To Appear in the 34th International Symposium on Computer Architecture (ISCA 2007)

Kyle J. Nesbit, James Laudon*, and James E. Smith

University of Wisconsin – Madison
Departmnet of Electrical and Computer Engr.
{ nesbit, jes }@ece.wisc.edu

Sun Microsystems, Inc. *
James.laudon@sun.com

Virtual Private Caches

Abstract

Virtual Private Machines (VPM) provide a framework for Quality of Service (QoS) in CMP-based computer systems. VPMs incorporate microarchitecture mechanisms that allow shares of hardware resources to be allocated to executing threads. VPMs can thereby provide applications with an upper bound on execution time regardless of other thread activity. Virtual Private Caches (VPCs) are an important element of VPMs, and VPC hardware consists of two major components: the VPC Arbiters, which manage shared resource bandwidth, and the VPC Capacity Manager. Both the VPC Arbiter and VPC Capacity Manager provide minimum service guarantees that, when combined, achieve QoS within a L2 cache.

Simulation-based evaluation shows that conventional cache management policies allow concurrently executing threads to affect each other significantly. In contrast, VPCs meet their QoS performance objectives and have a fairness policy for distributing excess resources that is amenable to general-purpose multi-threaded systems. On a CMP running heterogeneous workloads, VPCs improve throughput by eliminating negative interference, i.e., VPCs improve average performance by 14% (harmonic mean of normalized IPCs) and by 25% (minimum normalized IPC).

1. Introduction

To provide the combination of efficiency and high throughput, CMP-based systems rely heavily on resource sharing, especially in the memory hierarchy. Shared resources include both bandwidth, including interconnection paths with associated buffering, and storage capacity. Although main memory capacity is managed by OS page replacement algorithms, most of the other memory resources – main memory bandwidth and cache capacities and bandwidths – are typically managed through hardware mechanisms. For these shared resources, the hardware management mechanisms affect the amount of inter-thread interference, both destructive and constructive, and, in turn, this affects the threads' performance predictability, dependability, and fairness.

In many general purpose applications, execution threads must attain a minimum level of performance, irrespective of the other threads that might happen to be running concurrently. Some desktop applications have soft real-time constraints, for example those that support multimedia such as HD videos and resource-intensive video games. Operating systems sometimes have timing requirements that are critical to the correct operation of the system, for example when handling interrupts in a timely manner. In servers supporting multiple threads on behalf of independent service subscribers, it is important that resources be shared in an

equitable manner and that tasks perform in a responsive, timely way. To support these applications and others, future CMP-based systems should provide threads with quality of service (QoS) in order to preserve performance predictability and facilitate the design of dependable applications.

Historically, QoS has been of interest in computer networking, and recently it has become the subject of computer architecture research. The ultimate objective of QoS is to provide a bound on some performance metric. For networks, the objective is to provide an upper bound on the end-to-end delay through the network. For CMP systems, the objective is to provide applications an upper bound on execution time, or, alternatively, a consistent execution time regardless of other thread activity.

QoS objectives are usually achieved through resource provisioning; that is, by allocating a certain quantity of resource(s) to a given network flow (or computation thread in our case). The goal of the research reported here is to provide microarchitecture-level *mechanisms* whereby allocation of shared cache resources can be guaranteed. On the other hand, the *policies* that determine the actual allocations are beyond our scope. That is, we assume the desired resource allocations are given to us, presumably through a combination of application and system software, and our job is to assure that the requested allocations are provided by the shared cache hardware, thereby achieving QoS objectives.

We study and evaluate resource sharing mechanisms within the general framework of *Virtual Private Machines*, and *Virtual Private Caches* in particular. These are described more fully in the following two subsections. Briefly, a virtual private machine (virtual private cache) is defined through the allocation of a portion of shared resources. The objective is that a virtual private machine (virtual private cache) should provide performance at least as good as a real private machine (real private cache) having the same resources. The effectiveness of the proposed mechanisms is demonstrated through performance simulations that compare thread performance on a shared-cache CMP with performance on an equivalently provisioned private machine.

1.1. Virtual Private Machines

The Virtual Private Machine (VPM) framework is illustrated in Figure 1. A generic CMP-based system is in Figure 1a. In the most general case, it would contain multi-threaded processors with shared L1 caches, although in the work here we assume single-threaded processors with private L1 caches as shown in the figure. The system contains a shared L2 cache, main memory, and supporting interconnection structures. Of course, if there were an L3 cache, it would be shared in a similar manner.

By implementing hardware mechanisms such as those studied in this paper, the CMP is effectively divided into Virtual Private Machines (VPMs). In a VPM, shares of hardware resources are allocated to executing threads. A CMP with four hardware-supported threads could be divided into four VPMs where the resources, both bandwidths and capacities, are divided evenly among threads executing in the VPMs. In general, however, the resources need not be allocated evenly, and there may be left-over, unallocated resources that are dynamically shared among the VPMs in accordance with the hardware's *fairness policy*. For exam-

ple, in Figure 1b, VPM0 is given a significant fraction (50%) of resources to support a demanding multimedia application, while the other three VPMs are assigned a much lower fraction of resources (10% each). This leaves 20% of the cache memory resources unallocated. Overall, VPMs provide system software with a useful abstraction for maintaining control over shared microarchitecture resources.

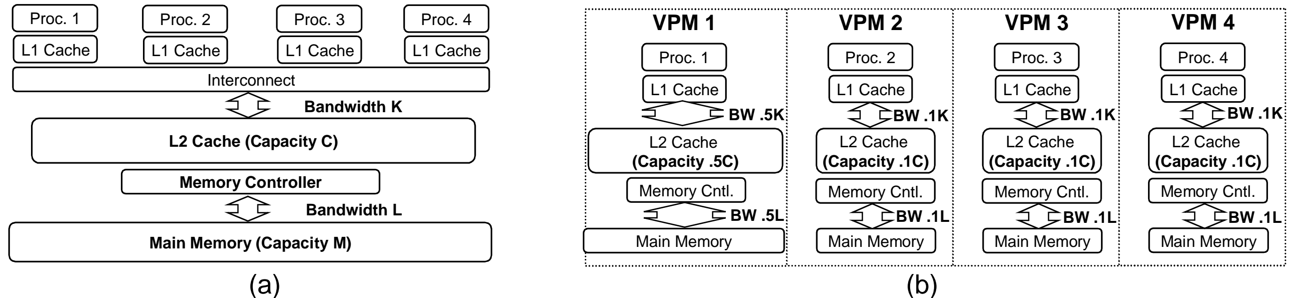

Figure 1. Virtual Private Machines: a) System Hardware b) Four Virtual Private Machines.

The goal is for the VPM to give performance at least as good as a standalone, real private machine having the same resources as allocated to the VPM. Moreover, if one or more of the VPMs do not need all their allocated resources and/or there are unallocated resources, then any excess is distributed among the other VPMs, potentially providing them with additional speedup. The main point, though, is that a VPM should never go slower than the equivalent standalone system, regardless of the other threads that are running.

1.2. Virtual Private Caches

In this paper, we develop Virtual Private Caches (VPCs), a critical component of the VPM framework (See Figure 1b). A VPC consists of two main parts: the VPC Arbiters, which manage shared bandwidth, and the VPC Capacity Manager. The VPC Arbiter design is based on network fair queuing (FQ) algorithms that are adapted to microarchitecture resources. Both the VPC Arbiter and VPC Capacity Manager provide minimum resource guarantees that allow their combination to achieve a global VPM QoS objective.

Cache capacity sharing has been studied and evaluated elsewhere [6][10][20], so we focus most of our discussion and evaluation on the VPC Arbiters. A simulation-based evaluation shows that existing cache arbiter policies allow threads sharing cache bandwidth to affect each other significantly, e.g., performance degradation of up to 87%. In contrast, the proposed VPC Arbiters limit the effect threads have on each other so that QoS performance objectives are met on all workloads studied, including workloads that intentionally inundate the shared cache with requests. Furthermore, we show that on a CMP running heterogeneous workloads, VPCs improve CMP throughput by eliminating negative interference, i.e., VPCs improve average performance by 14% (harmonic mean of normalized IPCs) and by 25% (minimum normalized IPC).

2. Related Work

2.1. Fair Queuing SDRAM Memory Systems

The FQ memory controller proposed by Nesbit et al. uses basic network FQ principles in order to provide QoS and fairness to threads competing for SDRAM memory system bandwidth [18]. The work presented here extends this basic application of FQ principles in several significant ways. First, the VPM is pro-

posed as an overall framework that extends across CMP subsystems. Second, we study shared caches which have both shared bandwidth (VPC Arbiters) and capacity (VPC Capacity Manager) components. Third, although the FQ memory scheduler and VPC arbiters are conceptually based on the same FQ algorithms, the FQ memory controller [18] uses approximate FQ methods. In contrast, the VPC Arbiter algorithms are derived from strict FQ methods. The result is an arbiter that guarantees threads a minimum amount of bandwidth and is amenable to managing microarchitecture resources. In addition, the VPC algorithms account for details that are ignored by the approximate FQ methods, e.g., preemption latencies and performance monotonicity. Fourth, in FQ Memory Systems, Nesbit et al. assume off-the-shelf software where threads are statically allocated equal portions of memory bandwidth. Here we study the effects of *differentiated service*, i.e., allocating threads different amounts of cache bandwidth.

2.2. Cache Capacity Sharing

As noted above, cache capacity sharing has been the topic of a significant amount of research. Many hardware cache sharing algorithms use a thread-aware replacement policy to partition cache capacity. Thread-aware replacements policies require few changes to the underlying cache structure and have little reconfiguration overhead [6][20]. In general, most cache capacity sharing algorithms optimize a performance metric related to aggregate throughput, although some cache sharing research has focused on providing a degree of QoS and fair sharing [6][10]. However, these existing QoS cache capacity sharing algorithms are based on definitions of QoS and fairness that only apply to cache capacity, and consequently, are not compatible with the QoS and fairness properties that can be offered by other shared microarchitecture resources, specifically bandwidth resources.

Rafique et al. [20] show that when system software is given access to appropriate hardware support, software policies can more effectively manage shared cache capacity than hardware algorithms alone. The advantage of software approaches is that OS APIs allow application developers to convey information about their applications' performance requirements. This information is essential for a sharing algorithm to accurately determine QoS, fairness, and performance objectives in a general-purpose environment.

3. Background

3.1. Shared Cache Microarchitecture

Processors in a CMP place considerable pressure on both shared cache capacity and bandwidth. A common way to increase cache bandwidth is to address-interleave requests across multiple cache banks [7]. However, recent work on shared cache crossbar interconnections [12] illustrates that cache banking does not scale well, i.e., cache banking can significantly increase latency, area, and power. Therefore, CMP designers are targeting higher cache utilization [13], thus making threads sharing cache bandwidth more susceptible to negative interference.

We consider a baseline cache hierarchy that closely resembles that used in IBM's Power4, Power5, 970 (Apple's G5), and Xbox 360 chips [23][12][8][11][2]. Write-through L1 caches are used because they re-

quire less L1 array bandwidth, eliminate many of the complexities of protecting on-chip data with ECC, and consequently, have lower latency [16]. The L2 caches are illustrated in Figure 2; this structure would also apply to shared L3 caches, if present. In the baseline cache microarchitecture, processors are connected to the shared cache banks via a crossbar interconnect (Figure 2a). Each processor has private read and write ports into each cache bank [12]. Each cache bank has a return data bus connected to all processors on the crossbar [12]. To reduce power the crossbar interconnection operates at half the core frequency.

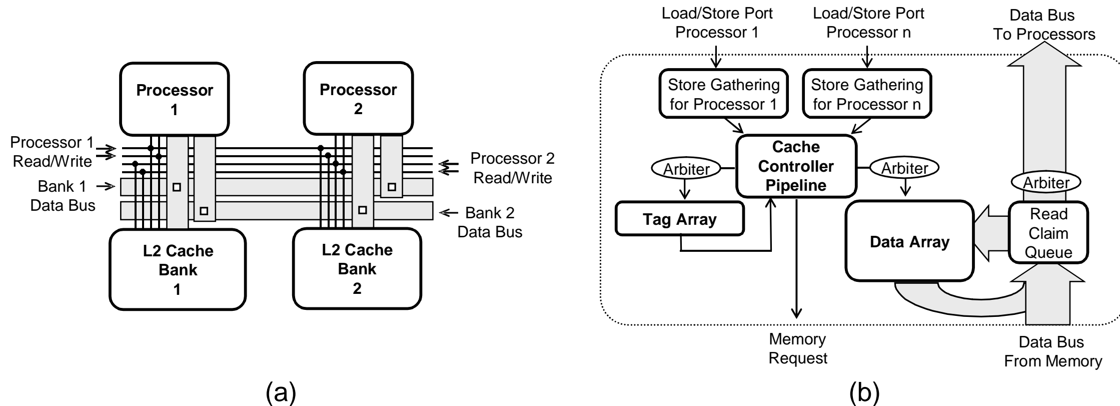

Figure 2. Shared L2 caches a) High-level cache structure. b) Cache bank logical structure.

The logical structure of a cache bank is illustrated in Figure 2b. Within a cache bank, read and write requests first arrive at the store gathering buffers (one per processors) [22]. Store gathering is an effective method to support write-through L1 caches efficiently. At the store gathering buffer, incoming stores are merged with other stores to the same cache line, or if there are no other stores to the same cache line, a new buffer entry is allocated. Loads bypass stores in the store gathering buffer after checking the store gathering buffer for memory dependencies, i.e., *Read-over-Write* (*RoW*) [7]. When a load encounters a store to the same line address, the conflicting store and any older stores are retired from the store gathering buffer to the L2 cache before the load is allowed to proceed, i.e., a *partial flush policy* [22] is implemented. When the store gathering buffer occupancy reaches a high water mark (*n*), the buffer begins retiring stores to the L2 cache, i.e., a *retire-at-n-policy* [22] is used. Loads are prevented from bypassing writes (*RoW inversion*) until the store buffer occupancy falls below its high water mark.

The cache controller uses round-robin selection from the threads' requests after they have been processed by the store gathering buffers [16]. After a request is selected, the controller checks that the request does not conflict with other pending L2 requests, i.e., the new request can not cause a race or violate the memory consistency (in this case PowerPC weak consistency) if it is allowed to proceed [8]. If the request does not create a conflict, it enters the controller pipeline and is allocated a cache controller state machine.

The first stage in the cache controller pipeline is arbitration for the tag array. When the tag array is available, the tag array arbiter selects the first-in request. After a request's tag array access is complete, the request's controller state machine is updated with the state information read from the tag array. If the tag state information indicates a cache hit, the request then enters arbitration for the data array. When the data

array is available, the data array arbiter selects the first-in request from the cache controller state machines. Write requests require two back-to-back data array accesses because ECC covers 32 byte segments. The first access reads the cache line out of the data array – the dirty data is then merged into the cache line and the line's new ECC is calculated – and the second data array access updates the cache line and the ECC. Read requests require a single data array access. After a read request accesses the data array, the request's cache line is sent through the read claim queue and out of the cache bank on the bank's data bus. The data bus arbiter prevents collisions between data coming directly from memory and data coming from the data array.

There are three shared resources in the baseline cache microarchitecture: 1) the tag array, 2) the data array, and 3) the data bus. Each resource has an arbiter (Figure 2b). For private caches, Read-over-Write, First-Come First-Serve (RoW-FCFS) arbiters [7][22] are effective at improving performance and optimizing resource utilization. RoW-FCFS first prioritizes types of requests (RoW) and then prioritizes request ordering (FCFS). In a multi-thread environment, however, a RoW-FCFS arbiter for shared resources allows an aggressive thread with many loads to starve other threads from receiving shared cache bandwidth [16]. This is demonstrated in evaluation Section 5.3.

3.2. Network Fair Queuing

Network FQ scheduling algorithms offer guaranteed bandwidth to simultaneous network flows over a shared link [1][14][26]. That is, the scheduling algorithm provides each flow its allocated share of link bandwidth regardless of the load placed on the link by other flows. A *fairness policy* distributes excess bandwidth in proportion to the flows' allocated shares. Excess bandwidth is either bandwidth that is not allocated to any flow or it is bandwidth that has been allocated but is not used by the flow to which it is allocated. A scheduler that redistributes unused bandwidth is *work-conserving* [4]. In general, the fairness policy can be any policy that distributes excess bandwidth, whether one would subjectively consider it to be "fair" or not. Different fairness policies may be more appropriate in different environments. An important characteristic of a scheduling algorithm's fairness policy is the extent to which a flow receiving excess bandwidth in a given time period will be penalized in later time period(s) [1][14][21][26].

In an FQ scheduling algorithm a flow i is given a share $0 < \phi_i \leq 1$ of the total link bandwidth. FQ scheduling algorithms often operate within a virtual time framework where the *virtual service time* equals the network packet's length L_i^k (expressed in units of link capacity) *time scaled* by the reciprocal of its share ϕ_i. When each packet p_i^k (kth packet of ith flow) arrives at time a_i^k, the FQ algorithm calculates the packet's virtual start-time S_i^k and virtual finish-time F_i^k. A packet's *virtual finish-time* is the time the request must finish (its deadline) in order to fulfill the minimum bandwidth guarantee under ideal conditions. The *virtual start-time* (Equation 1) of a packet is the maximum of its virtual arrival time and the virtual finish-time of the flow's previous packet. The virtual finish-time (Equation 2) is the sum of a packet's virtual start-time and its virtual service time. The packet and its virtual finish-time are placed in a priority queue that is ordered according to the packets' earliest virtual finish-time (packets can be entered anywhere in the priority queue,

depending on their virtual finish-time). Prioritizing packets earliest virtual finish-time first [1][21][26] yields earliest deadline first (EDF) scheduling [4].

[1] $S_i^k = \max \{ a_i^k , F_i^{k-1} \}$

[2] $F_i^k = S_i^k + L_i^k / \phi_i$

Even though a flow is guaranteed a bandwidth, in the presence of other flows, it is not guaranteed that individual requests will finish at the same time as they would in the absence of all other flows. The reason is that the resource may be active with another request (from a different flow) at the time a new request arrives. If the new request could preempt the current request immediately, then the deadline could be satisfied. However, if there is a non-zero preemption latency, the deadline may not be satisfied. With EDF scheduling, as long as the resource is not over allocated (e.g., $\sum\phi_i \leq 1$ [14][21]), a request will finish its service no later than the <*deadline*> + < *max preemption latency*> [14]. If a resource is not preemptible, then the max preemption latency is the maximum service time.

4. Virtual Private Caches

Virtual private caches (VPC) provide a QoS solution for sharing cache resources in CMP-based systems. The VPC controller described in this section has a set of control registers visible to system software that specify a VPC configuration for each hardware thread sharing the cache. For each active thread, the control register specify a share of cache capacity (α_i), and a share of tag array, data array, and data bus bandwidths (ϕ_i). In their full generality, the mechanisms described in this section would allow software to allocate each of the three bandwidth resources independently (via separate control registers), but we restrict the discussion to the case where the allocations are the same.

4.1. VPC Arbiter

Each shared cache resource has a VPC arbiter that selects a pending request to access the shared resource next. All the VPC arbiters are implemented in essentially the same way, taking into account the resources' different timing characteristics. Network FQ algorithms form the basis of the VPC arbiters.

4.1.1. *Arbiter Implementation*

The proposed arbiter implementation is based on the following observations. The first comes from network FQ research: even though a priority queue is not strictly FIFO, the requests from a single flow are serviced in FIFO order. This leads to an implementation with separate FIFO buffering for waiting requests from each thread (Figure 3) [26]. We go one step further: as long as a thread is offered its allocated share of the cache bandwidth, the exact order in which a thread's requests are serviced is not important. This means that the FIFO requirement within a thread's arbitration buffer can be relaxed while maintaining bandwidth guarantees. Exploiting this feature allows performance optimizations that re-order requests (e.g., Read over Write), after requests have entered arbitration. Because the baseline cache microarchitecture checks for memory consistency violations before a request enters arbitration for a shared resource, reordering requests

(subject to address dependencies) within the cache arbiters does not affect correctness. Recall that in the baseline microarchitecture reordering can cause starvation, not affect correctness.

A VPC arbiter has a small buffer for each thread (see Figure 3). When request m_i^k (the *k*th request from the *i*th thread) enters arbitration for the resource, the request's ID is added to its thread's buffer. A request ID is a reference to a controller state machine where the request's state information is stored. A request ID requires a few bits of storage, *log₂(<number of controller state machines>)* bits, and a thread's buffer consists of a few bytes (3 bytes for the configuration studied in the evaluation section).

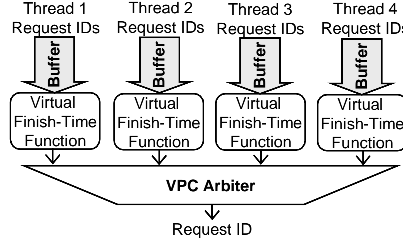

Figure 3. VPC Arbiter Hardware Implementation

A VPC arbiter maintains a clock register *R.clk* and two registers $R.L_i$ and $R.S_i$ for each thread[1]. The clock register in our implementation is a real time counter that simply counts clock ticks. The $R.L_i$ register stores a thread's virtual service time L_i^k / ϕ_i, where L_i^k is the service requirement and ϕ_i is the thread's fraction of the resource. The service requirement L_i^k is the latency of the shared resource (see Table 1 in the evaluation section). Because the latency of a shared resource does not change, the $R.L_i$ register only has to be calculated when the thread's service share ϕ_i is changed. The $R.S_i$ register tracks the time a *virtual resource* will become available for a thread's next request.

In a basic implementation (without reordering), request m_i^k's virtual start-time S_i^k is the maximum of its arrival time a_i^k and $R.S_i$ (Equation 3). Request m_i^k's virtual finish-time F_i^k is the sum of its virtual start-time and its virtual service time (Equation 4), which is stored in $R.L_i$. When a VPC arbiter selects a request m_i^k to access the shared resource, it first performs the functions of Equations 3 and 4, and updates the $R.S_i$ register (Equation 5) to reflect the virtual time the virtual resource will become available for the next request.

[3] $S_i^k = \max\{ a_i^k, R.S_i \}$ from Equation 1

[4] $F_i^k = S_i^k + R.L_i$ from Equation 2 ($S_i^k + 2*R.L_i$ for write requests on the data array)

[5] $R.S_i \leftarrow F_i^k$

Because each request's arrival time is used in Equation 3, the basic implementation requires the arbiter to store each request's arrival time. However, there is an optimization that eliminates this requirement. At the time the request m_i^k enters arbitration:

- If thread *i*'s queue *was not* empty when m_i^k arrived, i.e., the virtual resource was backlogged, then its arrival time will have no effect on its virtual finish-time. The reason is that the virtual resource

[1] In our notation, we use the *R.* prefix to distinguish hardware register values.

can not service requests faster than the real resource (because $\phi_i \leq 1$). When the request becomes the thread's oldest request, the virtual resource will still be backlogged (i.e., $a_i^k \leq F_i^{k-1}$). Consequently, the request's virtual start-time S_i^k will be the time the virtual resource becomes available F_i^{k-1}, which is the value held in the $R.S_i$ register at the time m_i^k is selected for service.

- If the thread's queue *was* empty at the time request m_i^k arrived, then m_i^k is by default the thread's oldest request. At this time the $R.S_i$ register is updated so that it holds m_i^k 's virtual start-time S_i^k. If $F_i^{k-1} > a_i^k$, then S_i^k is F_i^{k-1}, which is the value already held in the $R.S_i$ register. Otherwise, a_i^k is m_i^k 's virtual start-time, and this is the value held in the clock register (*R.clk*). Therefore, at the time m_i^k enters arbitration Equation 6 is used to conditionally update the $R.S_i$ register.

[6] *If thread i's queue is empty and* $R.S_i \leq R.clk$ *then* $R.S_i \leftarrow R.clk$

Consequently, when request m_i^k becomes the thread's oldest request, the request's correct virtual start-time S_i^k is held in $R.S_i$. This means that Equation 3 can be replaced with Equation 3′.

[3′] $S_i^k = R.S_i$

In the optimized implementation, a VPC Arbiter (Figure 3) compares the threads' oldest request's virtual finish-times (Equation 4) and selects the request with the earliest virtual finish-time. When the VPC arbiter selects a request m_i^k to access the resource, it updates the $R.S_i$ register (Equation 5) to reflect the virtual time the virtual resource will become available. In addition, the request's ID is removed from the VPC arbiter and is used by the cache controller to identify the request's controller state machine which initiates the access to the shared resource.

The implementation outlined in equations 3′, 4, 5, and 6 enables performance optimizations that use intra-thread request reordering, while maintaining the bandwidth guarantees and fairness properties of the basic FQ algorithm. With respect to the virtual resource, the $R.S_i$ register holds the maximum of 1) the virtual time the virtual resource last became backlogged and 2) the time the virtual resource will become available; both of these are independent of the specific requests that are pending. Therefore, any request in a thread's buffer can be selected to access the resource next, and doing so does not change the amount of service (bandwidth) the thread receives relative to other threads [14]. Consequently, requests within a thread's buffer can be reordered while still maintaining the bandwidth guarantees and fairness properties of the basic FQ algorithm. Intra-thread reordering occurs within the thread's buffer and does not affect the rest of the arbiter design. The primary reordering optimization we are interested in here is prioritizing Reads-over-Writes (RoW), although there are other common reordering optimizations that occur in a cache hierarchy, e.g., prioritizing demand-fetches over prefetches, but we do not consider them in this paper.

4.1.2. *Resource Scheduling and Preemption Latencies*

Servicing the thread with the earliest virtual finish-time first is the same as scheduling *earliest deadline first (EDF)* [4]. As described in the background section, with EDF scheduling and a non-preemptible re-

source, which is the case for shared cache resources, a request will finish its service by the <*deadline*> + <*resource's max service time*> [4]. That is, a cache request may be delayed by another thread's maximum service time L_i^k (in the worst case).

Despite the worst case behavior, the resource latency of shared cache resources does not often have a significant affect on a thread's performance (as will be shown in the evaluation section). General purpose applications tend to contain bursty L2 accesses [7], and the preemption latency is *paid only once per burst* [14]; the performance effect is analogous to filling a pipeline. Hence, the preemption latency is amortized over the burst of cache misses. However, applications with low memory level parallelism, i.e., where misses are less bursty, tend to be more susceptible to preemption latency.

4.1.3. *Fairness Policy*

A VPC arbiter mitigates the effects of preemption latency through a fairness policy that is tailored to general-purpose applications. The *VPC arbiters' fairness policy* distributes excess bandwidth to the back-logged thread that has received the least excess bandwidth in the past (relative to its service share ϕ_i) [18][21]. In addition to acting as a deadline, the virtual finish-time of a thread's next request is an indicator of the amount of excess service the thread has received normalized to its share of the resource. For example, if one thread is able to use a large amount of excess resource in a burst, its virtual finish-time will run ahead of the other threads' virtual finish times. Therefore, servicing threads earliest virtual finish-time (as described above) naturally distributes excess bandwidth to the backlogged thread that has received the least excess bandwidth in the past.

Threads that consume more cache bandwidth tend to increase the average preemption latency experienced by other threads. Furthermore, these highly consumptive threads tend to have more memory level parallelism and are less susceptible to preemption latency. Therefore, the highly consumptive threads should not receive excess bandwidth before threads that have received less excess bandwidth in the past. The VPC arbiters' fairness policy is very effective at averaging out preemption latency. However, in cases where a thread is sensitive to L2 latency and is allocated a high percentage of the cache bandwidth, artifacts of the preemption latency can still be observed in a thread's average performance. A detailed comparison of fairness policies and how a thread should be penalized for consuming excess service is a topic for future work.

4.2. VPC Capacity Manager

4.2.1. *Capacity Manager Implementation*

The VPC Capacity Manager implements a thread-aware replacement policy that requires few changes to the baseline cache microarchitecture. Hardware mechanisms that support similar replacements policies have been proposed in the past [6][10][20]. The VPC Capacity Manager provides each thread a VPC that has the same number of sets as the shared cache and at least $\lfloor \alpha_i$ * <*ways in the shared cache*> $\rfloor$ cache ways. We discuss the rationale behind this property later in Section 4.3, after we introduce the concept of *performance*

monotonicity. The VPC Capacity Manager's replacement policy chooses a victim cache line from the destination cache set that satisfies one of the following two conditions:

1) *It is the least recently used (LRU) line owned by another thread j, such that thread j occupies more than α_j of the ways in the destination cache set.*
2) *If there are no cache lines that satisfy the first condition, then it is the LRU line owned by the thread requesting the replacement.*

If Condition 1 is satisfied, then taking a cache line away from thread *j* will not cause the thread to fall below its allocated share of the cache ways. Furthermore, the LRU line owned by a thread occupying more than its share of the cache ways, would not have been in the thread's cache if it were executing out of a private cache with only α_i of the cache ways. If there are no threads that occupy more then their share of the cache ways (Condition 2), then all threads must occupy exactly their share of the cache ways. Therefore, the LRU cache line owned by the thread would be the same cache line replaced if the thread were executing out of a private cache with α_i of the cache ways.

4.2.2. *Fairness Policy*

As defined above, there may be multiple threads that occupy more than α_j of the ways in the destination cache set, and therefore, there may be multiple cache lines that satisfy the first condition of the replacement policy. The VPC Capacity Manger's fairness policy refines the replacement policy above by specifying how to select a victim when more than one thread occupies more than its share of the cache ways, and therefore, specifies how excess cache capacity should be distributed.

In contrast with the VPC arbiter, the VPC Capacity Manager cannot guarantee QoS and be *work conserving* without oracle knowledge. Once cache capacity is allocated to a thread, the capacity cannot be relinquished (and redistributed) as long as the thread remains active, even if it will go unused in the future. To redistribute such cache capacity would require knowing when a thread has accessed a cache line for the last time. Theoretically, the cache line is *unused* and could be redistributed without affecting the owner's QoS, but without an oracle, this cannot be done.

We leave the analysis of the VPC Capacity Manager's fairness policy for future work. As discussed in the background section, the fairness policy can be any policy that distributes excess resources, whether one would subjectively consider it to be "fair" or not. There are many existing capacity sharing algorithms that target "fairness", e.g. [6][20], or throughput which can be used to refine the VPC Capacity Manager's replacement policy and manage the excess cache capacity. Because these methods have been studied elsewhere, we focus on scenarios where all of the shared cache's resources are allocated so the VPC Capacity Manger's fairness policy has no effect.

4.3. Discussion

As discussed in the introduction, we use resource allocation as a means for achieving a desired QoS. In doing so we assume that a given level of performance is guaranteed if a minimum resource allocation is guaranteed. Resources above the allocated minimum may be provided to a thread if they are available. An implicit assumption is that *a thread's performance is a monotonically increasing function of the amount of resources the thread is offered.* That is, if a thread is offered more resources, the thread's performance will remain the same or increase – it *will not* decrease. Performance monotonicity, when combined with guaranteed minimum bandwidth will provide QoS.

In many computer systems, performance monotonicity is not always a safe assumption. Because of the complexity of modern processors, and their use of speculation, there are situations where having more resources can cause the ordering of events to shift or new events to be generated, and these can occasionally degrade performance. One example is prefetching – giving a thread more memory bandwidth may increase the number of prefetches. In some programs increasing prefetches may lead to net performance loss: performance losses due to cache pollution may not be offset by the performance gain from the prefetching.

We conjecture that if one were to constrain a system implementation so that performance monotonicity is strictly satisfied in all cases, then it would rule out many performance optimizations that significantly raise average performance. In other words, by guaranteeing performance monotonicity, one would reduce average performance only to avoid relatively rare cases where performance might degrade slightly. We feel that in many situations, this is not a good engineering choice, and therefore one might opt for systems which do not guarantee performance monotonicity. The wisdom of this choice can be demonstrated through simulations (as we have done), which show that in practical cases, QoS is in fact achieved, despite the absence of guaranteed performance monotonicity.

Similar comments apply to algorithms for sharing cache capacity. We conjecture that the proposed VPC Capacity Manager (partitioning cache ways) satisfies performance monotonicity. However, the proposed VPC method comes at a cost; it increases the granularity at which the cache can be partitioned. Therefore, when compared with more flexible capacity managers, e.g., capacity managers that partition cache capacity based on the usage of the entire cache [6][10][20], the VPC method may have lower average performance. However, these more flexible capacity managers do not satisfy performance monotonicity. For example, a capacity manager that does not distinguish between cache sets and cache ways may offer a thread more cache sets and fewer cache ways, which can increase the thread's conflict misses and decrease its performance.

In this paper we use the more restricted cache capacity manager because it facilitates a more targeted evaluation of the VPC arbiters. If we chose a capacity manager design that clearly does not satisfy performance monotonicity, it would be difficult to distinguish the performance effects of the VPC arbiters from the effects of an impaired capacity manager.

5. Evaluation

5.1. CMP Performance Model

We use a detailed structural simulator developed at IBM Research. The simulator adopts the ASIM modeling methodology [5], and is defined at an abstraction level slightly higher than latch-level. The model's default configuration is of a single processor IBM 970 system [8]. In its default configuration, the model has been validated to be ± 5% of the 970 design group's latch-level processor model. In this paper, we use an alternative simulator configuration (see Table 1) to avoid 970-specific design constraints.

The primary difference is a cache configuration that is more representative of future systems than the 970's cache configuration. We increased the size of the L2 cache to 16MB, decreased the line size (64 bytes) in order to reduce the demand on memory bandwidth, and added more cache banks in order to support more processors. The L2 tag array and data array latencies have been scaled to account for the increase in array size and to be consistent with 45 nm technology [25]; array bandwidth is the reciprocal of the array latency. The L2-to-L1 data bus is 16-bytes wide, which balances the additional data array latency with the data bus bandwidth. The total L2 latency to read the critical word of a cache line from the L2 is 16 processor cycles – to read the entire line takes 22 processor cycles (see the timing diagram in Figure 4). As in [12] the L2 controller state machines are partitioned – each processor is allocated eight state machines per cache bank. The VPC Capacity Manager allocates equal portions of cache ways, and there are no unallocated cache ways, e.g., for a four processor CMP $\alpha_i = .25$ for all i. The 970 instruction and data prefetchers are disabled. VPC supported prefetching is an important topic for future work.

Table 1. 2 GHz CMP System Configuration (latencies measured in processor cycles)

Processors	4 processors
Issue Buffers	20 entry BRU/CRU, two 20 entry FXU/LSU, 20 entry FPU
Issue Width	8 units (2 FXU, 2 LSU, 2 FPU, 1 BRU, 1 CRU)
Reorder Buffer	20 dispatch groups, 5 instructions per dispatch group
Load / Store Queues	32 entry load reorder queue, 32 entry store reorder queue
I-Cache	16KB private, 4-ways, 64 byte lines, 2 cycle latency, 8 MSHRs
D-Cache	16KB private, 4-ways, 64 byte lines, 2 cycle latency, 16 MSHRs
L1-to-L2 Interconnect	operates at ½ core frequency, 2 cycle interconnect latency, 16 byte data bus per bank
L2 Store Gathering Buffer	8 store gathering buffer entries per thread, read bypassing, retire-at-6 policy, partial-flush on read conflict
L2 Cache	operates at ½ core frequency, 2 banks, 16MB, 32-ways, 64 byte lines, 8 cache controller state machines per thread, 4 cycle tag array latency, 8 cycle data array latency
Memory Controller	operates at ½ core frequency, 16 transaction buffer entries per thread, 8 write buffer entries per thread, closed page policy
SDRAM Channels	1 channel per thread
SDRAM Ranks	2 ranks per channel
SDRAM Banks	8 banks per rank

For the uniprocessor baseline which has a private L2 cache, we use the RoW-FCFS arbiter policy. The same arbiter policy is applied to all shared resources: tag array, data array, and data bus. For the multiproc-

essor baseline, we use RoW-FCFS within the threads' private store gathering buffer (as described earlier), and FCFS for shared cache resource because RoW-FCFS can cause starvation (as will be shown).

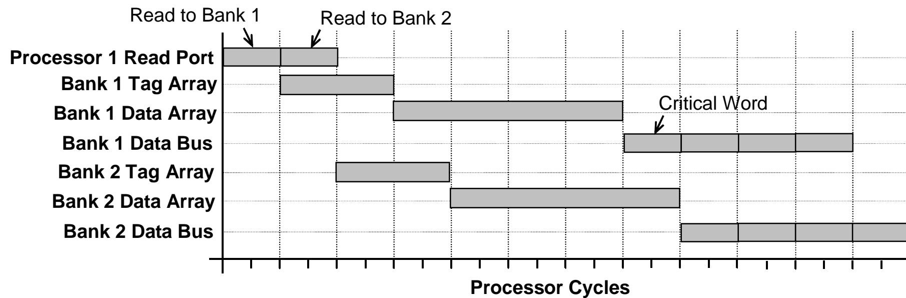

Figure 4. Cache Timing Diagram of back-to-back reads to different cache banks

The simulator has a cycle accurate model of an on-chip memory controller attached to a DDR2-800 memory system [17]. To isolate the effects of cache sharing, threads are allocated private SDRAM channels by interleaving memory requests across memory channels using the most significant bits of the physical address and controlling the virtual-to-physical address mapping such that the threads' physical address spaces differ in the most significant bits of the physical address.

5.2. Workloads

We use two microbenchmarks (Table 2) and the SPEC 2000 benchmark suite to evaluate the performance of the VPC arbiters. Each microbenchmark operates on a two-dimensional array of 32-bit words (int array[R][C]). The array's rows are 64 bytes long (the L1 cache line size) and the array's total size is 32KB, twice the size of the L1 data cache. The *Loads* microbenchmark stresses L2 load bandwidth by continuously loading (*lwz* in PowerPC) the first column of each row in the array, thus creating a constant stream of L2 read hits. The main loop is unrolled; otherwise, the 970 branch information queue (BIQ) [8] becomes a bottleneck. The *Stores* microbenchmark stresses L2 store bandwidth and is the same as the *Loads* microbenchmark but with store instructions (*stw* in PowerPC).

Table 2. Microbenchmarks in C / PowerPC hybrid

Loads	***Stores***
while(true) {	while(true) {
r2 <- &array[0][0]	r2 <- &array[0][0]
for(i = 0; i < R; i += 4) {	for(i = 0; i < R; i += 4) {
lwz r3,0(r2)	stw r3,0(r2)
lwz r3,64(r2)	stw r3,64(r2)
lwz r3,128(r2)	stw r3,128(r2)
lwz r3,192(r2)	stw r3,192(r2)
r2 <- r2 + 256	r2 <- r2 + 256
}	}
}	}

Figure 5 shows the cache utilization of the microbenchmarks with a varying number of L2 cache banks, e.g., *Loads 2B* uses the baseline cache configuration with 2 cache banks. The *Loads* benchmark fully utilizes two L2 banks and has about *80%* utilization on four L2 banks. Under ideal conditions the *Loads* benchmark

should be able to fully utilize four L2 banks. However, the 970's LSU reject mechanism causes loads to acquire LMQ [8] entries and enter the L2 cache out-of-order. Out-of-order cache accesses cause non-ideal bank interleaving, and the processor's in-order structures, the LRQ [8] and reorder buffer, fill up and stall dispatch. The data bus and data array utilization for the *Loads* benchmark are equal, illustrating that the cache design is properly balanced as in the timing diagram in Figure 4. The *Stores* benchmark is very aggressive; it fully utilizes the data array in eight cache banks. Write requests enter the L2 cache in-order, and therefore, have ideal bank interleaving.

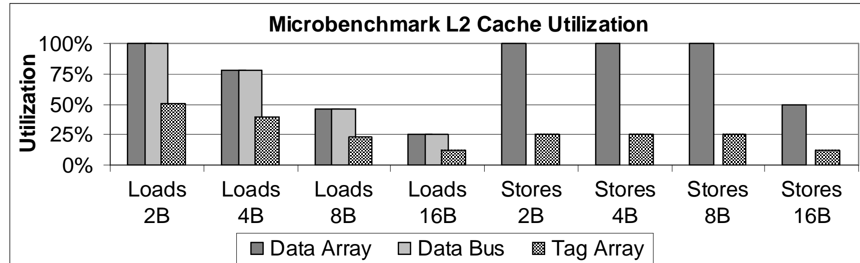

Figure 5. L2 Cache Utilization of the Microbenchmarks

We use the SPEC CPU 2000 benchmark suite as the second set of benchmarks because they are the best available source of heterogeneous applications. The SPEC benchmark simulations use twenty 100 million instruction sampled traces where each trace has been verified to be statistically representative of an entire SPEC application [9]. Figure 6 shows the cache utilizations of the individual SPEC benchmarks. As would be expected, the data array has the highest utilization, but for a few benchmarks (e.g., *equake* and *swim*) the tag array has greater utilization than the data array. *Equake* and *swim* have few L2 write requests (see Figure 7) and many L2 cache misses which require multiple tag array accesses.

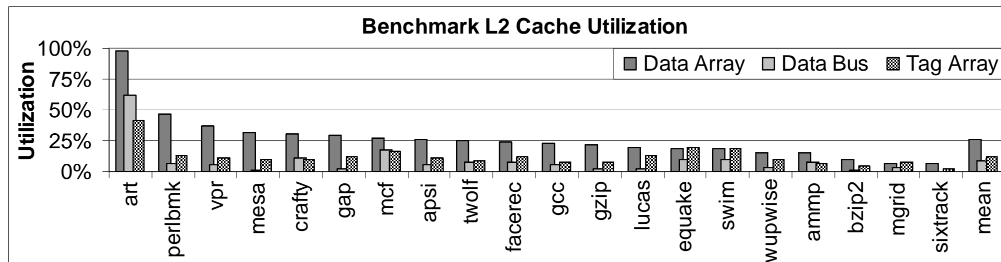

Figure 6. L2 Cache Utilization of the SPEC Benchmarks

Throughout our evaluation, the SPEC benchmarks are ordered by data array utilization; we use data array utilization as an indicator of a thread's *aggressiveness*. Figure 7 shows the percentage of L2 requests that are write requests (after store gathering) and the store gathering rate (the percentage of stores that are gathered with other stores in the store gathering buffer). On average, write requests account for *55%* of all L2 requests (after store gathering), and *80%* of stores are gathered and do not require a separate L2 access. The 970's write-through L1 cache combined with store gathering is nearly as bandwidth effective as write-back caches, but without the complexities of write-back caches.

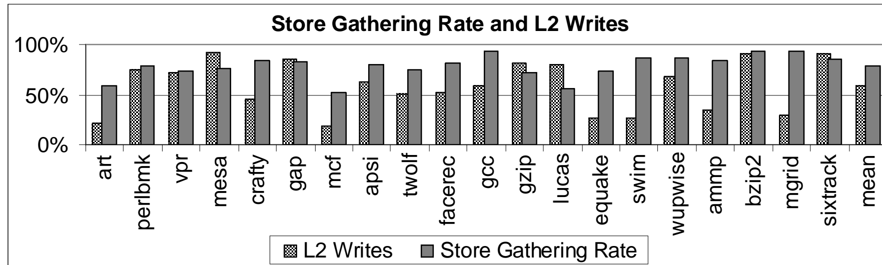

Figure 7. Percentage of L2 Requests that are Writes and the Store Gathering Rate

We chose a baseline cache configuration using the Figure 6 data as a guide. As described in the background section, CMP designers have to trade between cache bandwidth and cache power, latency, and area. In general, architects design for the common case. Because desktop and laptop systems typically have a small number of executing threads, high volume CMPs will tend to have relatively few banks. From Figure 5 we observe that a single-thread achieves utilization of *100%* for as many as eight cache banks. Obviously, designing for this case is unattractive, e.g., a four processor CMP with 32 cache banks. With the VPC arbiters, CMP designers can focus on designing for the common case, rather than the worst case. From Figure 6 we observe that on average a single-thread consumes *26%* of the cache bank bandwidth for our set of benchmarks. Therefore, we chose a design with two cache banks. Two banks provide ample bandwidth for what are likely to be the most common cases: single-thread and two thread workloads, while on a four thread workload, the cache approaches full utilization.

5.3. Results

For the first experiment we model the baseline CMP executing two threads: the *Loads* microbenchmark on processor 1 and the *Stores* microbenchmark on processor 2. We analyze the effects of the following cache arbiters: First Come First Serve (FCFS), Read-over-Write First Come First Serve (RoW), and Virtual Private Cache (VPC) with five VPC configurations. The IPC and data array utilization results are shown in Figure 8. We omit the utilizations for the other shared resources because the data array is the main bottleneck for these benchmarks (see Figure 5). The x-axis of Figure 8 specifies the cache arbiter policy, and for the VPC arbiters, the share of cache bandwidth allocated to the *Stores* benchmark – leftover bandwidth is allocated to the *Loads* benchmark. For example, the label VPC *25%* represents the configuration where the Stores benchmark is allocated *.25* of the cache bandwidths ($\phi_1 = .25$) and the *Loads* benchmark is allocated *.75* ($\phi_2 = .75$). The IPC graph includes the *target IPCs* for each VPC configuration.

A benchmark's *target IPC* is the benchmark's instructions per cycle (IPC) performance when running on a private cache with the same resources as the thread's allocated VPC. As described above, the benchmarks ideally should not go slower than this target, and if the thread is offered excess bandwidth by the VPC fairness policy, it may go faster. However, target IPC does not take into account the effects of resource preemption latencies. Therefore, even if performance monotonicity holds, benchmarks may not meet their target IPC; although we will show that in most cases they do. To determine a benchmark's target IPC, we

simulate a uniprocessor system with a private cache that has the same resources as the VPC. The private cache has the same number of sets as the shared cache and $\lfloor \alpha_i * \textless baseline\ cache\ ways \textgreater \rfloor$ cache ways. In the private cache all resource latencies are scaled by: $1/\phi_i$ * *<baseline resource latency>*. For example, for a VPC allocated *.5* of the cache bandwidth and *.25* of the cache ways ($\phi_i = .5$, $\alpha_i = .25$), we simulate a uniprocessor with a L2 cache that has 8 cache ways, an 8 cycle tag array latency, and 16 cycle data array latency. For the cases where $\phi_i = 0$ we set the target IPC to 0.

With the RoW-FCFS arbiter, the *Loads* benchmark prevents the *Stores* benchmark from receiving any cache bandwidth (likewise, any application with a long stream of loads would starve any other application's stores). In a real system, this would be a critical design flaw.

The FCFS arbiters perform well on the combined *Loads* and *Stores* workload. With FCFS, requests from the *Loads* and *Stores* benchmarks are interleaved uniformly, i.e., there is one store from the *Stores* benchmark shared cache for every load from the *Loads* benchmark. With a uniform interleaving of requests, the Stores benchmark receives *67%* of the data array bandwidth and the Loads benchmark receives *33%* of the data array bandwidth because stores require twice as much data array bandwidth as loads.

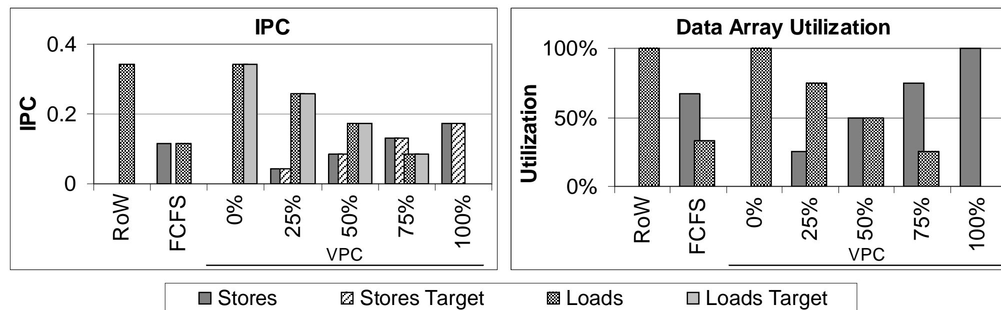

Figure 8. Loads and Stores Microbenchmarks IPC and Data Array Utilization

All five VPC arbiters precisely provide each benchmark its share of the cache bandwidth over a broad range of allocations. The VPC arbiters are able to divide bandwidth between the *Loads* and *Store* benchmarks so well because both benchmarks keep constant pressure on the L2 cache – there is always a request from both threads waiting for a shared resource. Because the performance of these benchmarks depends almost completely on L2 bandwidth, both benchmarks meet their target IPCs; there are no preemption latency effects.

For our second experiment we model the baseline CMP executing a SPEC benchmark on processor 1 (the *subject thread*) and the *Stores* microbenchmark on processors 2, 3, and 4 (*background threads*). This experiment illustrates the advantages of the VPC arbiters when a foreground task is co-scheduled with aggressive (possibly even malicious) background tasks. Figure 9 shows the IPC and data array utilization of the subject thread. We model the subject thread with three VPC bandwidth allocations: $\phi_1 = .25$, $\phi_1 = .5$, and $\phi_1 = 1$; leftover bandwidth is allocated equally amongst the background *Stores* benchmarks. Each IPC is normalized to the subject thread's target IPCs for $\phi_1 = 1$, i.e., the subject thread running on a private cache

with full cache bandwidth. In addition to the IPC of the subject thread, the IPC graph shows the target IPC for $\phi_I = .25$ and $\phi_I = .5$, and the *QoS IPC* (defined below) for $\phi_I = 1$.

A benchmark's *QoS IPC* takes into account the resources' maximum preemption latencies. To generate the QoS IPCs, we simulate a uniprocessor system as we do for the target IPCs, except we add the maximum preemption latency to the resource latencies and keep the resource bandwidths the same as for the target IPC simulation. For example, we simulate a uniprocessor system with the cache resource latencies equal to $1/\phi_i$ * *<baseline resource latency>* + *<max preemption latency>* cycles and bandwidths equal to ϕ_i / *<baseline resource latency>* accesses per cycle.

The FCFS arbiters do not meet the subject threads' Target IPCs on any of the benchmarks. The subject threads' average normalized IPC is *.3* (the harmonic mean of the normalized IPCs). Benchmark *mcf* has the worst normalized IPC, it receives only 4% of the cache bandwidth (far below its implicit share of the cache bandwidth) and has a normalized IPC of *.13* (an *8x* increase in runtime compared to the thread running alone). With the FCFS arbiters in a desktop environment, for example, an aggressive background task may prevent the user from watching a HD movie, even when there is ample processing power available. In a service oriented server environment, a subscriber can launch aggressive threads and inadvertently slowdown other subscribers' tasks. Or, as in the case of a denial of service attack, it may not be inadvertent.

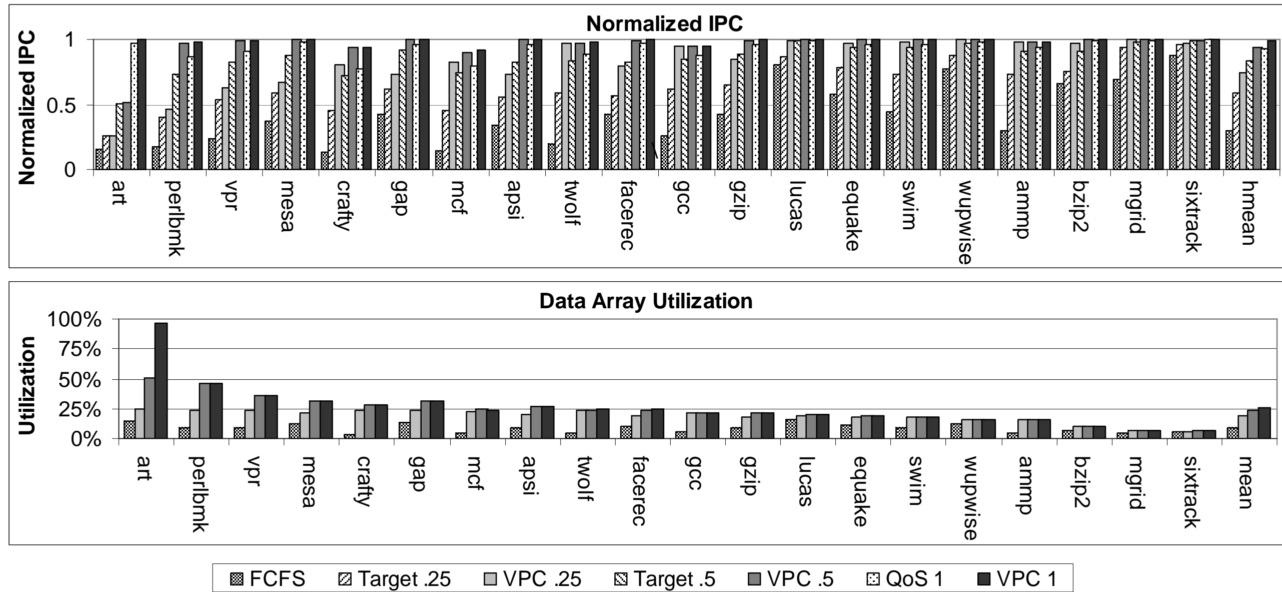

Figure 9. Subject Thread Normalized IPC and Data Array Utilization

The VPC arbiters meet the QoS objectives for all workloads on all VPC configurations. For the $\phi_I = .25$ and $\phi_I = .5$ VPC configurations, each subject thread's IPC is greater than its target IPC. The preemption latencies do not have a significant effect with these VPC configurations because 1) the preemption latencies are relatively small compared to VPC resource latencies, and 2) the fairness policy penalizes aggressive

threads for consuming excess resource, thus causing the less aggressive (more latency-sensitive) subject thread's requests to get priority as soon as they arrive at the cache controller.

For the $\phi_1 = 1$ VPC configuration, the subject threads meet their QoS IPCs, which take into account the resources' preemption latencies. For $\phi_1 = 1$, the subject thread's VPC resources are equivalent to the real cache, and therefore, the effects of the resources' preemption latencies can be observed in the benchmarks' IPCs. From these results we observe the varying degrees of memory level parallelism in the benchmarks. In general, benchmarks that have less memory level parallelism tend to be more sensitive to preemption latency.

As the subject thread's allocated share of cache bandwidth increases, the variation between the thread's actual IPC and its target IPC decreases. This is mostly due to the VPC arbiters' fairness policy. Requests from less aggressive threads (usually more latency-sensitive) are given higher priority, and therefore, their average latency tends to be closer to the real cache latency than the VPC latency. We intend to study the fairness policy in more detail in future work, specifically whether the fairness policy can provide tighter bounds on performance than those derived using network FQ theory.

The top eight benchmarks demand more than *.25* of the cache bandwidth. With the $\phi_1 = .25$ configuration, these threads are bandwidth constrained. On average, they receive *23%* of the cache bandwidth. The average normalized IPC of all the subject threads (with $\phi_1 = .25$) is *.75*, much better than FCFS, which has an average normalized IPC of *.3*. The only subject thread that demands more than .5 the cache bandwidth is *art*. With the $\phi_1 = .5$ VPC configuration, *art*'s normalized IPC is *.51*. Its performance depends mostly on L2 cache bandwidth and it receives *.5* of the cache bandwidth. The average normalized IPC excluding art is *.98*, and including *art* the average is *.94*.

For the $\phi_1 = 1$ VPC configuration, the average normalized IPC is*.99*. The worse case performance degradation is *mcf* with a normalized IPC of *.92*; *mcf* is very susceptible to preemption latency. These results show that the $\phi_1 = 1$ VPC configuration is good for prioritizing a foreground thread while running less important background tasks.

For our last experiment, we model the baseline CMP executing a four thread multiprogram workload. This experiment illustrates the advantages of the VPC cache arbiter with off-the-shelf system software running heterogeneous workloads under normal conditions. To generate workloads, we used a perl script that uses random selection without replacement to generate three four-processor workloads from the top twelve SPEC benchmarks. We ran this script three times to generate nine workloads. Therefore, each benchmark appears once in the first set of three workloads, once in the second, and once in the third. Figure 10 shows individual benchmarks' normalized IPCs as well as the harmonic mean of each workload's four IPCs, and shows the individual and mean data array utilizations. Cache bandwidth is allocated in equal proportions ($\phi_i = .25$ for i from 1 to 4). Each thread's IPC is normalized to its target IPC for $\phi_i = .25$.

For the multiprogram workload and the VPC arbiters, each benchmark meets its target IPC, i.e., the normalized IPC of each benchmark is greater than one. With the FCFS arbiters there are 7 out of 36 benchmarks that do not meet their target IPC; in the worst case, the normalized IPC is *.71*.

Overall, there are significant differences between the individual threads' normalized IPCs with the FCFS arbiters and with the VPC arbiters. The average relative performance difference is 28% , i.e. thread i's relative performance difference is: $| FCFS_i - VPC_i | / average (FCFS_i , VPC_i)$, where $FCFS_i$ is the thread's normalized IPC with FCFS and VPC_i is the thread's normalized with the VPC arbiters. This result emphasizes the significance of shared cache bandwidth and the importance of fair cache arbiters. Furthermore, there is *-.21* correlation between the threads' normalized IPCs with FCFS and the threads' normalized IPCs with the VPC arbiters, when *mcf* 's (an outlier) IPCs are excluded, the correlation is *-.72*, which is a strong negative correlation. With FCFS, more aggressive threads tend to have better performance and less aggressive threads tend to have worse performance. In contrast, with the VPC arbiters, all threads get their allocated share of the bandwidth. Therefore, relative to FCFS, more aggressive threads tend to have lower performance and less aggressive threads tend to have better performance.

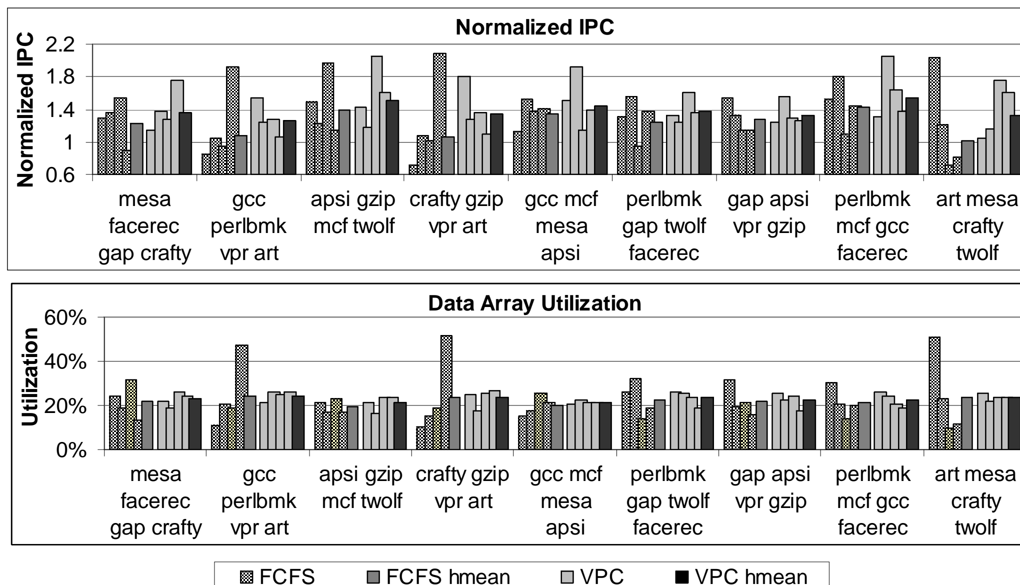

Figure 10. Multiprogram Workload Normalized IPC and Data Array Utilization

With FCFS arbiters, the average (arithmetic mean) data array utilization is *22%* and has a standard deviation of *10.3%*. With the VPC arbiters, the threads' data array utilizations closely track their allocated shares of cache bandwidth. The average data array utilization is *23%* (slightly more than FCFS) and has a standard deviation of *2.8%*. Most of the deviation results from workloads that do not demand their full share of the cache bandwidth, e.g., the benchmarks *gcc*, *gzip*, and *facerec* (see Figure 6).

Finally, we use two performance metrics (see Table 3) to compare the VPC and FCFS arbiters: the harmonic mean of each workload's normalized IPCs and the minimum normalized IPC of each workload. The harmonic mean of the normalized IPCs emphasizes both throughput and fairness [15]. It is a good metric for

multiprogram workloads that run continuously and the amount of work each program does is equally important. The minimum normalized IPC is a good metric for multiprogram workloads where each thread is run to completion. Here, the emphasis is on the time the entire workload completes.

The VPC arbiters eliminate negative interference which improves both performance metrics for all workloads studied. In the best cases, the VPC arbiter improves the harmonic of normalized IPCs by 32% and the minimum normalized IPC by 52%. Overall, the VPC arbiters improve average normalized IPC by *14%* and improve the average minimum normalized IPC by *25%*.

Table 3. Performance Improvement

	mesa facerec gap crafty	*gcc perlbmk vpr art*	*apsi gzip mcf twolf*	*crafty gzip vpr art*	*gcc mcf mesa apsi*	*perlbmk gap twolf facerec*	*gap apsi vpr gzip*	*perlbmk mcf gcc facerec*	*art mesa crafty twolf*	**Average**
Harmonic Mean of Normalized IPCs	11%	17%	8%	26%	7%	9%	5%	9%	32%	**14%**
Minimum Normalized IPC	27%	26%	3%	52%	1%	32%	10%	19%	48%	**25%**

6. Summary and Conclusions

The Virtual Private Machine framework is a means for supporting microarchitecture resource sharing. In this framework, hardware resource sharing mechanisms allow hardware resources to be allocated to executing threads, thus providing a Virtual Private Machine. VPMs provide threads QoS so that a thread's performance is at least as good as a standalone, real private machine having the same resources as allocated to the VPM.

We use shared caches as a vehicle to develop the VPM framework. Virtual Private Cache hardware consists of two major components: the VPC Arbiters, which manage shared resource bandwidth, and the VPC Capacity Manager. The VPC Arbiter is a significant component of the overall VPM framework. Although, the arbiter design is proposed in the context shared caches it can be applied to other shared microarchitecture resources. Both the VPC Arbiter and VPC Capacity Manager provide minimum service guarantees that when combined achieve the global VPM QoS objective. However, due to the nature of modern out-of-order processors we can not prove that VPMs meet their QoS performance objective unless we assume performance monotonicity. That is, a thread's performance is a monotonically increasing function of the amount of resources a thread is offered.

Our evaluation shows that existing cache arbiter policies allow threads that share cache bandwidth to significantly affect each other. In contrast, we show VPCs meet their QoS performance objective on all workloads studied and have a fairness policy amenable to general-purpose multithread systems. Furthermore, we show VPCs improve CMP throughput by eliminating negative interference.

In the short term, the mechanisms proposed here can be applied to eliminate negative interference in CMP systems with off-the-shelf software. In the long term, operating systems, virtual machine monitors, and

applications, can greatly benefit from implementing policies that control VPM hardware mechanisms. We plan to investigate the VPM hardware / software interface and VPM programming models in future work.

7. References

[1] Bennett, J. C., Zhang, H. Hierarchical packet fair queuing algorithms. In *Proc. on Apps. Tech. Arch. and Protocols for Comp. Comm.* (Palo Alto, Ca., Aug. 28 - 30, 1996).

[2] Brown, J., Application Customized CPU Design: The Xbox 360 Story, on *IBM Developerworks*, Dec. 2005.

[3] Cazorla, F. J., Ramirez, A., Valero, M., Fernandez, E. 2004. Dynamically Controlled Resource Allocation in SMT Processors. In *Proc. of the 37th Intl. Symp. on Microarcch.* Dec. 2004.

[4] Chetto, H. and Chetto, M. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Trans. on Software Eng.* 15, 10, Oct. 1989.

[5] Emer J., et al. Asim: A Performance Model Framework. *Computer* 35, 2, Feb. 2002.

[6] Kim, S., Chandra, D., Solihin, Y. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proc. of the 13th Intl.l Conf. on Parallel Arch. and Comp. Techniques*, Sept. 2004.

[7] Hennessy J. L., Patterson, D. A., *Computer Architecture: A Quantitative Approach , Third Edition,* Morgan Kaugmann, 2002.

[8] *IBM PowerPC 970FX RISC Microprocessor User's Manual, Version 1.6*, Dec. 2005.

[9] Iyengar, V. S., Trevillyan, L. H., Bose, P. 1996. Representative Traces for Processor Models with Infinite Cache. In *Proc. of the 2nd Symp. on High-Perf. Comp. Arch..* Feb. 1996.

[10] Iyer, R. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *Proc, of the 18th Intl. Conf. on Supercomputing* (Malo, France, June 26 2004.

[11] Kalla, R. Sinharoy, et al., IBM Power5 Chip: A Dual Core Multithreaded Processor, *IEEE Micro*, Mar/Apr 2004.

[12] Kumar, R., Zyuban, V., Tullsen, D. M. 2005. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In *Proc. of the 32nd Intl. Symp. on Comp. Arch.* June 2005.

[13] Kongetira, P., Aingaran, K., and Olukotun, K. 2005. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro* 25, 2, Mar. 2005.

[14] Le Boudec, J.Y., Thiran, P., Network Calculus, Springer Verlag, 2004.

[15] Luo, K., Gummaraju, J., Franklin, M.. Balancing throughput and fairness in SMT processors. In *Proc. of the Intl. Symp. on Perf. Analysis of Systems and Software*, Jan. 2001.

[16] Mak, P., et al. Shared-cache clusters in a system with a fully shared memory. *In IBM Journal of R&D* Vol. 41 July/Sept. 1997.

[17] *Micron. 1Gb DDR2 SDRAM Component: MT47H128M8B7-25E*, June 2006.

[18] Nesbit, K.J., Aggarwal, N., Laudon, J., Smith, J.E., Fair Queuing Memory Systems, In Proceedings of 39th Intl. Symp. On Microarchitecture, Dec 2006.

[19] Parekh, A. K. Gallager, R. G. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Networks* 1, 3 Jun. 1993.

[20] Rafique, N., Lim, W., Thottethodi, M. Architectural support for operating system-driven CMP cache management. In *Proc. of the 15th Intl. Conf. on Parallel Arch. and Comp. Tech.* (Seattle, WA, Sept.16 - 20, 2006). PACT '06.

[21] Sariowan, H., Cruz R.L., Polyzos G.C. Scheduling for quality of service guarantees via service curves. In *Proc. of the 4th Intl. Conf. on Comp. Comm. and Networks,* Sept. 1995.

[22] Skadron, K. Clark, D. W. Design Issues and Tradeoffs for Write Buffers. I*n Proc. of the 3rd Symp. on High-Perf. Comp. Arch.* Feb. 1997.

[23] Tendler, J. M., et. al., *Power4 System Mircoarchitecture, Technical white paper*, Oct. 2001.

[24] Verghese, B., Gupta, A., Rosenblum, M. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *Proc. of the 8th Intl. Conf. on Arch. Support For Prog. Lang. and Op. Sys.*, Oct. 1998.

[25] Wilton, S., Jouppi, N., CACTI: An Enhanced cache Access and Cycle Time Model , In *Journal of Solid-State Circuits*, Vol. 31, May 1996.

[26] Zhang H. Service Disciplines for Guaranteed Performance Service in Packet-switching Networks, In *Proc. of the IEEE*, vol.83, Oct. 1995.