Architectural Support for Dynamic Resource Allocation on Multi-core Platforms Using Bloom Filter Signatures

Abstract

The landscape of modern enterprise computing systems has been dramatically impacted by recent technological innovations. Advances in fabrication technology has enabled an exponential increase in the number of transistors per chip, bringing the microprocessor industry to the many-core era. A major challenge for hardware and software designers is to efficiently use these multiple cores to extend computing performance, improve reliability and security within a given power envelope. At the same time, advances in virtualization support allow for data-center applications to efficiently utilize these processing elements while maintaining fault isolation and increasing manageability. This paper considers computing platforms in which it is beneficial to execute large numbers of applications on a Multi-core system for power management or resource consolidation purposes. For such platforms, it provides support and methods for allocation and scheduling of applications onto physical cores, in order to minimize negative caching effects, specifically considering shared cache architectures. A typical example of such a deployment is a system running many virtual machines (VMs) on a multi-core processor. To facilitate management of computing resources in such platforms, we propose *Bloom Filter Signatures* (BFS), which constitute lightweight hardware extensions that allow software layer allocation algorithms to infer cache footprint characteristics and interference of applications or VMs on multi-core platforms. The overall goal is to improve system performance. Bloom filter signatures rely on hardware-level counting Bloom filters that efficiently summarize cache usage behavior, and provide interfaces that allow the OS to track application specific caching information. In the case of virtualization, the interface allows management and scheduling software within the Virtual Machine Monitor (VMM) to obtain VM specific caching information. Our evaluation of the collaborative BFS approach to cache interference-aware resource allocation is shown to provide application performance benefits of up to 54% in 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. We did a performance comparison with another fair cache sharing scheme called CacheScouts. We demonstrated significant performance improvement for three out of the four benchmarks whose improvement have been reported by CacheScouts.

1 Introduction

The ever increasing demands for performance and manageability in modern enterprise computing systems has directly affected recent innovations in both computer architecture and systems design. To address the performance criteria of enterprise workloads, earlier generations of server platforms provided high computational performance with a combination of architectural techniques and increased processor frequencies. The associated power and thermal issues of such approaches, however, have limited the scalability of improving performance in this manner. In response, modern platforms are providing increased performance by leveraging thread level parallelism via chip multiprocesosrs or multi-core architectures. Enterprise applications that, for example, process independent web requests can easily attain performance benefits with these emerging architectures.

A second, and equally important, requirement of enterprise systems is the ability to provide flexible and efficient use of resources via software management capabilities. Industry has responded to this need with the integration of hardware [16] and software solutions (e.g., Xen [4] or VMware [27]) for virtualization. Virtualization allows application components to obtain benefits such as fault [4] and

performance isolation [15].

Given the context of these two converging technologies, it is critical to be able 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 [25] as well as thrashing in shared cache [9] 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 negative caching effects of multiple applications running simultaneously on a multi-core processor. As a special case, we show how these implications extend to the virtualized case where multiple virtual CPUs (`vcpus`) are mapped to sets of physical processing units, e.g., cores sharing a physical cache. We highlight two problems that must be addressed in order to improve performance in this allocation decision, and propose hardware and software solutions towards that end.

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 extend the use of per-core counting Bloom filters to enable online profiling of cache usage behavior in hardware. The second problem, then, is that of determining the interactions between workloads based upon observed Bloom filter data. We project such interactions using hardware support for comparing *Bloom filter signatures* of workloads. This information is then provided to software policy 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 monitoring user 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 resource to physical resource mappings to improve system performance, while still providing some means of fairness across workloads. Our evaluation of these proposed solutions highlight the benefits achievable with 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, including its relevant hardware and software components, in Section 3. A description of our experimental methodology and implementation that allow us to evaluate this design appears in Section 4. Section 5 contains our results and discussion. Finally, we summarize related work in Section 6, and conclude with final comments in Section 7.

2 Motivation

2.1 Resource Allocation in Multi-core Processors

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

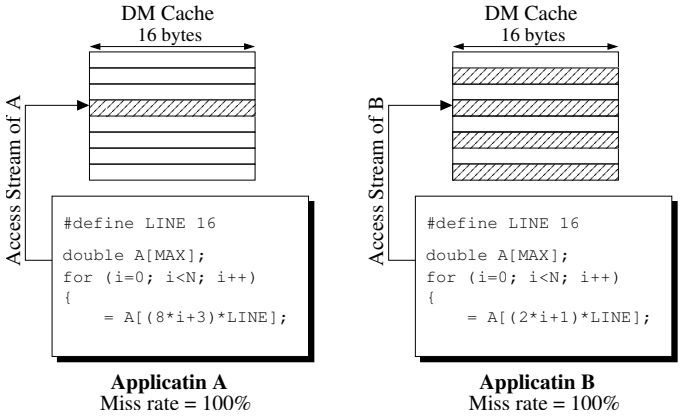


Figure 1: Different Cache Footprints with the Same Miss Rate

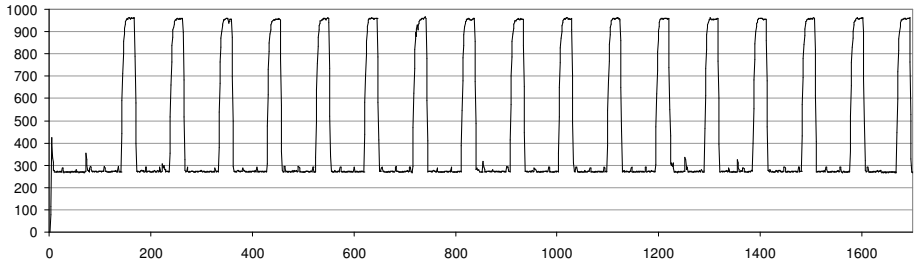
paper, we particularly consider improving the allocation decisions when mapping physical resources like a shared last level cache to virtual CPUs that run guest VMs. An important goal of resource allocation is to determine allocations that maximize performance by minimizing the types of negative caching effects described next.

2.2 Hardware Support for Multi-core Resource Management

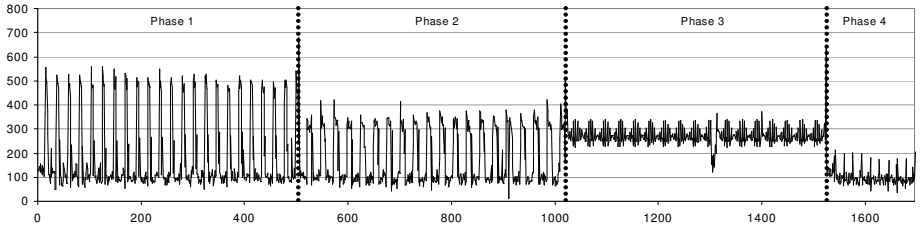
A large body of work has addressed cache affinity scheduling in shared memory multiprocessors. Devakonda *et al.* [5] showed the effects of cache affinity scheduling and also state that affinity scheduling works well only for a certain class of applications. This observation is substantiated by Salehi in [22] that showed that affinity scheduling improves throughput of certain network protocol processing applications. Furthermore, in [30], 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 cache performance of each application drastically. For instance, when multiple applications compete for a shared state-based resource like the cache, certain applications can have a destructive effect on the cache state of other applications. In response, there has been research to aid scheduling decisions based on the miss rate of caches [8]. However, predicting incompatibilities between running applications cannot be accurately done using performance metering based approaches like counting 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 workshing 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 accumulated L2 misses over the same time windows



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



(b) Accumulated L2 Misses

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

of 100ms and 200ms, respectively. The figure clearly demonstrates that the benchmark exhibits four phases of execution. In constrast, only two periodic phases are visible 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 [10] 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 Incompatibilities

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.

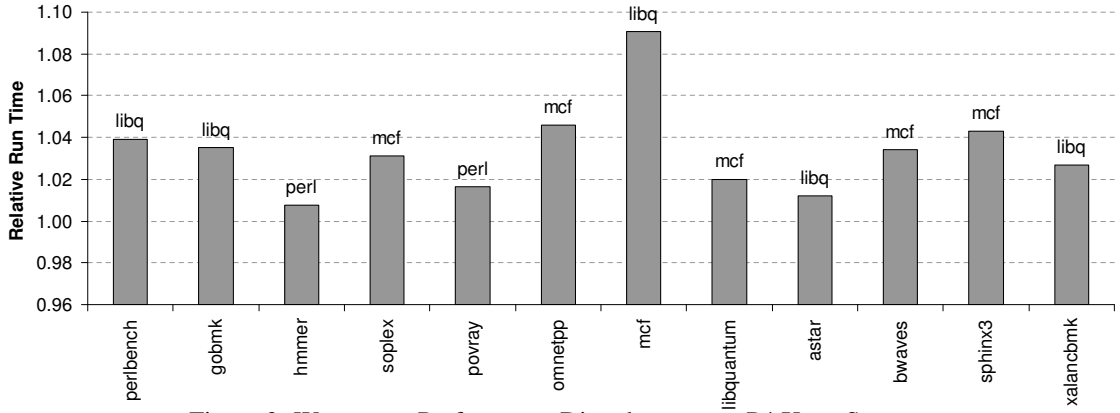


Figure 3: Worst-case Performance Disturbance on a P4 Xeon System

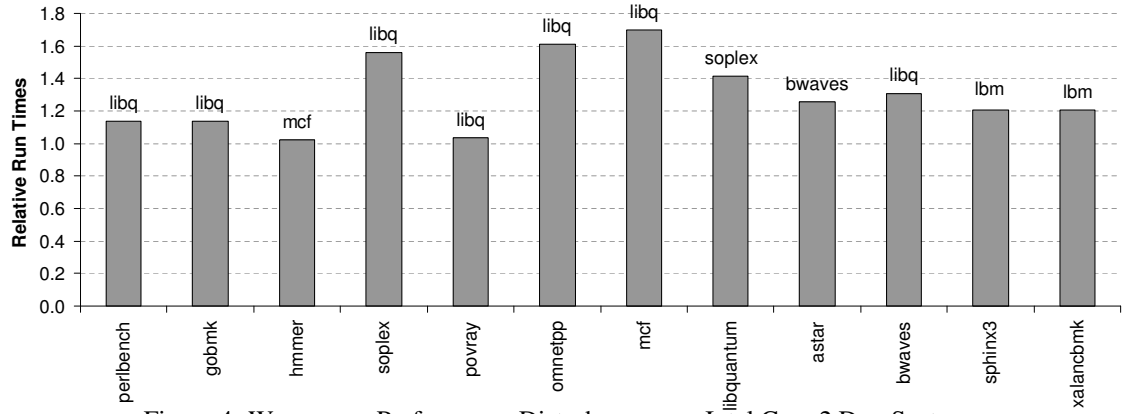


Figure 4: Worse-case Performance Disturbance on a Intel Core 2 Duo System

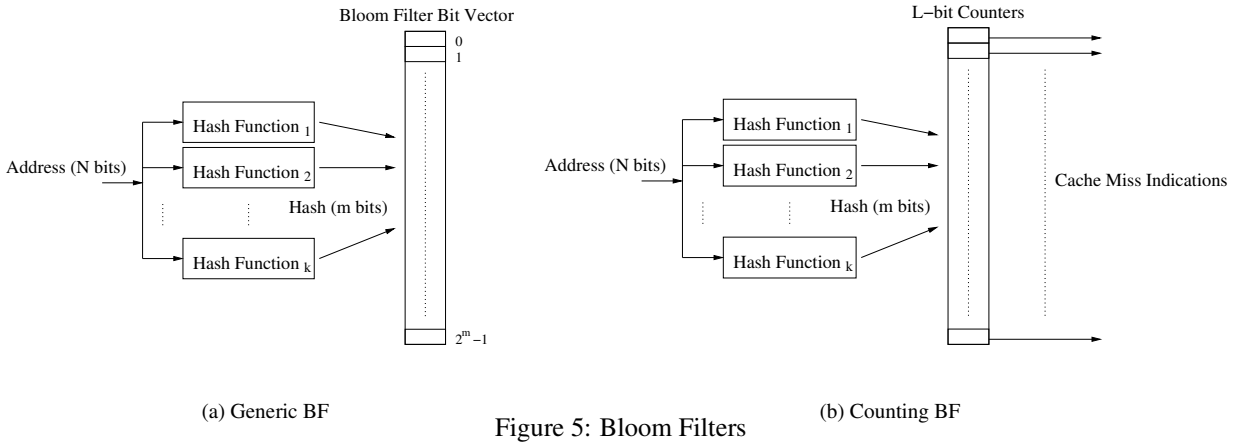
2.3.1 P4 Xeon SMP System

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 inteference 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

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 main-



(a) Generic BF

(b) Counting BF

Figure 5: Bloom Filters

tain 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 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) [7] 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 observed for addition to the Bloom filter, each *m*-bit hash index addresses to a specific counter in an *L*-bit counter

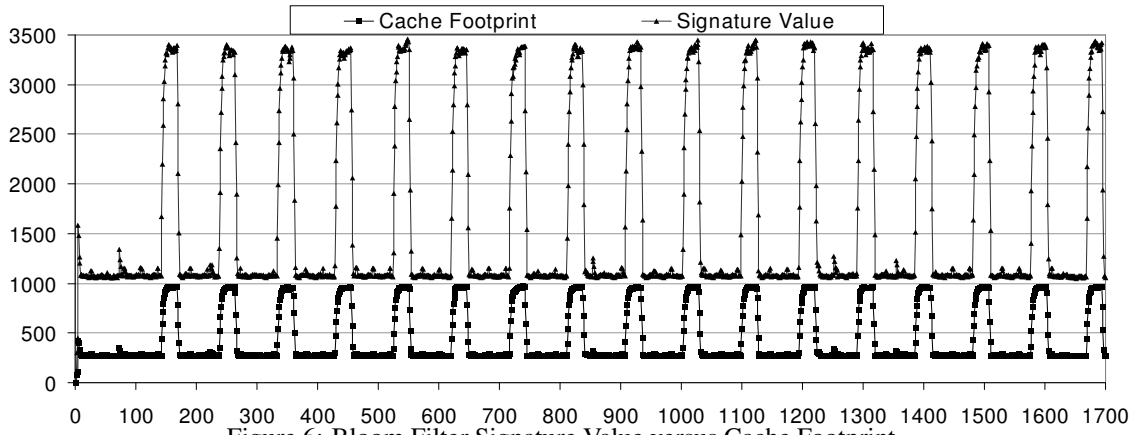


Figure 6: Bloom Filter Signature Value versus Cache Footprint

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 data 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.

3 System Design

3.1 Multi-Core Enhanced with Bloom Filter Signature

This section describes the CBF-based infrastructure that enables the OS to efficiently allocate computing resources for applications in a multi-core platform. The architecture is illustrated in Figure 7 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 multuple 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.

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

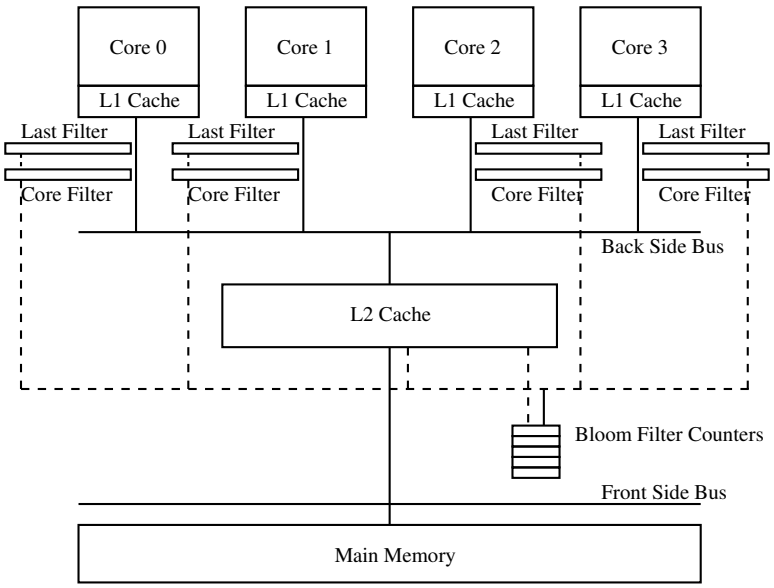


Figure 7: Multi-Core Enhanced with Working Set Signature

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 is described as follows. For every miss in the L2, 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 cache 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 one-to-one 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 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 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. 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). 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 an *interference metric* to be the sum total of the number of ones in the bit vector obtained by XORing the CF and the RBV. A high value of the interference metric indicates less inteference. A low value either means higher inteference, or that both vectors have a very low occupancy. These metrics of occupancy and interference 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 inteference. Note that we keep only the occupancy and interference

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 *interference metrics* and *occupancy weight* 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 *occpancy weight* and *interference metric* 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 architecture described above. In particular, for each interesting 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 the application is context switched from execution on a core, the data structure associated with the application is updated. The first entry of the structure is set to be the core ID of the last physical processor allocated to the application. The second entry is updated with the *occupancy weight*. The remainder of the N entries are used to store *interference metrics*.

While the OS handles utilizing underlying hardware support for updating interference 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 interfaces. It is important to 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 reponsible 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 in order 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 in order to obtain an updated allocation decision that is then passed along to the hypervisor using existing control interfaces.

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. The rationale of doing this, as explained in Section 2.3.1, is that the processes assigned to the same core will never execute together and will not affect each others' performance. The quantitative effects of the latter argument has been illustrated in Section 2.3.2. In this section we describe three algorithms we developed for this purpose.

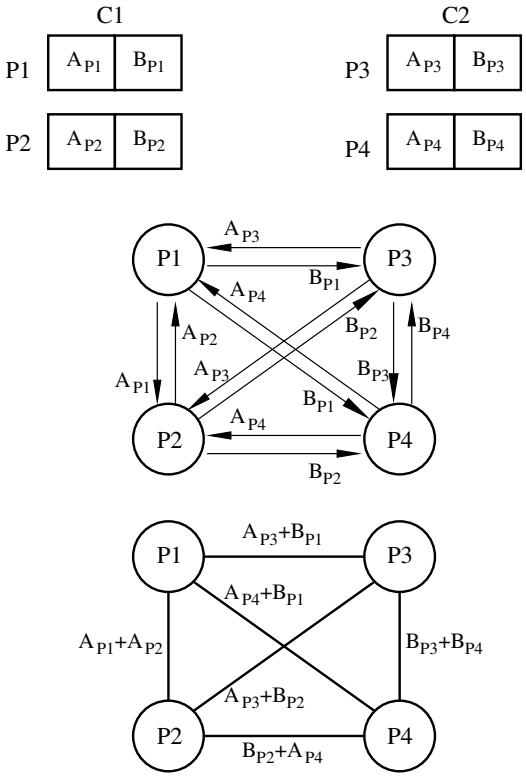


Figure 8: Forming an Interference Graph

3.3.1 Simple Sorting Algorithm

The sorting algorithm uses a very simple mechanism to detect the L2 cache occupancy of each process. The only metric used for this algorithm is the sum of the bits contained in the Running Bit Vector. This metric gives a reasonable idea of the cache footprint of a process. Upon every context switch, the Running Bit Vector is computed as a difference between the Core Filter and the Last Filter of the core where the process was running. After computing the RBV, the 1's of the bitvector are added and the weight is communicated and 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 *P/N*, where the group size determines the number of processes scheduled to one core. After sorting the algorithm, it simply forms groups of processes in their sorted order. The rationale behind this algorithm is that the process having a larger weight is more likely to affect performance of other processes. Thus, if the most *heavy* processes are in a group, they should not be running together, thereby their effect on other processes will be reduced significantly.

3.3.2 Interference Graph Algorithm

The Interference Graph algorithm is explained with the help of the example illustrated in Figure 8. In this example, we have a dual-core machine with cores C1 and C2. We need to schedule 4 processes: P1 through P4. The figure shows the interference metrics of each process when the algorithm is invoked. The interference metrics are simply the number of 1's obtained by XORing the Running Bit Vector with each Core Filter. For example, considering process P3 in the figure. The interference metrics A_{P3} and B_{P3} are obtained by counting the 1's in the bitvector obtained after XORing the Core Filter of each core with the Running Bit Vector of P3 when P3 was context switched out of C2.

Using the interference metrics, the interference graph is constructed as illustrated in 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 B_{P1}, because it is the interference of the process P1 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 to 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.

As discussed in Section 3, a lower value of the interference metric implies a high interference between processes. Thus the objective of this algorithm is to partition the graph into two groups such that the weights of edges within a group is minimized. Minimizing weights of edges within a group ensures that the processes will be allocated to the same core and thus won't 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 are maximized. This problem is better known as the MAX-CUT problem. Although a generic solution to the MAX-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 [1] to solve the problem.

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

3.3.3 Weighted Interference Graph Algorithm

The interference graph algorithm had one impediment. As explained earlier in Section 3, a lower value of the interference metric signifies a higher interference. However, this statement may not be always true. If the *weight* of the bitvectors whose interference being calculated is small, then the interference metric will come out to be small, but that does not imply that the two vectors heavily interfere. Therefore, we came up with a 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 divide the result obtained by the weight of the node. Therefore, the weight of the edge connecting nodes P_1 and P_3 will be $A_{P3}/W_{P3} + B_{P1}/W_{P1}$, where W_{P1} and W_{P3} are the weights of nodes P1 and P3.

Using this metric will ensure that if a node has a large weight, it will have a very low interference metric, making the algorithm perform more efficiently. We show in Section 5.4 that this is indeed the case.

3.3.4 Resource Allocation for Multithreaded Applications

The resource allocation algorithms described earlier must be adapted to be capable of resource allocation for mutithreaded applications. To enable proper resource allocation for multithreaded applications, we must collect occupancy weight and interference metric statistics at the thread granularity. For multithreaded applications, threads of the same application often heavily share data. Therefore, if we compute interference metrics between threads of the same process, we will get a very high interference value. This is misleading, as in this case the threads are actually sharing data rather than interfering with each other. To overcome this challenge, we need to adapt our resource allocation algorithms to perform resource allocation in two phases.

In the first phase, we consider multithreaded processes in isolation and perform resource allocation for threads of individual processes. Since the interference metric is not meaningful for threads of the same process, we use the occupancy weight sorting algorithm to decide for a given process which threads will be allocated to the same core.

After this decision is made for each multithreaded process, we begin the second phase of the resource allocation algorithm by forming the interference graph. In the second phase we perform the weighted interference graph algorithm as explained earlier at a

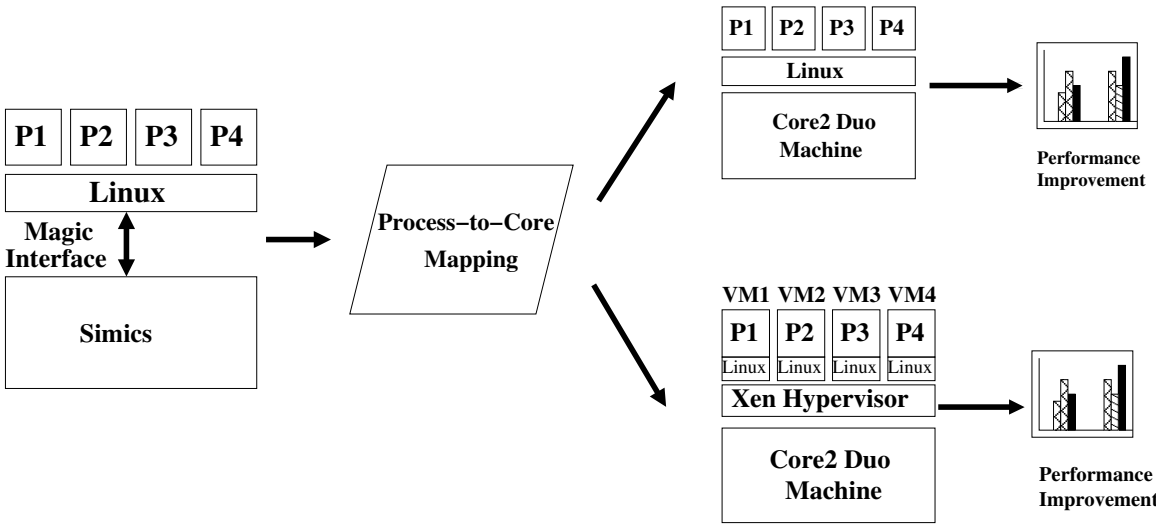


Figure 9: Evaluation Methodology

thread granularity. We use the results of phase one by making edge weights of threads that are allocated on the same core to zero. Similarly, we set the edge weights of threads that will be on different cores to have a very high value. The other edge weights are set according to the weighted interference graph algorithm. Pre-setting edge weights ensure that the Max Cut of the graph will always put threads that have zero edge weight on the same core and threads that have a very large edge weight will be allocated to different cores.

4 Evaluation Methodology

We conduct our experiments in two phases as shown in Figure 9. The first phase constitutes the simulation phase. The second phase is the experiment phase. We explain each of the phases in the following subsections.

4.1 Simulation Infrastructure

In this phase, we use Simics [3] to execute benchmark applications. Simics is a full system emulator that can run unmodified production software like full blown operating systems. 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 in this infrastructure has been implemented using Simics magic instructions. The Linux 2.6.16.33 kernel has been modified to incorporate calls to the special magic instruction during interesting context switches. We define interesting context switches as those context switches where an application of interest is context switched out by the OS. The applications of interest for which this scheduling is being done are specified by the user. In our simulations we use linux's proc interface to let the kernel know the PIDs of processes that are of interest. When a magic instruction is executed by the Simics simulator, it passes the control to the Simics magic handler. This magic handler has been modified to pass Bloom Filter Signatures. The Simics g-cache module has been modified to implement the Bloom Filter Signatures infrastructure. A kernel module reads these signatures and keeps them as a part of the process context. The Linux kernel has been 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 assigns the process to a particular processor core. The algorithms used to perform this allocation using interference information obtained from the process context, is explained in Section 3.3.

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

The simulation phase is responsible for providing a process-to-core mapping for a certain set of processes that maximizes system performance. This mapping is used in the experiment phase. The simulations were run 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 Experiments

We perform two sets of experiments. The first involved running sets of four benchmarks simultaneously on a Fedora Core Linux OS running on a 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 of benchmarks. We explain how we arrive at these two numbers with the help of 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 dual-core 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 tabulated in Table 1.

We should note that there are only three possible mappings for 4 processes running on a dual-core machine as shown in Table 1. Once we obtain this table, we look at our results obtained in the simulation phase and find that during our simulation the mapping (AD and BC) was preferred by our resource allocation algorithm for majority of the simulation time. We find from our results that benchmarks *Gobmk* and *Libquantum* have significant performance improvements for the chosen schedule while any schedule has no 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 applications. The benchmarks were chosen to have a diverse mix, the performance of some were affected significantly by other benchmarks; 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 in the pool of twelve SPEC benchmarks. It should be noted that due to limitations and scalability issues of executing virtualization solutions such as Xen in Simics, our simulation was performed using process based encapsulation of workloads instead of virtual machines. Our results (mapping of VMs to cores), though,

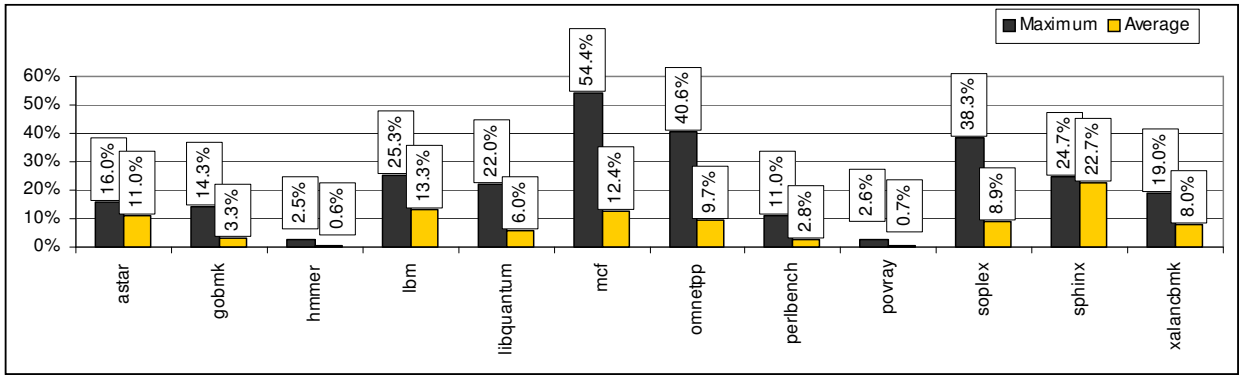


Figure 10: Maximum Performance Improvement for Each Benchmark in Native Machine

map directly to scenarios where workload processes are executed in independent VMs as opposed to a single operating system. Using this approach, the hypervisor functionality described in Section 3 is incorporated into the host kernel via modifications to the operating system 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 the *PARSEC* benchmark suite. 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 Experimental Results and Analysis

5.1 Performance Improvement

Figure 10 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 machine as explained in Section 4.2. We can see that our technique shows significant improvement for those heavily exercising the L2 cache like 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 like povray, gobmk and astar that are mainly compute-bound and do not depend much on the L2. The maximum improvement for gobmk is 14%. The second type of benchmarks are those memory-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. For being memory-bound and showing low locality the maximum performance benefit obtained for hmmer is only 2.5%.

Figure 10 also illustrates the average performance gain using the weighted interference graph algorithm obtained for each benchmark across all the mixes of benchmarks. We see that the trend of this figure follows the one for maximum performance. However, except for a few benchmarks the average performance is substantially reduced from the maximum improvement. The most significant average improvement is 22% for the xalancbmk. In the following subsection we report the performance improvements of different different benchmarks when encapsulated by VMs.

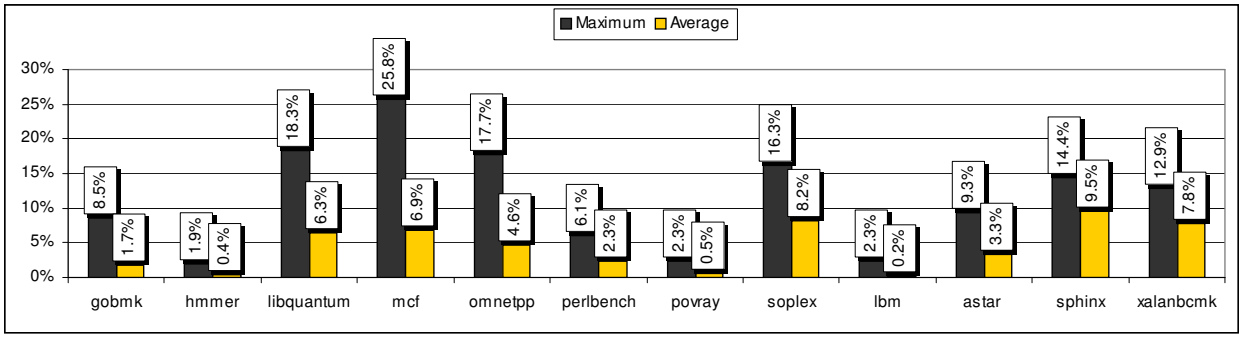


Figure 11: Performance Improvement for Each Benchmark Running in VMs

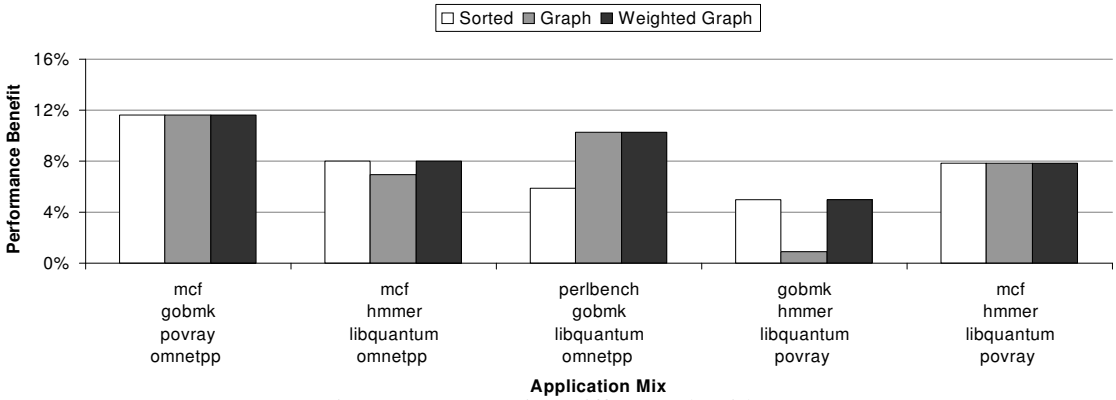


Figure 12: Comparing Different Algorithms

5.1.1 Performance Improvement for VMs

Figure 11 reports the maximum and average performance benefit obtained per benchmark running inside VMs using our weighted interference graph resource allocation algorithms. We can observe that the maximum and average performance improvement of benchmarks running inside VMs is significantly lower than if they were running on a native machine. For example, the maximum performance improvement for *mcf* is 26 % when running inside VMs, whereas it is 54 % when running on the native machine. One of the reasons for this significant reduction in relative performance improvement is the virtualization overhead of the benchmarks. However, even though the performance improvement has reduced, the relative trend of improvements among benchmarks remains the same. This shows that even though the benchmarks are encapsulated inside VMs, the negative caching effects of each of the benchmarks still have similar effects on the performance of other benchmarks.

5.2 Comparison of Different Resource Allocation Algorithms

Figure 12 shows a few representative mix of benchmarks showing the relative performance improvements for the simple sorting algorithm, the interference graph algorithm and the weighted interference graph algorithm. We were surprised to find that the simple sorting algorithm often gave the best results, as we see is the case with four of the five mixes shown in the figure. This tells us that the cache footprint is a very good metric for predicting performance effects on other processes/VMs sharing the cache resource. We can also see in Figure 12, that the weighted interference graph mechanism always does as good as the better of the sorting and the graph algorithm. This result is intuitive, as the weighted algorithm considers both the interference metrics and the occupancy weights. We believe that if we consider a larger size of groups consisting of six or eight benchmarks, the weighted interference algorithm will perform much better than the other two algorithms.

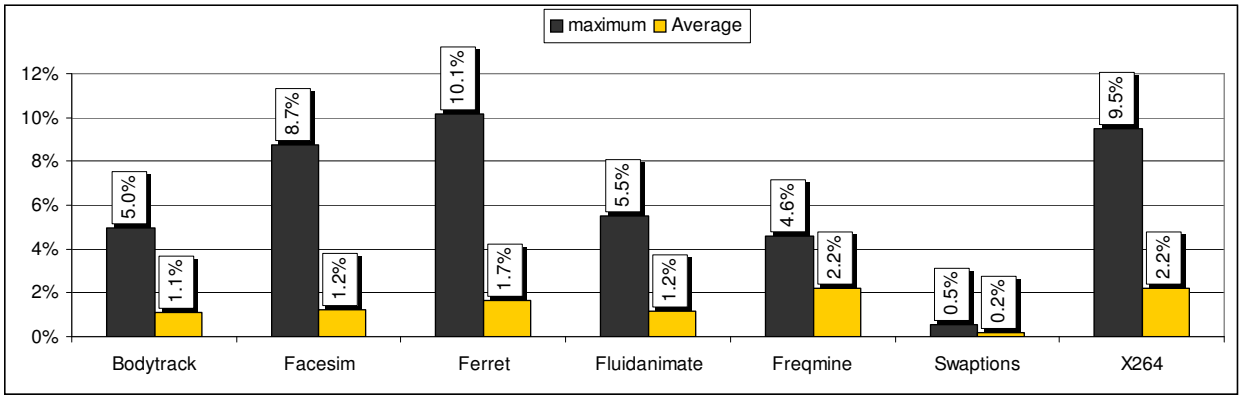


Figure 13: Performance Improvement with Multi-Threaded Workloads (PARSEC)

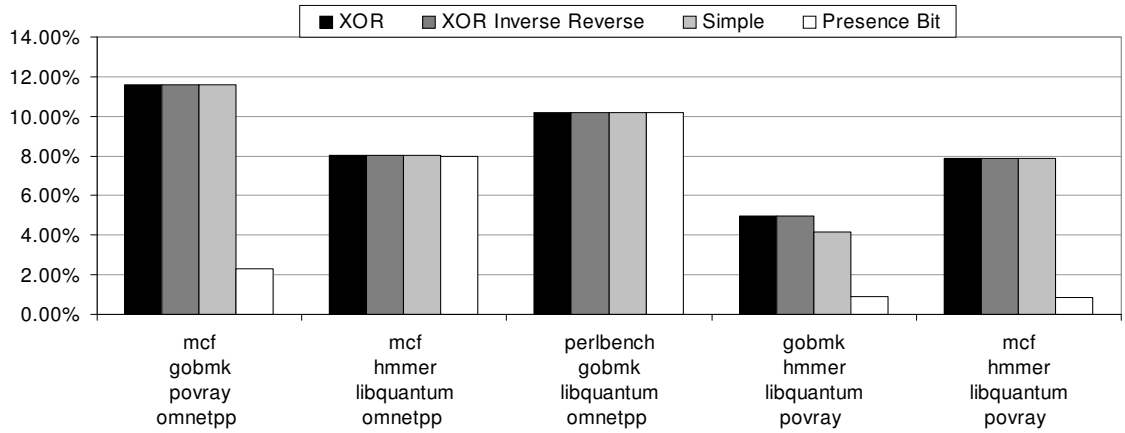


Figure 14: Comparing Different Hash Functions

5.3 Performance Improvement for Multithreaded Benchmarks

Figure 13 shows the maximum and average performance improvements for the multithreaded *PARSEC* benchmark. As with 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 the single threaded ones since the memory working set of *SPEC2006* is much larger than the *PARSEC* which focuses more on compute-bound workloads.

5.4 Comparison of Different Hash Functions

An important design criterion in designing the architecture was choosing a suitable hash function for the Bloom filters. Figure 14 shows a few representative mix of benchmarks showing the relative performance improvements for different hash functions. We chose three hash functions for our experiments:

- XOR: The block address is divided into several 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.
- Simple: This is a simple modulo operation of the block address with the Bloom filter size.

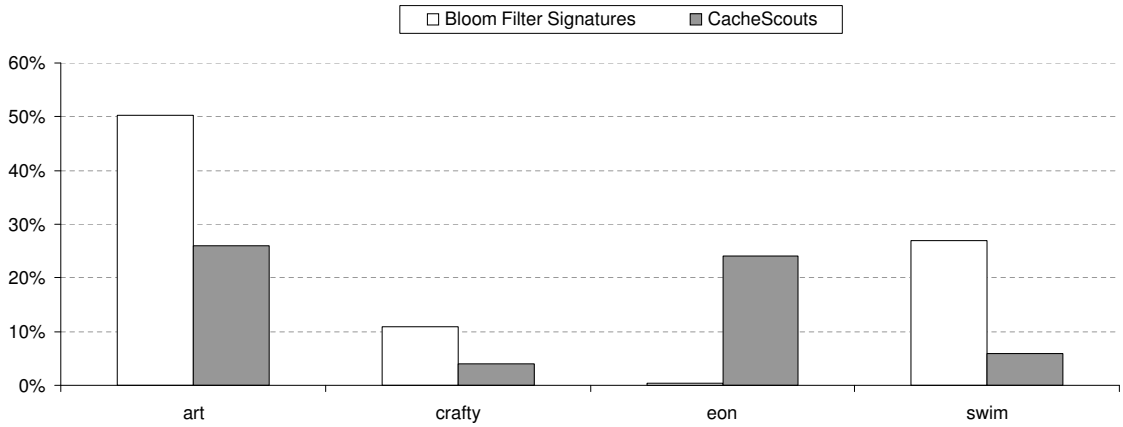


Figure 15: Comparison with CacheScouts

We found that for almost all the mixes the three hash functions perform identically. One exception is the mix with gobmk, hmmer, libquantum and povray where the Simple hash function performs marginally poorer. This shows that any of the above sufficiently random hash function is appropriate for implementing Bloom filter signatures.

Another interesting experiment performed was using presence bits. 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 illustrated in the fourth bar of Figure 14. We found that presence bit vectors have no effect on scheduling decisions. All the results have illustrated 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 was 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 very little information. This is exactly the same reason why we do not use multiple hash functions. Multiple hash functions set multiple bits in the bit vector for a single address entering the cache. This will saturate the Bloom filter and the technique will become ineffective just as in the case of presence bit vectors. Multiple hash functions would have been effective if we did not have a strict harware 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.

5.5 Comparison with CacheScouts

There have been several research papers in the area of fair cache sharing. Notable among them is by Li Zhao *et al.*, called CacheScouts [32]. This paper proposed to tag cache lines with software guided MIDs 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 15. We can see that except for the eon benchmark, our technique does significantly better 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]

[2]We did the comparison for only four applications because, the CacheScouts paper only reports performance improvement for these applications

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 benefits. Our technique uses CBFs that provide more accurate information of the address space used by the process. The second and more pertinent reason is that CachScouts only uses the simple sorting algorithm for performing Capacity/Interference based scheduling. In Figure 14 we demonstrate that our weighted interference graph algorithm performs better than a simple sorting algorithm.

5.6 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 minimal as 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 giga-hertz processor, this performance overhead for a fast heuristic algorithm is negligible.

Another aspect of the overhead is that of computing the weight and interference metrics for every context switch. If we assume that we have 1024 XOR gates for the computation, the operation will take less than 10 cycles. We must also 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, roughly once every 2-3 billion cycles. The most important aspect of overhead is the hardware overhead. 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 a *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. However this overhead is inordinately large in terms of real estate for the chip. 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 [19]. 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 comes down to only about 2.13% of the L2 cache size. This space overhead is comparable to the overhead of CacheScouts which states that their overhead is of the order of 2% of the L2 Cache size.

6 Related Work

Some papers that have focussed on the impact of L2 cache space partitioning on overall system performance are [11, 12, 13, 20, 21, 31]. All L2 Cache partitioning papers suffer from the problem that they need to change the interface of the normal caching mechanism. For example, [13] and [20] change the replacement policy of the cache for partitioning cache space, while [26] adaptively dedicates sets to processors to enable improved shared L2 cache behavior. [31] also modifies the LRU policy to implement pseudo partitioning in shared cache multicore processors. Other methods require software layers to explicitly specify application requirements for cache capacity and bandwidth, and provide mechanisms in hardware to uphold guarantees [17, 18]. 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 patitioning 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 [28][10] have also looked at some metrics like Cache Affinity for resource allocation. [14] 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 [24, 29]. Therefore, we adopt the use of bloom filter based hardware signatures based upon architectural extensions to enable intelligent software level management and resource allocation.

Settle et al [23] uses activity-factor as a metric to schedule threads in an SMT platform. Our technique is suited for multi-core platforms and uses cache occupancy as a metric. Dhodapkar et al [6] present several algorithms that dynamically collect and analyse program working set information to configure hardware resources like the instruction cache. Though the manner of collecting Working Set Signatures is similar to this paper, their paper is for a completely different purpose, to detect phase changes for architecture tuning. 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 driven two recent technological innovations. First, emerging processors are providing scalable performance with multi-core architectures that leverage thread level parallelism. At the same time, system support for virtualization enable improved manageability of workloads deployed on these chips. In this paper we have experimentally demonstrated the types of interactions such deployments can have on multi-core systems. In particular, we find that the cache interactions between VM workloads can incur significant performance penalties. To alleviate these issues, we propose the use of intelligent resource allocation schemes that map physical resources to virtual machines in a manner that improves overall performance.

To build an effective resource allocation architecture, we propose the use of hardware extensions that allow software management policies to estimate cache usage and interference amongst workloads. Towards this end, we identify Bloom Filter Signatures (BFS) as a useful solution, and evaluate it using a hybrid setup with Simics and real experimental hardware. Our results show that by using intelligent allocation algorithms based upon the occupancy weights and interference metrics derived from Bloom filters, benefits of up to 54%, or an average of 22% can be achieved. When the benchmark programs are running inside VMs, the maximum improvement of up to 26% and average improvement of up to 9.5% is achieved.

We believe that the resource allocation architecture presented in this paper will be invaluable for emerging multi-core platforms because it leads to significant improvement in performance with minimal changes to the operating system or the hypervisor.

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] 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.

[6] A. Dhodapkar and J. Smith. Managing multi-configuration hardware via dynamic working setanalysis. *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 233–244, 2002.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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 (MICRO-40)*, December 2007.

[12] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. pages 257–266, June 2004.

[13] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. pages 111–122, Sept. 2004.

[14] 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.

[15] 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.

[16] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. In *Intel Technology Journal (http://www.intel.com/technology/itj/2006/v10i3/)*, August 2006.

[17] K. Nesbit, J. Laudon, and J. Smith. Virtual private caches. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2007.

[18] 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.

[19] M. Qureshi, M. Suleman, and Y. Patt. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 250–259, 2007.

[20] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance Runtime Mechanism to Partition Shared Caches. Dec. 2006.

[21] 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.

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

[23] 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.

[24] 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 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008.

[25] M. Squillante and E. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[26] 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 (ASPLOS)*, March 2008.

[27] 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.

[28] 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.

[29] 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 (ASPLOS)*, March 2008.

[30] 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.

[31] 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.

[32] 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 (PACT 2007)-Volume 00*, pages 339–352, 2007.