Cache Restoration for Highly Partitioned Virtualized Systems

David Daly and Harold W. Cain
IBM Thomas J. Watson Research Center
Yorktown Heights, NY
{dmdaly, tcain}@us.ibm.com

Abstract

The economics of server consolidation have led to the support of virtualization features in almost all server-class systems, with the related feature set being a subject of significant competition. While most systems allow for partitioning at the relatively coarse grain of a single core, some systems also support multiprogrammed virtualization*, whereby a system can be more finely partitioned through time-sharing, down to a percentage of a core being allotted to a virtual machine. When multiple virtual machines share a single core however, performance can suffer due to the displacement of microarchitectural state.*

We introduce cache restoration, *a hardware-based prefetching mechanism initiated by the underlying virtualization software when a virtual machine is being scheduled on a core, prefetching its working set and warming its initial environment. Through cycle-accurate simulation of a POWER7 system, we show that when applied to its private per-core L3 last-level cache, the warm cache translates into 20% on average performance improvement for a mixture of workloads on a highly partitioned core, compared to a virtualized server without cache restoration.*

1. Introduction

Architectural support for virtualization has emerged as an essential feature provided by every major server vendor, allowing the sharing of hardware resources by multiple guest operating system instances, variously called logical partitions, containers, domains, or virtual machines. We refer generically to such containers as *partitions,* and the underlying virtualization software as the *partition manager.* Virtualization support has been motivated by cost-savings through server consolidation and improved systems management. With the recent support in x86 servers, it has been estimated that the average utilization of such systems has increased from 36% to 56% [4]. While a significant gain, a utilization of 56% leaves considerable opportunity for further server consolidation, which would result in significant savings in power, cooling, and facilities costs. In comparison, IBM mainframe installations, which have the longest history of support for virtualization, are said to run at average utilizations of 80% [15]. To increase the opportunity for such consolidation, a system must support a large number of partitions. For example, in one case-study IBM reports the consolidation of 3,900 x86-based servers onto 30 System z mainframes, a ratio of 130 virtual machines per server [11]. The ongoing shift away from dedicated workstations to virtualized desktop infrastructure (VDI) environments is another trend in this direction but also a serious challenge; the low cpu-utilization of individual desktops leads to high consolidation rates, while also requiring low (unnoticeable) interactive response times to desktop users. As an example of consolidation rates in this context, VMWare lists several case studies of customers transitioning from dedicated workstations to VDI, with virtual desktops per server ranging from 4-to-1 to 15-to-1 for those customers [18]. Botelho summarizes the difficulty in calculating an estimated desktop to server ratio due to wide variations in desktop usage, but does recommend a rule-of-thumb of six to eight virtual desktops per core on the host [3].

While cost-reduction pressures are one factor leading to more partitions per system, other factors also contribute to this trend. In some environments (e.g. cloud computing installations) it may be required to separate applications belonging to different customers into individual operating system instances for security reasons. System management convenience is also a contributing factor, allowing for different software installations tailored to specific needs (e.g. allowing for one OS image per user, or testing of multiple operating system or library upgrades). Although some systems limit the number of active partitions to prevent a greater number of active partitions than cores, others include support for what we call *multiprogrammed virtualization*, in which two or more partitions are assigned to a single core or hardware thread, which is multiprogrammed by the underlying partition manager [10, 19].

Unfortunately, multiprogrammed virtualization can af-

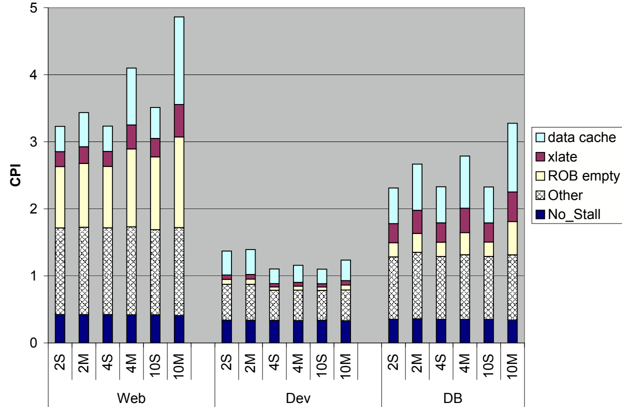


Figure 1. CPI for three workload mixes (web serving, development, and database), for sharing levels of 2, 4, and 10, with multiprogrammed virtualization enabled (M) and disabled (S: Serial workload execution).

fect performance in a number of ways. First, there is overhead incurred by the partition manager to switch partitions. But a more significant impact is caused by the pollution of a partition's working set; at the time that the victimized partition is subsequently rescheduled, most or all of its working set has been evicted from the core's caches, branch predictor, and TLBs. While the effects of this displacement can be amortized by maximizing the amount of time each partition runs before being switched out, requirements for acceptable real-time interactive response times of the guest operating systems dictate that each be permitted to execute within a fixed time interval, as frequently as every 10 ms in some systems [10]. Figure 1 shows the result of a study measuring this negative impact as the number of partitions that are active on a single core is varied, with configurations of 2,4,and 10 partitions, each being allocated a uniform portion of a 10 ms scheduling window. As one would expect, CPI increases with the number of partitions per core. At two partitions per core, performance suffers between 2% and 11%, depending on the workload type. At 4 partitions per core, this range increases to 4% to 25%, and at 10 partitions, varying from 12% to 40%.

While degradation caused by i-cache and branch predictor pollution (as indicated by ROB-empty stalls), and translation misses are significant, especially in the database and web serving workloads, stalls due to data cache misses are the dominant source of overheads, accounting for over half of the increased stall cycles. As discussed in Section 4, others have described the cache polluting effects of context switches, but our work is the first to present the effects of multiprogrammed virtualization, whose response time constraints require an additional degree of multiprogramming above and beyond OS-level multiprogramming, inducing severe cache pollution.

In response to this pollution, we introduce *cache restoration,* a hardware-based prefetching mechanism initiated by the partition manager when a partition is being scheduled on a core, prefetching its working set and thereby warming its initial environment. Through cycle-accurate simulation, we show that the warm cache translates into 20% average performance improvement in a simulated POWER7 system when applied to its private per core L3 last-level cache[16], for a mixture of workloads on a highly partitioned core, compared to a virtualized server without cache restoration.

Because partition switches made by the partition manager occur in addition to any multiprogramming performed by the operating system, context switching will occur to a greater degree than reported in previous studies of multiprogramming cache effects [1, 8, 12, 14, 13, 17]. Prefetching for operating-system level multiprogramming has also been a subject of recent work by Cui and Sair, who proposed a *global history list* (GHL) [8]. We also evaluate applying the GHL prefetcher to partition switches, and show that cache restoration prefetching approaches the performance of GHL prefetching while requiring fewer and simpler hardware resources.

In summary, this work makes the following contributions:

- We demonstrate the harmful performance effects of multiprogrammed virtualization, caused by the pollution of microarchitectural structures. Although similar issues have been explored in the context of multiprogramming by the operating system , to our knowledge, this is the first work exploring the challenges of multiprogrammed virtualization stemming from the increased multiprogramming rates of this environment.
- We describe and evaluate a novel prefetching mechanism that restores cache contents on behalf of a newly rescheduled partition, showing that it improves performance up to 31% in a highly partitioned environment. We also show that this prefetcher approaches the performance of global history list prefetching [8], while requiring fewer on-chip resources.
- We show that restoring only a portion of the cache has significant benefits, and describe a method of programming the prefetcher that allows for control of performance vs. bandwidth consumption.
- We show that a simple adjustment to the replacement algorithm that preferentially chooses cache lines from another partition for replacement yields small but significant performance gains across the workload mixes.

We will first describe cache restoration in detail, followed by analysis of its performance in Section 3 using cycle-accurate full system simulator. Discussion of related work is presented in Section 4.

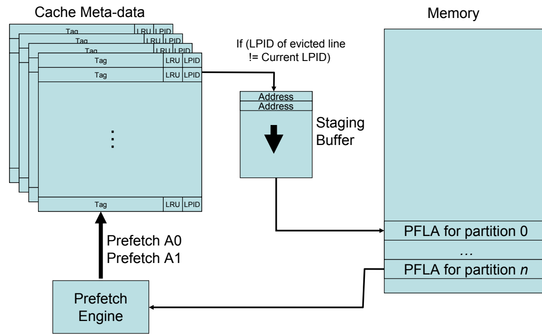


Figure 2. Block diagram of the cache restoration prefetcher.

Tag Array	4 extra bits per line
Staging Buffer	128B single ported per buffer. 2-16 buffers
Control Registers	4b active partition + 64b pointer/staging buffer
Cache Logic	4 bit comparator
Prefetch Logic	Read from buffer and send address to existing prefetcher with additional mux
Buffer Logic	Detect buffer full or empty conditions, inducing write or read to memory

Table 1. Summary of additional hardware due to Cache restoration

2. Cache Restoration Prefetching

Cache restoration prefetching consists of two steps: the collection of the cache footprint for a partition (the victim) that has been descheduled by the partition manager, and the subsequent prefetching of that victim's footprint at the time that it is rescheduled. Cache restoration prefetching could be applied to any address-based cache structure, in this paper however we assume it is used with an L3 cache, since L3 misses are the source of most processor stalls in our experimental system. A high-level block diagram of the prefetching mechanism is shown in Figure 2, which is described in detail in the following subsections. Table 1 details the hardware overheads.

2.1. Collecting the Cache Footprint Lazily

Rather than attempt to save the victim's footprint eagerly while the partition is executing, or at the time the partition is descheduled, we adopt a lazy scheme that records the addresses of cache lines belonging to the victim partition at the time that they are evicted from the cache. In order to identify the victim's lines and distinguish those from lines belonging to other partitions, the cache is augmented with a *logical partition ID* (LPID) field associated with each cache line, which indicates the particular partition that is using the line. LPID 0 is treated specially to indicate that the cache line is shared by multiple partitions (or shared with the partition manager). Assuming up to 16 partitions per core (a 4-bit LPID), LPID storage overhead represents less than 10% of the cost of the existing tag array (which is itself small relative to the data array). [1]

At the time that a partition is scheduled, a register is set by the partition manager indicating that the LPID for the newly scheduled partition is the active LPID (and therefore all other LPIDs are inactive). On a cache miss, lines with inactive LPID values are preferentially chosen for eviction, regardless of LRU state, since it is unlikely that an inactive partition's line would be used by the current partition. Among the cache lines with inactive LPID values, LRU status is still used to decide which inactive line to evict. The shared LPID is always considered active. If the current partition requests a line resident in the cache with an LPID value different from the partition's LPID, the line's LPID value is updated to 0, indicating a shared LPID.

The LPID extension to the cache enables the lazy collection of footprint data. Rather than eagerly recording cache-resident addresses during steady-state execution, or at the time of a partition switch, the recording of addresses is performed lazily. Once a line is chosen for eviction, the cache checks if the line's LPID is active. If the LPID is inactive, a hardware machine associated with the cache writes the address of the cache line to a memory-resident log of cache lines that have been evicted for that partition. The log represents the evicted footprint of the partition, and resides in a private region of memory allocated to the partition manager but invisible to each partition, called the *Partition Footprint Log Area* (PFLA). The size of the PFLA per partition is at most the width of a cache line address times the number of lines in its cache (39 bits x 32k lines = 156KB for our simulated system), so a tiny fraction of total partition memory.

To avoid writing to the PFLA on every replacement, a set of staging buffers is associated with the cache, with each staging buffer being the size of a cache line. The address of the victim line is appended to the staging buffer associated with the victim's LPID. When the buffer is full, its contents are written to the PFLA for the LPID.

Since the existing LRU mechanism is used to decide which inactive line to evict from the set, the partition footprint log contains a partial order among the addresses in the saved footprint. Consider two lines X and Y for a given inactive partition and a given set, with Y being less recently used than X. Y will be evicted before X, and therefore will be recorded earlier in the footprint than X. Lines that map to the same set will be ordered in the log, however lines in different sets will not be ordered by their recency of access.

[1]Assuming a system with 48 bits of metadata (tag, coherence state, ECC, LRU stack) per cache line, which corresponds to our baseline POWER7 system.

It should be noted that the width of the LPID field does not limit the number of partitions supported by the system, or even the number of partitions that can be assigned and multiplexed on one core. The LPID only controls this prefetching and replacement mechanism. If a partition manager chooses to reuse an LPID for a given partition, its only potential negative effect is that any cache-resident lines belonging to the old partition at the time that the LPID is reassigned may be subsequently written to the PFLA of the new partition, and be subject to subsequent prefetches. This may affect prefetch accuracy, but not correctness. Since a partition manager should choose for reassignment the LPID corresponding to the partition that executed least recently, only those cache lines that have survived the prior 15 partition executions (assuming a 4-bit LPID) could be subject to such unnecessary prefetching, which is expected to be a marginal amount.

While our cache restoration prefetcher uses a simple in-memory list of addresses, prior work by Wenisch et al. has also proposed the in-memory storage of prefetcher meta-data, in that case allowing meta-data to exceed on-chip capacity in correlation-based prefetchers [20]. Such in-memory storage has also been leveraged for increased meta-data capacity for other forms of predictors [6]. While this prior work also used memory as a repository of meta-data, the employed prefetching algorithms are significantly different from our cache restoration prefetching mechanism.

2.2. Footprint prefetching

A simple hardware prefetcher uses the footprint addresses stored in the PFLA to prefetch the partition's cache footprint back into the cache. The prefetch engine sequences through the list of blocks that were saved in the footprint, reading a cache-line sized block of addresses at a time from memory, and subsequently issuing prefetch requests for each address in the line. When the partition is scheduled, the partition manager sets a special purpose register with the address of the PFLA, which triggers the prefetcher to begin prefetching using the addresses in that log. In theory this register could be set early by the partition manager as soon as it knows the LPID of the partition that will be scheduled, in order to give the prefetching mechanism a head start. We have not investigated this further however, and in our experiments conservatively assume that prefetching is delayed until the newly scheduled partition begins execution.

Since LRU members have been evicted first and saved first, closer to the head of the footprint list, for timeliness the prefetch engine traverses the list in the reverse order, from tail to head. The prefetcher loads the prefetched line into the LRU state. However, since there are lines belonging to inactive partitions in the set during the prefetch period, subsequent prefetches evict the inactive lines instead of the recently prefetched line. As each set fills, the prefetched lines are pushed onto the top of the LRU stack, moving the lines prefetched earlier toward the MRU position.

We envision a cache restoration prefetcher design that is tightly integrated with a conventional stride or stream-based prefetcher, in order to leverage the cache interface and as much pre-existing logic as possible. The stream prefetcher is augmented with a base register for storing the PFLA address, as well as two cache line-sized buffers, into which PFLA entries are read; two are used so that prefetches can be issued from one buffer while the other buffer is being filled from the PFLA. Addresses are read from the buffer, and issued to the existing prefetch logic.

2.3. Prefetch Throttling

The footprint prefetch also introduces bandwidth overhead that could adversely affect the system. Any prefetch that is eventually used before eviction does not introduce bandwidth overhead, but any prefetch that is not used does introduce overhead. We propose a simple limit to the number of prefetches to address the bandwidth overhead. With the addition of a single counter, we allow the partition manager to set a programmable limit to the number of lines in the footprint to prefetch back into the cache, directly limiting the number of wasteful prefetches possible. We evaluate the prefetcher using a range of limits in Section 3.

3. Experimental Results

We evaluate the effectiveness of the cache restoration prefetcher using three workload mixes that represent common usage scenarios for virtualized servers. Each mix consists of a set of traces from an IBM System *p* server running AIX.

- Web Serving Mix: A collection of traces from systems running the Daytrader J2EE benchmark, using the WebSphere Application Server v 7.0 and DB2 9.1.
- Database Serving Mix: A collection of traces from a system running an OLTP benchmark using DB2 v9.0.
- Development Mix: A collection of traces of development related applications from the SPEC CPU benchmark suite: gcc, perl, bzip2

For each mixture, a round-robin approach is used to mimic the multiprogrammed virtualization that would be incurred by the underlying partition manager. Each of the mixtures is run in a 2-way, 4-way, and 10-way partitioned configuration, corresponding to a system with a 10 ms scheduling wheel divided equally among the partitions. Since we are simulating a core with a frequency of 4GHZ, this corresponds to time quantums of 4M, 10M, and 20M cycles for the 10x, 4x, and 2x configurations respectively.

For comparison, the mixture for each configuration is also executed without partitioning, running each trace in its entirety before moving on to the next. These runs, marked *serial* in the results, indicate system performance for each mixture without the negative effects of multiprogrammed virtualization.

Each mixture of traces is simulated on a version of Mambo[2], which has been heavily modified with a cycle-accurate model of a POWER7 system, including out-of-order core, cache hierarchy, Powerbus, and memory subsystem [16]. A detailed machine description used for all experiments is listed in Table 2. For each data point, 1B instructions are executed in total. During simulation, a partition-unique offset is added to the physical addresses of references by the simulated partition, mimicking the partitioning of memory that occurs in a virtualized system.

For simplicity we assume a single 128B staging buffer per partition in our experiments, however a more realistic design would provision a small fixed set of staging buffers; since most cache lines from a victimized partition will have been evicted after a few intervening partitions have executed, benefits from writing subsequently evicted lines from that partition will be small.

We modeled five configurations:

- Baseline: No prefetching
- LPID-based eviction: Cache lines with inactive LPID values are preferentially evicted
- Cache Restoration Prefetching: Cache Restoration Prefetching in conjunction with LPID-based eviction, while modeling all bandwidth and latency costs of prefetching.
- Perfect Prefetching: Cache Restoration Prefetching in conjunction with LPID-based eviction, in a "perfect" zero-latency/zero-bandwidth memory system for prefetches. These results are used to gauge the performance opportunity lost due to bandwidth constraints and prefetch timeliness.
- Serial: Execution without partition switching. Each trace is processed to completion, in order to measure performance in the absence of virtualization effects.

The LPID-based eviction strategy will allow lines that survive in the cache from the time the partition is descheduled, until it is rescheduled, to be productively used by the cache, while cache restoration prefetching also performs a prefetch based on the saved footprint. The perfect prefetch serves as a comparison point to cache restoration prefetching for analyzing the impact of the bandwidth overhead and timeliness of the prefetches.

3.1. Performance

Although we have evaluated the cache restoration prefetcher when used at both the L2 and L3 caches, and against the 2-way, 4-way, and 10-way multiprogrammed virtualized systems, due to space constraints we present only the 10-way results when applied at the L3; we have found the impact of perfect L2 prefetch to be minor compared to prefetching at the L3, and we focus on the 10-way workloads since they are the most strenuous and we observe similar ability to overcome the effects of virtualization in the less heavily partitioned systems.

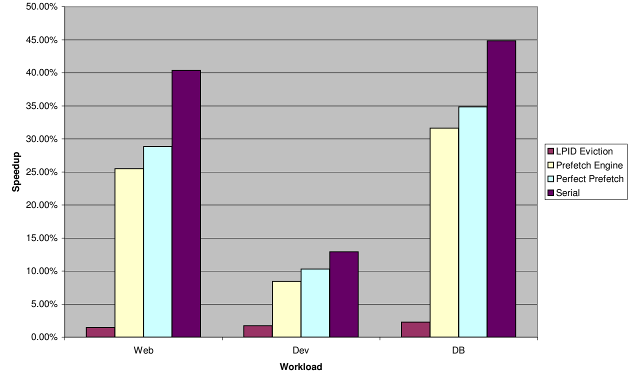


Figure 3. Speedup for workload footprint prefetch techniques.

Speedup results are shown in Figure 3, relative to the multiprogrammed virtualized system without any cache restoration. Cache restoration is able to restore the majority of performance loss caused by micro-architectural displacement, closely following the "perfect" prefetching mechanism. The difference between cache restoration and perfect prefetching is due to both the reduced timeliness of the prefetches, as well as the bandwidth due to prefetching degrading memory latency for demand references. Perfect prefetching falls short of the non-virtualized serial execution due to remaining pollution of the i-cache, branch predictor, and translation buffers, which is not addressed by our technique. We also note that the LPID based eviction has a small but noticeable improvement on performance, indicating that even with the 10-way partitioned workload, some of the partition's working set is still cache-resident when the workload is rescheduled. The LPID-based eviction prevents those lines from being evicted before newer, but inactive lines are evicted. The cache restoration speedup results directly follow improvements in L3 miss rate, shown in Figure 4(a). Figure 4(b) shows a breakdown of CPI for each configuration. In addition to the reduced commit stalls for data cache misses, we also observe fewer i-cache miss stalls due to an increased number of L3 hits on i-cache misses, and fewer TLB miss stalls since we observe the L3 hit rate is also improved for the hardware page table walker.

Out-of-order core	4 GHz 16-stage pipeline, 20 entry reorder buffer (six grouped instructions per entry, subject to grouping restrictions). 64 entry load buffer, 64 entry store buffer.
Fetch/Issue/Commit width	8/6/6
Issue queues	Pair of 24 entry queues for FXU/FPU/LSU, 12 entry branch queue, 8 entry compare queue.
Functional units	2 FXU, 2 FPU, 2 LSU, 1 branch, 1 compare.
Branch prediction	Combining predictor with 16k entry global/8K entry local/8K entry select. 16-entry call/return stack. 128 entry indirect branch target cache.
L1 instruction cache	32KB 4-way associative, 128-byte lines, 1 cycle latency, max 4 outstanding misses.
L1 data cache	32KB 8-way associative, 128-byte lines, 2 cycle latency, max 8 outstanding demand load misses and 4 outstanding prefetches.
L2	Private 256KB 8-way associative, 128-byte lines, 8 cycle latency, 64-entry STQ/RC machine, gathering on 128-byte block basis.
Conventional Prefetcher	A sequential stream prefetcher supporting up to 12 active streams, as well as software prefetch instructions. These prefetching mechanisms are enabled in all simulations, with and without cache restoration.
Interconnect	Power7 Powerbus model connecting L2/L3 controllers into a local coherence domain.
L3	Private 4MB 8-way associative, 128B lines, 22 cycle latency
Memory	640 MHz DDR3, 400 cycle best case latency. 2 memory controllers interleaved on cache block boundary.

Table 2. Experimental machine configuration.

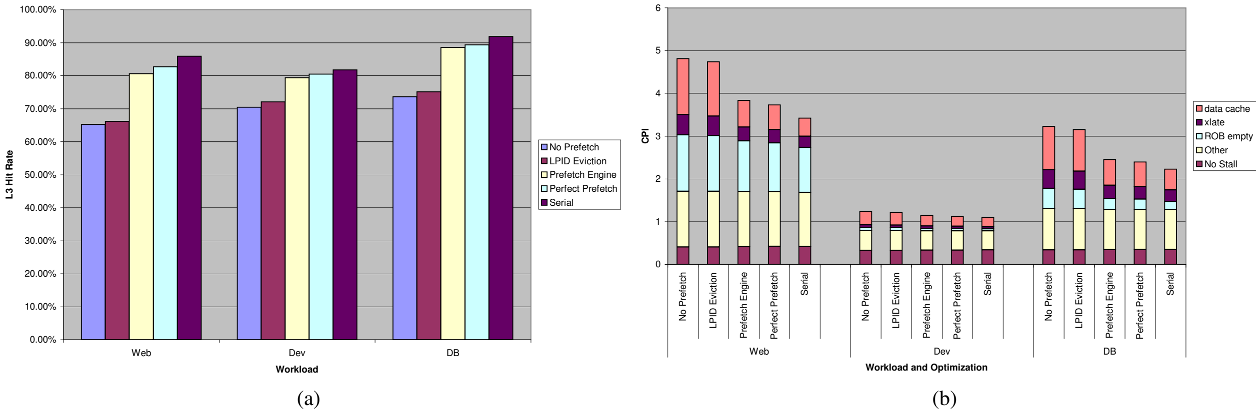


Figure 4. Effect of workload footprint prefetch techniques on (a) L3 hit rates and (b) CPI breakdown.

3.1.1 Bandwidth overhead and mitigation

While the performance gains due to cache restoration are considerable, its bandwidth requirements are also nontrivial. In this section, we explore the prefetch throttling mechanism (previously described in Section 2.3), which limits the total number of prefetches that are issued when a partition is rescheduled by the underlying partition manager. Figure 5(a) illustrates the impact of this throttling on performance, when limiting the number of prefetches to 2000, 4000, 8000, 16000 (out of 32K lines in the 4MB cache). As one would expect, we see an increase in performance as more lines are prefetched. However, the performance improvement gained by prefetching further into the saved footprint is sub-linear to the number of prefetched lines, implying that the lines prefetched first are more likely to be used than the lines prefetched later, which is expected based on the LRU properties of the stream. MRU items are prefetched first and LRU items later, so one would expect prefetches closer to the head of the list to be more accurate than those near the tail. Additionally, these prefetches are more likely to be timely since they are issued first.

Depending on the workloads and the system configuration, memory bandwidth may be at a premium. Figure 5(b) shows the increase in bandwidth associated with cache restoration prefetching for the various prefetch limits. The bandwidth overhead is for all read and write traffic to the memory controller, and is normalized to the virtualized system with no optimizations, not the serialized case. We note that there is already a bandwidth overhead associated with running highly partitioned, as the serial case requires up to 60% less memory bandwidth than the virtualized case.

In general the increased performance of allowing more prefetches is balanced by the growing bandwidth overhead. The bandwidth overhead consists of prefetched lines that were not used as well as lines that were unnecessarily castout from the caches. We experience diminishing returns in L3 hit rate and performance at the largest prefetch limits, and therefore see an increasing proportion of unnecessary prefetches and higher memory overhead.

However, we also note that the most constrained case of 2000 prefetches, actually lowered overall memory bandwidth while still providing a system performance boost. These results imply that the technique should be tuned according to the running system to be as aggressive as possible, while making sure not to drive the memory bandwidth to full utilization and increasing latency. We believe prior work in this area would be another means of preventing the prefetcher from overburdening system resources [9].

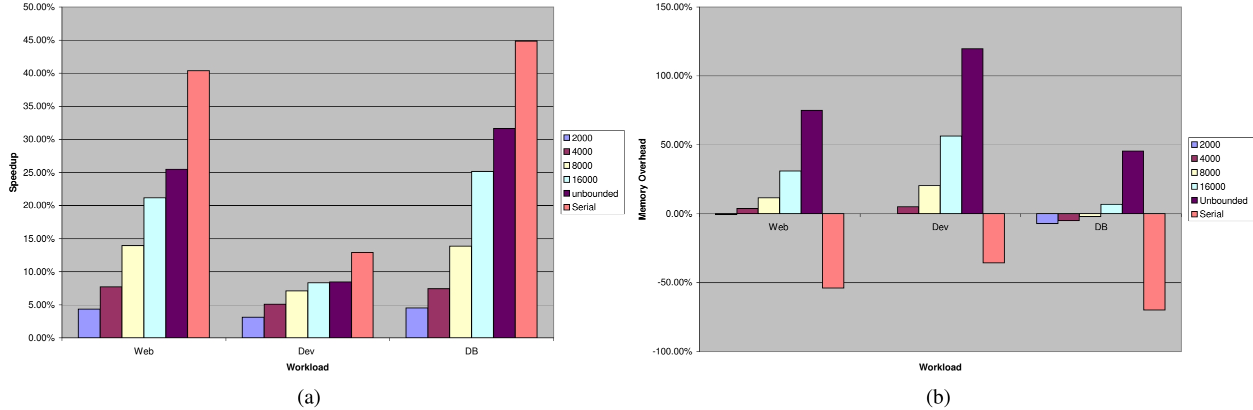


Figure 5. (a) Speedup and (b) Increase in memory bandwidth, while varying he number of prefetches performed per partition switch.

3.1.2 Prefetcher accuracy and coverage

We now present some standard prefetcher metrics for cache restoration prefetching. We measured the accuracy and coverage of the prefetcher, where the accuracy is the percentage of prefetches that were useful (i.e., directly led to a cache hit), and the coverage is the miss rate reduction the prefetcher provides. Figure 6(a) shows the miss rate reduction for the prefetcher with varying number of prefetches, which mirrors the performance data shown in Figure 5(a). As expected the miss rate reduction increases as the number of prefetches goes up, although it does so sub-linearly.

The accuracy results are shown in Figure 6(b). As expected, the prefetch accuracy decreases as the number of allowed prefetches becomes large. Surprisingly, however, the accuracy actually increases a small amount as the number of prefetches is increased from 2000 to 4000 for the Web workload, and increases more dramatically for the DB workload as the number of prefetches per partition is increased from 2000-8000, before then trailing off beyond 8000. We suspect this may be due to the incomplete LRU information available in the PFLA. Recall that the LRU information is maintained among lines within a set, but the ordering of lines between sets is driven by the next partitions memory usage. We explore this idea more in the next section.

3.2. Impact of global LRU information

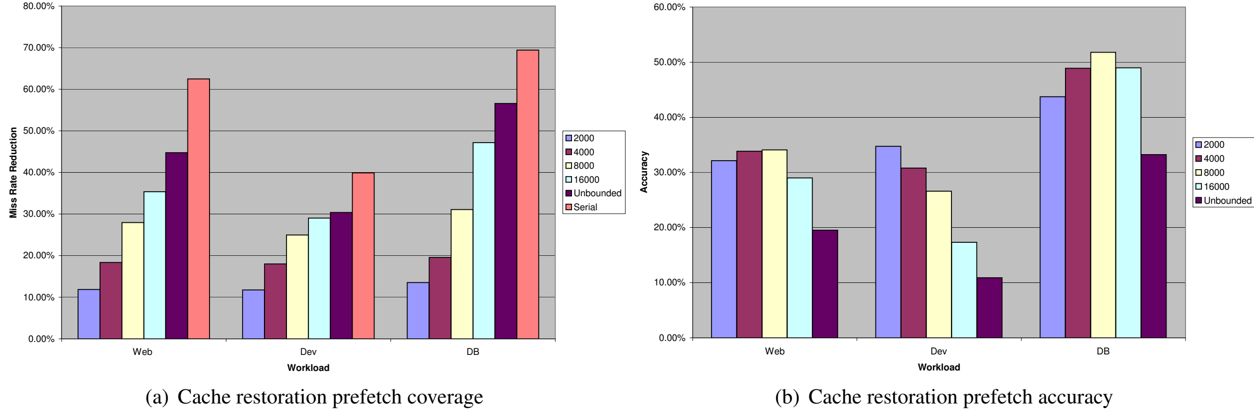
We implemented a more idealistic version of cache restoration prefetching in which the PFLA is ordered based on global LRU information across all sets in order to analyze the performance impact of the partial LRU information collected by our technique.

System speedup results with and without the global LRU are shown in Figure 7(a). The DB workload shows the largest performance improvements from prefetching the saved footprint according to the global LRU, almost achieving the same performance as doubling the number of allowed prefetches. We note performance improvements for all the workloads and all the configurations using the global LRU. Such performance improvements show that some form of cross-set LRU tracking can provide valuable improvements.

Figure 7(b) shows the effect of the global LRU on prefetch accuracy. We note that the prefetch accuracy monotonically decreases as the number of prefetches is increased, which was not the case for cache restoration (shown in Figure 6(b)). Additionally, the global LRU information improves prefetch accuracy for the lower prefetch limits, while having a small effect on the unbounded cases (approximately 1.8% relative change in speedup or 0.34% absolute change in speedup). The unbounded case only benefits from the increased timeliness of prefetches using the global LRU information, demonstrating that the timeliness has a minor impact on accuracy. The bounded cases benefit from increased timeliness, as well as improved selection of which lines to prefetch. This improved accuracy is particularly beneficial for the 2000 and 4000 limited cases, with less pronounced benefits beyond that.

3.3. Cache restoration vs. GHL prefetching

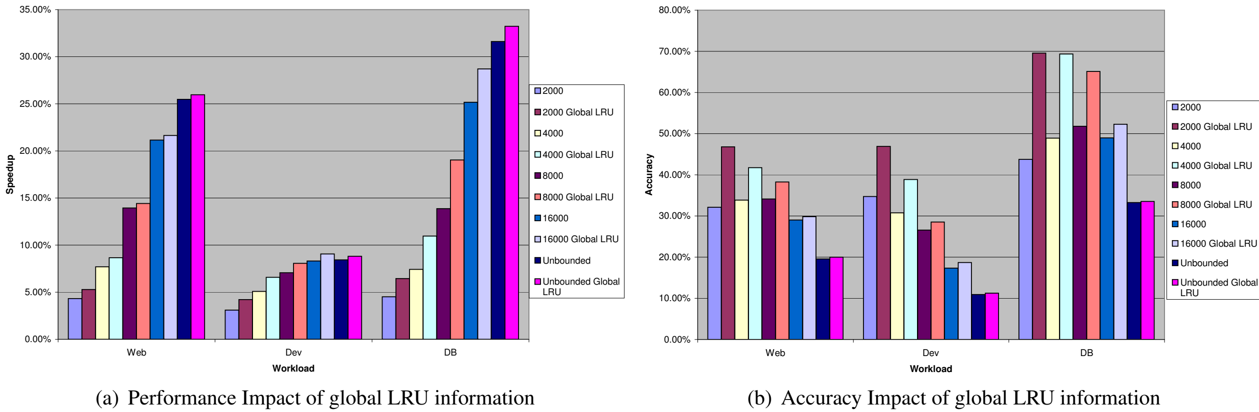
We now provide a comparison of cache restoration prefetching to the only prior work in prefetching across context-switches: Global-history-list prefetching (GHL) [8]. GHL maintains a complete list of cache lines that is precisely ordered by recency of use, maintained using a hardware-implemented doubly-linked list, part of which resides on chip, but which spills into a memory region as lines fall out of the on-chip structure. Duplicate entries are removed from the list by also maintaining pointers with



(a) Cache restoration prefetch coverage

(b) Cache restoration prefetch accuracy

Figure 6. Prefetch coverage and accuracy for cache restoration prefetching.



(a) Performance Impact of global LRU information

(b) Accuracy Impact of global LRU information

Figure 7. Impact on (a) performance and (b) accuracy of prefetching using global LRU information.

each cache line in the L2 to the corresponding entry in the list. On a context switch the processor pauses to write the hardware-contained portion of this list to memory, loads the global history list of the new process, and begins prefetching.

Although GHL was developed to address cache misses due to context switches of processes rather than partitions, the concept can be applied to partitions. GHL's mechanism for saving addresses is different from ours, but the resulting address list is similar, with one significant difference: while our technique preserves LRU information only within a set, their technique saves LRU information across all lines, although it requires significantly more hardware resources to do so. In our evaluation, since the idealized cache restoration prefetcher evaluated in Section 3.2 also uses a globally-ordered LRU list, we use these results to give an upper bound on the performance of GHL, since GHL also preserves LRU information globally (across sets). There are two differences between the GHL prefetching and the cache restoration prefetching with global LRU information that we present, which make the global LRU results more optimistic than GHL: the bandwidth and latency of saving/restoring the on-chip portion of the GHL on context switch are not included in the global LRU results, and the global LRU model includes the preferential eviction of lines from inactive partitions, which GHL does not explicitly support but could support easily enough. Both of these differences are in favor of GHL prefetching.

Based on the global LRU performance data, the largest potential performance gap between GHL and cache restoration occurs for the database workload when prefetching 8000 lines, where despite the additional hardware resources the GHL prefetcher outperforms our technique by at most 4.5% in this extreme case. On average the performance difference is less than 1.5%, and is less than 1% for the unbounded case. While the GHL prefetching mechanism is able to more accurately restore cache contents following a context switch, we consider these differences to be quite small, especially considering that they also come with significant implementation cost and complexity caused by the need to maintain a complete access history list. We now summarize these costs in detail.

Tag Array	14 extra bits per line
Entry List	New 1K x 46b list, multiported
Staging Buffer	2 x 128B single ported
Control Registers	4b active partition + 64b pointer/staging buffer
Cache Logic	Complex logic on each L2 access: add new entry, remove duplicates, updated pointers, and evict to memory if needed
Prefetch Logic	Read from buffer and send address to existing prefetcher
Buffer Logic	Detect buffer empty and read from memory

Table 3. HW overheads associated with GHL

For a comparison of overheads due to storage, we must consider cache impact, GHL storage, and staging buffers. Table 3 lists GHL's hardware costs and can be compared to cache restoration's hardware costs shown in Table 1. We point out the increased tag size, the multiported entry list, and the logic to update that list as requiring significantly more hardware, design, and verification resources than cache restoration. As described in [8], on each L2 access an entry is removed from the free list and inserted at the tail of the address list (requiring at least three writes, two for the insert and one for the removal, in addition to the maintenance of the head/tail pointers). GHL also employs a mechanism for removing duplicates from the list, requiring an additional read and two writes for list removal and two writes in order to add the removed entry to the free list. While the list is being maintained, the tail of the list also needs to be evicted to the off-chip GHL, requiring more updates to the history list and free list. Although a description of the implementation envisioned for the GHL was omitted from that work, one would expect it to either be done with a highly multiported SRAM, or with a finite state machine sequencing through the described steps over multiple cycles, or some combination. Considering that typical L2 caches support multiple concurrent accesses, several of these operations on the GHL and free list will be required in parallel.

4. Related Work

Although there is no prior work on prefetching for multiprogrammed virtualization, there is a long history of work on the performance impact caused by the conventional multiprogramming performed by an operating system.

Stone and Thiebalt first described the cache polluting effects of context switches, and developed an analytical model that could predict their impact on cache performance [17]. Agarwal et al. experimentally measured the performance effects of context switches and operating system interactions, and evaluated the effects of different cache organizations on miss rates [1]. Mogul and Borg also studied the cache effects of context switches on multiprogrammed workloads, finding that the additional overhead of cache displacement accounts for an additional 10 to 400 microseconds [14]. As predicted by these prior studies, per-core cache sizes and memory latencies have grown to the extent that the latency required to restore a core's cache can consume a significant fraction of the the time quantum allotted to a virtual machine or a conventional multiprogrammed thread. For example, in a typical Nehalem EX system, an L3 cache miss requires 79 ns to receive the data from DRAM. With 3MB of L3 cache per core and 64B lines, 49,152 requests must be sent to the memory system to completely restore a core's footprint. Fulfillment of these requests would require 3.8 ms (assuming no memory-level parallelism), or 1.9ms (optimistically assuming a MLP of 2[2]). Given that many operating systems operate using a 10 ms time quantum, this 1.9 to 3.8 ms constitute 20-40% of a time quantum spent mitigating the cold-start phenomenon.

More recently, Koka and Lipasti have shown that for commercial applications that exhibit frequent context switches due to I/O, a significant fraction of data cache misses are due to context switches [12]. Although the concept of restoring cache state across context switches is mentioned in their work, they instead focus on minimizing context switch misses through better scheduling by the OS. Liu et al. further studied context switch misses, highlighting the significance of LRU stack perturbation as a source of evictions that continue to occur even after a swapped-out thread has been rescheduled [13].

Despite this long history of work, the only prior work in prefetching across context-switches was recently proposed by Cui and Sair [8], which was discussed in Section 3.3.

Brown et al. investigate cache working set prediction as an enabler for mechanisms that may need frequent migration, for example in systems exploiting loop or task-level parallelism, as well as speculative multithreading and helper threading [5]. The time intervals explored in their work are significantly shorter (1 instruction to 1M instructions) than those explored in our work, which affects the predictability of the working set over that interval. For these smaller intervals, they find that a bulk copy of cache contents from one cache to another is detrimental to performance (due to competition between demand misses and prefetches for MSHRs and bandwidth), but that a small table recording the MRU memory locations is the most effective subcomponent of their working set predictor. This corroborates our own results as well as those of Cui and Sair [8], in which an MRU-ordered list of cache lines is shown to be an effective source of prefetch addresses. Also, we mitigate the competition between prefetches and demand misses by capping the number of outstanding prefetches (at four) leaving additional MSHRs (eight) for demand misses. (This policy follows the implementation of POWER7 [16], our baseline machine.)

[2]We consider an MLP of 2 optimistic since prior studies have shown a maximum MLP of 1.38 for a set of server workloads [7].

5. Conclusions

This paper introduces the problem of restoring cache contents for newly rescheduled partition, and proposes cache restoration prefetching to address that problem. Cache restoration allows a system to support significantly more virtual machine workloads at a given quality of service than would otherwise be possible. By remembering cache state at the last time a workload is active, and using that history to proactively prefetch when the workload is subsequently rescheduled, cache miss penalties are significantly reduced, resulting in up to 31% performance improvement, on average 20% percent across several workload mixes with 10 partitions per core. It is understood that improvements of this magnitude will not provide the illusion that the individual partitions are being executed on a dedicated core, however they will help keep the negative effects of multiprogrammed virtualization linear with respect to the number of partitions, rather than causing a precipitous degradation.

Cache restoration's performance is comparable (within 1-2% overall performance) to the best known existing technique (GHL prefetching) for the related problem of restoring cache context after context switches, while being considerably simpler to implement than GHL, requiring only minor changes to the existing cache hierarchy.

We have shown that cache restoration prefetching has mostly solved the cache pollution problem, however nontrivial overheads remain due to translation buffer, i-cache, and branch predictor pollution. In future work, we plan to explore cache restoration for these structures as well, in addition to improved mechanisms for reducing prefetch bandwidth without sacrificing coverage.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, 1988.

[2] P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold, H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo: A full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, 2004.

[3] B. Botelho. Virtual machines per server, a viable metric for hardware selection? Web Article. http://itknowledgeexchange.techtarget.com/server-farm/virtual-machines-per-server-a-viable-metric-for-hardware- selection/.

[4] J. S. Bozman and G. P. Chen. Optimizing hardware for x86 server virtualization. IDC White Paper, August 2009.

[5] J. Brown, L. Porter, and D. Tullsen. Fast thread migration via cache working set prediction. In *Proc. of the 17th Symp. on High Performance Computer Architecture*, pages 193–204, February 2011.

[6] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi. Predictor virtualization. In *Proc. of the 13th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems.*, pages 157–167, 2008.

[7] Y. Chou, B. Fahs, and S. G. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proc. of the 31st Intl. Symp. on Computer Architecturew*, pages 76–89, 2004.

[8] H. Cui and S. Sair. Extending data prefetching to cope with context switch misses. In *Proc. of the 2009 IEEE International Conference on Computer Design*, pages 260–267, 2009.

[9] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proc. of the 42nd Intl. Symp. on Microarchitecture*, December 2009.

[10] C. Hales, C. Milsted, O. Stadler, and M. Vagmo. PowerVM virtualization on IBM System p: Introduction and configuration. IBM Redbook, May 2008.

[11] IBM Corporation. *Shrinking 3900 Distributed Servers to 30 Linux Mainframes*, August 2007. Press Release.

[12] P. Koka and M. H. Lipasti. Opportunities for cache friendly process scheduling. In *Proc. of the Workshop on Interaction between Operating Systems and Computer Architecture*, October 2005.

[13] F. Liu, F. Guo, Y. Solihin, S. Kim, and A. Eker. Characterizing and modeling the behavior of context switch misses. In *Proc. of the 17th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 91–101, October 2008.

[14] J. Mogul and A. Borg. The effect of context switches on cache performance. *Proc. of the 4th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[15] B. Reeder. Networking in a virtualized environment. *IBM Systems Magazine*, March/April 2010.

[16] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. IBM POWER7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1–29, 2011.

[17] H. Stone and D. Thibaut. Footprints in the cache. *SIGMETRICS Performance Evaluation Review*, 14(1):4–8, 1986.

[18] VMWare Inc. *Enabling End-to-End Virtualization Solutions for Mid-market and Enterprise Customers: Featured Case Studies*. www.vmware.com/solutions/partners/alliances/hp-vmware-customers.html.

[19] VMWare Inc. *VMWare vSphere 4.0 and Sphere 4.0 Update 1: Configuration Maximums*, 2009. http://www.vmware.com/pdf/vsphere4/r40/vsp_40_config_max.pdf.

[20] T. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Practical off-chip meta-data for temporal memory streaming. In *Proc. of the 15th Intl. Symp. on High-performance Computer Architecture*, pages 79–90, February 2009.