Virtual Private Caches

Kyle J. Nesbit
University of Wisconsin- Madison
Department of Electrical and Computer Engineering
kjnesbit@ece.wisc.edu

James Laudon
Sun Microsystems Inc.
james.laudon@sun.com

James E. Smith
University of Wisconsin – Madison
Department of Electrical and Computer Engineering
jes@ece.wisc.edu

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, thus providing applications with an upper bound on execution time regardless of other thread activity. Virtual Private Caches (VPCs) are an important element of VPMs. VPC hardware consists of two major components: the VPC Arbiter, which manages shared cache bandwidth, and the VPC Capacity Manager, which manages the cache storage. Both the VPC Arbiter and VPC Capacity Manager provide minimum service guarantees that, when combined, achieve QoS for the cache subsystem.

Simulation-based evaluation shows that conventional cache bandwidth management policies allow concurrently executing threads to affect each other significantly in an uncontrollable manner. The evaluation targets cache bandwidth because the effects of cache capacity sharing have been studied elsewhere. In contrast with the conventional policies, the VPC Arbiter meets its QoS performance objectives on all workloads studied and over a range of allocated bandwidth levels. The VPC Arbiter's fairness policy, which distributes leftover bandwidth, mitigates the effects of cache preemption latencies, thus ensuring threads a high-degree of performance isolation. Furthermore, the VPC Arbiter eliminates negative bandwidth interference which can improve aggregate throughput and resource utilization.

Categories and Subject Descriptors

C.0 [***Computer System Organization***]: hardware/software interface, system architectures

General Terms

Management, Performance, Design, Experimentation.

Keywords

Chip Multiprocessor, Shared Caches, Quality of Service, Performance Isolation, Soft Real-Time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA'07, June 9–13, 2007, San Diego, California, USA.
Copyright 2007 ACM 978-1-59593-706-3/07/0006...$5.00.

1. INTRODUCTION

To provide both efficiency and high throughput, CMP-based systems rely heavily on resource sharing, especially in the memory hierarchy. Shared resources include both storage capacity and bandwidth, comprising ports, interconnection paths, and associated buffering. 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 managed through hardware mechanisms. For these shared resources, the hardware mechanisms affect the amount of inter-thread interference, both destructive and constructive, and, in turn, this affects the threads' performance predictability.

In many general purpose applications, execution threads must attain a minimum level of performance, irrespective of the other threads that 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. 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 systems.

Historically, QoS has been of interest in computer networking, and recently it has become the subject of computer architecture research. The 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, a QoS objective is to provide applications an upper bound on execution time, or, alternatively, a consistent execution time regardless of other thread activity.

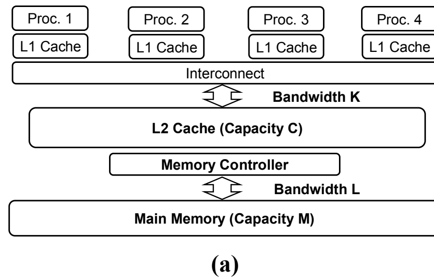
QoS objectives are usually achieved through resource provisioning; that is, by allocating a certain quantity of resource(s) to a given network flow (or computation thread in our case). The goal of the research reported here is to provide microarchitecture-level *mechanisms* whereby allocations of shared 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 policies, and it is the proposed mechanisms that assure 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* more specifically. These are described more fully in the following two subsections. Briefly, a virtual private machine (virtual private cache) is defined through the allocation

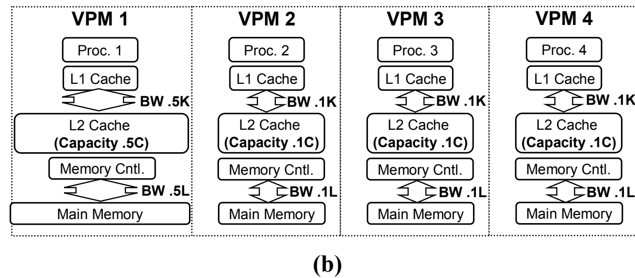
of a portion of shared resources. The objective is that a virtual private machine (virtual private cache) should provide performance at least as good as a real private machine (real private cache) having the same resources. The effectiveness of the proposed mechanisms is demonstrated through simulations 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 work here we assume single-threaded processors, although the concepts extend to multi-threaded processors. The system contains private L1 caches, 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.



(a)



(b)

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

By implementing 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. For example, a CMP with four hardware-supported threads can be divided into four VPMs where the resources, both bandwidths and capacities, are divided evenly among executing threads. This configuration is well suited for general purpose operating systems that distribute processor resources using time slices – a time slice is a unit of a processing resources. Allocating each time slice to an equally provisioned VPM maintains the existing abstraction provided to the OS by ensuring that each time slice represents a consistent amount of processing resources.

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 a *fairness policy*. For example, 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.

The goal is for a VPM to yield thread performance at least as good as on standalone, real private machine having the same resources, regardless of any other threads that might be running concurrently (i.e., *performance isolation*). Furthermore, if one or more of the VPMs do not need all of their allocated resources and/or there are unallocated resources, then any excess is distributed among the other VPMs, potentially providing them with additional performance.

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 Arbiter, which manages shared bandwidth, and the VPC Capacity Manager, which manages storage resources. The VPC Arbiter design is based on network fair queuing (FQ) algorithms that are adapted to microarchitecture resources.

Cache capacity sharing has been studied and evaluated elsewhere [7][10][13][22][23][29], 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 provide threads with performance isolation 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 for heterogeneous workloads, VPCs improve throughput by eliminating negative bandwidth interference.

2. RELATED WORK

Fair Queuing Memory Systems *(FQMS)* uses basic network FQ principles in order to provide QoS and fairness to threads competing for SDRAM memory system bandwidth [21]. The work presented here extends FQMS in several significant ways. First, the VPM is proposed 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, the FQ memory scheduler uses approximate FQ methods to account for the complex timing constraints of an SDRAM memory system. In contrast, the VPC arbiters are a strict implementation of FQ algorithms. The result is an arbiter that guarantees minimum bandwidth and is amenable to managing microarchitecture resources. Finally, we study many details of bandwidth sharing that are ignored in the FQMS work, e.g., preemption latencies, performance monotonicity, differentiated service, and strict isolation.

Performance isolation has become an important topic in operating system and virtual machine research [1][28][31][8]. Performance isolation and QoS are closely related. The objective of both performance isolation and QoS is to ensure a task a minimum level of performance, regardless of other tasks in the system. General-purpose operating systems and virtual machines attempt to provide tasks performance isolation by running a task at a fraction of its speed on a dedicated system [26], regardless of other tasks in a shared system. In general, this abstraction is usually implemented by time multiplexing tasks onto shared resources.

Most performance isolation research focuses on software policies. However, software policies are unable to provide a sufficient level of performance isolation in CMP-based computer systems, primarily because the hardware does not support appropriate low level resource sharing mechanisms. Such mechanisms, applied to shared caches, are the topic of this paper.

Carzola et al. proposed hardware-based QoS policies to dynamically adjust resource allocations in SMT processors based on previous time period(s) [4]. This approach is based on prediction, which may not work for all applications and does not offer any QoS guarantees.

METERG is a methodology to test and deploy real-time applications on multiprocessor systems with shared microarchitecture resources [17]. The basic premise of the METERG methodology is consistent with the VPM framework; both provide QoS through hardware mechanisms that provide service guarantees. In contrast, METERG is primarily an application development methodology; Lee and Asanović provide few details of hardware mechanisms. The QoS policies and mechanisms presented in this paper can be applied to the METERG methodology.

Cache capacity sharing has been the topic of a significant amount of research [7][10][13][22][23][29]. Most cache capacity sharing algorithms optimize a performance metric related to aggregate throughput[22][29], although there is some cache sharing research that has focused on metrics that are subjectively fair [7][10]. Aggregate throughput and fairness sharing algorithms tend to be very similar – they use the same basic algorithms, but optimize different metrics. Often fairness metrics are based on proportionate slowdown [7][10] where all concurrently executing threads slow down by the same percentage, as compared with running alone on the same hardware.

Techniques that target aggregate metrics do not provide performance isolation, thus making them incompatible with most operating system and virtual machine performance abstractions. For example, with proportionate slowdown, if one thread is resource constrained in a multithreaded execution, it will cause all threads in the system to run slower. This is an undesirable behavior for a general-purpose system. Furthermore, the operating system should be the primary resource manager in a computer system because it has a global view of the system's resources and workload objectives. Hardware implemented policies targeted at aggregate metrics may compete with the operating system's policies and prevent the operating system from effectively accounting for and managing resource sharing.

Iyer discusses cache QoS and a framework of possible solutions [13]. However, Iyer's discussion is based on a definition of QoS that may not apply to certain types of shared resources, in particular, bandwidth resources. In contrast, we propose bandwidth and capacity QoS policies that are both based on the same, well established QoS definitions and solutions from computer networking research.

3. BACKGROUND

3.1 Shared Caches

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 [9]. However, recent work on shared cache crossbar interconnections illustrates that cache banking does not scale well [14]; cache banking can significantly increase latency, area, and power. Therefore, CMP designers are targeting higher cache utilization [15], thus making threads sharing cache bandwidth more susceptible to 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 [30][11][3]. Write-through L1 caches are used because they require less L1 array bandwidth, eliminate many of the complexities of protecting on-chip data with ECC, and consequently, have lower latency [19]. A L2 cache is illustrated in Figure 2; this structure would also apply to shared L3 caches, if present. In the baseline shared cache microarchitecture, processors are connected to the cache banks via a crossbar interconnect (Figure 2a). Each processor has private read and write ports into each cache bank. Each cache bank has a return data bus connected to all processors on the crossbar [14]. To reduce power the crossbar interconnection operates at half the core frequency.

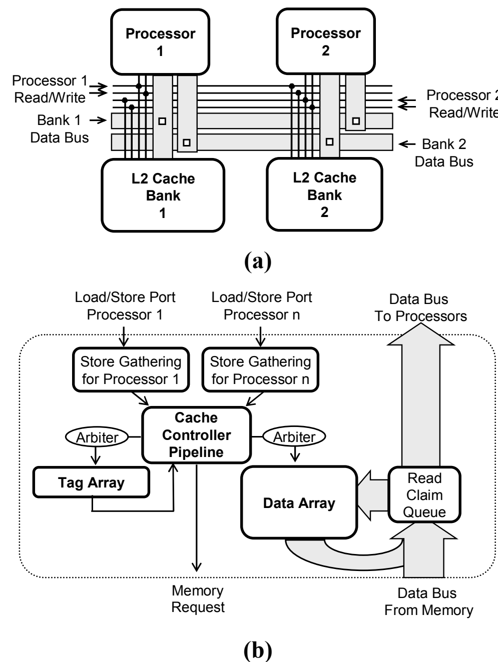


Figure 2. Shared L2 caches a) Processor-cache interconnect. b) Cache bank structure.

The logical structure of a single cache bank is illustrated in Figure 2b. Read and write requests first arrive at the store gathering buffers (one per processor). Store gathering is an efficient method for supporting write-through L1 caches [27]. At the store gathering buffer, incoming store data for the same cache line are merged; 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. 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* is implemented [27]. 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* is used [27].

After a request passes through the store gathering buffers, the controller checks that the request does not conflict with other pending L2 requests, i.e., to assure that the new request will not cause a race or violate the memory consistency if it is allowed to proceed [19]. If the request does not create a conflict, it is allocated a cache controller state machine. The cache controller state machine initiates tag array, data array, and external coherence requests in order to fulfill the request. 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 stores the cache line and its 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.

There are three shared resources in the baseline cache microarchitecture: 1) the tag array bandwidth, 2) the data array bandwidth, and 3) the storage capacity. Coherence state machines are statically partitioned [14].

The tag array and data array bandwidth resources are managed by arbiters that multiplex threads' requests onto the shared resources (Figure 2b). In addition, the data bus back to the processors is shared, but it operates at the same rate as the data array; therefore, the data array arbiter controls access to both resources. For private caches, Read-over-Write, First-Come First-Serve (RoW-FCFS) arbiter policies are effective at improving performance and optimizing resource utilization [9][27]. RoW-FCFS first prioritizes read over write requests (RoW) and then prioritizes the oldest request (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 [19]. Therefore, for the multithreaded case, we use the FCFS arbiter without RoW as our baseline. We also consider round-robin arbitration, which is sometimes considered "fair", but, as we will show, it does not provide performance isolation.

Cache capacity resources are managed through cache replacement [7][13][22][23][29]. In general, each line in a shared cache is associated with a specific thread. When a thread causes a cache miss, the replacement policy takes cache capacity associated with one thread and assigns the capacity to the thread that caused the cache miss. In effect, the replacement policy multiplexes threads onto the shared cache capacity resources.

3.2 Fair Queuing

A fair queuing scheduling algorithm offers guaranteed bandwidth to simultaneous network flows over a shared link [2][16][24][25][33]. 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 allocated, but unused bandwidth is *work-conserving* [5][16]. 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).

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 the flow's share ϕ_i. In a basic implementation, when 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. Prioritizing packets earliest virtual finish-time first [2][16][24] yields earliest deadline first (EDF) scheduling [5].

[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 its deadline could be satisfied. However, if there is a non-zero preemption latency, the deadline may not be satisfied. In general, with EDF scheduling, as long as the resource is not overloaded (e.g., $\sum\phi_i \leq 1$), a request will finish its service no later than the <*deadline*> + <*max preemption latency*> [5][16]. 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 QoS mechanisms 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 registers specify a share of cache capacity (α_i), and a share of tag and data array bandwidth (ϕ_i). In their full generality, these mechanisms allow software to allocate each of the bandwidth resources independently (via separate control registers), but, to simplify the discussion, we consider the case where the shares are the same.

The VPC controller consists of the VPC Arbiters, which manage the tag and data array bandwidth resources, and the VPC Capacity Manager, which manages the cache capacity resources. The VPC Arbiter and VPC Capacity Manager are compatible– they are based on the same definition of QoS and fairness – and in combination they achieve the VPC QoS objective.

4.1 VPC Arbiter

Each tag and data array has a VPC arbiter that selects a pending request to access the shared resource next (see Figure 2b in the background section). All VPC arbiters are implemented in essentially the same way. The basic implementation is parameterized and each arbiter instance is configured to manage its resource's specific timing characteristics. For example, the

data array's arbiter is configured for the data array's bandwidth constraints and the fact that write operations on the data array require two array accesses.

4.1.1 Arbiter Implementation

The proposed arbiter implementation is equivalent to the FQ algorithm outlined in the background section, but is optimized to reduce circuit complexity and support common performance optimizations that reorder requests after they have entered arbitration. In particular, this implementation reduces circuit complexity by eliminating the priority queue in the implementation outlined in the background section. Instead of the priority queue, a VPC arbiter has a FIFO buffer for each thread (see Figure 3). When a request enters arbitration for a resource, the request's ID is added to its thread's arbiter buffer. A request ID is a reference to a controller state machine, where the request's state information is stored. The arbiter buffers are small; a request ID only requires a few bits of storage.

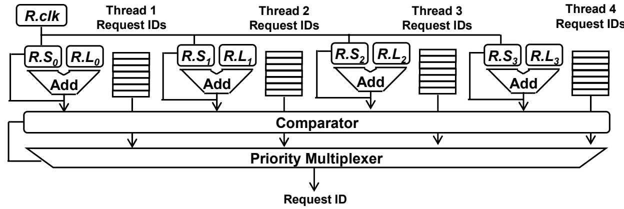


Figure 3. VPC Arbiter Hardware Implementation

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 the VPC implementation is a real time counter that simply counts clock ticks and is used for determining arrival times a_i^k (Section 3.2). The $R.L_i$ register stores a thread's virtual service time L / ϕ_i, where L is the service requirement of request m_i^k (the *k*th request from the *i*th thread) and ϕ_i is the thread's fraction of the resource. The service requirement L is the latency of the shared resource (see Table 1 in the evaluation section). For write requests on the data array, the service requirement is twice the latency of the data array. The $R.L_i$ register only has to be calculated when the thread's service share ϕ_i is changed.

A key element of the implementation is the $R.S_i$ register which tracks the time the *virtual resource* will become available for a thread's next request, i.e., the next request's virtual start-time S_i^k. At the time a request is selected for service, then its $R.S_i$ register is updated to indicate the time the virtual resource will be available for the next request. Following paragraphs describe the implementation algorithm in more detail.

When a request from thread *i* arrives at the FIFO buffer and the buffer is empty, then the virtual resource is either available immediately (if $R.S_i < R.clk$) or it will not be available until $R.S_i$. (the time the preceding request finishes with the virtual resource). This is shown in register transfer statement 3.

Arrival

[3] *If thread i's queue is empty and* $R.S_i < R.clk$ *then* $R.S_i \leftarrow R.clk$*; else* $R.S_i$ *is unchanged*

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

If thread *i*'s FIFO buffer *is not* empty when m_i^k arrives, then the virtual resource is backlogged, and m_i^k's arrival time will have no effect on its virtual start-time. Rather, its virtual start time will be the virtual finish time of thread *i*'s immediately preceding request, and, as will be described shortly, the preceding request will update $R.S_i$ accordingly at the time it is selected.

The VPC Arbiter inspects the thread requests at the heads of the FIFO buffers and calculates their virtual finish times. The virtual finish time of request m_i^k is F_i^k, and it is calculated (statement 4) as the sum of m_i^k's virtual start-time, which is stored in the $R.S_i$, and its virtual service time, which is stored in $R.L_i$.

Abriter Calculation

[4] $F_i = R.S_i + R.L_i$

($R.S_i + 2{*}R.L_i$ for write requests on the data array)

The arbiter selects the request with the earliest virtual finish-time. It then updates the $R.S_i$ register with the just-computed finish time of the selected request (statement 5) to reflect the time the virtual resource will become available for thread *i*'s next request. It removes the request's ID from the queue and uses the ID to identify the request's controller state machine which initiates the access to the shared resource.

Arbiter Selection

[5] $R.S_i \leftarrow F_i$

The implementation outlined in statements 3, 4, and 5, enables performance optimizations that use intra-thread request reordering, while maintaining the bandwidth guarantees and fairness properties of the basic FQ algorithm. Any request in a thread's buffer can be selected to access the resource, and doing so does not change the amount of service (bandwidth) the thread receives relative to other threads [16].

Intra-thread reordering occurs within the thread's buffer (i.e., the buffer is no longer strictly FIFO) and does not affect the rest of the arbiter design. The primary reordering optimization we are interested in here is prioritizing intra-thread 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 Scheduling and Preemption

Servicing the thread with the earliest virtual finish-time first is the same as scheduling *earliest deadline first (EDF)* [5]. As described in the background section, with EDF scheduling and a non-preemptible resource, which is the case for shared cache resources, a request will finish its service by the <*deadline*> + <*resource's max service time*> , where a request's deadline is its virtual finish-time. That is, a cache request may be delayed by another thread's maximum service time (in the worst case).

Despite the worst case behavior, the resource latency of shared cache resources often does not affect a thread's performance (as will be shown in the evaluation section). General purpose processor traffic tends to contain bursty L2 accesses, and preemption latency is *paid only once per burst* [16]; 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 (where misses are less bursty) are more susceptible to preemption latency effects.

4.1.3 Arbiter Fairness Policy

VPC arbiters mitigate the effects of preemption latency through a fairness policy that is tailored to general-purpose processor traffic. As described above, a fairness policy distributes excess bandwidth. Excess bandwidth is bandwidth that is not allocated to any thread or it is bandwidth that has been allocated but is not used by the thread to which it is allocated. The *VPC arbiters' fairness policy* distributes excess bandwidth to the thread that has received the least excess bandwidth in the past (relative to its service share ϕ_i) [21][24]. This fairness policy differs from the fairness policy commonly used in networks. In contrast with network fairness, the VPC arbiter's fairness policy takes into account past bandwidth usage.

The VPC arbiter uses a different fairness policy because a general-purpose processor and its memory system behave like a closed system, i.e., the rate that a processor injects requests into the memory system depends on the average latency of the requests. Threads that consume more cache bandwidth tend to increase the average latency experienced by other threads – primarily by increasing the average preemption latency experienced by other threads. These highly consumptive threads also tend to have more memory level parallelism and are less sensitive to latency. Therefore, the highly consumptive threads should not receive excess bandwidth before threads that have received less excess bandwidth in past time periods. In general, the VPC arbiters' fairness policy is very effective at averaging out preemption latency. However, in cases where a thread is very 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.

The VPC arbiter implementation described in subsections 4.1.1 and 4.1.2 implements the desired fairness policy. VPC arbiters schedule requests earliest virtual finish-time first. In addition to acting as a deadline, the virtual finish-time of a thread's next request is an indicator of the amount of 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-times will run well ahead of the clock register (*R.clk*) and the virtual finish times of other threads. Therefore, servicing threads according to earliest virtual finish-time (as described above) ensures threads meet their QoS guarantees, and naturally distributes excess bandwidth to the backlogged thread that has received the least excess bandwidth in past time periods.

In contrast with the VPC arbiter, network fair queuing algorithms often use a virtual clock to implement their fairness policy. A virtual clock determines request arrival times (a_i^k) and computes virtual finish-times [2][33]. A virtual clock advances faster when fewer threads are backlogged, which increases the rate at which the non-backlogged threads' virtual finish-times advance, thus reducing the penalty for the backlogged threads that are consuming excess service. A detailed comparison of fairness policies and the extent to which 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 way-partitioned replacement policy that requires few changes to the baseline cache microarchitecture. Hardware mechanisms that support similar way-partitioned replacements policies have been proposed in the past [7][13][22][23][29]. The VPC Capacity Manager provides each thread with 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, where α_I is thread i's allocated share of the cache. We discuss the rationale behind this property later in Section 4.3, after we introduce the concept of performance monotonicity. Note that in general the sum of the α_I's may be less than one, i.e., there may be unallocated cache capacity. 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 thread i, such that thread i occupies more than α_i 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 *i* 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 Capacity Manager Fairness Policy

As defined above, there may be multiple threads that occupy more than α_i 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 a cache line will go unused in the future. Theoretically, the frame holding such a cache line could be redistributed to another thread without affecting the owner's QoS, but without an oracle, this cannot be done. We plan to explore the topic of work conserving capacity managers in 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. [7], 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 our evaluation 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

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 when an associated 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 resources will provide QoS.

Many computer systems do not strictly satisfy the performance monotonicity assumption. Because of the complexity of modern processors and their use of speculation, there are situations where 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 due to cache pollution.

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 VPC Capacity Manager (way-partitioning) satisfies performance monotonicity. However, the proposed VPC method comes at a cost; it increases the minimum 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, 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 set and cache way resources may offer a thread more cache sets and fewer cache ways, which can increase the thread's conflict misses and decrease its performance.

5. EVALUATION

5.1 Performance Model

We use a detailed structural simulator developed at IBM Research to evaluate the VPC arbiters. The simulator adopts the ASIM modeling methodology [6], 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 [11]. 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 L2 cache size, decreased the line size in order to reduce the demand on memory bandwidth, and added another cache bank in order to support additional 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 [32] – 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 receive the entire line takes 22 processor cycles (see the timing diagram in Figure 4). The L2 controller state machines are partitioned [14] – each processor is allocated eight state machines per cache bank. Threads are allocated equal portions of the cache ways, and there are no unallocated cache ways, e.g., $\alpha_i = .25$ for all i. The 970 instruction and data prefetchers are disabled. VPC supported prefetching is a topic for future work. For the uniprocessor baseline, we use the RoW-FCFS arbiter policy. The same arbiter policy is applied to the tag and data array. For the multiprocessor baseline, we use FCFS because RoW-FCFS can cause starvation.

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

Processors	4 processors operating at 2 Ghz
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 latency, 16 byte data bus per bank
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, 8 store gathering buffer entries per thread, read bypassing, retire-at-6 policy, partial-flush on read conflict latency
Memory System	DDR2-800, controller operates at ½ core frequency, 16 transaction buffer entries per thread, 8 write buffer entries per thread, closed page policy, 1 channel per thread, 2 ranks per channel, 8 banks per rank

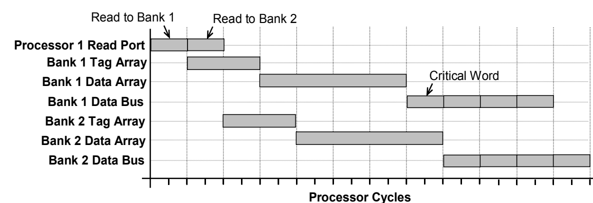


Figure 4. Cache Timing Diagram of back-to-back reads to different cache banks

The simulator has a cycle accurate model of an on-chip memory controller attached to a DDR2-800 memory system [21]. 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 (see Table 2) and the SPEC 2000 benchmark suite to evaluate the performance of the VPC arbiters. The microbenchmarks are designed to stress performance isolation features. 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) [11] 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

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 [11] 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 [11] 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. A single-thread of the *Stores* benchmark 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. Our results show that with the VPC arbiters, CMP designers can focus on designing for the common case, rather than the worst case.

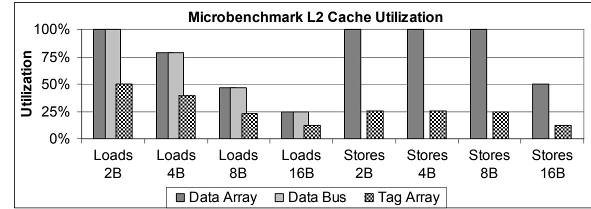


Figure 5. L2 Cache Utilization of the Microbenchmarks

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. Each trace has been verified to be statistically representative of an entire SPEC application [12]. Figure 6 shows the cache utilizations of the individual SPEC benchmarks. As one would expect, 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. Throughout our evaluation, we use data array utilization as an indicator of a thread's *aggressiveness*.

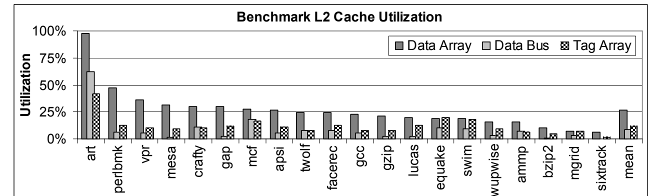


Figure 6. L2 Cache Utilization of the SPEC Benchmarks

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.

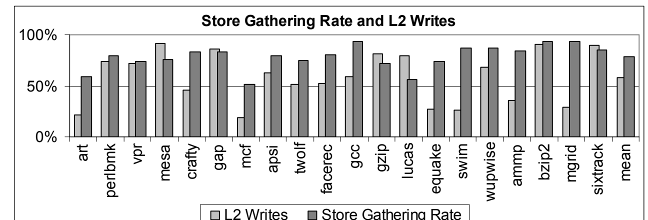


Figure 7. Percentage of L2 Requests that are Writes and the Store Gathering Rate

5.3 QoS Performance

To illustrate performance isolation and QoS we provide results from a number of experiments. Most experiments involve multi-threaded execution, and to gauge QoS we define two standards for comparison: target IPC (instructions per cycle) and QoS IPC.

A benchmark's *target IPC* is the benchmark's IPC performance when running on a uniprocessor cache with the same

resources as the thread's allocated VPC. As described above, a benchmark ideally should go no slower than this target, and if the thread is offered excess bandwidth, it may go faster. However, target IPC does not take into account the effects of resource preemption latencies. Therefore, benchmarks may not meet their target IPC; although we will show that they do in most cases. 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 $\lfloor \alpha_i * <baseline\ cache\ ways> \rfloor$ 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.

A benchmark's *QoS IPC* takes into account the resources' worst case preemption latencies. QoS IPC is a lower bound on a thread's IPC. 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 bandwidth the same. For example, we simulate a uniprocessor system with the latencies equal to $1/\phi_i$ * *<baseline resource latency>* + *<max preemption latency>* cycles and bandwidth equal to ϕ_i / *<baseline resource latency>* accesses per cycle.

5.4 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), Round-Robin (RR), and VPC with five different VPC bandwidth 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 (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.

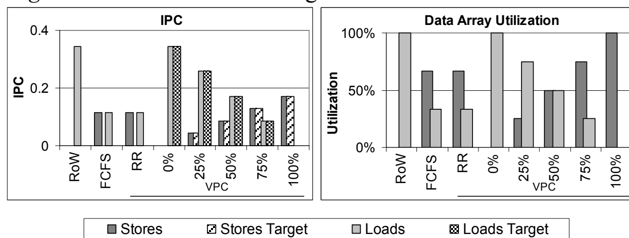


Figure 8. Loads and Stores Microbenchmarks IPC and Data Array Utilization

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.

With FCFS, requests from the *Loads* and *Stores* benchmarks are interleaved uniformly, i.e., there is one store from the *Stores* benchmark for every load from the *Loads* benchmark. With a uniform interleaving of requests, however, the data array bandwidth is unevenly shared, and the demands of the two threads appear to interfere with each other. 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. The round-robin arbiters also interleave requests uniformly, and, consequently, they provide the same performance and data array utilization as the FCFS arbiters.

The VPC arbiters precisely provide each thread its allocated share of the cache bandwidth over a broad range of allocations. The VPC arbiters are able to divide bandwidth so well because both the *Loads* and *Stores* benchmarks keep constant pressure on the L2 cache – there are always pending requests from both threads. Because the performance of these benchmarks depends almost completely on L2 bandwidth, both benchmarks meet their target IPCs; there are no preemption latency effects. In addition, the VPC arbiters' fairness policy has no effect on their performance because there is no excess bandwidth to distribute.

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*). We model the subject thread with FCFS, round-robin, and three VPC bandwidth allocations: $\phi_1 = .25$, $\phi_1 = .5$, and $\phi_1 = 1$. Remaining bandwidth is allocated equally amongst the background threads. For example, the label VPC *.25* represents the configuration where the subject thread is allocated *.25* of the cache bandwidth ($\phi_1 = .25$) and each background thread is allocated *.25* ($\phi_i = .25$).

Figure 9 shows the IPC and data array utilization of the subject thread. The IPCs are 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 normalized IPC of the subject thread, the graph shows the target IPC for $\phi_1 = .25$ and $\phi_1 = .5$, and the QoS IPC for $\phi_1 = 1$.

This experiment illustrates 1) the VPC arbiters' ability to isolate a foreground task from aggressive (possibly even malicious) background tasks, 2) the effects of the VPC arbiters' fairness policy, and 3) the VPC arbiters' ability to provide differentiated service.

The FCFS and round-robin arbiters do not provide performance isolation. The FCFS arbiters fail to meet the subject threads' Target IPCs (Target .25) on all 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). FCFS is unable to provide performance isolation because it is a greedy policy. Threads that have higher resource demands are given unimpeded access to the shared bandwidth resources.

The round-robin arbiters fail to meet the subject threads' Target IPCs on 12 out of 20 benchmarks – the subject threads average normalized IPC is *.54*. The round-robin arbiters provide

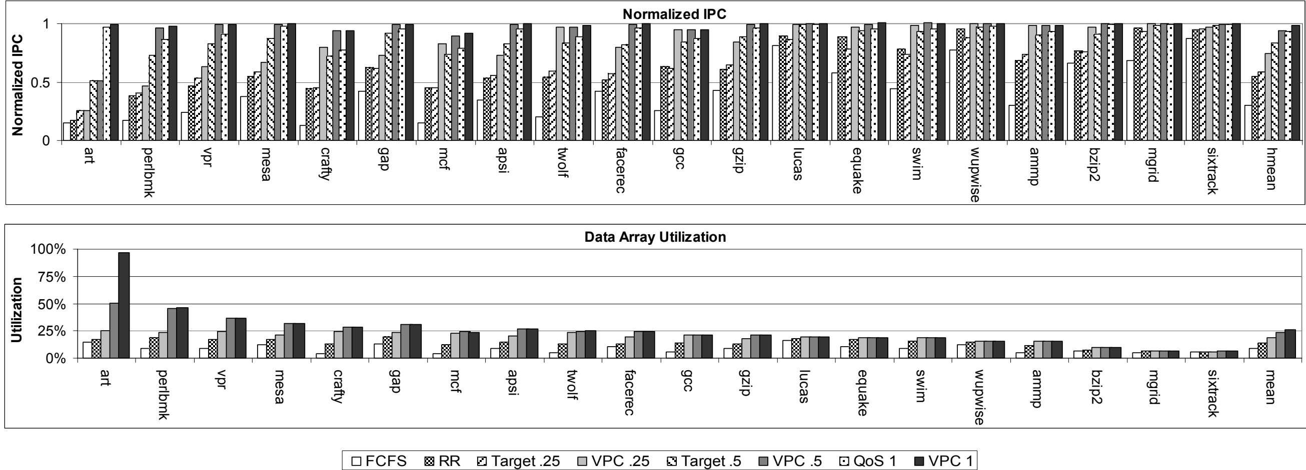


Figure 9. Subject Thread Normalized IPC and Data Array Utilization

better isolation than the FCFS arbiters because they distribute bandwidth more fairly than FCFS. However, the round-robin arbiters still do not provide a sufficient level of QoS and performance isolation. There are two reasons why round-robin fall short. First, round-robin does not account for requests' different service requirements; threads that have a higher percentage of stores receive more service. Second, round-robin does not take into account past resource usage. Therefore, aggressive threads are able to increase the average latency of less aggressive threads, and force the less aggressive threads to backoff.

In contrast with the FCFS and round-robin arbiters, 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. Preemption latency does not have a significant effect with these VPC configurations because 1) the preemption latencies are relatively small compared to the allocated VPC resource latencies, and 2) the fairness policy penalizes aggressive threads for consuming excess resource, thereby causing the less aggressive (more latency-sensitive) subject thread's requests to receive service soon after they arrive at the cache controller.

For the $\phi_I = 1$ VPC configuration, the subject threads meet their QoS IPCs – the QoS IPCs take into account the resources' preemption latencies. In this case, 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 decreases, the variation between the thread's actual IPC and its target IPC increases. This occurs for a two reasons. As the allocated share decreases, the range of guaranteed cache latencies increases – the range is roughly [*<baseline latency>* , $1 / \phi_i *$ *<baseline latency>* + *<maximum preemption latency>*]. Furthermore, with 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 much closer to the *<baseline latency>* end of the range rather than the $1 / \phi_i *$ *<baseline latency>* + *<maximum preemption latency>* end of the range. We intend to study the fairness policy in more detail in future work, specifically whether an alternative fairness policy can provide tighter bounds on performance than those derived using network FQ theory.

The top eight benchmarks in Figure 9 demand more than *.25* of the cache bandwidth (see Figure 6). With the $\phi_I = .25$ configuration, these threads are bandwidth constrained. On average, they receive *23%* of the cache bandwidth. The average normalized IPC of the lower twelve subject threads is *.95*. The average normalized IPC of all the subject threads (with $\phi_I = .25$) is *.75*, much better than FCFS and round-robin.

The only subject thread that demands more than .5 the cache bandwidth is *art*. With the $\phi_I = .5$ 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 $\phi_I = 1$ 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 $\phi_I = 1$ VPC configuration is good for prioritizing a foreground thread while running less important background tasks.

For our last experiment, we compare the performance of FCFS (a greedy technique), RR (which has some fairness attributes), and VPC (a QoS and fair technique) on multiprogram SPEC 2000 workloads. The purpose of this experiment is to show that providing QoS and fairness improves aggregate throughput and resource utilization when compared to a greedy technique. In real-time system, this is often not the case – providing QoS often reduces utilization. For a general-purpose system low resource utilization is unacceptable, and as this experiment illustrates, it is not the case with the VPC arbiters. 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 workloads with the baseline 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*. With the VPC arbiters, each benchmark meets its target IPC, i.e., the normalized IPC of each benchmark is greater than one.

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 between the FCFS and VPC arbiters 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 QoS. Furthermore, there is *-.21* correlation between the threads' normalized IPCs with FCFS and the threads' normalized IPCs with the VPC arbiters, when *mcf* 's IPCs are excluded (*mcf* is an outlier), 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 and excess bandwidth is offered to the less aggressive threads first. Therefore, relative to FCFS, more aggressive threads tend to have lower performance and less aggressive threads tend to have better performance.

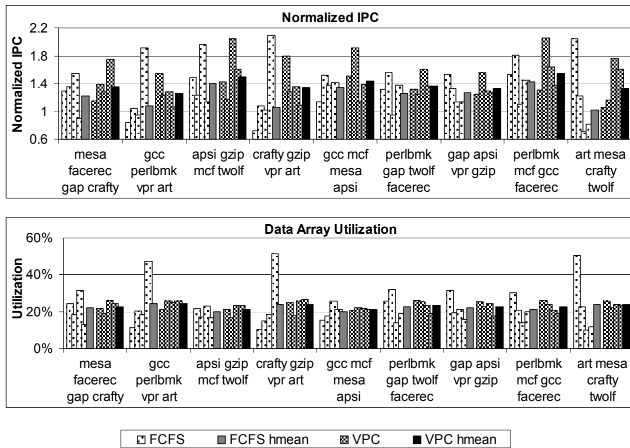


Figure 10. Multiprogram Workload Normalized IPC and Data Array Utilization

With the FCFS arbiters, the average (arithmetic mean) data array utilization per thread 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 per thread is *23%* 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*. On average, the VPC arbiters improve data array utilization by *4%* on the combined workloads.

We use two performance metrics to compare the performance of the VPC, RR, and FCFS arbiters when running the multiprogram workloads: the harmonic mean of each workload's normalized IPCs and the weighted speedup. The harmonic mean of the normalized IPCs emphasizes both throughput and fairness [18]. The weighted speedup is the sum of the normalized IPCs; it measures aggregate throughput. Results are in terms of performance improvements with respect to the FCFS baseline. On average, the VPC arbiters improve the harmonic mean of normalized IPCs by *14%* and the weighted speedup by *10%*. The RR arbiters improve the harmonic mean of normalized IPC by *10%* and the weighted speedup by *4%*. This result illustrates that providing QoS and fairness actually improves throughput not only with respect to the greedy FCFS policy, but also when compared with the more fair RR policy. This result is very positive considering that improving an aggregate performance metric was not our primary objective – our primary objective was to develop consistent (performance isolated) QoS mechanisms for managing shared cache resources. Note that we could have exploited performance isolation by adjusting the ϕ_i values in order to optimize one of the popular aggregate performance metrics. However, as described in the related work section, aggregate performance metrics alone are incompatible with existing operating system and virtual machine performance abstractions. Furthermore, performance depends on a collection of resources (e.g, the processor, caches, and SDRAM memory system), and consequently, should be accounted for by a policy that has a global view of the system – not just a policy managing a single resource. Meaningful objectives and effective global policies are a topic for future work.

6. SUMMARY AND CONCLUSIONS

The Virtual Private Machine framework is a means for supporting microarchitecture resource sharing. In this framework, hardware mechanisms allow hardware resources to be allocated to executing threads, thus realizing 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.

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 implementation presented in this paper 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 will 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 affect each other significantly and in an uncontrollable manner. With the FCFS or round-robin arbiters in a desktop environment, for example, an aggressive background task may prevent the user from watching a movie, even when there is ample processing power available. 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 can improve CMP throughput by eliminating negative interference.

7. ACKOWLEDGEMENTS

This work was supported by an IBM Fellowship and by equipment donations and financial support from Intel. The first author would like to thank Professor Parmesh Ramanathan for his clarity and insights in teaching real-time systems. We would also like to thank Ravi Nair and Dan Prener of IBM for their advice and support during the development of simulation infrastructure.

8. REFERENCES

[1] Banga, G., Druschel, P., and Mogul, J., Resource containers: A new facility for resource management in server systems, In *Proc. of the 3rd USENIX Symp. On Operating Systems and Design Implementation,* Feb. 1999. pp 45-58.

[2] Bennett, J. C., and Zhang, H., Hierarchical packet fair queuing algorithms. In *Trans. On Networking,* Oct. 1997. pp 675-689.

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

[4] Cazorla, F. J., Ramırez, A., Valero, M., Knijnenburg, P. M. W., Sakellariou, R., and Fernandez., E., QoS for High-Performance SMT Processors in Embedded Systems. *IEEE Micro*, 2004. pp 24–31.

[5] Chetto, H., and Chetto, M., Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Trans. on Software Engineering.* 15, 10, Oct. 1989. pp 1261-1269.

[6] Emer J., et al., Asim: A Performance Model Framework. *IEEE Computer,* Feb. 2002. pp 68-76.

[7] Kim, S., Chandra, D., and Solihin, Y., Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proc. of the 13th Intl. Conf. on Parallel Architecture and Compiler Techniques*, Sept. 2004. pp 111-122.

[8] Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A., Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proc. of the USENIX 7th Intl. Middleware Conference,* Dec.2006.

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

[10] Hsu, L. R., Reinhardt, S. K., Iyer, R., and Makineni, S., Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proc. of the 15th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2006. pp 13-22.

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

[12] Iyengar, V. S., Trevillyan, L. H., and Bose, P., Representative Traces for Processor Models with Infinite Cache. In *Proc. of the 2nd Symp. on High-Performance Computer Architecture,* Feb. 1996. pp 62-72.

[13] Iyer, R. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *Proc, of the 18th Intl. Conf. on Supercomputing*, June 26, 2004. pp 257-266.

[14] Kumar, R., Zyuban, V., and Tullsen, D. M., Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In *Proc. of the 32nd Intl. Symp. on Computer Architecture,* June 2005. pp 408-419.

[15] Kongetira, P., Aingaran, K., and Olukotun, K., Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro,* 25, 2, Mar. 2005. pp 21-29.

[16] Le Boudec, J.Y., and Thiran, P., *Network Calculus*, Springer Verlag, 2004.

[17] Lee, J. W. and Asanovic, K., METERG: Measurement-Based End-to-End Performance Estimation Technique in QoS-Capable Multiprocessors. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symp,* April 2006. pp 135-147.

[18] Luo, K., Gummaraju, J., and Franklin, M., Balancing throughput and fairness in SMT processors. In *Proc. of the Intl. Symp. on Performance Analysis of Systems and Software*, Jan. 2001. pp 164-171.

[19] 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. pp 429-448.

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

[21] Nesbit, K.J., Aggarwal, N., Laudon, J., and Smith, J.E., Fair Queuing Memory Systems, In *Proc. of 39th Intl. Symp. On Microarchitecture*, Dec 2006. pp 208-222.

[22] Qureshi, M. K. and Patt, Y. N. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Intl. Symp. on Microarchitecture*, Dec. 2006. pp 423-432.

[23] Rafique, N., Lim, W., and Thottethodi, M. Architectural support for operating system-driven CMP cache management. In *Proceedings of the 15th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2006. pp 2-12.

[24] Sariowan, H., Cruz R.L., and Polyzos G.C., Scheduling for quality of service guarantees via service curves. In *Proc. of the 4th Intl. Conf. on Computer Communication and Networks,* Sept. 1995. pp 512-520.

[25] Shreedhar, M., and Varghese, G., Efficient fair queueing using deficit round robin. In *Proc. of the Conference on Applications, Technologies, Architectures, and Protocols For Computer Communication*, August 1995. pp 231-242.

[26] Silberschatz, A., Galbin, P. B., and Gagne, G., *Operating System Concepts, Seven Edition,* John Wiley & Sons, Inc., 2004.

[27] Skadron, K., and Clark, D. W., Design Issues and Tradeoffs for Write Buffers. In *Proc. of the 3rd Symp. on High-Performance Computer Architecture.* Feb. 1997. pp 144-155.

[28] Stewart, D. B., and Mortier, R., Virtual private machines: user-centric performance. In *Proc. of the 11th Workshop on ACM SIGOPS European Workshop: Beyond the PC*, Sept., 2004. pp 36-40.

[29] Suh, G. E., Devadas, S., and Rudolph, L., A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proceedings of the 8th Intl. Symp. on High-Performance Computer Architecture*, Feb. 2002. pp 117-128.

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

[31] Verghese, B., Gupta, A., and Rosenblum, M. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *Proc. of the 8th Intl. Conf. on Architecture Support For Programming Language and Operating Systems*, Oct. 1998. pp 181-192.

[32] Wilton, S., and Jouppi, N., CACTI: An Enhanced cache Access and Cycle Time Model, In *Journal of Solid-State Circuits*, Vol. 31, May 1996. pp 677-688.

[33] Zhang H., Service Disciplines for Guaranteed Performance Service in Packet-switching Networks, In *Proc. of the IEEE*, vol.83, Oct. 1995. pp 1374-1398.