phases. First, we collected the dynamic signature information from Simics runs, and ran them through the proposed three resource allocation algorithms. We then applied these results to guide our modified Linux and executed the same workloads on a real Intel Core 2 Duo platform. We showed an improvement of

[2]Therefore, we did not present the results of unsampled systems as the mapping decisions were found identical to those of sampled ones.

22% on average (up to 54%) when applications run natively. When running inside the VMs, the speedup is 9.5% on average (up to 26%). Given that the future trend is to increase the number of cores and the degree of sharing, integrating more intelligence using runtime information for resource sharing as proposed will become very critical to minimize performance downfall due to resource conflicts.

8. ACKNOWLEDGMENT

This research is supported by NSF grants CCF-0326396, CCF-0811738, CNS-1017297, an NSF CAREER Award CNS-0644096, and generous gift given by Intel Corp. The authors also thank Tejas Iyer for discussion on the algorithm design. This research was performed when Dr. Ghosh and Dr. Nathuji were affiliated with Georgia Tech.

9. REFERENCES

[1] The AIM IX Independent Resource Benchmark Suite. http://sourceforge.net/projects/aimbench.

[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 ACM Symposium on Operating Systems Principles*, 2003.

[3] C. Bienia and K. Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[4] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the International Conference on Supercomputing*, 2007.

[5] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *Proceedings of the International Symposium on Microarchitecture*, 2006.

[6] M. Devarakonda and A. Mukherjee. Issues in implementation of cache-affinity scheduling. *Proceedings of theWinter 1992 USENIX Technical Conference and Exhibition*, pages 345–357, 1992.

[7] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working setanalysis. *Proceedings of the Annual International Symposium on Computer Architecture*, 2002.

[8] L. Fan, P. Cao, J. Almerda, and A. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 2000.

[9] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Throughput-Oriented Scheduling On Chip Multithreading Systems. *Technical ReportTR-17-04, Harvard University, August*, 2004.

[10] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proceedings of the USENIX Annual Technical Conference*, April 2005.

[11] A. Fedorova, M. Seltzer, M. Smith, and C. Small. CASC: A Cache-Aware Scheduling Algorithm For Multithreaded Chip Multiprocessors. http://research.sun.com/scalable/pubs/CASC.pdf.

[12] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *Proceedings of the IEEE International Symposium on Microarchitecture*, 2007.

[13] M. Hammoud, S. Cho, and R. Melhem. Dynamic Cache Clustering for Chip Multiprocessors. In *Proceedings of the International Conference on Supercomputing*, 2009.

[14] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based qos techniques for cache/memory in cmp platforms. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 479–488, 2009.

[15] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *Proceedings of the 2004 International Conference on Supercomputing*, pages 257–266, June 2004.

[16] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques*, 2008.

[17] M. Kandemir, S. P. Muralidhara, S. H. K. Narayanan, Y. Zhang, and O. Ozturk. Optimizing shared cache behavior of chip multiprocessors. In *Proc. of the International Symp. on Microarchitecture*, 2009.

[18] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 2004 International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, Sept. 2004.

[19] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE MICRO*, 28(3):54–66, 2008.

[20] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An analysis of performance interference effects in virtual environments. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2007.

[21] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating System Support for Overlapping-ISA Heterogeneous Multi-core Architectures. In *Proceedings of the International Conference on High Performance Computer Architecture*, 2010.

[22] F. Liu, X. Jiang, and Y. Solihin. Understanding How Off-Chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance. In *Proceedings of the International Conference on High Performance Computer Architecture*, 2010.

[23] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. In *Intel Technology Journal*, August 2006.

[24] K. Nesbit, J. Laudon, and J. Smith. Virtual private caches. In *Proceedings of the Int'l Symp. on Computer Architecture*, 2007.

[25] K. Nesbit, J. Smith, M. Moreto, F. Cazorla, B. Supercomputing, A. Ramirez, and M. Valero. Multicore Resource Management. *IEEE MICRO*, 28(3):6–16, 2008.

[26] M. Qureshi. Adaptive Spill-Receive for Robust High-Performance Caching in CMPs. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2009.

[27] M. Qureshi, M. Suleman, and Y. Patt. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. *Proceedings of International Symp. on High Performance Computer Architecture*, 2007.

[28] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance Runtime Mechanism to Partition Shared Caches. In *Proceedings of the Interntaionl Symposium on Microarchitecture*, 2006.

[29] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural Support for Operating System-Driven CMP Cache Management. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 2–12, Sept. 2006.

[30] J. Salehi, J. Kurose, and D. Towsley. The effectiveness of affinity-based scheduling in multiprocessor network protocol processing. *IEEE/ACM Transactions on Networking*, 4(4):516–530, 1996.

[31] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural Support for Enhanced SMT Job Scheduling. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[32] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *SIGARCH Comput. Archit. News*, 28(5):234–244, 2000.

[33] M. Squillante and E. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2), 1993.

[34] S. Srikantaiah, M. Kandemir, and M. Irwin. Adaptive set-pinning: Managing shared caches in chip multiprocessors. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[35] G. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[36] D. Tam, R. Azimi, L. Soares, and M. Stumm. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 121–132. ACM, 2009.

[37] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, 1995.

[38] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 26–40, 1991.

[39] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms. *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[40] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.