Purdue University
Purdue e-Pubs

ECE Technical Reports | Electrical and Computer Engineering

7-21-2006

Architectural Support for Operating System-Driven CMP Cache Management

Nauman Rafique
Purdue University

Won Taek
Purdue University

Mithuna Thottethodi
Purdue University

Follow this and additional works at: http://docs.lib.purdue.edu/ecetr

Rafique, Nauman; Taek, Won; and Thottethodi, Mithuna, "Architectural Support for Operating System-Driven CMP Cache Management" (2006). *ECE Technical Reports.* Paper 337.
http://docs.lib.purdue.edu/ecetr/337

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Architectural Support for Operating System-Driven CMP Cache Management

Nauman Rafique, Won-Taek Lim, Mithuna Thottethodi
School of Electrical and Computer Engineering, Purdue University
West Lafayette, IN 47907-2035
{nrafique,wlim,mithuna}@purdue.edu

ABSTRACT

The role of the operating system (OS) in managing shared resources such as CPU time, memory, peripherals, and even energy is well motivated and understood [22]. Unfortunately, one key resource—lower-level shared cache in chip multi-processors—is commonly managed purely in hardware by rudimentary replacement policies such as least-recently-used (LRU). The rigid nature of the hardware cache management policy poses a serious problem since there is no single best cache management policy across all sharing scenarios. For example, the cache management policy for a scenario where applications from a single organization are running under "best effort" performance expectation is likely to be different from the policy for a scenario where applications from competing business entities (say, at a third party data center) are running under a minimum service level expectation. When it comes to managing shared caches, there is an inherent tension between flexibility and performance. On one hand, managing the shared cache in the OS offers immense policy flexibility since it may be implemented in software. Unfortunately, it is prohibitively expensive in terms of performance for the OS to be involved in managing temporally fine-grain events such as cache allocation. On the other hand, sophisticated hardware-only cache management techniques to achieve fair sharing or throughput maximization have been proposed. But they offer no policy flexibility.

This paper addresses this problem by designing architectural support for OS to efficiently manage shared caches with a wide variety of policies. Our scheme consists of a hardware cache quota management mechanism, an OS interface and a set of OS level quota orchestration policies. The hardware mechanism guarantees that OS-specified quotas are enforced in shared caches, thus eliminating the need for (and the performance penalty of) temporally fine-grained OS intervention. The OS retains policy flexibility since it can tune the quotas during regularly scheduled OS interventions. We demonstrate that our scheme can support a wide range of policies including policies that provide (a) passive performance differentiation, (b) reactive fairness by miss-rate equalization and (c) reactive performance differentiation.

1. MOTIVATION AND INTRODUCTION

Operating systems deal with a large number of policy options in managing shared resources. For example, operating systems have traditionally satisfied the wide variety of CPU sharing requirements by deploying a number of scheduling policies. Specifically, the `priocntl()` system call in Solaris offers four different scheduling policies such as fixed priority, real-time priority, fair-sharing and time-sharing [25]. Each policy may be appropriate under different circumstances.

A similar situation arises in the management of shared caches in chip multiprocessors (CMPs), which are increasingly seen as the mainstream platform of choice for most high end machines [16]. As with CPU-sharing, there is a rich space of cache management policies for shared CMP caches that may be appropriate under various sharing scenarios. LRU and other traditional replacement algorithms may be appropriate for many domains in which there is minimal interference between sharers, and access patterns are rich in temporal locality. For other domains where there is potential for interference among applications, policies that ensure fairness by equalizing the impact of cache sharing and/or maximizing instruction throughput may be more appropriate [9, 26]. Finally, for domains such as *service oriented computing* [2, 17, 19, 27] wherein third party IT providers agree to provide minimum service levels to their customers, performance differentiation and performance assurance are key concerns. Even within a single organization, performance differentiation may be needed since some applications might be more critical than others.

The above observation leads us to the following dilemma. On one hand, involving the operating system (OS) in managing shared caches offers policy flexibility since policies are implemented in software. But caches are widely managed in hardware because of the high overhead of OS intervention for every L2 cache miss (the disadvantages of software managed caches (SMCs) [3, 6, 8, 12] are further explained in Section 5). On the other hand, hardware cache management minimizes the performance impact but it does not offer policy flexibility.

This paper resolves the above dilemma by developing architectural support that enables efficient OS-level cache management. Our solution consists of three components: a hardware cache quota enforcement mechanism, an OS interface and a set of OS-level policies for orchestrating quotas. The hardware mechanism enforces OS-specified, per-sharer cache quotas in shared lower level (L2 and beyond) caches. The OS can implement various cache management policies by manipulating the quotas via the quota specification interface. As a result, the caches are still managed in hardware, and OS is only rarely invoked to orchestrate quotas. Moreover, our scheme offers the OS an additional tool beyond CPU-scheduling to influence the performance of processes. This paper demonstrates that our scheme serves as an effective and flexible platform for the OS to implement policies with various goals. In the following paragraphs, we

define the precise semantics of the cache quota guarantee that the hardware mechanism enforces and provide a high level overview of the proposed implementation.

Guarantee Semantics and Terminology: A minimum portion of a cache or a set in the cache that is guaranteed to an entity is referred as its *quota* (denoted by symbol q). The quota is specified by the OS in terms of the number of cache ways (for the reasons explained in Section 2.1). We explore two granularity options of enforcing quota: at the overall cache level as well as at the set level in a set-associative cache. The two options have their own advantages and dis-advantages which we discuss in Section 2.1.3. The actual cache space allocation might be more than the quota and is referred to as *share*. An entity which is guaranteed a specific quota is referred to as a *sharer*. A sharer could be a thread, a process or any group of threads/processes that the OS wishes to treat as a single entity as far as cache sharing is concerned. Thus multiple threads in a sharer group share the same cache quota. Section 2.2 describes how the OS can specify a sharer to the hardware. The subset of sharers that actually contend for a cache or a set in the cache are referred to as *contenders*.

The semantics of quota guarantee can be explained with an example. Consider a set of n sharers $S = \{1, 2, 3, \ldots, n\}$. Each sharer $i \in S$ has an explicitly specified quota: q^i. Now, if there are k contenders, it is guaranteed that each contender c_i is allocated $q^{c_i}/\Sigma_{j=1}^{k}(q^{c_j})$ fraction of the space (either at the set or cache granularity). The above definition incorporates two key desirable features: First, it ensures that each sharer receives some minimum guaranteed cache space. This is because the guaranteed fraction of cache space of a sharer i is no less than $q^i/\Sigma_{j=1}^{n}(q^j)$ when all sharers are contending for the cache (or the specific set in the cache). Second, it can be observed that cache space is allocated depending on the number of contenders. When there is no contention, the sole contender (say i) gets all of the resources irrespective of the quota since its fraction of the share amounts to $q^i/q^i = 1$. This is a key advantage of our technique over some other cache partitioning schemes [4, 7, 11] since cache allocation is not limited in the absence of contention.

We use the term *mechanism* to describe the hardware architecture to enforce quotas. The term *policy* is used for the OS level policy used to control cache quotas. The combination of mechanism, OS interface and a set of OS level policies is referred to as our *scheme*.

Implementation: We propose a new set-level quota enforcement mechanism. Cache-level quota enforcement mechanism has used before [9, 23] for different hardware-based cache management schemes. Our results shows that set-level quota enforcement mechanism provides stronger quota guarantees and shows reasonable performance in all cases. We also propose a new optimization for quota enforcement mechanisms (both set-level and cache-level) that can be used to tune how strongly the quota guarantees are enforced. Using this optimization, the quota enforcement mechanism can be tuned to provide strong guarantees (for minimum service level expectation systems), no quota guarantees (to switch back to LRU) or some intermediate level of quota guarantee (to use the cache space efficiently while still maintaining quotas). The hardware quota enforcement mechanisms have no timing overhead. Their area overhead is less than 10% of the L2 tag array (for a 4MB, 16-way associative L2 cache), and less than 1% over the whole cache. The OS interface is also very simple and requires only 28 bits if 4 sharers have to be supported per CMP. We demonstrate the flexibility of our approach by developing a variety of policies including policies for performance differentiation and fair sharing. The timing overhead of the OS level policies depends on the complexity of the policy. One of the policy that we have shown has 0 overhead, while the other has 2% overhead. Hence our scheme has little overhead; however, the corresponding benefits, in terms of enabling OS-level management of cache quotas are significant.

Related Work: There have been many recently proposed *reactive* schemes for shared CMP cache management. These schemes are reactive in the sense that cache space allocation/deallocation decisions are driven by miss-rate and/or performance feedback and aim to (a) equalize the miss-rate or performance degradation [9] or (b) optimize throughput by giving more cache space to the users who benefit the most from it [20, 26]. Our work is orthogonal to the above work since the hardware quota enforcement mechanisms in our scheme are policy-agnostic and are fully compatible with the aforementioned schemes. The only difference is that while some of those schemes assumed a purely hardware implementation, our scheme requires that the OS must reactively tune the quotas to achieve the desired goals (equalizing miss rate, maximizing instruction throughput etc.). This separation of mechanism and policy is inherently a good design practice since CMPs are required to operate in a wide range of systems where different policies may be appropriate.

Summary: The major contributions of this paper are: (a) We propose a new scheme for the management of shared L2 cache in CMPs. Our scheme includes a hardware cache quota enforcement mechanism, an OS interface and a set of OS level quota management policies. Our scheme is flexible in the sense that it enables OS to implement a variety of policies including policies for fair cache sharing and achieving differentiated instruction throughput. (b) We develop a new mechanism that enforces OS-specified, minimum cache quota guarantees. The area overhead of the quota enforcement mechanism is minimal (less than 10% over tag array and less than 1% over whole cache). There is no delay overhead of the mechanism since it works off the critical path. (c) We propose a simple, yet powerful interface for the OS to specify cache quotas. The interface, which involves a single write to a special register allows threads running on different CMP cores to share the same cache quota. Moreover, it allows thread migration and context switching. (d) We demonstrate how OS-level policies can use the hardware infrastructure proposed by our scheme. Simulations of multiprogrammed scientific workloads and multi-threaded commercial workloads confirm that the desired goals of the policies are indeed achieved.

The rest of the paper is organized as follows. Section 2 describes the components our scheme i.e., the hardware quota enforcement mechanisms, OS interface and how various policies may be implemented using our scheme. Section 3 explains the experimental methodology. Results of our simulations are presented in Section 4. Section 5 discusses related work and finally Section 6 concludes the paper.

2. CACHE MANAGEMENT SCHEME

Our cache management scheme consists of three essential components: a hardware quota enforcement mechanism, an

SQVP: On a cache miss for a set S by a sharer S_i:
Let B_i be the total number of blocks in the set S currently used by S_i.
Calculate the share of S_i in the set ($share_i$).
If $B_i \geq share_i$,
Select new replacement candidates until one is found which is owned by S_i.
else
Select new replacement candidates until one is found which is owned by a sharer (S_k) such that $B_k > share_k$.

Figure 1: Set Quota Violation Prohibition

CQVP: On a cache miss for a set S by a sharer S_i:
Let T_i be the total number of blocks in the whole cache currently used by S_i.
If $T_i \geq q_i$,
Select new replacement candidates until one is found which is owned by S_i.
If no such candidate is found, select a random candidate.
else
Select new replacement candidates until one is found which is owned by a sharer (S_k) such that $T_k > q_k$.
If no such candidate is found, select a random candidate.
If the replacement candidate is owned by sharer S_k and $S_k \neq S_i$
Increment the bcounter of S_i and decrement the bcounter of S_k

Figure 2: Cache Quota Violation Prohibition

interface between hardware and OS, and a set of OS level policies. We first describe two hardware quota enforcement mechanisms in Section 2.1, their advantages and limitations. Section 2.2 describes the OS interface of our scheme and finally Section 2.3 explains how the OS can use this interface to implement different policies.

2.1 Hardware Quota Enforcement Mechanism

Our scheme uses a hardware mechanism that enforces quotas for different sharers. In general, the quotas can be enforced at a set-level or the whole cache space level. Based on this criteria, there can be two mechanisms which are explained in this section.

Both mechanisms that we describe here enforce quotas at the time of cache block replacement. They act as "overlays" that modify the behavior of an underlying quota-oblivious replacement algorithm. We do not require LRU replacement policy. Any replacement policy (this policy should not be confused with the OS-level policy in our scheme) that can generate a sequence of replacement candidates is acceptable. In addition to the tags and status bits maintained by caches, the mechanisms require additional state to be maintained. We maintain the "ownership" information of each cache block which requires a "Tag-owner-ID" to be stored along with each tag. We define the "owner" of a block as the sharer that last touched the block. If a cache block is used by multiple sharers, they will most likely have the same SID. If they have different SIDs, then ideally the block's ownership (and hence its contribution to quota utilization) must be simultaneously assigned partially to each sharer. We avoid the complexity of determining partial ownership but achieve a similar effect since the block's ownership is distributed among sharing threads over time (the ownership changes hands between the sharers as they use the cache block).

The replacement choice can be decided concurrently with cache block fetch and is not on the critical path. The same assumption is made by other replacement optimizations proposed in the past [4, 18]. Hence the quota enforcement mechanisms we describe here introduce no timing overhead.

2.1.1 Set-Level-Quota Enforcement Mechanism

In a shared cache, the issue of cache space management arises if there are multiple sharers contending for the same set. If there is no contention at a set, no sharer would interfere with others and there is no need for cache space management. This argument motivates to maintain quota at a cache set level. If quotas are enforced at a larger granularity, the behavior at the cache set level is still arbitrary. Hence, the first mechanism that we describe here implements quota enforcement at the cache set level.

This mechanism is called *Set Quota Violation Prohibition* (*SQVP*). It allocates a set level share to each sharer. The share of each sharer depends on the quotas of other sharers contending for the set. If there are k sharers using the cache set, the $share_i$ of a sharer S_i is calculated as follows:

$$share_i = q^i / \Sigma_{j=1}^{k} (q^j) * cache_associativity$$

$share$ should not be confused with quota. Quota (q) is the OS supplied value which is the minimum value of $share$. $share$ could potentially be equal to cache associativity if there are no other sharers using the set. On a cache replacement, a sharer replaces its own block if it is already using its $share$. If it is below its $share$, it replaces the block of another sharer that is exceeding its $share$. Hence, this mechanism prohibits any replacement that violates the minimum quota of a sharer. If a sharer S_i has not exceeded its quota of frames in the set, no other sharer may replace any of S_i's blocks, even if the primary replacement candidate[1] is owned by S_i. In such a situation, subsequent replacement candidates are considered and the first encountered replacement candidate that is owned by a sharer that has exceeded its quota is selected for replacement. $SQVP$ mechanism requires that the sum of set-level quotas of all sharers must not exceed the cache associativity; that ensures that $SQVP$ is guaranteed to find a replacement candidate. This algorithm is formally described in Figure 1. In Figure 1, the order in which the replacement candidates are selected is determined by the underlying replacement policy (e.g., LRU). Next, we describe the mechanism that enforces quotas at the whole cache space level.

2.1.2 Cache-Level-Quota Enforcement Mechanism

Another approach for cache quota enforcement is doing it at the cache level. Cache level quota enforcement has been used by other researchers before [9, 23]. We call this mechanism *Cache Quota Violation Prohibition* (*CQVP*). The granularity of quota allocation here is taken to be "number of cache blocks in a cache way" (n_w). The OS only specifies the number of cache ways to be allocated to a sharer and it is multiplied by n_w to get the cache level quota of each sharer. At the time of replacement, a sharer replaces its own cache blocks if it is already using its assigned quota (or more than its quota). If a sharer is using less than its quota, it replaces the cache block of a sharer which is over its quota. If an appropriate cache block is not found in the set, a cache block is replaced at random. In addition to the "owner" information, $CQVP$ also requires counters to keep track of

[1] A primary replacement candidate is the candidate that is first picked up by a replacement policy. For example, if LRU is the replacement policy, the least-recently-used block is the primary replacement candidate.

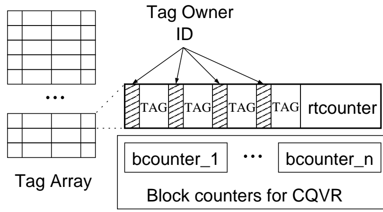


Figure 3: Hardware Modifications for Quota Enforcement Mechanisms

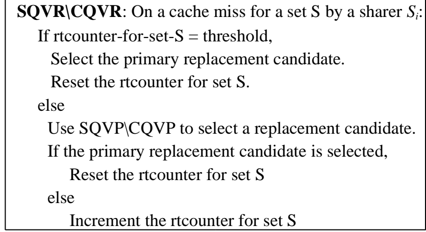


Figure 4: Quota Violation Reluctance Optimization

total cache blocks used by each sharer in the cache. We will name these counters as `bcounters` (Block Counters). The `bcounter` of a sharer is incremented whenever a new cache block is allocated to it. Figure 2 describes this algorithm. In Figure 2, quota q of sharers has already been multiplied by n_w so that we can directly compare it with block count T. Again, the order of selection of replacement candidates depends upon the underlying replacement policy.

2.1.3 Comparison of Set and Cache Level Quotas

We describe here the major trade-offs between set- and cache-level quotas. **First**, $SQVP$ manages cache at the granularity of a cache set. Thus, it can manage the hot sets according the quota allocation of sharers. $CQVP$, on the other hand, enforces cache level quotas. At a hot set, its behavior depends on the cache block count of the sharers contending for the hot set, and cannot be controlled directly. This limitation of $CQVP$ is apparent in the results shown in Section 4.2. **Second**, as $SQVP$ have cache quotas specified at the set level, it is guaranteed to find a replacement candidate, and never resorts to picking a random replacement candidate. $CQVP$, on the other hand, might not find the right replacement candidate; it picks a random candidate for replacement in such situations. Hence $SQVP$ provides harder quota guarantees as shown by results in Section 4.1. **Third**, $SQVP$ enforces set level quotas irrespective of the cache usage of the sharers in the rest of the cache. $CQVP$ is more powerful in this regard—if a sharer is using less space in some parts of the cache, it will be allocated more space in the other parts. This advantage of $CQVP$ over $SQVP$ is demonstrated by the results in Section 4.2.

2.1.4 Optimizing Quota Enforcement Mechanisms

$SQVP$ and $CQVP$ both suffer from one potential weakness. If a block owner with fewer cache blocks than its quota discontinues (or greatly reduces the frequency of) using its blocks, other sharers are not able to reclaim the rarely used cache frames. This problem is also present in the other cache partitioning schemes [4, 9, 11, 24]. To address this shortcoming, we modify the quota allocation mechanisms, incorporating *reluctance* rather than the outright *prohibition* on replacements that violate quotas. This optimization works as follows: each time a primary replacement candidate in a particular set is spared from replacement because its owner is consuming less than its share, a per-set counter is incremented. We name this counter as `rtcounter` (Reluctance Threshold counter). When `rtcounter` of a set exceeds a *reluctance threshold*, the primary replacement candidate is replaced even if the replacement violates quotas. The flavors of $SQVP$ and $CQVP$ that include this optimization are referred to as $SQVR$ and $CQVR$ respectively. Figure 3 shows a tag array with the new structures needed for a 4-way cache, supporting n sharers. Note that in addition to the structures needed for $SQVR$, $CQVR$ also requires a per sharer cache block counter (`bcounter`) to keep track of total cache blocks used by each sharer in the cache. Also notice that the `rtcounter` is per set only and not per cache block. Figure 4 describes $SQVR$ and $CQVR$ mechanisms.

In general, $SQVR_{rep,n}$ (or $CQVR_{rep,n}$) describes a concrete mechanism with rep as the underlying replacement policy and n as the reluctance threshold. In the rest of this paper, we will assume LRU to be the common underlying replacement strategy and vary the reluctance threshold (i.e., $SQVR_{lru,n}$ or $CQVR_{lru,n}$). Note, $SQVR_{lru,0}$ and $CQVR_{lru,0}$ simplify to LRU because there is no reluctance to deviate from the LRU policy. $SQVR_{lru,inf}$ and $CQVR_{lru,inf}$ are equivalent to $SQVP_{lru}$ and $CQVP_{lru}$ respectively which can never reclaim cache space from a sharer due their extreme reluctance to violate quotas.

Area overhead: The area overhead for the implementation of the quota enforcement mechanisms is minimal. Consider a CMP which supports 4 sharers, and has a 4-MB, 16-way L2 cache with 64 byte blocks. The additional "tag-owner-id" field requires 2 ($=lg_2(4)$) bits per tag or a total of 32 bits per set. The size of the `rtcounter` depends on the maximum threshold allowed. Even an extreme threshold of 15 would require only 4 bits. If 4 bits are used for threshold, a threshold of 16 can be treated as a special case to represent infinite threshold. Thus, the total overhead is 36 bits per cache set. Assuming a physically tagged L2 with 32-bit physical addresses and 4 bits of miscellaneous book-keeping state, the overhead of $SQVR$ is almost 10% over tag storage alone but it is less than 1% when considering the cache as a whole. $CQVR$ requires additional cache block counters (`bcounters`) for each sharer. Each of these `bcounters` need to be 16 bits (to count all the cache blocks in 4MB cache). Thus $CQVR$ has additional storage overhead of 64 bits over the whole cache.

In the rest of this section, we describe the interface of hardware quota enforcement mechanism to the OS and how OS can use this interface to implement different cache management policies.

2.2 The OS Interface

Our scheme enables the OS to decide if a sharer is a thread, a process, a user or any other well-defined group. If the sharer is not an individual thread, all threads of the group (e.g., process or user) share the same quota.

The hardware quota enforcing mechanisms require the sharer's identity to be associated with each access. To achieve this, we associate a *sharer identifier* (`sid`) register with each processor to identify the sharer currently using that proces-

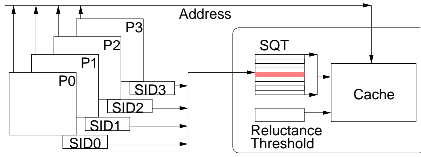


Figure 5: Interface Exposed to OS

sor. Whenever a processor makes a request, its associated `sid` accompanies the request and is used to lookup a *sharer quota table* (`SQT`) which stores the quota of each sharer. In addition to these structures, the optimization mentioned in Section 2.1.4 requires the OS to specify a reluctance threshold. These structures are shown in Figure 5. It is the operating system's responsibility to update the `sid` in the register on each context switch. Solaris already places pointer to the thread data structure in global register %g7, which is updated on context switch [5]. Thus `sid` can be stored in the thread data structure and can be updated when %g7 is updated. This scheme allows the threads of a sharer to move from one core to another core. Moreover, if a sharer is a process or any other group of threads, its threads can run on multiple cores of the CMP at a given time and still share the same quota.

Number of `sids`: An important design consideration is the choice of number of unique `sid`s. One option is to use as many unique `sid`s as there are processors in CMPs. In this case, `sid`s will be reused by sharers upon context switches. When a sharer's `sid` is reused by another sharer, the new sharer may replace the cache blocks of the old sharer over time. This design choice implies that a sharer has no guarantee that its quota will be respected when it is not actively running. But this behavior is no worse than any other replacement policy like LRU.

Our scheme can be easily extended in scope by increasing the number of sharers beyond the number of processors, thus providing quota enforcement across context switches. The area overhead will correspondingly increase due to increases in per-cache-line owner-id storage (Section 2.1) and an increase in the number of `SQT` entries. Another advantage of increasing the number of `sid`s is that it enables the use of our scheme in the context of single-core processors to ensure that quotas are respected across context switches. This may facilitate affinity scheduling techniques as well. We leave the detailed analysis of these benefits for future work and use the option of "`sid`s equal to the number of cores on CMP" for this study.

Area overhead: The area overhead of the OS interface is minimal too. Let us consider a 4-processor CMP which supports 4 sharers and has 16-way L2 cache. We need 8 (2 sid bits $*4$) bits for `sid` registers. The area overhead of `SQT` in terms of bits is 16 (4 quota bits $*4$) bits. The reluctance threshold requires 4 bits. Thus the total overhead of OS interface would be 28 bits.

Given the above architecture, the OS is required to specify (a) the quotas for each sharer in the `SQT` (b) the `sid` of the sharer using a processor core and (c) the reluctance threshold for the $SQVR$ and $CQVR$ mechanisms. Since the 28-bits can be captured in a special 32-bit register, any modifications to the quotas, `sid`s and/or the reluctance threshold can be achieved with a single register write/copy. Similar interfaces to read and write special-purpose hardware performance counters already exist. The modifications to this new register would have to be in privileged mode since applications cannot be allowed un-moderated control of their own quotas.

Next we discuss different OS level policies.

2.3 OS Level Policies

The mechanisms and OS interface described above provide the basic architectural support for the specification and enforcement of cache quotas. This architectural support can be used by OS to implement a variety of cache management policies. As a proof-of-concept, we implemented three concrete OS level policies that are described below.

First, we describe a passive policy (Section 2.3.1) in which cache quotas are fixed for the duration of the run. Next, we examine *reactive* policies that use performance feedback to tune the cache quotas. The feedback can be gathered from hardware performance counters that are present in most modern processors. We evaluate two reactive policies. The first reactive policy (Section 2.3.2) is an implementation of a hardware fairness policy proposed by Kim *et.al.* [9]. This policy attempts to equalize the miss rates of all sharers by controlling each sharer's cache quota. The second reactive policy (Section 2.3.3) attempts to equalize the sharers' instruction throughput ratio to their OS-specified service-level priority ratio by manipulating their cache quotas. Next, we present details of each of these policies.

2.3.1 Passive Performance Differentiation

In this policy, the cache quotas are kept static by OS, i.e., the OS specifies cache quotas and does not change them. We demonstrate this policy for two purposes. **First**, we use this policy for comparison of the quota enforcement mechanisms as it is simple and does not include secondary effects like the *reactive* policies. **Second**, we use it to enforce unequal cache quotas for performance differentiation. Unequal cache quotas provide the performance differentiation since each sharer gets an unequal share in the cache. Performance differentiation can be used to prioritize critical sharers and to improve the profits for an IT infrastructure provider by prioritizing high-paying customers. Our results in Section 4.2 indicate that this passive policy successfully provides performance differentiation.

2.3.2 Reactive Miss Rate Equalization

We demonstrate the flexibility of our scheme by implementing the algorithm suggested by Kim *et.al.* [9] for their fairness scheme. We will briefly describe the algorithm here. For details, we refer the reader to [9]. The algorithm tries to minimize a metric across all sharers in an attempt to improve fairness. Kim *et.al.* studied five different metrics. In this paper, we will present results for only one metric, namely the difference of miss rate referred to as M_4 in [9]. Thus the algorithm will try to minimize the difference of miss rates (or equalize the miss rates) for all the sharers. We refer to this policy in our scheme as MM_4 (Minimizing M_4).

The algorithm starts out by dividing the cache space equally (so we set the quota for each sharer to be equal). Then the following two steps are executed at regular intervals/epochs: repartitioning and rollback. In the **repartitioning** step, M_4 for each thread in the last epoch is calculated. Then, we consider the two threads with the largest and smallest value of

M_4; the quota of the thread with the larger value of M_4 is increased and the quota of the thread with the smaller value of M_4 is decreased. This process is repeated until all the threads have been considered. In the **rollback** step, we look at all the repartitionings done in the last invocation of the algorithm. If, as a result of repartitioning, the metric M_4 was improved by a *rollback-threshold*, we keep the repartitioning; otherwise we undo/rollback the repartitioning. The rollback step is actually performed before the repartitioning, and any threads involved in rollback are not considered for repartitioning.

We used a rollback-threshold of 20% (suggested to be best by Kim *et.al.*). We test the effectiveness of this fairness policy by introducing a program that acts as a cache `hog`. A cache `hog` is designed to be an enthusiastic user of cache space. It creates a big matrix equal to the size of L2 cache, and tries to keep all the elements of the matrix in the cache by accessing them frequently. It not only touches a lot of cache blocks, but goes back and accesses them again, so that its blocks are not evicted by demand based replacement policies like LRU. The code for the `hog` program is presented in Appendix A. Our results in Section 4.3 show that this policy significantly improves performance, as it gets rid of the malicious effect of `hog`.

2.3.3 Reactive Performance Differentiation

The goal of this policy is to provide reactive performance differentiation. In this policy, the algorithm used is similar to the one used for Miss Rate Equalization (MM_4) described in Section 2.3.2, with some modifications. The *first* modification is a new metric M_{pd} (Metric for Performance Differentiation). The M_{pd} metric aims to achieve performance (IPC) ratios proportional to OS-specified *service-level priorities* (say p). The desired IPC ratio and M_{pd} metric are calculated as follows:

$priority_{ratio} = p_i/\Sigma_j(p_j)$, $IPC_{ratio} = IPC_i/\Sigma_j(IPC_j)$,
$M_{pd} = IPC_{ratio} - priority_{ratio}$

In minimizing this metric, we are trying to make the IPC_{ratio} as close as possible to the $priority_{ratio}$. The *second* modification is that this algorithm does not have a rollback step. This policy can be used for performance differentiation only under certain conditions: (a) IPC of each sharer is similar. (b) Performance of sharers is sensitive to their cache space allocation.

We refer to this policy in our scheme as MM_{pd} (Minimizing M_{pd}). It starts with equal quota allocation and tweaks (increases or decreases) the quota allocation until the metric M_{pd} is minimized. It can be used to increase profits of the third party IT infrastructure providers, as they can use to it assign higher priority to more valuable (read high paying) customers. Similar policies can also be used to prioritize more critical applications.

Timing overhead: The overhead for the execution of policy enforcement algorithm in OS depends on the complexity of the policy. We envision the OS to provide a range of policies from simpler to complex ones. For the passive policy described above, there is no OS level overhead as quotas are kept constant. For the two reactive policies, we used an epoch length of 2 million cycles. That translates to 1`ms` for the processor frequency used in this study (2GHz). Thus this algorithm can be executed by system clock handler which is executed every 10`ms` by default but can be set to execute 1`ms` (system clock handler already performs significant amount of processing [15]). This algorithm should work well with epoch time of 10`ms` as well; we set it to 1`ms` so that there are enough epochs during our simulation run. The algorithm is around 50 lines of C++ code including all the book-keeping. We found the algorithm execution time to be around 20 `micro-seconds`, on average (measured using `gettimeofday` system call). As the algorithm is invoked once every 1`ms`, that amounts to a 2% timing overhead. We did not make any attempt to optimize the algorithm running time. If the algorithm is optimized and/or the epoch time is increased to 10`ms`, the overhead can be reduced to much less than 2%.

For all the OS-level policies that we mentioned here, the reluctance-threshold is kept constant. In practice, the OS can tweak the reluctance-threshold to change how strongly the quota guarantees are enforced. To show the effect of different reluctance-thresholds, we present all results with three different values of reluctance thresholds.

Workload	Benchmarks	Fourth
Mix_1	ammp, mcf, lucas	parser
Mix_2	vpr, facerec, gcc	twolf
Mix_3	twolf, parser, applu	art
Mix_4	parser, art, gcc	twolf

Table 1: Multiprogrammed workloads

Benchmarks	**SPECjbb** Trans. (Insts.)	**Apache** Trans. (Insts)
System Warm-up	100,000	20,000
Cache Warm-up	40,000	500
Simulation	4000 (≈100million)	100 (≈100million)

Table 2: Commercial Multi-threaded Benchmarks

3. SIMULATION METHODOLOGY

In this section, we describe the workloads and simulation platform we use to evaluate our scheme.

3.1 Benchmarks

We consider two distinct workload types: multiprogrammed scientific and engineering workloads, and multi-threaded commercial workloads. Our multiprogrammed workload mixes represent applications from SPEC2000 which have a large L2 cache usage and a high miss rate [28]. We made this choice because cache space management is not an issue if there is no resource contention. Table 1 lists all the multiprogrammed workload mixes we used for our experiments. For multiprogrammed workloads, caches are warmed up for 2 billion instructions and 100 million instructions are simulated in detail.

The commercial workloads used in the study are listed in Table 2 with the number of transactions (and corresponding number of instructions) used for different phases of the simulation. We run **SPECjbb** with 16 warehouses, with one client per warehouse. **Apache** was configured to use prefork multi-processing module. `Surge` is used as `apache`'s client to generate web requests [1]. It simulates 300 (75 * 4) users. As Surge itself takes significant processor resources, we run it on a separate simulator and the results reported

Processor parameters	
• Number of chips: 1 • Processors per chip: 4	
• Feature size: 65nm • Cycle time: 0.5ns	
• Pipeline width: 4way • Instruction window: 64	
• Return address stack: 64 • ROB size: 128	
• Branch Predictor: 1KB YAGS	
Memory hierarchy parameters	
L1 I/D cache	128KB, 4-way, 2 cycles
L2 cache	4MB, 16-way shared,
	4-way banked, 8 cycles
L2 miss latency	400cycles
DRAM size	1GB
Disk Latency	Fixed 10ms

Table 3: Simulation Parameters

here refer to the server-side statistics only.

We simulate the OS's interactions with our scheme by simulation without modifying the OS. The code for OS level policy is executed in the simulator and directly modifies `SQT`. We statically assign a `sid` to each process/thread. Our simulation environment does not provide a straightforward way of identifying a process/thread running on a processor. To overcome this limitation, we bind all threads with the same `sid` to a particular processor using the `pbind` command in Solaris. This binding of threads to processors prohibits the migration of threads between cores but does not qualitatively change the results of our experiments. This setup allows us to associate a static `sid` with a each processor; thus each processor is exclusively serving one sharer, with a quota specified by `SQT`. As the CMP we simulate has 4 cores, there are 4 sharers in system (they will be numbered 0, 1, 2 and 3). For multi-programmed workloads, we bind one benchmark to each processor. For commercial workloads, we bind the same number of threads to each processor. Each group of threads bound to one processor constitutes one sharer and all threads in the group share the same quota. `SPECjbb` uses a single thread to simulate a client and a server. As we simulate 16 clients in total for `SPECjbb`, each processor will have 4 such threads bound to it. For `Apache`, a server thread is used exclusively to serve one client/user. Thus each processor has 75 threads bound to it as there are a total of 300 (75 * 4) clients.

We also perform experiments with `hog` (Section 2.3.2) to demonstrate that our mechanism can be used to provide fairness. For those experiments, one of the processors runs only a `hog`. For multi-programmed workloads, the `hog` replaces the fourth benchmark (listed as "Fourth" is Table 1) in the mixes. We do not perform experiments with `hog` for commercial workloads.

3.2 Simulation Platform

We use the Simics [13]-based GEMS [14] full system simulation platform. Our simulated system runs an unmodified Solaris operating system version 5.9 and simulates a 4-core CMP. Each core is modeled by OPAL module in GEMS. OPAL simulates an OoO processor model, implements partial SPARC v9 ISA, and is modeled after MIPS R10000. For our memory subsystem simulation, we use the RUBY module of GEMS which is configured to simulate an MSI directory-based cache coherence protocol. We use an aggressive 4-way 128KB L1 cache conservatively (using a smaller cache would only make our results look better). Our L2 cache is 4-way banked, with a point-to-point interconnect between the L2 banks and L1. We used CACTI-3.0 [21] to model the latency of our caches. Table 3 lists the key parameters of the simulated machine.

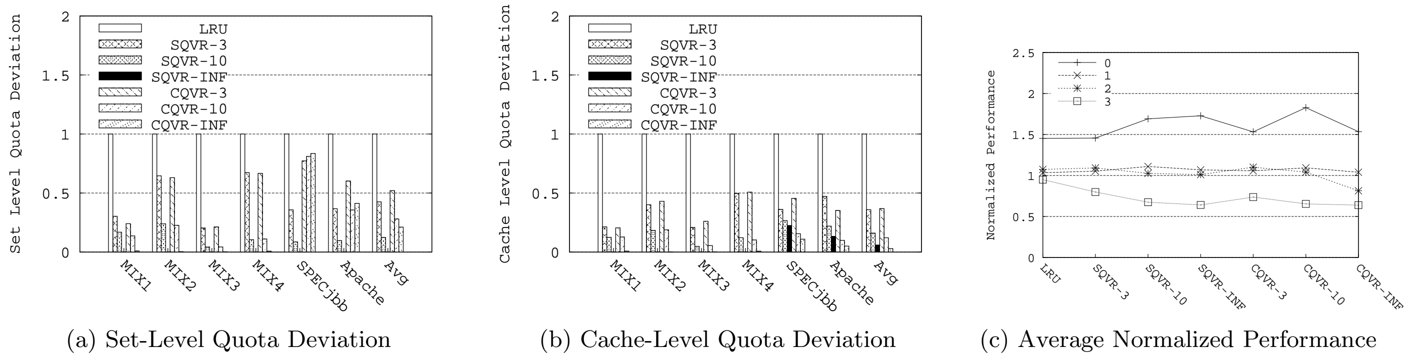
4. RESULTS

The primary conclusions from our results are fourfold: (a) Enforcing set-level quotas is broadly comparable to enforcing cache-level quotas. But set-level quota enforcement mechanism can provide harder quota guarantees than cache-level quota enforcement mechanism. Moreover, cache-level quota enforcement mechanism shows poor performance at high reluctance-thresholds in some cases. (b) The passive policy is effective in providing performance differentiation (c) MM_4 shows a significant improvement in performance by avoiding the malicious effect of the `hog`, and (d) MM_{pd} successfully achieves performance differentiation; however, the policy requires further study to achieve stable performance. We expand on each of the above results in the rest of this section.

4.1 Cache-level vs Set-level Quotas

In this section, we compare the efficacy of the quota enforcement mechanisms ($SQVR_{lru,n}$ and $CQVR_{lru,n}$) in terms of how well they maintain the quotas that they aim to achieve, for different reluctance thresholds. For this purpose, we collect set-level and cache-level quota deviation metrics. The set-level quota deviation metric aggregates the deficit (i.e., quota minus held blocks) for the owner of the replaced block if it is/becomes under (set-level) quota after a replacement. The cache-level quota deviation metric counts the instances when the owner of the replaced block is/becomes under (cache-level) quota after a replacement. For the sake of comparison, we also present the set-level deviation of $CQVR_{lru,n}$ and the cache-level deviation of $SQVR_{lru,n}$.

Figure 6 (a) and (b) show the results for set-level quota deviation and cache-level quota deviation for all the benchmarks. A smaller value of deviation shows that the mechanism closely follows the quota allocation. All the numbers are normalized to their value for LRU. Figure 6 (a) confirms that as the reluctance threshold is increased from 3 to infinity, $SQVR_{lru,n}$ improves (decreases) the deviation from the quota. $SQVR_{lru,inf}$'s deviation is zero for all workloads as expected. Hence, a high value of threshold ensures that quota guarantees are preserved. Figure 6 (a) also shows that the impact of reluctance threshold on set-level quota deviation varies from workload to workload. Mix_1 and Mix_3 shows little deviation at smaller reluctance thresholds, while the rest of the workloads show comparatively larger deviation. The rate at which the deviation decreases with increasing reluctance threshold is also different for different workloads. The value of deviation at different reluctance thresholds depends on the access pattern of the workloads and is hard to predict. We recommend high value of thresholds if guaranteeing cache quotas is important. $CQVR_{lru,n}$ also shows a small value of set-level quota deviation for the multiprogrammed workloads, but for commercial workloads, its set-level quota deviation is high and unpredictable (as for `SPECjbb`, the quota deviation increases with increasing threshold). Figure 6 (b) shows similar trends - the cache-



(a) Set-Level Quota Deviation (b) Cache-Level Quota Deviation (c) Average Normalized Performance

Figure 6: Passive Performance Differentiation

level quota deviation decreases as the reluctance threshold is increased. But as opposed to $SQVR_{lru,inf}$ which has zero set-level quota deviation, $CQVR_{lru,inf}$ has some non-zero value of cache-level quota deviance (visible at least for commercial workloads). The reason is that $CQVR_{lru,inf}$ has to let a sharer store at least one block in a cache set, even if it exceeds its quota overall. In such situations, the replacement candidate is picked at random. Thus $CQVR_{lru,n}$ does not provide hard quota guarantees as such (as mentioned in second point in comparison of $SQVP$ and $CQVP$ Section 2.1.3). Figure 6 (b) also shows that $SQVR_{lru,n}$ exhibits little cache-level quota deviation on average (though significant deviation is observed for commercial benchmarks).

Next, we will show the results for different OS level policies. The results of each policy are presented using both $SQVR_{lru,n}$ and $CQVR_{lru,n}$ to show how these mechanisms effect the function of different policies. We show only performance (IPC) results for the different policies; the miss rates correspond with performance trends and are not presented here.

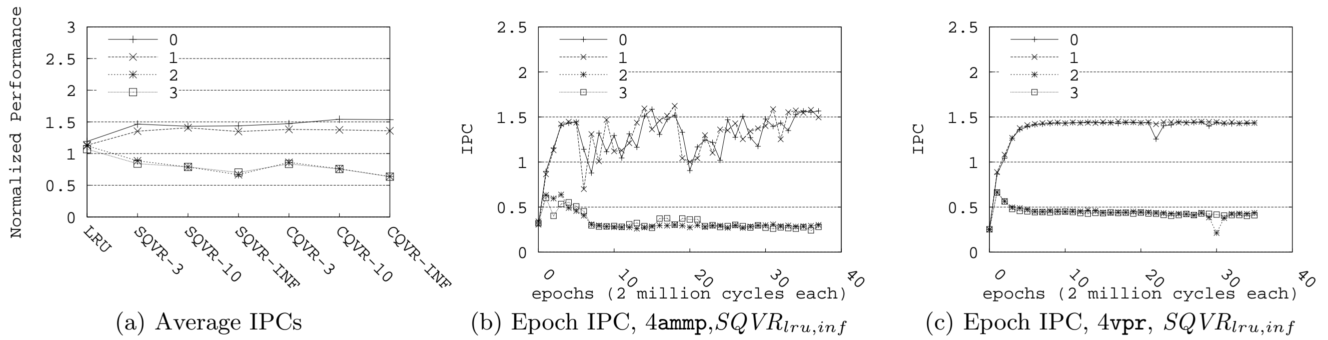
4.2 Passive Performance Differentiation

We mentioned in Section 2.3.1 that a passive policy of fixed, but different quotas can be used to provide differentiated service. For that purpose, we set the quotas of the four sharers to be $< 6, 4, 4, 2 >$. The sum of quotas does not exceed the cache-associativity for the reasons explained in Section 2.1.1.

We determine the success of the passive policy by how well it differentiates the performance. For comparison, we use column caching [4] with equal way division, i.e., each sharer is allocated equal number of cache ways (even if they do not use it). Hence, we normalize the IPC of each sharer to its IPC in column caching. We do not use the IPC in LRU for normalization because LRU is demand-based and might allocate different cache space to each sharer; hence, its not a good choice for normalization. Figure 6 (c) plots the instruction throughput (IPC) of each sharer averaged for all the workloads (individual misses and performance numbers for each workload are shown in Figure 9 and Figure 10 respectively, in Appendix B). The results are shown for $SQVR_{lru,n}$ and $CQVR_{lru,n}$ for different thresholds. The performance of each sharer is also shown for the demand-based LRU mechanism for comparison. Note here that the performance of each sharer in LRU depends on its cache access pattern; a sharer with more frequent cache accesses would secure more cache space and might show better performance than others. Thus, the efficacy of the passive performance differentiation policy should not be judged by its comparison to LRU, but by the difference in performance between sharers with high and low quotas. As sharer 0 and 3 are assigned quotas of 6 and 2 respectively, sharer 0 should benefit the most, while sharer 3 should suffer, depending on their sensitivity to the cache space allocated to them. This trend is visible for smaller thresholds and become more prominent at larger thresholds as expected (larger thresholds mean stronger quota enforcement). In general, it is possible that a sharer is able to grab more cache space in LRU than the quota allocated to it in $SQVR_{lru,n}$ or $CQVR_{lru,n}$. That happens in a few workloads for sharer 0 (numbers not shown here); that is why sharer 0 shows a good performance for LRU too. It can be noticed that $CQVR_{lru,3}$ and $CQVR_{lru,10}$ appear to be more effective than $SQVR_{lru,3}$ and $SQVR_{lru,10}$, respectively (sharer 0 shows slightly better performance for $CQVR_{lru,3}$ and $CQVR_{lru,10}$). The reason is that for $CQVR_{lru,n}$, if sharer 0 is not using some part of the cache, it can have greater share in the rest (the third point mentioned while comparing $SQVP$ and $CQVP$ in Section 2.1.3). One can also observe that for $CQVR_{lru,inf}$, sharer 0, 1 and 2 show a significant drop in performance. The reason is that $CQVR_{lru,inf}$ does not provide any quota guarantees at the set-level, and a benchmark might be penalized in hot sets if it is consuming its quota in the rest of the cache (first point mentioned in comparison of $SQVP$ and $CQVP$ in Section 2.1.3). Hence, we do not recommend using $CQVR_{lru,n}$ with high value of reluctance-thresholds. The overall instruction throughput (combined for all sharers) is slightly better than LRU for all mechanisms but $CQVR_{lru,inf}$ (not apparent from the numbers presented). We also notice that this overall throughput decreases slightly with increasing reluctance-threshold. The reason is that at high reluctance-thresholds, cache blocks are kept longer in cache even if they are not reused. Hence, if performance differentiation is not the primary concern, we recommend using smaller thresholds.

4.3 Reactive Miss Rate Equalization

The reactive policy MM_4 that aims to equalize miss rates to improve fairness is used is this study to prevent the malicious effect of the `hog`. We present the results with the `hog` for multiprogrammed workloads only. Figure 7 presents the performance of this policy with different quota enforce-



(a) Average IPCs (b) Epoch IPC, 4`ammp`,$SQVR_{lru,inf}$ (c) Epoch IPC, 4`vpr`, $SQVR_{lru,inf}$

Figure 8: Reactive Performance Differentiation

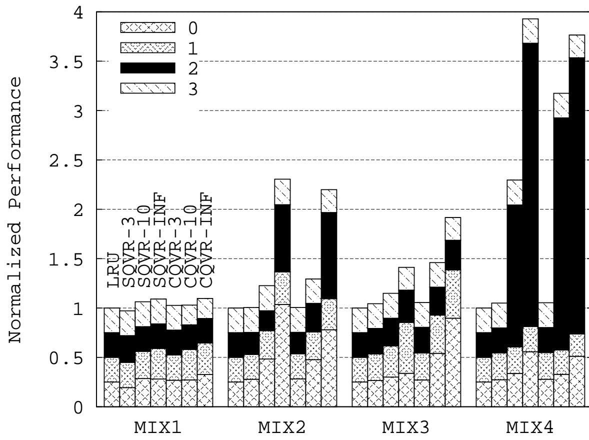


Figure 7: Performance of Reactive Fairness policy with hog

ment mechanisms (the corresponding normalized misses per instruction are shown in Figure 11 in Appendix B). The performance of each sharer is normalized to its performance in the LRU case and is shown by a box in the stack. As expected, this policy improves the performance significantly over the demand based LRU mechanism, by minimizing the effect of the `hog`. Mix_2, Mix_3 and Mix_4 all experience a big improvement in performance over LRU (Mix_4's performance improves by 3.9 times for $SQVR_{lru,inf}$), as their miss rates (not shown) were significantly reduced. Mix_2 and Mix_4 achieve particularly good performance because of `gcc`'s sudden and drastic reduction in miss-rate (`gcc` is very sensitive to its cache space allocation). Mix_1 does not show a significant improvement over LRU, because the benchmarks in Mix_1 use the cache aggressively and are not significantly effected by the `hog`. It can also be observed that, as the reluctance-threshold is increased, the performance improves. This is as expected because with lower thresholds, the `hog` still takes away cache blocks from other sharers.

4.4 Reactive Performance Differentiation

In this section, we show the results obtained by reactive performance differentiation policy MM_{pd} outlined in Section 2.3.3. For these experiments, the service-level priorities for sharers are $< 50, 50, 5, 5 >$. As mentioned in Section 2.3.3, the algorithm starts with equal cache quotas, and tries to make the ipc_{ratio} as close as possible to the $priority_{ratio}$. We also mentioned in Section 2.3.3 that this policy works only if IPC of each sharer is similar. Thus, instead of using multiprogrammed workload mixes mentioned in Table 1, we create 9 new workloads with the same benchmark running on all 4 processors (`4ammp, 4mcf, 4parser, 4vpr, 4facerec, 4gcc, 4twolf, 4applu, 4art`). Commercial workloads mentioned in Table 2 are still used as they have threads with similar IPC. Figure 8(a) plots IPC for different replacement mechanisms for all the 4 sharers, averaged over all the workloads. Each sharer's performance is normalized to its performance for column caching with equal way division (for the reasons mentioned in Section 4.2). Individual normalized IPC and misses per instruction are shown in Figure 13 and Figure 12 respectively, in Appendix B. As sharer 0 and 1 have high service-level priority, they are expected to show better performance than the other two sharers. This trend is clearly visible, but it does not become significantly more prominent as the reluctance-threshold is increased. This is because our feedback-driven policy further increases the quotas of high priority sharers if IPC does not change significantly at smaller thresholds. Thus even though higher reluctance thresholds enforce quotas strongly (sharer 2 and 3 lose performance), they offer little or no performance improvements to sharer 0 and 1 as compared to smaller reluctance-thresholds. The overall instruction throughput of $SQVR_{lru,3}$ and $CQVR_{lru,3}$ is slightly better than LRU on average (again not apparent from the graphs presented). $SQVR_{lru,10}$ and $SQVR_{lru,inf}$ show degradation in overall throughput by 2% and 7% respectively; while $CQVR_{lru,10}$ and $CQVR_{lru,inf}$ show degradation in overall throughput by 3% and 8% respectively.

To further analyze the operation of this reactive policy, Figs 8(b) and 8(c) plot IPC (not normalized) for every epoch of 2 million cycles (i.e. the interval at which our algorithm is invoked). These results are presented only for two workloads and one mechanism ($SQVR_{lru,inf}$); the rest of the results show the similar trend and are shown in Figure 14, Figure 15 and Figure 16 (in Appendix B) for $SQVR_{lru,3}$, $SQVR_{lru,10}$ and $SQVR_{lru,inf}$ respectively. Figs 8(b) and 8(c) show that sharers in the two workloads presented achieve the expected performance differentiation and maintain the difference with moderate stability. More robust control methods to tune quotas and reluctance-threshold are left for future work.

Throughout this section, we observe that using a higher reluctance threshold provides better results, as it enforces

quotas strongly and benefits the less aggressive users of cache space. The disadvantage of using higher reluctance thresholds is that the cache blocks stay in the cache longer even if they are not reused. Thus, the overall throughput (combined for all sharers) is higher at smaller thresholds in some cases. Note, however that OS policies are not always designed purely to maximize global throughput. A policy that enforces quotas strictly with a high reluctance threshold might be necessary if the OS has to guarantee minimum service levels to sharers.

5. RELATED WORK

We are not aware of any other work which studies flexible OS level policies for the management of shared CMP caches. Yeh *et.al.* [26] proposed a scheme which assumes equal priorities in hardware and attempts to optimize throughput in the shared cache while providing fairness guarantees. They use a "scratch pad array" to predict cache miss rate, which has a 10% overhead over the overall cache area. Settle *et.al.* [20] propose two different kinds of cache management schemes. They assumed per-cache-line re-use counters, thus their scheme has significant hardware overhead too. Though they mentioned the possibility of assigning high priority to one of the threads, they did not propose/evaluate a scheme for doing it. Kim *et.al.* [9] studied a hardware scheme for implementing fairness in cache. They propose 5 different metrics that can be used for fairness, and propose an algorithm which tries to equalize those metrics. We show in Section 4 how their fairness algorithm can be mapped to our scheme by an OS level policy. Suh *et.al.* [23] studied partitioning the cache among sharers by modifying the LRU replacement policy. They use per cache block counters that indicate the marginal gain as cache allocation of a sharer is increased. Their scheme can be used to minimize overall miss rate. Chiou *et.al.* [4] proposed column-caching that allows software to restrict data to certain portions of cache. If column-caching is used to enforce quotas, space will be reserved for a sharer even if it is not using it (i.e., quota will not only be a minimum guarantee, but a maximum limit too). We notice an increase in miss-rate of upto 14% over $SQVR_{lru,inf}$ if column-caching is used. Hence, the mechanisms studied in this paper are strictly better than column-caching. Liu *et.al.* [11] also suggested a way of splitting L2 cache. Though their scheme allows the OS to manage the cache space allocated to a sharer, they statically split the cache using profiling information. Moreover, their splitting mechanism has the same problem as column-caching i.e., cache space is allocated to a sharer even if it does not use it.

Our scheme is differentiated from the above-listed work in two key ways. First, our $SQVR$ mechanism differs from previous cache partitioning mechanisms in that it offers fine-grain, opportunistic quota enforcement that guarantees minimum cache allocation without unnecessarily limiting the maximum. Second, our work exploits the mechanism-policy decoupling to enable significant policy flexibility as opposed to single policy goals such as fairness, maximizing throughput or differentiated service.

Iyer *et.al.* [7] proposed a scheme for QoS support in shared CMP caches. They suggested that there could be many different approaches for cache priority classification, cache priority assignment, and cache priority enforcement. But they only studied cache priority enforcement in detail. One of their cache priority enforcement technique (they called it dynamic set partitioning) is similar to $SQVP$. But their technique supports only two priority levels and does not allow the low priority sharer to occupy all cache lines in a set, even if they are not used by the high priority sharer.

Some researchers have proposed to manage the lower level cache entirely in software [3, 6, 8, 12]. The involvement of software for handling L2 misses is costly/infeasible for the following two reasons. **First**, the software handling of L2 misses is expensive and it is really the reduction in miss rate (due to full associativity enabled by software management) that compensates for the overhead. While true for low-associativity L2 caches, this benefit is decreasing due to trend of increasing associativity (e.g, pentium D is 8-way and niagara is 12-way set-associative [10]). The benefit for 16-way associative caches used in our evaluation would be negligible because there is very little further reduction in miss-rate going from 16-way to fully associative caches. (Hallnor *et.al.* [6] Figure 3.) **Second**, the above mentioned proposals argue that the overhead of software miss handling could be overlapped with DRAM access. This argument is true if we consider only the critical path latency of a single thread. However, it ignores the execution bandwidth and the CPU power consumed by the instruction overhead of context switch and software miss handler. Our scheme solves this problem by reducing the frequency of OS intervention to once in a milli-second (instead of once for each L2 miss).

6. CONCLUSION AND FUTURE WORK

While operating systems have the ability to ration each process' CPU time, they are unable to exercise control over its cache space allocation. Existing cache management schemes are predominantly hardware based and offer little or no policy flexibility. The major contribution of this paper is the design of a flexible, low-overhead scheme for OS-level management of shared CMP caches. The scheme consists of three components: a hardware quota enforcement mechanism, an OS interface and a set of OS level quota management policies. The hardware mechanism enforces OS-specified quotas for each sharer and the OS can vary the policy by tuning the quotas. We propose a hardware quota enforcement mechanism with little area overhead (less than 10% tag area overhead and less than 1% cache area overhead for 4MB, 16-way L2 cache with 64 Byte blocks) and no timing overhead. The key advantage of our quota enforcing mechanism is that it enforces the minimum quota guarantees in the presence of contention without unnecessarily holding back the cache space in the absence of contention. The OS interface we propose is powerful and has area overhead of only 28 bits if 4 sharers are supported. Our scheme is versatile and provides support for a wide range of policies at the OS level including (a) passive performance differentiation, (b) reactive fair cache sharing policies and (c) reactive performance differentiation policies. The overhead of OS policies depends on their complexity. The policies we have implemented have timing overhead ranging from 0 to 2%. Our results indicate that our scheme successfully provides fairness and differentiated service. Extending this work to include (a) application-negotiated cache allocations and (b) application adaptation to allocated cache quotas is part of ongoing research.

Acknowledgments

We would like to thank anonymous reviewers for their feedback. This work is supported in part by Purdue Research Foundation XR Grant No. 690 1285-4010 and Purdue University.

7. REFERENCES

[1] Paul Barford and Mark Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. Technical Report, Boston, MA, USA, 1997.

[2] M. J. Buco, R. N. Chang, L. Z. Luan, C. Ward, J. L. Wolf, and P. S. Yu. Utility Computing SLA Management Based Upon Business Objectives. *IBM Systems Journal*, 43(1):159–178, 2004.

[3] D. R. Cheriton, A. Gupta, P. D. Boyle, and H. A. Goosen. The VMP Multiprocessor: Initial Experience, Refinements, and Performance Evaluation. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 410–421, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[4] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic Cache Partitioning via Columnization. In *Proceedings of Design Automation Conference, Los Angeles*, june 2000.

[5] Joseph R. Eykholt, Steve R. Kleiman, Steve Barton, Roger Faulkner, Anil Shivalingiah, Mark Smith, Dan Stein, Jim Voll, Mary Weeks, and Dock Williams. Beyond Multiprocessing: Multithreading the SunOS Kernel. In *Proceedings of theSummer 1992 USENIX Technical Conference and Exhibition*, pages 11–18, San Antontio, TX, USA, 1992.

[6] Erik G. Hallnor and Steven K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 107–116, New York, NY, USA, 2000. ACM Press.

[7] Ravi Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*, pages 257–266, New York, NY, USA, 2004. ACM Press.

[8] B. Jacob and T. Mudge. Software-Managed Address Translation. In *HPCA '97: Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, page 156, Washington, DC, USA, 1997. IEEE Computer Society.

[9] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *IEEE PACT*, pages 111–122, 2004.

[10] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[11] Chun Liu, Anand Sivasubramaniam, and Mahmut T. Kandemir. Organizing the Last Line of Defense before Hitting the Memory Wall for CMP. In *International Symposium on High Performance Computer Architecture*, pages 176–185, 2004.

[12] P. Machanick, P. Salverda, and L. Pompe. Hardware-Software Trade-Offs in a Direct Rambus Implementation of the RAMpage Memory Hierarchy. In *Proc. 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 105–114, San Jose, CA, October 1998.

[13] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[14] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 2005.

[15] Jim Mauro. The Solaris Process Model: Managing Thread Execution and Wait Times in the System Clock Handler, 2000.

[16] Kunle Olukotun and Lance Hammond. The Future of Microprocessors. *ACM Queue*, 3(7), September 2005.

[17] M. P. Papazoglou and D. Georgakopoulos. Service-oriented Computing: Introduction . *Communications of the ACM*, 46(10):24–28, 2003.

[18] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In *International Symposium on Computer Architecture*, pages 544–555, 2005.

[19] Parthasarathy Ranganathan and Norman Jouppi. Enterprise IT Trends and Implications for Architecture Research. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 253–256, Washington, DC, USA, 2005. IEEE Computer Society.

[20] Alex Settle, Dan Connors, Enric Gibert, and Antonio Gonzalez. A Dynamically Reconfigurable Cache for Multithreaded Processors. *Journal of Embedded Computing: Special Issue on Single-Chip Multi-core Architectures*, December 2005.

[21] P. Shivakumar and Norm P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power and Area Model. Technical Report, 2001.

[22] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts.* Addison-Wesley, 1998.

[23] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 117, Washington, DC, USA, 2002. IEEE Computer Society.

[24] G. Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic Partitioning of Shared Cache Memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[25] sun.com. Solaris Containers-Resource Management and Solaris Zones. In *System Administration Guide*.

[26] Thomas Y. Yeh and Glenn Reinman. Fast and Fair: Data-stream Quality of Service. In *CASES '05: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 237–248, New York, NY, USA, 2005. ACM Press.

[27] Li Zhang and Danilo Ardagna. SLA Based Profit Optimization in Autonomic Computing Systems. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 173–182, New York, NY, USA, 2004. ACM Press.

[28] Zhichun Zhu and Zhao Zhang. A Performance Comparison of DRAM Memory System Optimizations for SMT Processors. In *Proceedings of 11th International Sysmposium on High Performance Computer Architecture*, pages 213–224, 2005.

APPENDIX

A. CODE FOR HOG

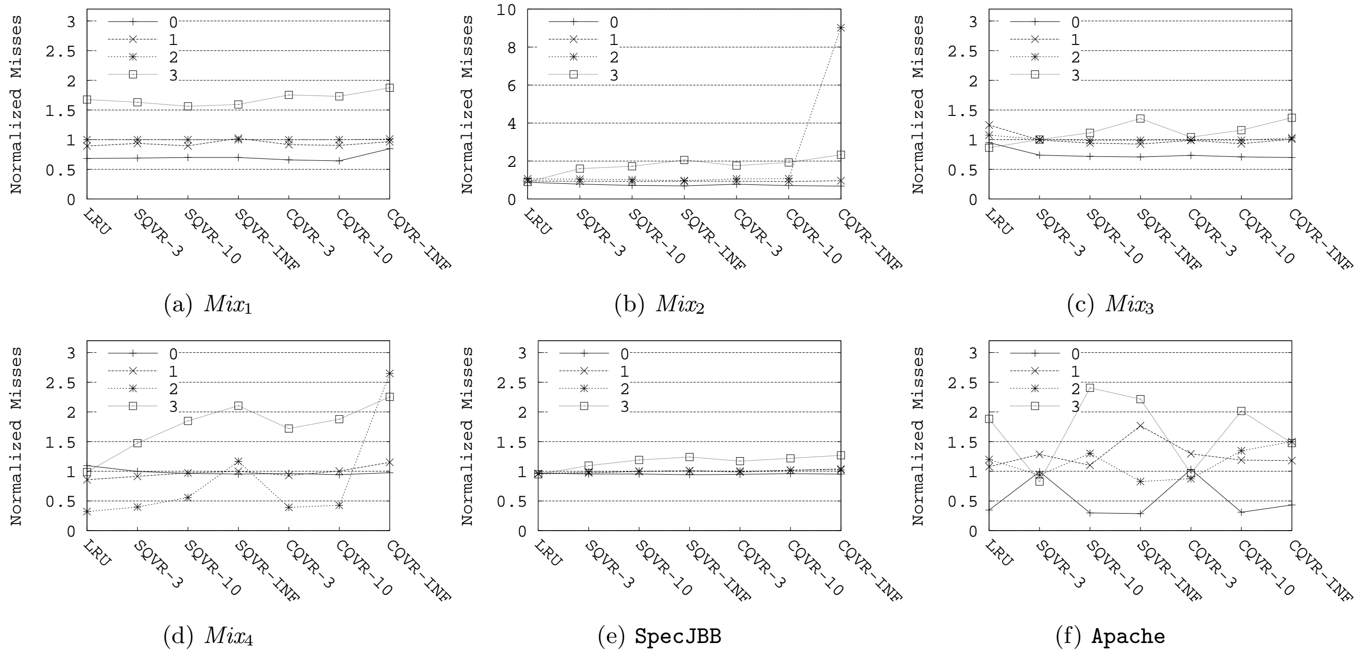
The code for hog is compiled with gcc version 3.4.4 using optimization flag -O3. The listing of the exact code used is provided here.

```
/***
This code is designed to be a cache hog. i.e. it creates a lot of
cache misses by accessing the elements of an array. The size of the
array is set to be equal to the cache size. It also goes back and
re-accesses the elements again so that they are not evicted from
cache.
Autors: Nauman Rafique and Won-Taek Lim
(nrafique,wlim@purdue.edu)
***/

#include <sys/types.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <stdio.h>

#define MEGABYTE ((size_t)(1024 * 1024))
#define FOUR_MEGABYTE ((size_t)4 * MEGABYTE)

/* Cache parameters */
#define CACHE_LSIZE 64
#define CACHE_ASSOC 16
#define CACHE_SETS 4096

/* Calculated parameters */
//Total number of cache lines
#define CACHE_LINES (CACHE_SETS * CACHE_ASSOC)
//Size of the array equal to the cache size
#define NUM_ARRAYS (CACHE_SETS * (CACHE_LSIZE / 8)  * ( CACHE_ASSOC ))
//Arbitrary large number for total iterations
#define NUM_ITER 400000000

int making_miss(double *array0) {
  register int burst_base_index,
    way_index,
    set_index,
    redo_index,
    iter_index;
  // Number of array elements to skip to get to the next cache line
  register int NEXT_LINE = (CACHE_LSIZE / 8);
  // Number of array elements to skip to get past 20 cache lines
  register int BURST_SIZE = (NEXT_LINE * 20);
  // Number lines to skip to get to the next cache way in the same set
  register int NEXT_WAY = (CACHE_SETS * NEXT_LINE);
  double sum;

  // Perform the sequence of accesses again and again
  for(iter_index = 0; iter_index < NUM_ITER; iter_index++) {
    sum = 0;

    /*
      The sequence of accesses is as follows: We first touch lines
      only in one of the ways before going to the next way. In one
      iteration of the innermost loop, we touch 20 conescutive cache
      sets. The number 20 is chosen arbitrarily and is used to
      amortize the loop overhead (loop unrolling).  Each way is
      touched twice before going to the next way, to keep the cache
      lines from being evicted.
     */
```

```
    // Loop over all the ways
    for (way_index=0; way_index < NUM_ARRAYS; way_index+=NEXT_WAY) {
      // Touch the same way twice before going to next
      for(redo_index = 0; redo_index < 2; redo_index++) {
// Access all sets in the way in burst of 20
for(burst_base_index=way_index,set_index=0;
    set_index < CACHE_SETS;
    burst_base_index+=BURST_SIZE,set_index+=20) {
  sum +=
    array0[burst_base_index+(0*NEXT_LINE)]+
    array0[burst_base_index+(1*NEXT_LINE)]+
    array0[burst_base_index+(2*NEXT_LINE)]+
    array0[burst_base_index+(3*NEXT_LINE)]+
    array0[burst_base_index+(4*NEXT_LINE)]+
    array0[burst_base_index+(5*NEXT_LINE)]+
    array0[burst_base_index+(6*NEXT_LINE)]+
    array0[burst_base_index+(7*NEXT_LINE)]+
    array0[burst_base_index+(8*NEXT_LINE)]+
    array0[burst_base_index+(9*NEXT_LINE)]+
    array0[burst_base_index+(10*NEXT_LINE)]+
    array0[burst_base_index+(11*NEXT_LINE)]+
    array0[burst_base_index+(12*NEXT_LINE)]+
    array0[burst_base_index+(13*NEXT_LINE)]+
    array0[burst_base_index+(14*NEXT_LINE)]+
    array0[burst_base_index+(15*NEXT_LINE)]+
    array0[burst_base_index+(16*NEXT_LINE)]+
    array0[burst_base_index+(17*NEXT_LINE)]+
    array0[burst_base_index+(18*NEXT_LINE)]+
    array0[burst_base_index+(19*NEXT_LINE)];
}// Burst
      }// Redo
    }// Ways
  }// Iterate
  return sum;
}


/*
   The main function just allocates space for the array and calls the
   array access function.
*/
int main(int argc, char *argv[])
{
  struct memcntl_mha mha;
  double *src0,sum;

  printf("Hog parameters: ");
  printf("Cache lines = %d, Line size = %d, Assoc = %d\n",
 CACHE_LINES,CACHE_LSIZE,CACHE_ASSOC);
  printf("Cache size = %d KB, Array size= %dKB\n",
 (CACHE_LINES*CACHE_LSIZE/1024),NUM_ARRAYS*8/1024);

  /* Set pagesize to 4MB for heap */
  mha.mha_cmd = MHA_MAPSIZE_BSSBRK;
  mha.mha_flags = 0;
  mha.mha_pagesize = FOUR_MEGABYTE;
  memcntl(NULL, 0, MC_HAT_ADVISE, (char *)&mha, 0, 0);

  /* Allocate memory for the array*/
  src0 = (double *)memalign(FOUR_MEGABYTE,sizeof(double)*NUM_ARRAYS);

  /* Call the function which makes misses*/
```

```
  sum = making_miss(src0);

  printf("array access kernel was successfully done with sum = %.cf !\n",sum);
}
```

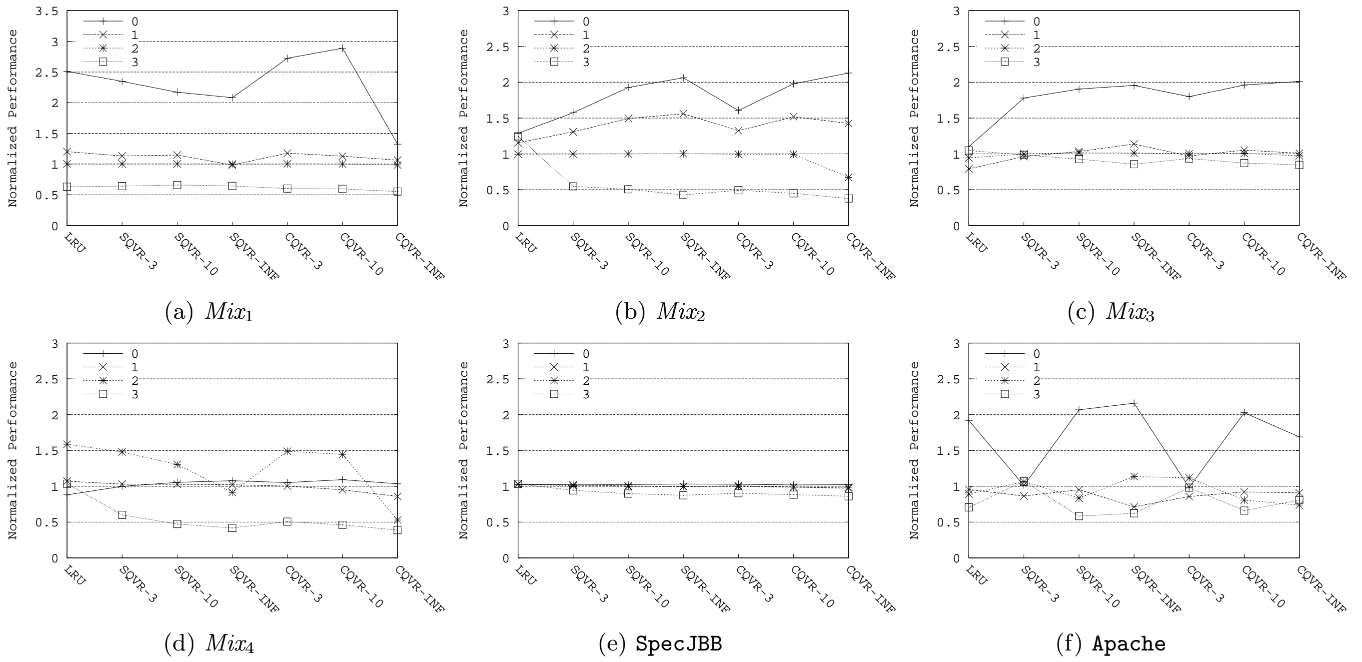
B. DETAILED RESULTS



(a) Mix_1 (b) Mix_2 (c) Mix_3

(d) Mix_4 (e) SpecJBB (f) Apache

Figure 9: Passive Performance Differentiation: Misses per instn. normalized to column-caching



(a) Mix_1 (b) Mix_2 (c) Mix_3

(d) Mix_4 (e) SpecJBB (f) Apache

Figure 10: Passive Performance Differentiation: IPC normalized to column-caching

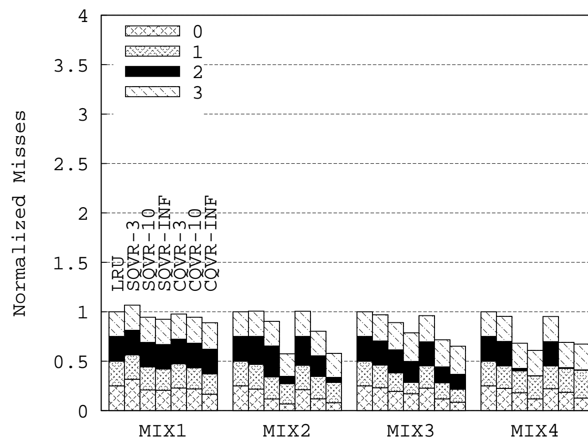
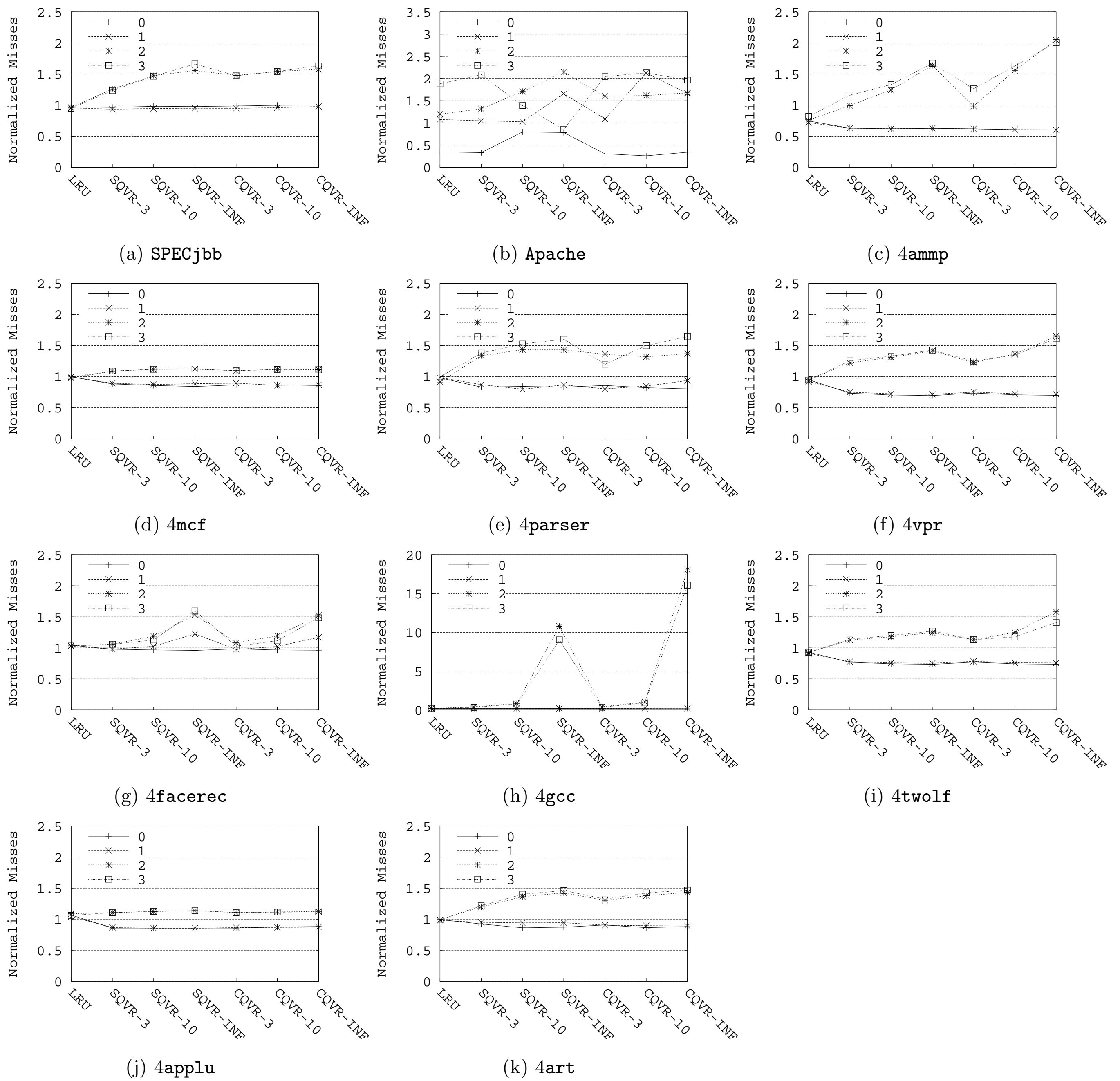


Figure 11: Misses per instn. of Reactive Fairness policy with hog



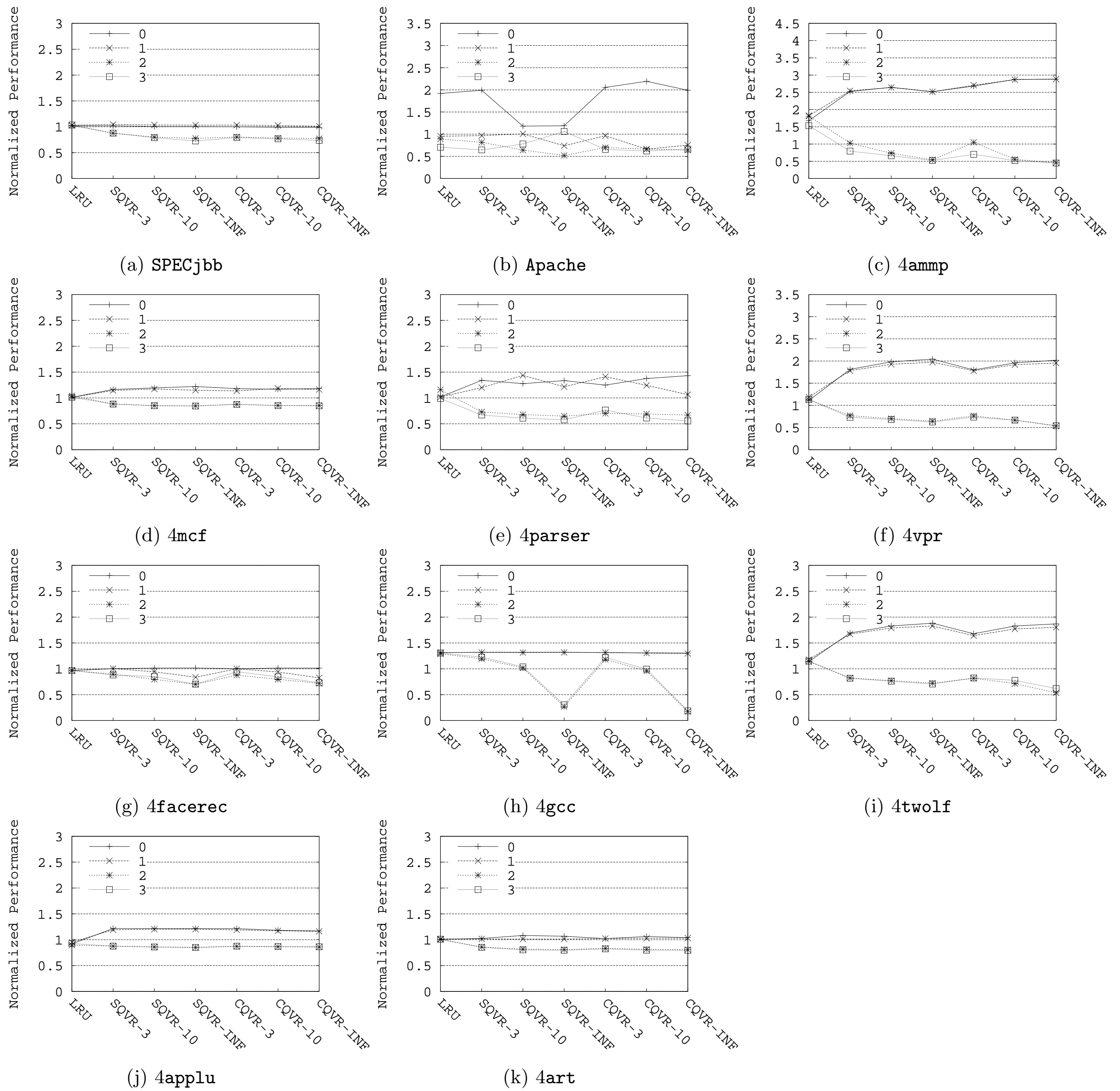
(a) SPECjbb (b) Apache (c) 4ammp

(d) 4mcf (e) 4parser (f) 4vpr

(g) 4facerec (h) 4gcc (i) 4twolf

(j) 4applu (k) 4art

Figure 12: Reactive Performance Differentiation: Misses per instn. normalized to column-caching



(a) SPECjbb (b) Apache (c) 4ammp

(d) 4mcf (e) 4parser (f) 4vpr

(g) 4facerec (h) 4gcc (i) 4twolf

(j) 4applu (k) 4art

Figure 13: Reactive Performance Differentiation: IPC normalized to column-caching

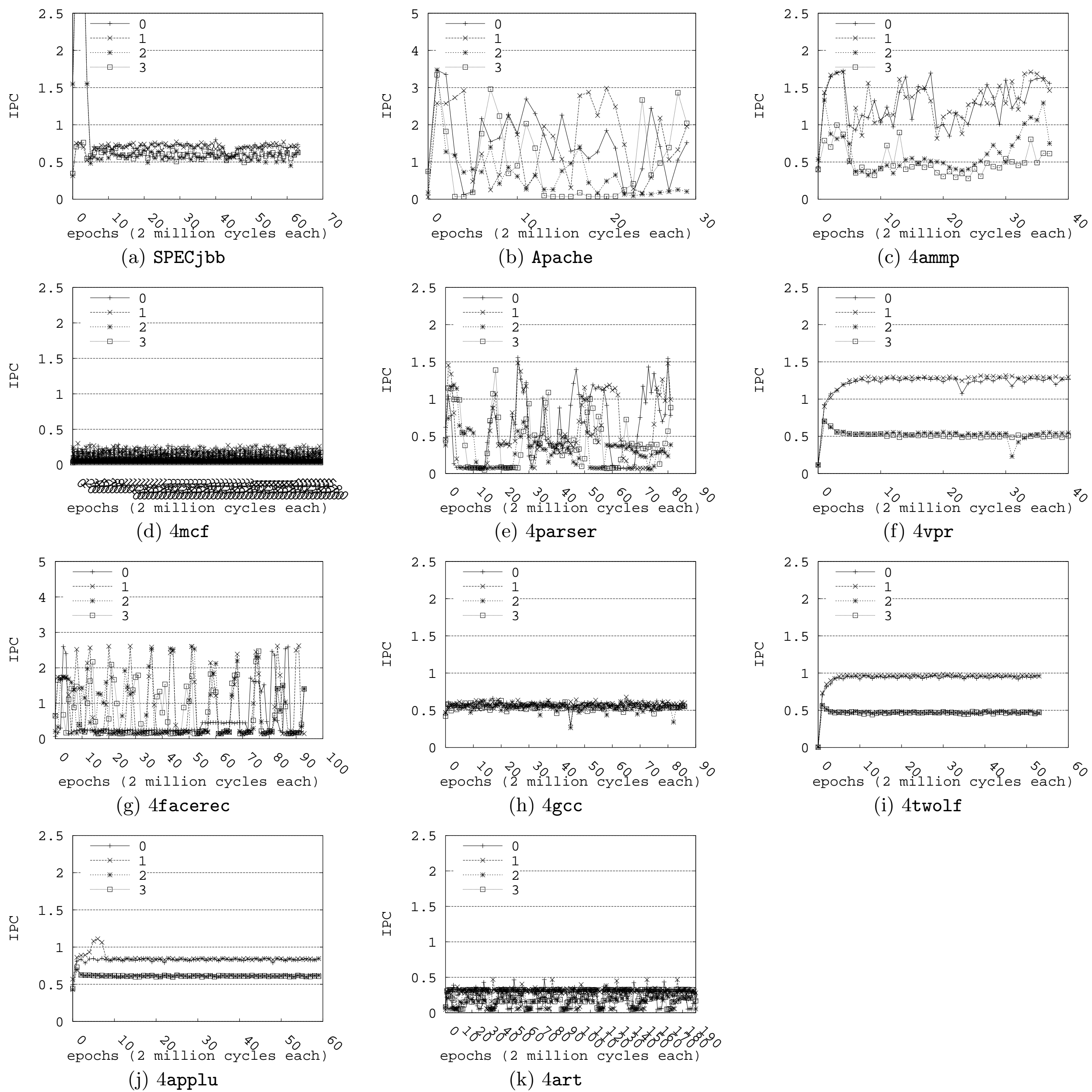


Figure 14: Reactive Performance Differentiation: Epoch IPC with $SQVR_{lru,3}$

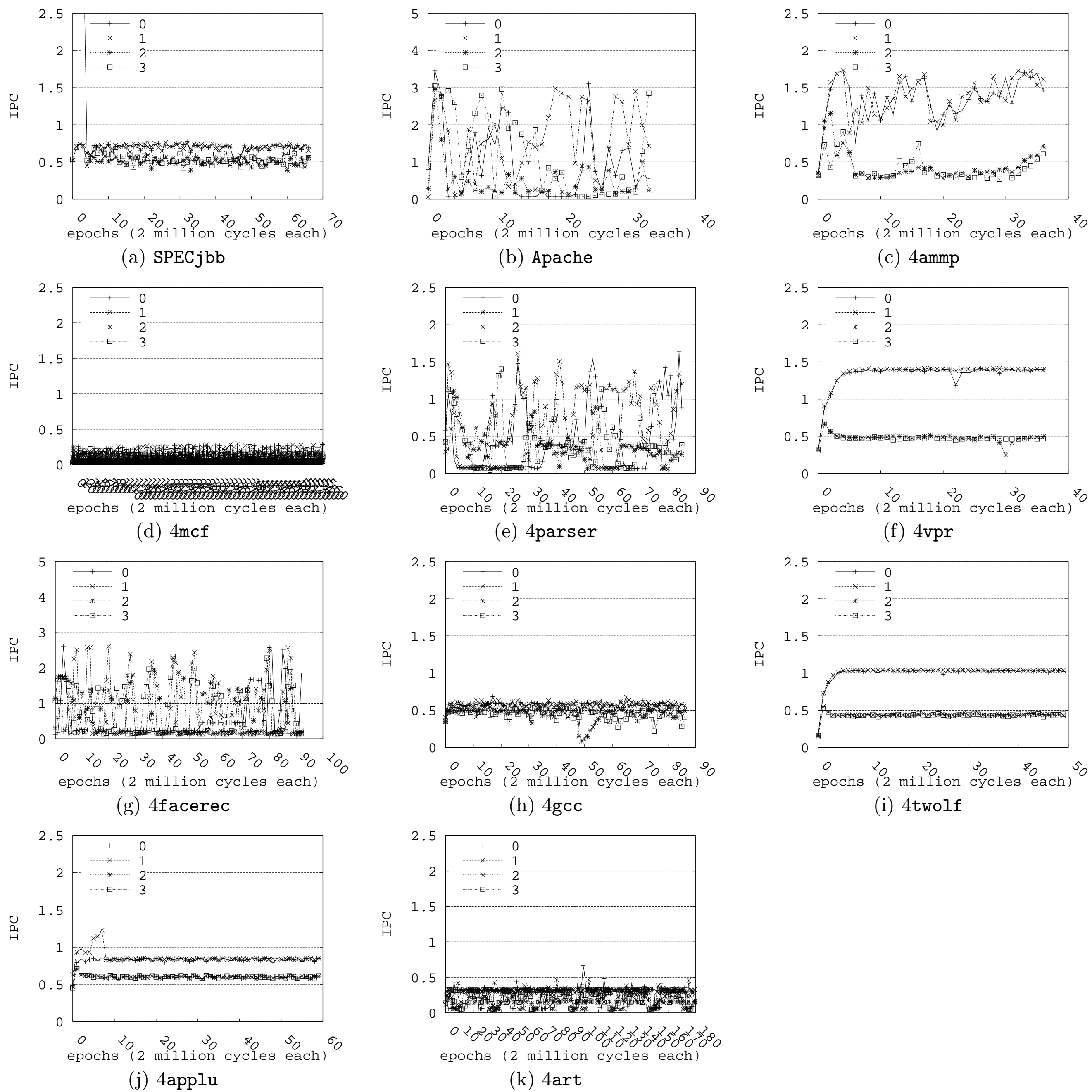


Figure 15: Reactive Performance Differentiation: Epoch IPC with $SQVR_{lru,10}$

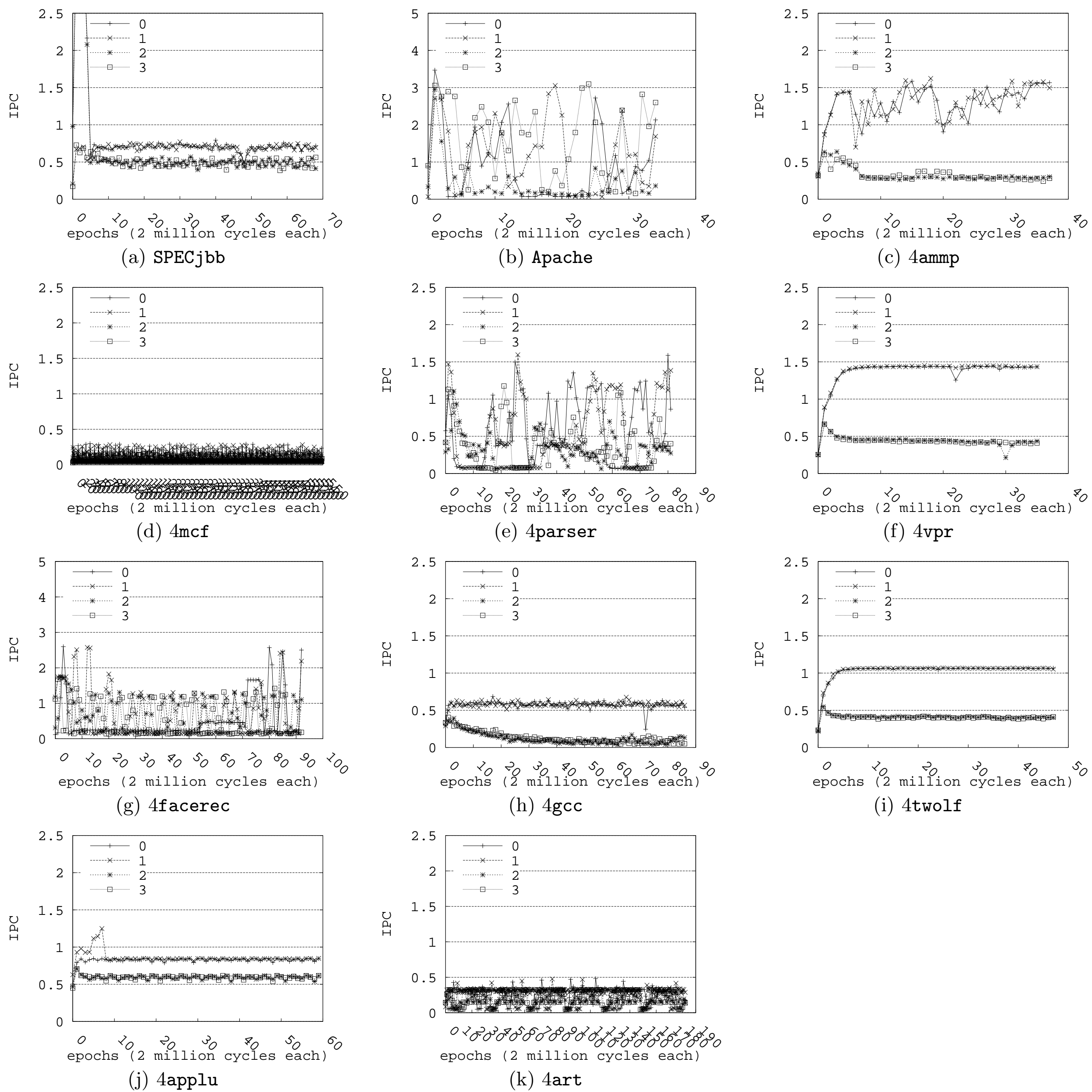


Figure 16: Reactive Performance Differentiation: Epoch IPC with $SQVR_{lru,inf}$