Architectural Support for Symbiotic Scheduling on Shared Cache Multi-Core Systems Using Bloom Filter Signature

ABSTRACT

As the trends of more cores sharing common resources on a single die and more systems crammed into enterprise computing space continue, optimizing the economy of scale for a given compute capacity is becoming increasingly 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 (e.g., the shared L2), thereby slowing down all processes.

To ameliorate the allocation/management policy of a shared cache on a multi-core or on a VM, 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 the caches to efficiently summarize cache usage behavior on a per-core, per-process or per-VM basis. We then proposed and studied three algorithms to determine the optimal process-to-core mapping with a goal of minimizing interference in the L2. Unlike many prior works completely relied on simulations and their own improvement metrics, we validate our performance improvement by executing applications using allocation generated by our new process-to-core mapping algorithms on an Intel Core 2 Duo machine and report the actual run time speedup. Our run time results showed an improvement of up to 54% (or 22% on average) when the applications are run natively, and an improvement of up to 26% (or 9.5% on average) when running inside VMs. Also, we compared our scheme against CacheScouts and showed significant speedup improvement.

1. INTRODUCTION

The ever increasing demands for performance and manageability in modern enterprise computing systems has directly affected the innovation in both computer architecture and systems design. On one end, the modern computing platforms provide multi-core processors with simplified cores to improve performance by exploiting thread-level parallelism while keeping thermal and power consumption in control. On the other hand, it has become a requirement for enterprise systems to provide flexible and efficient use of resources via software management capability. Industry has responded to this need with the integration of hardware and software solutions (e.g., Xen or VMware) for virtualization [4, 20, 33]. Virtualization allows applications to obtain benefits such as fault and performance isolation [4, 19].

Given the context of these two emerging technologies, it is critical to address their interactions for achieving the best possible performance, and preventing unnecessary performance degradation with available runtime information. Previous work has considered cache interference of different processes due to affinity effects and thrashing in shared cache architectures. These studies have shown the possibility of significant performance implications when multiple execution instances, or processes, are scheduled in an unaware fashion onto processors. In this paper, we quantify the performance implications of destructive caching effects of multiple applications running simultaneously on a multi-core processor. As a special case, we show how these implications extend to both native mult-core machine and the virtualized execution where multiple virtual CPUs (vcpus) are mapped to sets of physical processing cores sharing a cache.

Here we highlight two problems that must be addressed in order to make an optimized process-to-core allocation decision. First, there is the issue of determining cache usage characteristics of workloads. As experimentally validated in this paper, utilizing online measurement mechanisms such as event-based performance counters do not always reflect the cache footprints of workloads. In response, we employ per-core counting Bloom filters to enable online profiling of cache usage behavior in the hardware. The second problem is that of determining the interactions between workloads based on collected Bloom filter data. We project such interactions using hardware support for comparing *Bloom filter signatures* of workloads. This information is then provided to software-based allocation algorithms. These policies facilitate the efficient management of computing resources among applications to maximize system performance. The algorithms may be implemented either 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) in the case of the Xen hypervisor. For VMs, these policies determine the virtual-to-physical resource mappings to improve system performance, while still providing some means of fairness across workloads. Our evaluation by gathering Bloom filter signature on Simics and running real execution on an Intel Core 2 Duo system highlight the benefits achieved by our system, including performance improvements of up to 54% for the best case and up to 22% on the average. For VMs, the benefits range from 26% in the best case and up to 9.5% on the average.

The remainder of this paper is organized as follows. Section 2 motivates the need for hardware support for efficient resource allocation in multi-core systems. We then proceed to outline the design of our system, in Section 3. Our evaluation methodology is described in Section 4. Section 5 analyzes our results. Finally, we summarize related work in Section 6, and conclude in Section 7.

2. MOTIVATION

2.1 Resource Allocation in Multi-core

In a multiprocess multi-core environment, workloads usually compete for a common set of computing resources. As a result, the job of

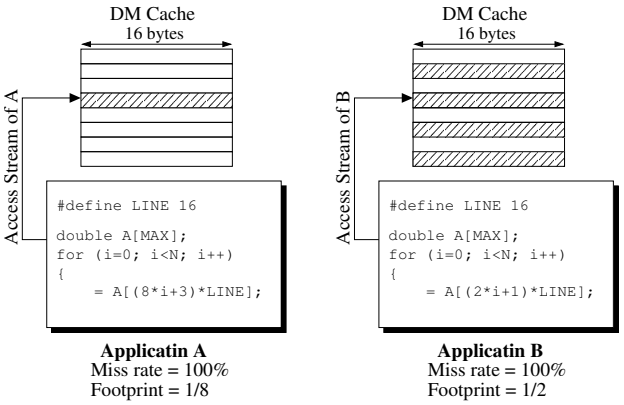


Figure 1: Different Cache Footprints with the Same Miss Rate

the OS in dispatching workloads to physical cores will become even more critical to achieve the best possible performance. In addition, future OSes will also need to consider power management, assess formation of hotspots, avert the risk of thermal runaway, and guarantee a certain hierarchy of QoS among the workloads. This problem is particularly interesting in the case of providing support for virtualization technology 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 utilize intelligent decision-making schemes to perform allocations that provide improved system characteristics. In the context of this paper, we particularly consider improving the allocation decisions when mapping physical resources like a shared last level cache to virtual CPUs that run guest VMs. An important goal of resource allocation is to determine allocations that maximize performance by minimizing the types of negative caching effects described next.

2.2 Hardware Support for Multi-core Resource Management

A large body of work has addressed cache affinity scheduling in shared memory multiprocessors. Devakonda *et al.* [7] showed the effects of cache affinity scheduling and also state that affinity scheduling works well only for a certain class of applications. This observation is substantiated by Salehi in [27] that showed that affinity scheduling improves throughput of certain network protocol processing applications. Furthermore, in [38], the authors stated that cache affinity scheduling is not very effective over other simple scheduling policies.

One conclusion from these prior work is that in the scenario of multiple applications running on different processors sharing a uniform L2 cache, the co-scheduling of incompatible applications may affect the performance of each application drastically. For instance, when multiple applications compete for a shared state-based resource (e.g., cache), certain applications can have a destructive effect on the cache state of other applications. In response, there was research to aid scheduling decisions based on the miss rate of caches [10]. However, predicting incompatibilities between running applications cannot be accurately done using performance metering based approaches like 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 contains repeated accesses to the same cache set, having a 100% miss rate, even though they are all different cache lines. Nonetheless, the *footprint* of A is just one line or 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 shared cache space, contributing a greater detrimental effect on 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 [2]. 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 to find a perfect correlation between them.

The conclusion from the above analysis is that performance counter based approaches employed in prior studies [12] 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.

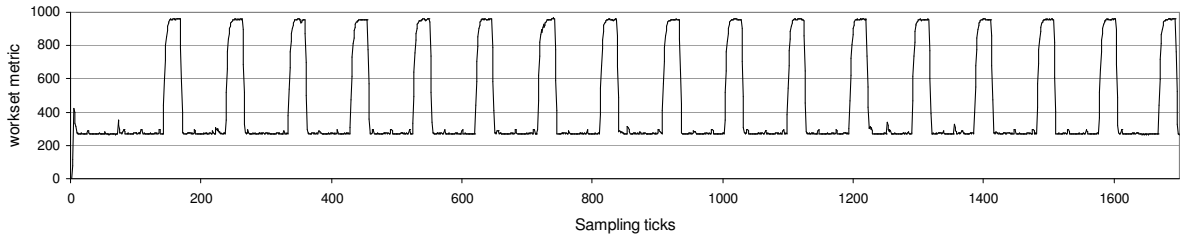
2.3 Quantifying Application Incompatibility

We used 12 SPEC2006 benchmark programs to quantify the effects of incompatibilities between applications 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. The first machine has a Xeon SMP wherein the selected two processes were constrained to run on the same processor. The second one is a shared cache multi-core system, where the pair of processes ran on different cores and contended for the L2 cache space. The results of the performance degradation are shown in the following subsections. Each bar in the graphs 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. The name on top of each bar indicates the benchmark with which it was paired.

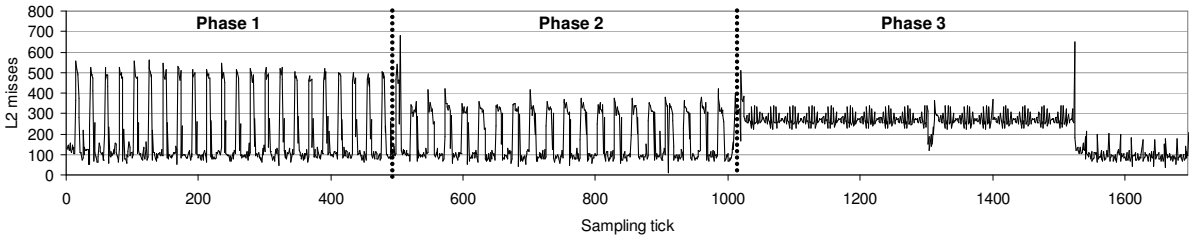
2.3.1 P4 Xeon SMP System

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

2.3.2 Shared Cache in Intel Dual-Core



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



(b) L2 Misses

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

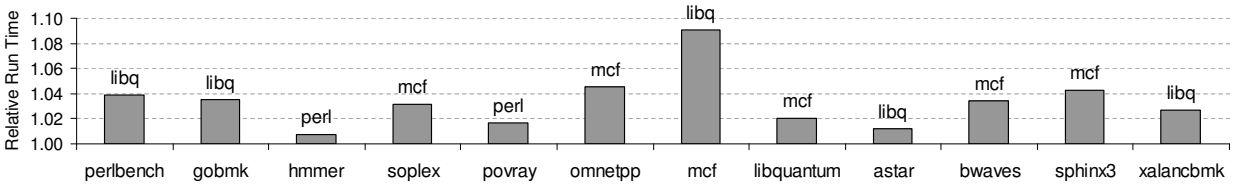


Figure 3: Worst-case Performance Disturbance on a P4 Xeon System (Run on Single P4)

For a shared cache architecture, we ran our experiments on an 2.34 GHz Intel Core 2 Duo (Conroe) system with a 4MB 16-way *shared L2* cache. We scheduled the two processes on different cores but sharing the same L2. As shown in Figure 4, even though the L2 is twice larger than the P4 Xeon, yet the relative performance degradation is much more severe in the case of Core 2 Duo. We found that the maximum degradation is 67% for the mcf paired with libquantum. These experimental results demonstrate that the application resource allocation problem is more interesting and serious in a multi-core shared cache architecture, with a potential performance improvement of as much as 67% as indicated in the figure. These results on actual systems motivate the need for a better core allocation mechanism for applications based on their cache footprints.

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 compatibilities. Based on the previous results, we can expect that an OS that uses a metric of determining compatibilities between applications for co-scheduling and resource allocation, will achieve significant throughput improvements over the current state-of-the-art. The goal of our work is to provide such compatibility methods. Toward this, we will describe a novel resource allocation infrastructure for multi-core architectures in the following sections. The infrastructure consists of an architecture scheme that monitors cache resource utilization along with a software layer that uses such monitoring information to intelligently allocate resources for each application. The architectural extension involves the use of a modified version of the Counting Bloom Filter (CBF), which is a low-cost data structure known for its high efficiency in maintaining signatures of large data sets. Now we will discuss the basics of the CBF to give a better understanding with respect to how it can be used for application resource allocation purposes.

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 5(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 build memory-efficient database applications. .

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

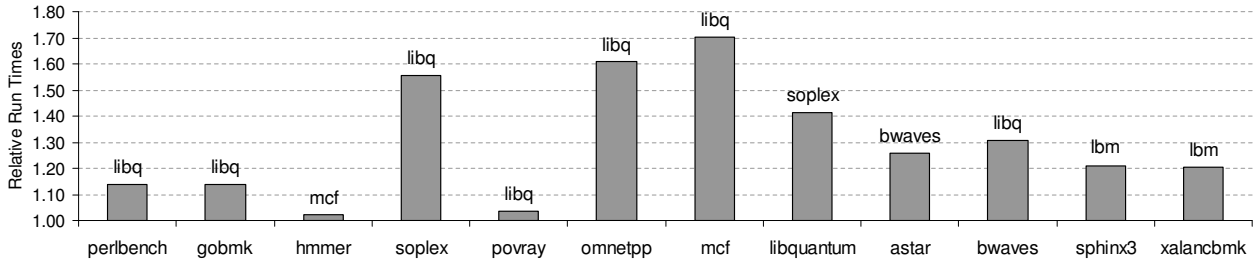
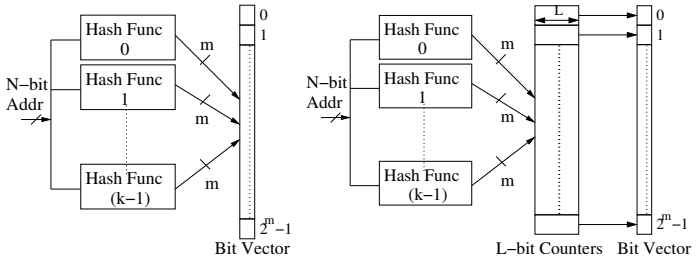


Figure 4: Worse-case Performance Disturbance on a Intel Core 2 Duo System



(a) Original (b) Counting Bloom Filter

Figure 5: Bloom Filters

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 during a query 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) [9] was proposed to allow deleting entries from the filter as shown in Figure 5(b). The CBF reduces the number of false hits by introducing counters in lieu of a bitvector. 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* of 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 6, 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 following section describes in detail the new architecture-supported OS resource allocation scheme developed in our research. The scheme is shown to substantially improve performance of applications in actual multi-core systems.

We demonstrate that counting Bloom filters effectively monitors the cache footprint in Figure 6. The benchmark used for this experiment is the same aim9_disk benchmark as used in Figure 2(a). We define *signature value* as the number of ones in the bit vector of the counting Bloom filer. We can easily find in Figure 6, that the signature value follows the cache footprint size.

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

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 multiple applications in a multi-core platform sharing L2. An architecture example is depicted in Figure 7(a) which consists of four cores sharing an L2 cache via the back side bus.

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-based 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 cache 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 cache. Another simplification to the CBF is that we just use one hashing 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 likely saturate the bit-vectors faster. Also, using multiple hash functions incur much larger hardware overhead.

The detailed operation of our proposed CBF architecture is described as follows. For each L2 miss, since the miss will eventually cause a linefill, the corresponding counter in the counter array indexed by the address hash is incremented. Along with incrementing the counter, the corresponding index of the Core Filter (CF) of the core from which the miss originated is also set to 1. As such, the CF is only responsible for keeping track of memory requests originating from the core to which it was attached. The replacement of a line in the L2 causes the corresponding hashed counter to be decremented. If the counter becomes zero after decrementing, all core filters are accessed, and the bit corresponding to the decremented counter index is 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 Core Filter only counts one entry, an artifact due to alias that will underestimates 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

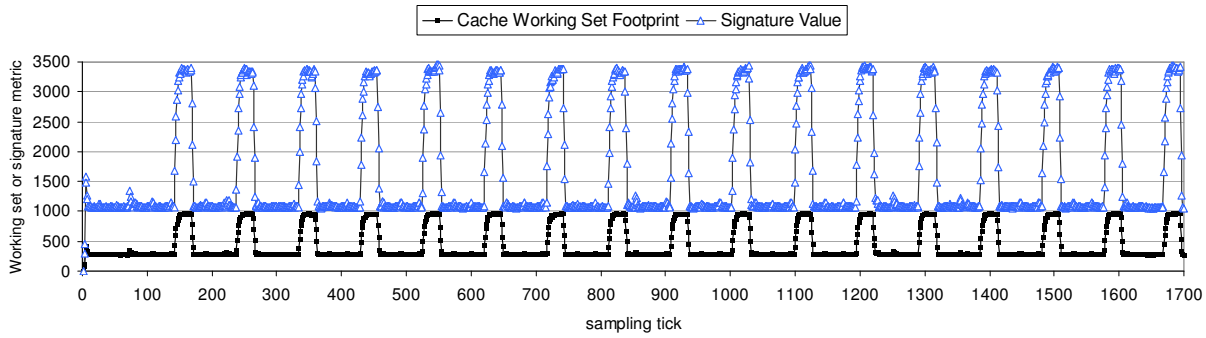
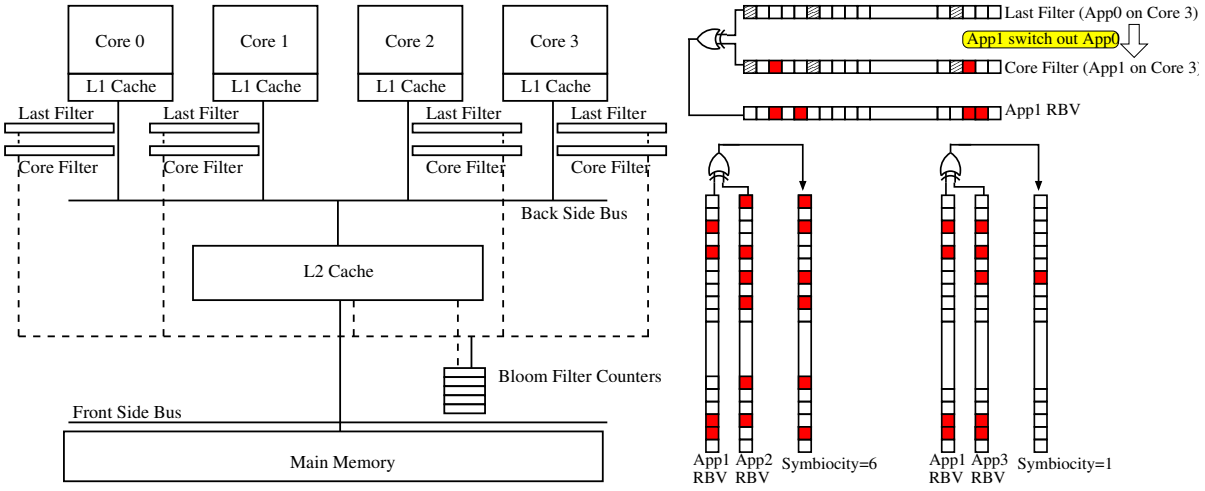


Figure 6: Bloom Filter Signature Value versus Cache Working Set Footprint



(a) Multi-core with CBF

(b) Symbiosis Calculation

Figure 7: Multi-Core Enhanced with Signature for Symbiotic Scheduling

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 that we call the Last Filter (LF) for each core. The LF keeps a snapshot copy of the CF whenever a new application is scheduled to run on the core. The topmost bitvector in Figure 7(b) shows such a snapshot when App0 is being swapped out of Core 3 by App1. This state information kept just before the new application or VM accesses the cache helps identify exactly how much of the cache resources will be 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 7(b). Counting the number of ones in the RBV is a metric of the cache *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 less interference. A low value either means higher interference, or that both vectors have a very low occupancy. We show an example at the bottom of Figure 7(b) where App1/App2 generates better (higher) symbiosis(=6) than App1/App3 (=1), that says, App1 will co-exist better with App2 in the L2 than with App3. These metrics of occupancy and symbiosis with other cores is kept along with the application as a part of its context. The OS can use these metrics to allocate a process with minimum cache interference. Note that we keep only the occupancy and symbiosis metrics (three integers for a dual core machine) as a part of the process context. We do not keep any of the bit vectors as a part of the context and thus the overhead of additional information is very low. This also explains why we need to only maintain the LF in hardware as opposed to keeping track of bitvectors for all running applications.

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 $(N + 2)$ entries, where N is the number of physical cores on the platform. Whenever an application is context switched out from a core, the data structure associated with the application is updated. The first entry of the structure is set to the core ID of the last physical processor allocated to the application. The second entry

is updated with the *occupancy weight* while the remainder of the N entries store the *symbiosis metrics*.

While the OS handles utilizing underlying hardware support for updating symbiosis and occupancy data structures, 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 executing applications of interest. As described in more detail in the implementation description, this information can then be incorporated into different algorithms in order to obtain an updated allocation decision that is then passed along to the OS using existing system call interface. Note that, 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.

The software for resource allocation in VMs is very similar to that of the application as explained above. In the case of VMs, our resource allocation system is distributed between the hypervisor and the Xen management or control domain, Dom0. Virtualization solutions such as Xen utilize this privileged domain to execute management and control components to provide extensibility while maintaining a thin hypervisor [4]. 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 data structure. 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 hypercall 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 allocate 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. This may appear odd at the first glance. The rationale, as explained earlier, is that the processes assigned to the same core will never execute simultaneously, minimizing their mutual-conflicting effect 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. In this section we describe three algorithms we developed for this purpose.

3.3.1 Weight Sorting Algorithm

The sorting algorithm uses a simple mechanism to detect the L2 cache occupancy of each process. The only metric used here is the sum of the bits inside the Running Bit Vector (RBV). This metric gives a reasonable idea of the cache footprint of a process. Upon every context switch, the RBV is computed as described previously in Section 3.1 and Figure 7(b). After computing the RBV, the 1's of the bitvector are reduced to one weight which is then kept as a part of the process context.

The user-mode algorithm collects the weights of all the processes it wants to schedule for. 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 $\lceil \frac{P}{N} \rceil$, equivalent to 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 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 this algorithm 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 against other processes.

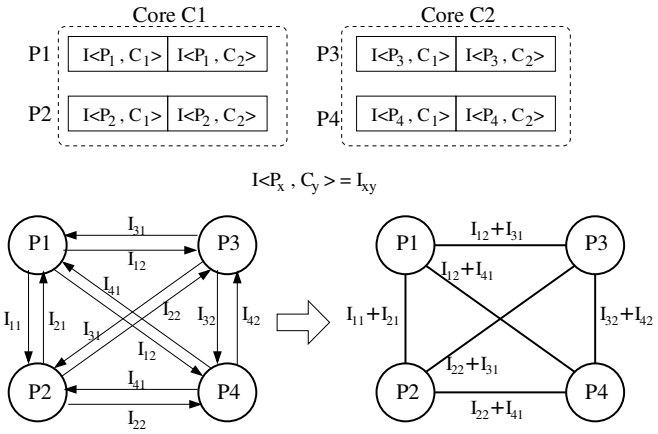


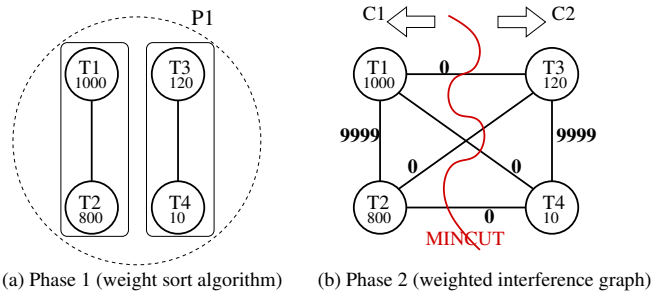
Figure 8: Forming an Interference Graph

3.3.2 Interference Graph Algorithm

The Interference Graph algorithm is explained with the example illustrated in Figure 8. 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* (defined in Section 3.1 and Figure 7(b)) for constructing an interference graph. We also define the notation $I{<}P_x, C_y{>}$ to represent the degree of Interference of *process x* on *core y*. The top of Figure 8 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 Filter of cores C1 and C2 (self-core) with the Running Bit Vector of P3 when P3 was context switched out of C2.

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 weights 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 the interference of 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.



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

Figure 9: Allocation for Multi-threaded Applications

We use the SDP solver [1] to solve the problem.

This algorithm provides a good solution for the case where there are 2 cores. It can be easily 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 5.3 that this is indeed the case.

3.3.4 *Resource Allocation for Multi-threaded Apps*

The resource allocation algorithms described earlier 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 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 challenge, we adapt our resource allocation algorithms to perform resource allocation in two phases. Notice that, this adaption for threads will be applied together with the process-level allocation we just discussed the previous sections.

In the first phase, we consider multithreaded processes in isolation and perform resource allocation for threads for each multithreaded 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 9(a) where the occupancy weight is shown under the thread ID in each node.

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 9(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.

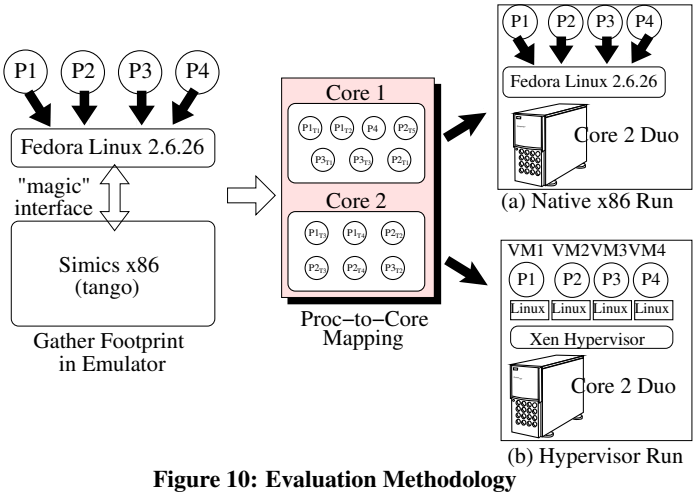


Figure 10: Evaluation Methodology

4. EVALUATION METHODOLOGY

We conduct our experiments in two phases as shown in Figure 10. 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 [3] 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 the 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. 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 processor core. The algorithms were explained in Section 3.3.

The goal of this phase to provide a process-to-core mapping for a certain set of processes for performance optimization. This mapping will be used in the runtime phase 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.

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

4.2 Real Machine Execution

We perform two sets of executions. 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 of benchmarks. We run all possible mappings of these four benchmark programs on a Core 2 Duo machine and record their user run-time to completion. For this example, the user run times in seconds of the benchmarks 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 obtained 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. We find from our results 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, the performance of some were affected significantly by others; for other benchmarks, it was not affected at all. 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 pool of 12 SPEC benchmarks and encapsulated them in a VM. Four VMs were configured on the Xen hypervisor. Each VM ran Fedora Core Linux and one benchmark from the pool. Notice 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 virtual machines. 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 multithreaded applications we use *PARSEC* suite [5]. We run all possible combinations of four benchmarks from this suite. For the multithreaded 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 11 reports the maximum performance benefit obtained per benchmark application using our resource allocation algorithm. This 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 chosen by our weighted interference graph algorithm over the worse-case performance for that group of four benchmarks. All reported results were obtained from running the mix of applications on a real Intel Core 2 Duo system. We can see that 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 benchmarks. The first are benchmarks such as povray that is mainly compute-bound and does not depend much on the L2. The second type is bandwidth-bound such as hmmer. Hmmer is a sequence profile searching package, which needs frequent accesses to a protein database, and thus shows low locality and high memory traffic. Figure 11 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 12 reports the maximum and average performance benefit obtained per benchmark running inside a Xen hypervisor using our weighted interference graph resource allocation algorithms. We can observe that the maximum and average performance improvement of benchmarks running inside VMs is lower than if they were running on a native machine. For example, the maximum performance improvement for *mcf* is 26 % when running inside Hypervisors, whereas it achieved 54% when running on a native machine. One reason for the reduction is the virtualization overhead of the benchmarks. Despite the lowered performance improvement, the relative trend of improvements among benchmarks remains much the same. It implies that even though the benchmarks 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 13 shows the maximum and average performance improvement for the multithreaded *PARSEC* benchmark. Similar to single threaded benchmark, the multithreaded benchmark programs also show reasonably good performance improvement. The maximum performance improvement of 10.1% is observed in the *ferret* benchmark. The average performance improvements of up to 2.2% are observed for *freqmine* and *x264*. The performance improvements for the multithreaded workloads seem more modest than their single threaded counterparts as the memory working set (footprint) of *SPEC2006* 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

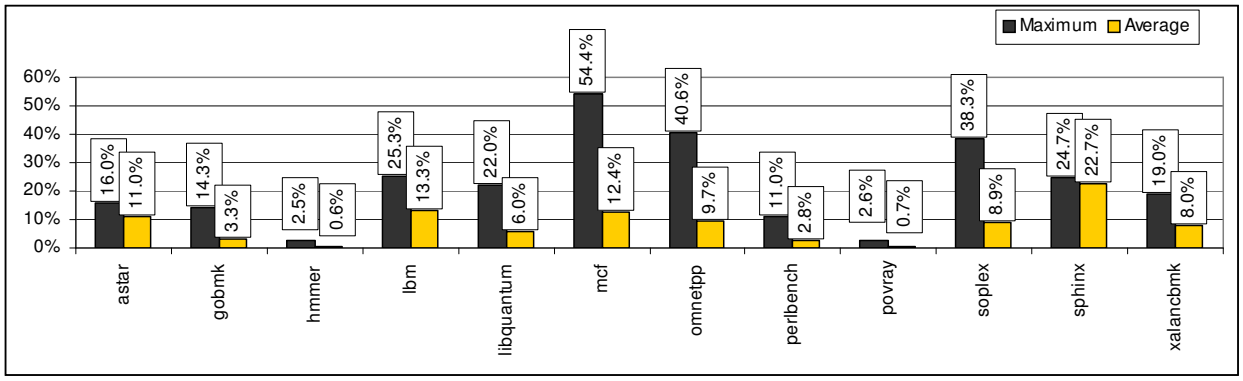


Figure 11: Maximum Performance Improvement for Each Benchmark (Native run on Intel Core 2 Duo)

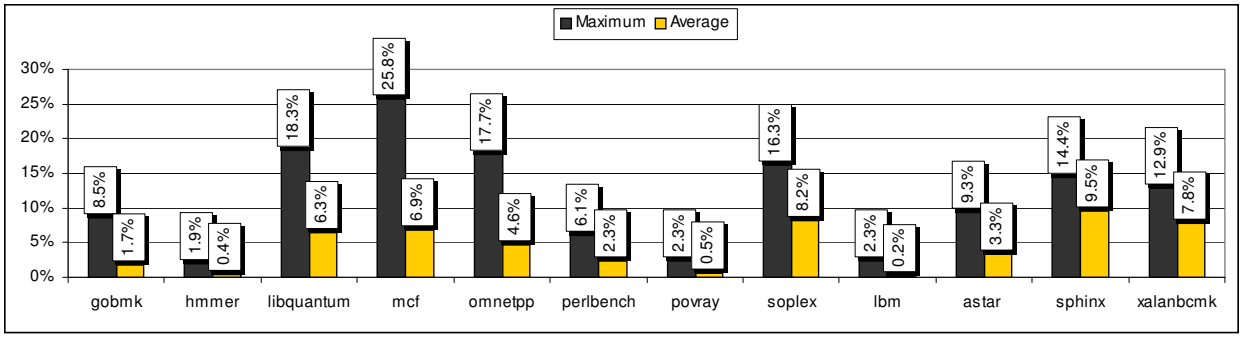


Figure 12: Maximum Performance Improvement for Each Benchmark (Run in Xen Hypervisor)

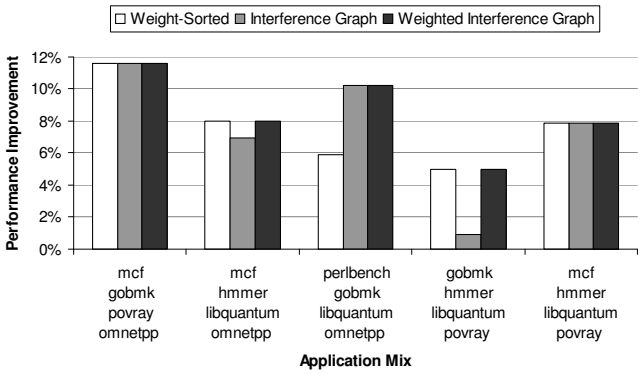


Figure 14: Resource Allocation Algorithms

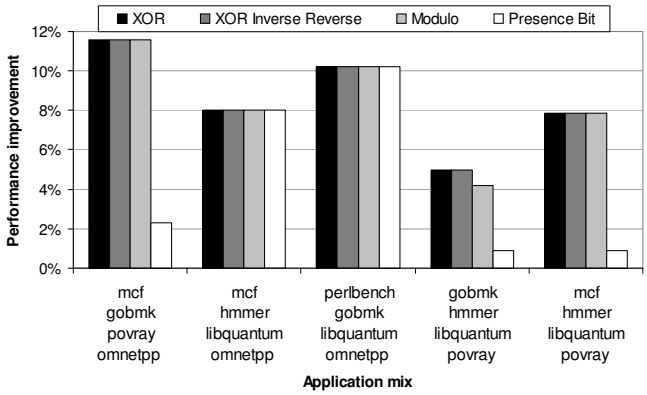


Figure 15: Comparing Different Hash Functions

Figure 14 shows the relative performance improvement of a few representative benchmark mix for three algorithms we proposed: the weight sorting algorithm, the interference graph algorithm, and the weighted interference graph algorithm. Interestingly, the weight sorting algorithm, in spite it is simple, 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 cache. We can also observe that the weighted interference graph mechanism achieved as good or the better performance among three. This result is not surprising as the weighted algorithm considers both the interference metrics (or symbiosis) and the occupancy weights.

5.3 Comparison of Different Hash Functions

One important design criterion in our proposed architecture is choosing a suitable hash function for the Bloom filters. In addition to minimizing collisions, a low-complexity hash function is more desirable as it will be implemented inside our proposed hardware. Figure 15 shows a few representative mix of benchmarks showing the relative performance improvements for different hash functions. We four hash functions for our evaluation:

- XOR: The block address is divided into equal chunks of hash index long and they are bitwise-XORed to obtain a single 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

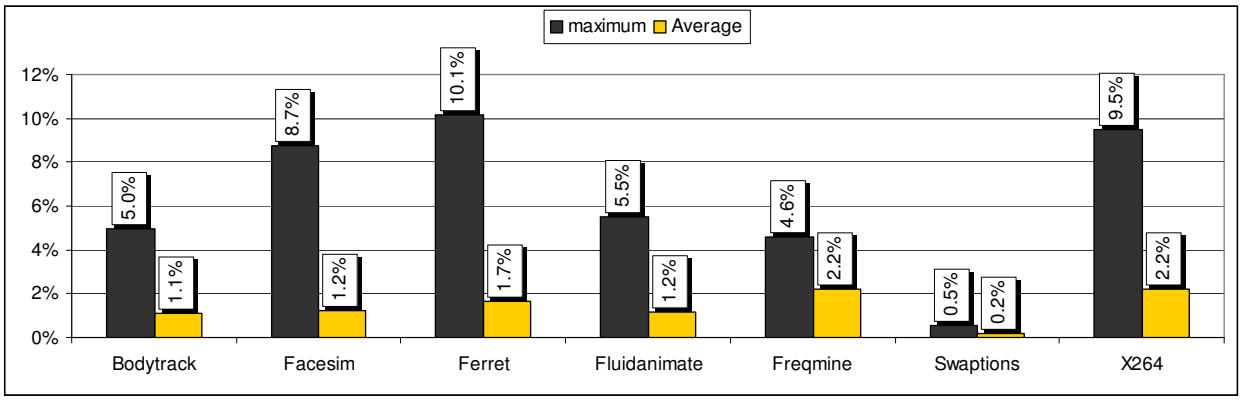


Figure 13: Maximum Performance Improvement for Multithreaded PARSEC (Native run on Intel Core 2 Duo)

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 the Modulo hash function performs slightly worse. In general, XOR-based hash will be sufficient in terms of performance and hardware cost.

As explained, he 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 15. 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 assigns processes to processor cores, it will not affect Linux's scheduling algorithm of using distributed queues for a multicore system.

5.4 Comparison with CacheScouts

There have been several research papers in the area of fair cache sharing. Notable among them is *CacheScouts* by Zhao *et al.* [40]. They proposed to tag cache lines with software guided monitoring IDs to track cache capacity, and perform occupancy and interference analysis to perform resource allocation. We compare the performance improvement from their resource allocation technique with ours in Figure 16. Note that, our results are actual run time improvement measured on a real Intel Core D2 Duo machine while theirs were based on simulations that resemble these machine classes. We can see that except for the eon benchmark, our technique does significantly better

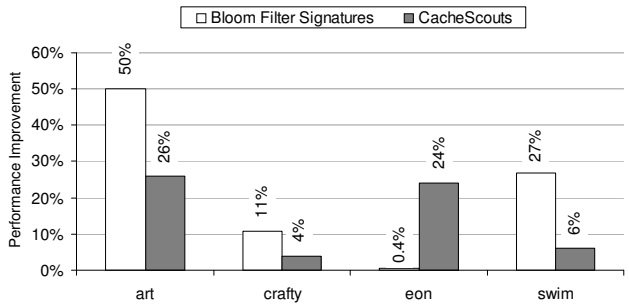


Figure 16: Comparison with CacheScouts

in all other benchmarks. For example, we get a best-case performance improvement of over 50% using our algorithm for art compared to CacheScouts 26%. We find similar trends in the swim and crafty benchmarks.[2]

There are a couple of reasons why our technique does significantly better than CacheScouts with comparable overheads. The first reason is that CacheScouts uses the tags of individual lines for their analysis. The use of tags is similar to using presence bits. We have demonstrated in Figure 15 that using presence bits gives very little performance benefit. Our technique uses Counting Bloom Filters that provide more accurate information of the address space (working set or footprint) used by the process. The second and more pertinent reason is that CachScouts only uses a simple sorting algorithm for performing Capacity/Interference based scheduling. In Figure 14 we also demonstrated that our weighted interference graph algorithm can outperform a simple sorting algorithm.

5.5 Implementation Overheads

We consider two different aspects of the overheads for this technique. The first is the software overhead. This 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

[2]We did the comparison for only four applications because, the CacheScouts paper only reports performance improvement for these applications while we did have more runtime results for other benchmark programs.

100 ms in a giga-hertz processor, this performance overhead for a fast heuristic algorithm is also negligible.

Another aspect of the overhead is that of 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 or exactly 16 bus transfers on a 64 byte wide bus with every context switch, for roughly once every 2-3 billion cycles.

The most important 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 cache 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 [24]. We perform 25% sampling of data sets and find that the correctness of our algorithm is not affected by the sampling for the benchmarks we considered. Thus our total overhead can be boiled down to only about 2.13% of the L2 size. This space overhead is comparable to prior art, e.g., the overhead of CacheScouts which stated that their overhead is of the order of 2% of the L2.

6. RELATED WORK

Some recent work has focused on the impact of L2 cache space partitioning on overall system performance are [6, 13, 14, 15, 16, 17, 25, 26, 34, 35, 39]. All L2 cache partitioning papers suffered from the problem that they need to change the interface of the normal caching mechanism. For example, [17] and [25] change the replacement policy for partitioning cache space, while [32] adaptively dedicates sets to processors to enable improved shared L2 cache behavior. [39] also modifies the LRU policy to implement pseudo partitioning in shared cache multicore processors. [6] uses multiple time sharing cache partitions that increase throughput of individual threads by affecting fairness. This paper first calculates the L2 miss rates for each thread, estimates IPC with expanded and reduced capacities and then enforces multiple time sharing cache partitions to increase the throughput. [34] uses an analytical model to predict miss rate, calculates the marginal gain of adding capacity to each process and then comes up with an adequate cache partition for each process and modifies the cache replacement policy to enforce the allocation of a specified number of cache blocks for a particular process. In [16], the TADIP scheme changes the LRU policy to improve cache sharing. In [35], the authors propose using N counters (for an N-way cache) per running thread to calculate the relative footprint. Since modern multitasking systems have many running threads, realizing such a set of hardware counters will be less practical.

Other cache partitioning methods require software layers to explicitly specify the application requirements for cache capacity and bandwidth, and provide mechanisms in hardware to uphold guarantees [21, 22]. The architectural support for this research does not change the normal cache functionality or require direct specification of requirements for virtual resources. It uses lightweight monitoring to do resource allocation at the user level. This less invasive approach makes this work more implementable. However, should a proper cache partitioning scheme exist in the architecture, our technique can be used along with the partitioning scheme. The information that different cores access different cache-partitions can be used to optimize process to core mapping.The dynamic cache-partitioning can then optimize hit-rates after the mapping is done.

Researchers have also looked at some metrics like cache affinity for resource allocation [12, 11, 31, 36]. [18] leverages the OS to capture misses per cycle and cache access information on a per thread basis and applies scheduling policies to improve performance. However, these techniques have been shown to be not very good indicators for resource allocation because they lack a good hardware infrastructure. Previous work has illustrated the ability to effectively manage applications based upon signatures provided in hardware [29, 37]. Therefore, we adopt the use of bloom filter based hardware signatures based upon architectural extensions to enable intelligent software level management and resource allocation.

In [23], a Spill-Receive architecture was proposed for managing CMPs with private L2 Caches. In this architecture, an L2 cache can either spill or receive cache lines to/from neighboring caches. With the dynamic spill/receive architecture, the policy dynamically chooses the best spill receive combination. Settle *et al.* [28] uses an activity-factor as a metric to schedule threads in an SMT platform. One advantage of our technique over Settle's is that ours is OS-agnostic, while Settle's scheme requires to rewrite the OS scheduler. Though the hardware mechanism is similar, the resource allocation algorithms are completely different. We also consider resource allocation for multithreaded benchmarks, not considered in Settle's scheme. Another technique for scheduling for SM was proposed by Snavely *et al.* [30]. They compute the diversity, balance, and resource conflicts between competing possible schedules in SMT and performs symbiotic scheduling to achieve maximum throughput, which is drastically different from our use of memory footprint signature.

Dhodapkar *et al.* [8] presented several algorithms that dynamically collect and analyze program working set information to configure hardware resources like the instruction cache. Though the manner of collecting *Working Set Signatures* is similar in principle, their paper is for a completely different purpose, to detect phase changes for architecture tuning, and has nothing nor any algorithms to do with process scheduling. Also note that this paper does not assume that any architecture resource can be changed.

7. CONCLUSIONS

Demands for performance and manageability in modern computing systems have led to two technological trends: first, to employ a shared-cache multi-core architecture for exploiting thread-level parallelism, and second, to provide system support for virtualization to improve manageability of workloads on large-scale systems. Due to resource sharing (e.g., the last level cache) on these emerging platforms, the overall performance could be degraded for lacking a more informed and intelligent resource allocation policy even though they were designed to improve the throughput by exploiting concurrency.

In this paper, we proposed an architectural support to mitigate the interference caused by L2 cache contention. We also proposed and studied three resource allocation algorithms with a goal of minimizing interference and finding 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 their mutual symbiosis dynamically to minimize destructive performance side effect when simultaneous processes are running on multiple cores that share cache.

Unlike previous studies completely relied on simulators and their own improvement metrics, we validate our performance results by executing our new process-core mapping on an Intel Core 2 Duo machine and report the run time speedup. Our experiments were done in two phases. First, we collected the dynamic signature information from Simics runs, and ran them through our 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. Our run time results showed an improvement of up to 54% (or 22% on average) can be achieved when the applications are run natively. When running inside VMs, the speedup is up to 26% (or 9.5% on average). Given 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 in our technique will become more critical to warrant maximal performance on a given compute capacity.

8. REFERENCES

[1] Semidefinite Programming Solver. http://www.stanford.edu/ yyye/Col.html.

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

[3] Virtutech Simics. http://www.simics.net.

[4] 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 (SOSP)*, 2003.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. 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] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. pages 111–122, Sept. 2004.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

[24] M. Qureshi, M. Suleman, and Y. Patt. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. *Proceedings of 13th International Symposium on High Performance Computer Architecture*, 2007.

[25] 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.

[26] 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.

[27] 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.

[28] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural Support for Enhanced SMT Job Scheduling. In *PACT: Proceedings of the 13 th International Conference on Parallel Architectures and Compilation Techniques*, volume 29, pages 63–73, 2004.

[29] K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart, and X. Zhang. Hardware counter driven on-the-fly request signatures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

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

[31] 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.

[32] 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.

[33] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference*, 2001.

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

[35] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, 2002.

[36] 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.

[37] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas. Softsig: Software-exposed hardware signatures for code analysis and optimization. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[38] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 26–40, 1991.

[39] Y. Xie and G. Loh. Pipp: Promotion/insertation pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36nd Annual International Symposium on Computer Architecture*, June 2009.

[40] 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.