ReCaP: A Region-Based Cure for the Common Cold (Cache)

Jason Zebchuk*, Harold W. Cain†, Xin Tong*, Vijayalakshmi Srinivasan†, and Andreas Moshovos*

*Dept. of Electrical and Computer Engineering
University of Toronto
{zebchuk,xtong,moshovos}@eecg.toronto.edu

†T.J. Watson Research Center
IBM
{tcain,viji}@us.ibm.com

Abstract

Virtualization has become a magic bullet to increase utilization, improve security, lower costs, and reduce management overheads. In many scenarios, the number of virtual machines consolidated onto a single processor has grown even faster than the number of hardware threads. This results in multiprogrammed virtualization where many virtual machines time-share a single processor core. Such fine-grain sharing comes at a cost; each time a virtual machine gets scheduled by the hypervisor, it effectively begins with a "cold" cache, since any cache blocks it accessed in the past have likely been evicted by other virtual machines.

Recently, cache restoration prefetchers *have been shown to reduce the cold cache effects caused by multiprogrammed virtualization. However, these prefetchers waste large amounts of bandwidth prefetching many useless blocks and writing and reading metadata to and from memory. Using block reuse based filtering, we enhance two existing cache restoration prefetchers and reduce their bandwidth overheads by* 16% *and* 32%*, with slowdowns of only* 0.5% *and* 0.6%*.*

We also propose a Region-Based Cache Restoration Prefetcher *(*ReCaP*), which groups cache blocks into coarse-grain regions of memory, and predicts which regions contain useful blocks to prefetch the next time the current virtual machine executes. Based on these predictions, and using a simple compression technique that exploits spatial locality,* ReCaP *provides a robust prefetcher that improves performance by up to* 42% *for some applications and only uses* 14% *more bandwidth and* 3.5% *more power than a stride prefetcher, on average. Compared to other prefetchers designed for multiprogrammed virtualization,* ReCaP *provides comparable performance while using an average of* 12% *to* 27% *less bandwidth, and reducing the energy-delay product of the L2 cache and main memory by* 12% *to* 13% *on average.*

1 Introduction

Server consolidation, virtual desktop infrastructure (VDI) environments, and cloud computing trends dominate the landscape of new server purchases. The growth of these trends has led not only to a much wider adoption of hardware virtualization, but also to an increasing number of virtual instances, or *partitions*, being consolidated onto each physical system. For example, IBM has reported a case study of consolidating 3,900 servers onto only 30 mainframe systems [13] and a number of VMWare case studies have reported consolidation ratios from 4:1 to 15:1 [24]. As another example, Botelho has suggested that in VDI environments a good rule of thumb is to combine six to eight virtual desktop instances per processor core [2]. In the future, the number of partitions on each machine is expected to continue to increase. Botelho quotes Anne Skamarock, a research director at Focus Consulting, as saying that customers, "are running 30 VM's per 8-core system and expect to increase that to 50 VMs per system" [2]. Even with the increasing number of hardware threads present in modern processors, such large consolidation ratios result in each partition being assigned only a fraction of a real hardware execution context.

Consolidating this many partitions onto a single system generally requires some form of *multiprogrammed virtualization* in which multiple partitions time-share a single hardware thread. To meet QoS constraints and provide real-time interactive response times, the execution interval for each partition is kept relatively short. For instance, the PowerVM partition manager will allocate some portion of a 10 ms dispatch window to each active partition, such that a system with five partitions, for example, might execute each one for only 2 ms at a time within the window.

978-1-4673-5587-2/13/$31.00 ©2013 IEEE

VMWare case studies describe a number of scenarios that require short response times for VDI environments used with “mission critical” applications, e.g., in urgent care departments in hostpitals [24].

Multiprogrammed virtualization incurs overheads each time the hypervisor switches partitions, with much of this slowdown coming from the loss of microarchitectural state in the processor. While a partition is switched out, other partitions pollute the processor’s caches, branch predictors, and TLBs. By the time the first partition is subsequently scheduled for its next execution interval, the intervening partitions might have evicted all of its state, resulting in an almost perpetual cold cache. While these effects could be amortized by executing each partition for longer periods of time, the need to maintain fast response times limits the applicability of this solution. In this paper, we explore alternative hardware mechanisms to reduce the performance impact caused by the common cold cache problem created by multiprogrammed virtualization.

Daly and Cain recently explored the effects of multiprogrammed virtualization and found that it can reduce performance by as much as 40% compared to running applications in isolation [9], and our results show slowdowns of up to 35% for some applications from SPEC CPU 2000. To reduce this performance penalty, Daly and Cain proposed the CRP prefetcher and Cui and Sair proposed the GHL prefetcher [8]. However, both of these prefetchers have significant bandwidth overheads for wasted prefetches and for prefetcher metadata. We propose two modified prefetchers, SCAN-HITS and GHL-HITS that use block reuse based filtering to signficantly reduce the bandwidth overheads without significantly reducing the performance benefits.

Additionally, we propose RECAP, a *Region-based Cache Restoration Prefetcher*, that exploits coarse-grain patterns in an application’s memory access stream to restore the cache working set after a partition switch with little bandwidth overhead. RECAP uses a dual-grain approach, tracking reuse of individual blocks within 4 KB regions. This approach allows RECAP to effectively reduce the number of useless prefetches, and also to efficiently compress the list of block addresses to prefetch. RECAP can track reuse either for an individual cache block or for an entire region of memory. This reuse information is then used to predict which blocks are most likely to experience additional reuse after the next partition switch and are thus most beneficial to prefetch.

Overall, RECAP achieves speedups of up to 42% for some applications, with a mean speedup of 4.5% over a stride prefetcher. The performance benefits out-weigh the bandwidth and power overheads, resulting in an average 5% reduction in the energy-delay product of the L2 cache and main memory compared to a stride prefetcher. Compared to previously proposed prefetchers that target the same cache pollution problem, RECAP achieves the same or better performance, while lowering the energy-delay product by 12% to 13% and reducing bandwidth by 12% to 27% less bandwidth on average.

In summary, this paper makes the following contributions:

- It demonstrates that only prefetching reused cache blocks signficiantly reduces the bandwidth overhead of cache restoration prefetchers while only slightly reducing their overall benefit.
- It shows that tracking a relatively small number of coarse-grain regions can capture most of an application’s cache working set.
- It demonstrates that tracking block reuse within regions can effectively predict which blocks are most likely to be useful to prefetch.
- It proposes RECAP, a bandwidth-efficient region-based cache restoration prefetcher that provides better overall performance improvement than previous solutions with less power and bandwidth overhead.

Section 2 describes the slowdowns caused by multiprogrammed virtualization and the bandwidth overheads that can result from naïve prefetching approaches. In Section 3 we explain our experimental methodology and how we model multiprogrammed virtualization. Section 4 describes two state-of-the-art prefetchers designed to address problems with frequently switching partitions, and illustrates their high bandwidth overheads. In Section 5, we demonstrate an elegant optimization that reduces bandwidth overhead for both previously proposed prefetchers without significantly sacrificing performance. Section 6 describes our new RECAP design and illustrates its benefits over other prefetchers. The storage overheads of the various prefetchers are compared in Section 7, and Section 8 describes some additional related work. Finally, Section 9 summarizes the benefits of RECAP and details some possible future work.

2 Motivation

The cache pollution effects of multiprogramming increase with deep cache hierarchies and larger cache sizes [1]. For any cache, the maximum number of additional misses that result from cache pollution is limited by the cache capacity. For typical L1 caches, the performance loss of 512 to 2048 additional cache misses is small even with a relatively short interval of a few million cycles. However, this section shows that performance losses can be significant for typical higher level caches with 32K cache blocks or more.

We constructed three workloads, each consisting of eight partitions running different SPEC CPU 2000 applications.

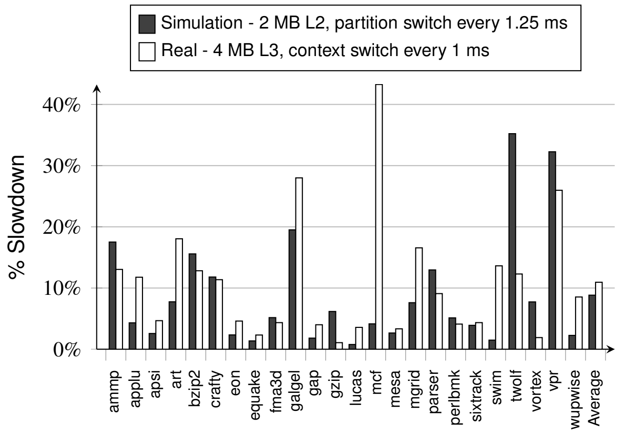

Figure 1. Percentage slowdown for Spec CPU 2000 applications when run as part of multi-programmed workload with 8 partitions compared to running in isolation.

We ran these workloads both on a real system and on a detailed timing simulator to estimate the slowdown that results from frequent partition switches. The simulated system has a 2 MB L2 cache and models separate virtual machines each allocated an equal portion of a 10 ms dispatch window. The real system is a dual-core, 2.4 GHz Intel® Xeon® CPU with a 4 MB L3 cache and with the hardware prefetcher disabled. All applications were run in a single Linux OS instance as real-time processes pinned to the same core. We modified the Linux real-time scheduler to perform context switches every 1 ms, the closest interval possible to the 1.25 ms used in simulation.

Figure 1 shows the slowdown of each application in the virtualized workloads compared to the same application running in isolation for both the simulated and real systems. On average, the applications experienced a slowdown of 9% in the simulated environment and 11% on the real computer. Some applications (galgel, mcf, twolf and vpr) incurred over 30% slowdowns as a result of cache pollution and other effects of virtualization.

We expected that there would be differences in the measured slowdown between the simulated and actual systems, more so given that the core and memory hierarchies are different. For most applications the results from simulation and the real system are surprisingly close, with simulation often underestimating the impact of virtualization. The results also show that cache pollution remains a problem even when the last level cache is 4 MB. The rest of this work focuses on the simulated system with a 2 MB L2 cache per core which is representative of server class machines.

2.1 Overheads of Naïve Cache Restoration

An obvious way to reduce the effects of cache pollution for these workloads might be to try to "preload" the cache when the active partition switches. Just as the hypervisor saves and restores all architecturally visible state, such as the registers, the hypervisor or a hardware assist could save and restore the L2 cache contents. A list of blocks that ought to be restored will have to be saved as well.

Preloading the entire cache wastes off-chip bandwidth in two ways: (1) by prefetching data blocks that will not be used before evicted, and (2) by writing and reading the list of cache blocks to prefetch. Past work has shown that under normal operation, a block is "dead" for an average of 86% of the time it resides in the cache [14]. The frequent partition switches of multiprogrammed virtualization exacerbate this problem by evicting cache blocks that might otherwise simply have long reuse intervals.

Figure 2 shows bandwidth overheads with naïve cache restoration. For this experiment we flush the cache on every partition switch. Bandwidth is normalized with respect to that required when running the application in isolation (lower ■ bar). The bandwidth overhead includes preloading blocks that were evicted due to multiprogramming and later accessed again (■ bar), and wasted bandwidth due to prefetching "dead" blocks (□ bar).

On average, multi-programming alone increases bandwidth by 46%, but in the worst case it imposes more than a 30× overhead. The naïve prefetcher further imposes an average 51% overhead for prefetching useless blocks and in the worst case, for vpr, the bandwidth wasted on over-prefetching is more than 9× the bandwidth required when running the application in isolation.

In addition to bandwidth used for prefetching blocks, any prefetcher that attempts to preload the cache needs to use some bandwidth to write a list of block addresses to main memory so they can later be read back and the appropriate prefetches can be issued. For our 2 MB L2 cache doing so would require 15.6% additional bandwidth assuming 46 bit physical addresses, and 64 byte cache blocks. Figure 2 illustrates how signficiant this metadata overhead is (▨ bar). For the worst case of vpr, the metadata bandwidth is more than 6× the bandwidth of the standalone application.

In summary, multiprogrammed virtualization causes cache polution that reduces performance by 9% on average and more than 30% for several applications. Moreover, naïve cache restoration increases bandwidth by 51% on average and by more than 100% for several applications.

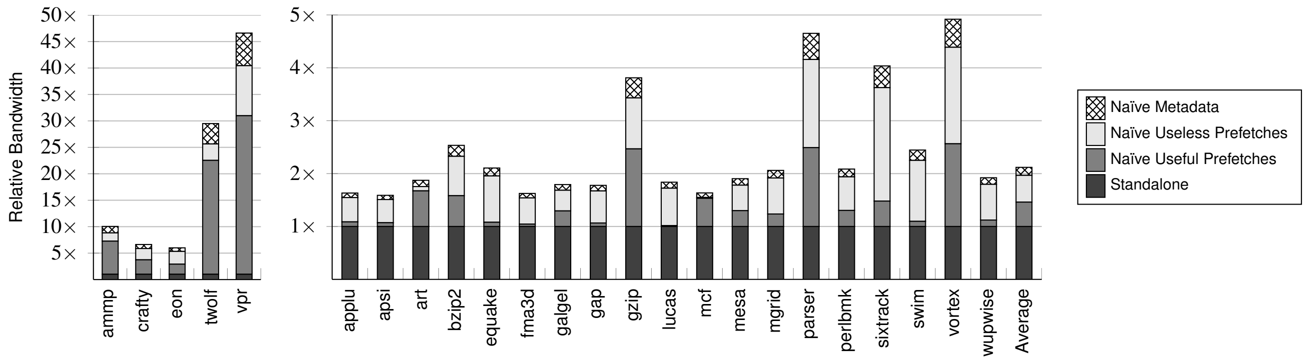

Figure 2. Bandwidth overhead from multiprogrammed virtualization and naïve prefetching normalized to standalone bandwidth.

Table 1. Workload Descriptions

High-MPKI	applu, art, equake, galgel, lucas, mcf, mgrid, swim
Mid-MPKI	ammp, apsi, bzip2, gap, parser, twolf, vpr, wupwise
Low-MPKI	crafty, eon, fma3d, gzip, mesa, perlbmk, sixtrack, vortex

Table 2. Processor core configuration

Processor Core	4 GHz UltraSPARC III ISA 8-stage, out-of-order 256-entry ROB, 64-entry LSQ decode/issue/commit any 4 instrs/cycle
Branch Predictor	8K GShare, 16K bi-modal, and 16K selector 2K entry, 16-way BTB, 2 branches/cycle
Fetch Unit	Up to 8 instrs/cycle, 32-entry fetch buffer
L1D/L1I	64 KB, 64 B blocks, 4-way, 1 cycle
Unified L2	2 MB, 16-way, inclusive of L1 3/12-cycle tag/data latency
Additional Prefetcher	various designs, max 4 outstanding, 32-entry write-back buffer

3 Methodology

In the following sections, all simulations modelled a single-core, single-thread processor with the configuration described in Table 2. The simulator is based on the full-system simulator *Flexus* [12], which uses WindRiver Simics [10], and includes a memory timing model based on DRAMSim2 [20]. All simulated systems ran the Solaris 8 operating system.

Each virtualized workload consists of eight applications chosen from the SPEC CPU 2000 benchmark suite, as indicated in Table 1. We expect that cache miss rates will have a large impact on the effectiveness of any cache restoration prefetcher, so we have grouped the applications based on the MPKI of the L2 caches for each application running in isolation.

Switching between "virtual machines" is modelled by stoping the Simics simulation of one application and starting a new simulation for a different application. To avoid overlapping address spaces for different applications, we modify all physical addresses by mapping the virtual machines to distinct portions of a 64 GB address space.

Each individual application is run for 5 million cycles before switching to a new application. With a 4 GHz clock, this represents eight partitions given equal share of a 10 ms scheduling interval [11]. Simulations were run for a total of four billion cycles of execution (500 million cycles per application). Functional simulations of an additional 2 billion instructions (250 million cycles per application) were used to warm the branch predictor and cache state before starting the timing simulations.

When measuring performance, we report the "Fair Speedup" metric [6], calculated as the harmonic mean speedup of the eight applications in each workload. For bandwidth, we report the average of the normalized bandwidth per instruction. All results are relative to the baseline system running the same virtualized workload with no hardware prefetcher.

The simulations used to measure the slowdown shown earlier in Figure 1 ran each application in isolation for 500 million cycles after 250 million cycles of functional warmup (the same total execution as one application in the virtualized workloads). The simulated system included a twelve-entry stride prefetcher for both the virtualized workloads and the applications run in isolation.

3.1 Prefetcher Configurations

We model a stride prefetcher and a number of new and existing cache restoration prefetchers described in the following sections. Each prefetcher was configured to issue prefetches to the L2 cache, and the number of outstanding prefetches was limited to four to avoid excessive bandwidth contention with demand accesses. All prefetchers used a similar 32-entry write-back buffer when writing data to memory. Except for the baseline, all configurations include the same twelve-entry stride prefetcher located between the L1 and L2 caches. The following sections briefly describe the configuration of each prefetcher.

4 Previously Proposed Prefetchers

Two previous works have proposed prefetchers that address the cache pollution problem caused by multiprogrammed virtualization. Daly and Cain recently proposed a *cache restoration prefetcher* (CRP[1]) [9], and Cui and Sair have proposed a *Global History List Prefetcher* (GHL) [8]. Both prefetchers attempt to restore the entire cache contents after a partition switch with little concern for the usefulness of each prefetch.

4.1 Cache Restoration Prefetcher

CRP attempts to restore the cache contents across context switches by recording addresses as cache blocks are evicted. Then, on a partition switch, CRP prefetches all the blocks belonging to the incoming partition that have been evicted since that last time it ran. The hypervisor reserves a portion of physical memory to store a list of evicted blocks for each logical partition. To associate evicted blocks with the partition that allocated them CRP stores a *logical partition id* (LPID) with each cache block. CRP also maintains a hardware table with the mappings from LPID to the location of each prefetch list.

For a small number of partitions, or when most of the partitions' working sets can reside in the cache simultaneously, CRP can use a small amount of bandwidth to prefetch the few cache blocks that have been evicted. However, when many partitions share the same processor, all of a partition's cache blocks will most likely be evicted between consecutive execution intervals. This is what we observe in our experiments where most partitions do not have any blocks that survive between subsequent execution intervals, and no partitions see more than 1.5% of the cache blocks survive. In this scenario, CRP is comparable to scanning the cache contents after a partition switch and prefetching all cache blocks. Given this result, we use a SCAN prefetcher to demonstrate the behavior of a CRP-like prefetcher.

The SCAN prefetcher scans the entire cache and behaves similarly to the naïve prefetcher described in Section 2.1. As Figure 2 showed, this approach wastes significant bandwidth with useless prefetches. To reduce the amount of bandwidth wasted on useless and un-timely prefetches, Daly and Cain recommend only prefetching at most half of the cache contents. Following this recommendation, our simulations model a SCAN-8 prefetcher which scans eight of the sixteen cache ways in MRU order. We explored other options and found this design offered the best overall average performance.

[1]The CRP acronym is used here for convenience, it was not proposed by Daly and Cain.

4.2 Global History List Prefetcher

GHL was proposed to improve performance when multiprogramming creates frequent context switches. Conceptually, GHL maintains a large LRU list of all blocks present in the cache. On a partition switch the LRU list is written to memory, and the next time the partition executes, block addresses are read from the list in memory and blocks are prefetched starting with the most recently accessed blocks.

For large last level caches, a single LRU list would impose a large hardware overhead. To avoid this, GHL instead uses a smaller hardware LRU supplemented with a large in-memory FIFO. The hardware LRU is implemented as a double-linked list stored in SRAM. Block addresses are written to the in-memory FIFO once they are evicted from the LRU list. GHL also adds a pointer to each block in the cache to identify its location in the LRU list.

In theory, the global history list improves the timelines of prefetches by prioritizing the most recently accessed blocks. However, separating the list into a small on-chip LRU and an in-memory FIFO creates many duplicate entries in the in-memory FIFO. For our workloads, on average approximately half of the addresses stored by GHL were redundant.

In this work, we scaled the GHL structures in proportion to the size of the L2 cache size. All simulations use a 2K entry on-chip LRU structure and a 30K in-memory FIFO.

4.3 Performance and Bandwidth

Figure 3 compares the speedup and bandwidth for the multiprogrammed workloads when using just a stride prefetcher, or a stride prefetcher combined with either a GHL or SCAN-8 prefetcher. A breakdown of the amount of memory bandwidth per instruction is shown for demand misses (■), useful prefetches (▩), wasted prefetches (▨), and prefetcher metadata (□). The bandwidth is measured relative to the demand misses for the baseline system with no prefetcher. The figure also shows speedup over the baseline. Section 3 provides additional details of the simulated processor, the workloads, and our evaluation methodology.

GHL and SCAN-8 both increase bandwidth by over 50%. While this may be less than the naïve approach it is still significant. In the worst case, for the Low-MPKI workload, SCAN-8 increases the bandwidth by 2.5× over the baseline system while achieving only a 3.5% speedup.

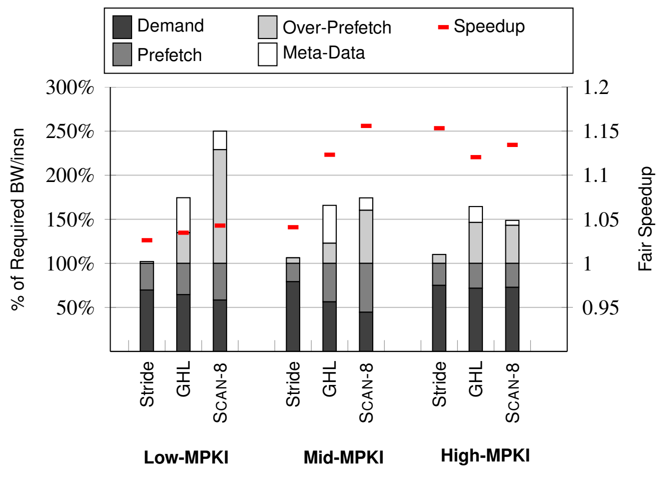

Figure 3. Breakdown of per-instruction memory bandwidth normalized with respect to demand misses and useful prefetches.

For GHL, meta-data increases the bandwidth per instruction by an average of 34%. This overhead is a result of continual writes to memory as blocks are evicted from the on-chip hardware LRU structure and added to the in-memory FIFO.

SCAN-8 has a lower meta-data overhead since it only writes the prefetch list to memory once for each context switch. However, SCAN-8 incurs a larger overhead for prefetching more useless cache blocks compared to GHL. SCAN-8 prefetches half of each cache set (see Section 3.1), so it consistently attempts to prefetch 1 MB of data for workloads with sufficiently large working sets. GHL tracks up to 32K cache blocks containing 2 MB of data, but the majority of the block addresses are stored in an in-memory FIFO that can contain many duplicate entries. As a result, we found that for our workloads, GHL attempted to prefetch fewer unique cache blocks on average than SCAN-8, even though it could prefetch up to twice as much data in an ideal scenario. Unfortunately, much of the extra data prefetched by SCAN-8 was never used and resulted in an average 77% bandwidth overhead per instruction. These results demonstrate the importance of new prefetching techniques that reduce overhead by prefetching fewer useless blocks and by using less bandwidth to read and write prefetcher meta-data.

5 Eliminating Useless Prefetches

A significant portion of the overhead of cache restoration prefetchers comes from prefetching useless blocks. A filtering mechanism that predicts whether a block will be useful could reduce this overhead. While an ideal predictor would filter all useless blocks and only useless blocks, any realistic predictor will filter some useful blocks and miss some useless blocks. Thus, a good predictor needs to balance among these two cases, be simple to implement, and have low cost. This section proposes such a predictor.

To identify blocks that are likely to be useful to prefetch, we use the intuition that history tends to repeat itself. Specifically, we observe that blocks in the cache that have experienced cache hits are likely to see additional reuse in the future, while blocks that have never seen any cache hits are less likely to experience future reuse. Although this may not be true of all blocks, it can serve as a first-order filter to predict which blocks are unlikely to be useful if they are prefetched after a partition switch.

To illustrate the validity of this intuition, Figure 4 depicts the contents of the cache right before a partition switch, with each block being classified based on two criteria. First, blocks are classed as being either *Useful* or *Useless* depending on whether or not they would experience a cache hit if they were still present in the cache the next time the same partition executes, i.e., it would be useful to prefetch these blocks. Second, blocks are classified based on whether they have experienced any reuse while in the cache. Blocks which have experienced a cache hit since the last partition switch are indicated as having *Block Reuse*, and blocks that have not experienced any cache hits in the last-level cache have *No Reuse*.

The top (▨) and bottom (■) portions of each bar validate our intuition. On average blocks that experience reuse are more likely to be useful prefetches (■) rather than useless prefetches (▨). Of course, not all applications follow this trend, for instance most blocks with block reuse for swim would be useless prefetches. However, on average 73% of blocks with cache reuse would be useful to prefetch.

Based on this intuition and the trends shown in Figure 4, we propose two new cache restoration prefetchers:

GHL-HITS This prefetcher works the same as the original GHL prefetcher proposed by Cui and Sair [8], except that block address are only recorded in the history list when a cache hit occurs. Similarly, the LRU order is only changed on cache hits. Cache misses have no effect on the history list.

SCAN-8-HITS This prefetcher scans the cache after each partition switch and records all of the blocks in the cache that have experienced cache hits. Block addresses are written to a prefetch list stored in a reserved portion of main memory. The next time that partition executes, the prefetch list is read from memory and the blocks are prefetched into the cache. Each block in the cache requires an additional *reuse* bit to indicate which blocks have experienced cache hits, and a *partition id* field to identify which partition brought the block into the cache. This simplifies the process of scanning the cache for outgoing blocks

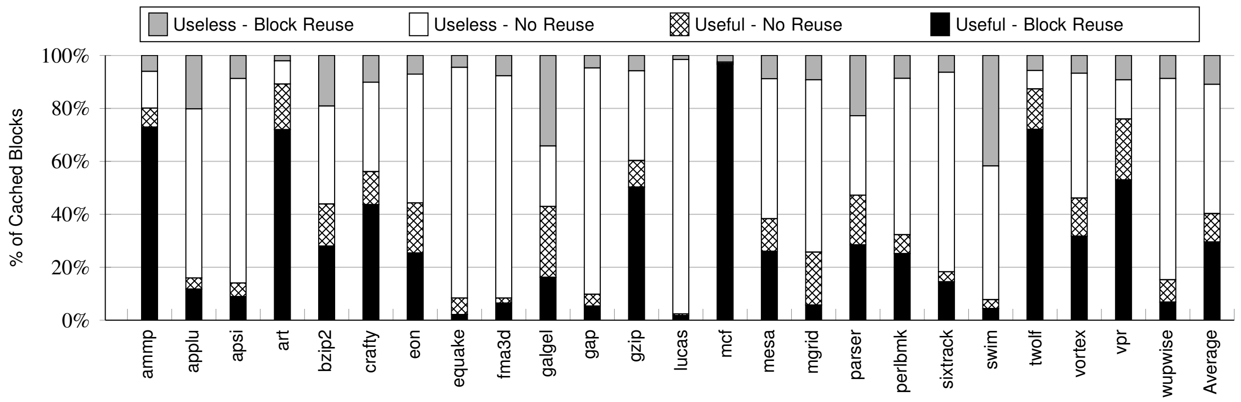

Figure 4. Breakdown of blocks stored in the cache at the end of a partition switch.

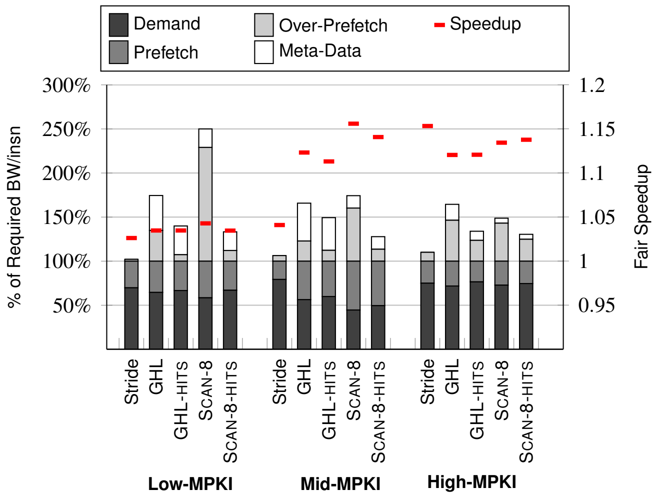

Figure 5. Fair speedup and breakdown of per-instruction memory bandwidth.

while simultaneously executing and prefetching blocks for the new partition.

5.1 Reuse Filtering Benefits

Figure 5 shows that reuse-based filtering significantly reduces the bandwidth overhead of the GHL and SCAN-8 prefetchers with only a small reduction in their benefits. Switching from GHL to GHL-HITS reduces the bandwidth overhead from 68% to 41% while only reducing the mean speedup from 9.1% to 8.8%. Similarly, switching from SCAN-8 to SCAN-8-HITS reduces the bandwidth overhead from 91% to 31% while only reducing the mean speedup from 10.9% to 10.2%.

In absolute terms, the savings when switching from SCAN-8 to SCAN-8-HITS are as much as 1 GB/s for some applications, and 0.43 GB/s on average. For GHL, the savings of switching to GHL-HITS are as high as 1.16 GB/s for some applications, with an average of 0.30 GB/s in bandwidth savings across all applications. Note that for the high-MPKI workload all cache restoration prefetchers perform worse than the simple stride prefetcher which uses less bandwidth and offers better performance.

Reuse-based filtering significantly reduces the amount of bandwidth wasted on useless prefetches. However, it does not reduce the amount of bandwidth required for reading and writing the prefetcher meta-data. As the number of useless prefetches is reduced, the meta-data becomes the dominant source of bandwidth overhead. Assuming 46-bit physical addresses and 64-byte cache blocks, every prefetch incurs an overhead of 10 bytes of bandwidth to first write the block address to memory and then read it back again before it can be prefetched.

6 RECAP: Region-Based Cache Restoration Prefetcher

To reduce the harmful effects of cache pollution caused by multiprogrammed virtualization, we propose RECAP, a region-based cache restoration prefetcher. RECAP attempts to conserve bandwidth through two key mechanisms. First, it uses a space-efficient representation of block addresses when writing and reading the list of blocks to prefetch for a partition. Second, it predicts which blocks are most likely to be useful after a partition switch and only prefetches those blocks. By conserving bandwidth, RECAP minimizes the power consumption and contention for off-chip memory.

RECAP uses a simple, space-efficient representation of the blocks to prefetch that reduces the metadata overhead by 7× on average compared to the previous approaches. RECAP groups data blocks into contiguous 4 KB regions of memory[2] and uses bit-vectors to identify individual blocks within a given region. Assuming 64-byte cache blocks and 46-bit physical addresses, in the best-case scenario up to

[2]We use 4 KB regions because it is a common page size; we did not explore other region sizes for this work.

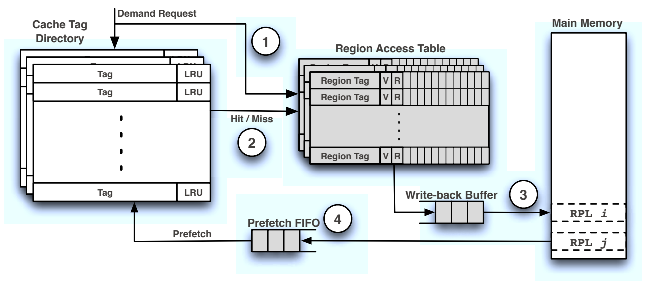

Figure 6. Block diagram of RECAP. New structures required for RECAP are highlighted in grey.

64 cache blocks can be identified using a 34-bit *region tag* and a 64-bit vector for an overhead of 1.5 bits per block. In practice, for the workloads studied the regions present in the cache at the time of a partition switch contained an average of 17 blocks each. This results in an overhead of 5.6 bits per block, seven times less than the 40 bits required for simply listing the block addresses. Thus, RECAP's approach significantly reduces the bandwidth to read or write a list of blocks to prefetch, and it also allows a relatively small hardware structure to track a large number of cache blocks to consider for potential prefetching.

The region-based design of RECAP enables two types of reuse-based prefetch filtering. Similar to the GHL-HITS and SCAN-HITS prefetchers proposed in Section 5, RECAP can restrict prefetches to blocks that experience block reuse by only track cache hits. In addition, RECAP can use a more conservative filtering approach by prefetch blocks based on *region reuse* which uses the intuition that nearby blocks likely experience similar reuse patterns. By prefetching all blocks that have been accessed from any region where at least one block has experienced reuse, RECAP can prefetch some additional blocks that would not have been prefetched using strict block reuse filtering.

The remainder of this section overviews the RECAP design (Section 6.1), describes its key structures (Section 6.2), details its operation (Section 6.3), and compares it against other prefetchers (Section 6.4).

6.1 High-level Design of RECAP

RECAP exploits common behavior within coarse-grain regions of memory to provide a space-efficient structure that tracks a sub-set of memory blocks that are very likely to be accessed after a partition switch. The block diagram in Figure 6 illustrates how RECAP performs the following key tasks:

1. Track accesses to individual blocks using the Region Access Table.
2. Track either region reuse or individual block reuse.
3. Save selected regions in a Region Prefetch List in memory when a partition is switched out.
4. Read the region prefetch list from memory, and issue prefetches when a partition is rescheduled.

By combining coarse-grain and fine-grain information, RECAP accomplishes these tasks with low memory and bandwidth overhead and small hardware structures. With the same basic design, RECAP can be adapted to prefetch either all blocks, only blocks with block reuse, or only blocks with region reuse.

6.2 RECAP Organization

As shown in Figure 6, RECAP consists of four key components: a *region access table (RAT)*, a set of *region prefetch lists (RPLs)*, a small write-back buffer, and a small prefetch FIFO.

RECAP tracks regions using the *region access table* (RAT). Each entry in this set-associative lookup structure contains a region tag, a valid bit (V), an optional reuse bit (R), and a presence vector containing one bit for each cache block in a region. All designs explored in this work use 64 byte cache blocks and 4 KB regions, resulting in 64-bit presence vectors. The size of the RAT is determined in proportion to the size of the cache and the average amount of spatial locality. Our design space explorations have shown that a reasonable RAT contains two region entries for every 4 KB of cache capacity. For a 2 MB cache, this results in a 1024 entry RAT, which requires roughly 12 KB of on-chip storage. This work uses an 8-way set-associative RAT.

RECAP uses a set of *region prefetch lists* (RPLs) stored in main memory. Each RPL contains a list of region tag and presence vector pairs similar to the RAT entries. The hypervisor, or partition manager, is responsible for allocating RPLs in a reserved portion of main memory, and for assigning one RPL for every partition that will use the RECAP mechanism. The hypervisor should allocate enough memory for each RPL so it can contain as many entries as the RAT.

Finally, RECAP uses two small on-chip buffers. The write-back buffer and prefetch FIFO both contain 64-byte buffers used to pack and un-pack 98-bit RPL entries as they are written to and read from main memory. This paper conservatively used a 32 entry write-back buffer and a four entry prefetch FIFO, which require a total of 2.25 KB of on-chip storage.

6.3 RECAP Operation

The hypervisor, or partition manager, software is responsible for initiating the operation of RECAP. When RECAP is first activated, it resets all valid bits, reuse bits, and presence vectors in the RAT.

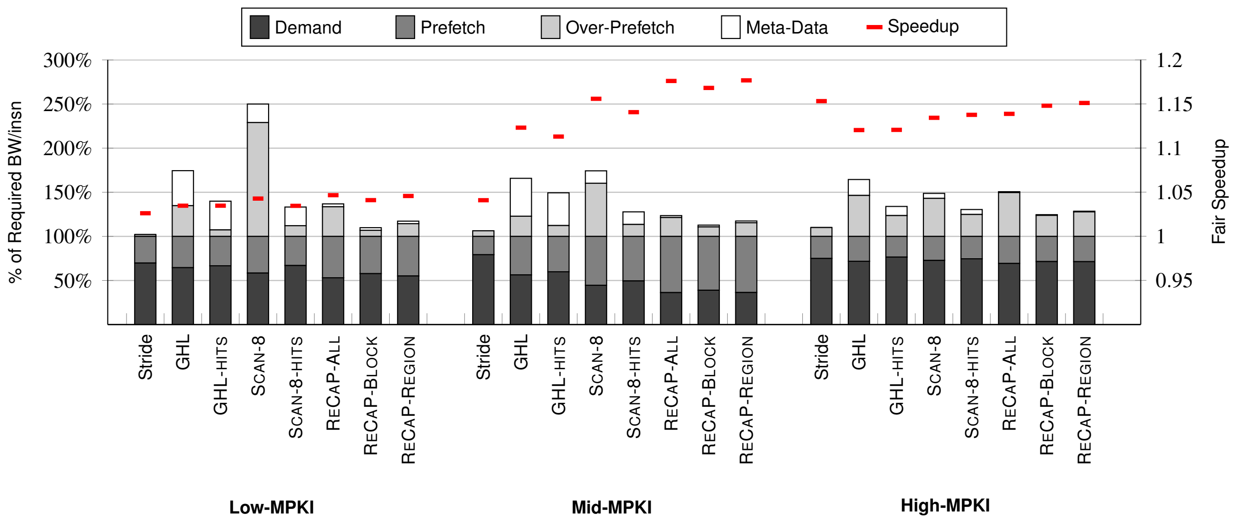

Figure 7. Fair speedup and breakdown of per-instruction memory bandwidth.

Each cache access searches the RAT for an entry with a matching region tag. If none is found, then the least recently used RAT entry (or an invalid entry, if one exists) is allocated to track the encompassing region. Once an entry has been selected, the corresponding presence vector bit is set. If the access results in a cache hit, then the RAT entry's reuse bit is set. This procedure allows RECAP to prefetch all blocks that have region reuse. To prefetch all blocks, the same procedure is followed except no reuse bit is required. To only prefetch blocks with block reuse, the reuse bit is not required, and the process of accessing the RAT and setting the presence bit is only performed for cache hits.

This process repeats for every demand request received by the cache until the hypervisor initiates a partition switch. On a partition switch, the hypervisor notifies RECAP and provides the address of the RPLs for the *outgoing* and *incoming* partitions, if they exist.

Once the hypervisor has finished these partition switch operations, normal execution begins for the newly active partition. In parallel with the execution of the new partition, RECAP begins reading RPL entries in 64-byte blocks into the prefetch FIFO. RECAP issues prefetches for all blocks marked *present* in each RPL entry. As the blocks are prefetched, RPL entries are removed from the FIFO and new RPL entries are read from memory as space becomes available. This process continues until all RPL entries have been read and all prefetches have been issued. To avoid excessive bandwidth use, RECAP limits the number of outstanding prefetches.

While RECAP is issuing prefetches from the prefetch FIFO, it simultaneously scans the RAT and writes the contents to the RPL for the outgoing partition. RECAP scans every entry in the RAT, and copies valid entries to the write-back buffer. To prefetch only blocks with region reuse, only RAT entries with their reuse bits set are copied, otherwise all RAT entries are copied. The write-back buffer collects entries into 64-byte blocks, and writes them back to memory. To reduce memory contention, write-back requests have a lower priority than demand requests.

Normally, any L2 cache accesses from the new partition would update the RAT, causing contention with the process of scanning and writing back the RAT entries. To avoid this contention, RECAP discards any updates while it scans the RAT. Once the scan completes, regular RAT operation continues as described above. For the workloads studied here, the scan process requires around 20K cycles on average and less than 1% of RAT updates are discarded, and no performance is lost as a result.

6.4 RECAP Performance and Bandwidth

We evaluate three RECAP designs: RECAP-ALL tracks all block accesses, and RECAP-BLOCK and RECAP-REGION implement block and region reuse filtering resepectively. All three designs use a 1K-entry 8-way set-associative RAT.

All three RECAP techniques out-perform the previously proposed prefetchers and their reuse filtering enhanced versions proposed in Section 5. This performance advantage is in part the result of eliminating the metadata overhead — all RECAP designs incur less than 2% metadata overhead on average. RECAP-REGION offers the best bandwidth-performance tradeoff as it provides the best performance overall and uses only slightly more bandwidth than RECAP-BLOCK. RECAP-REGION captures all of the blocks with block reuse, and the additional blocks with region reuse provide a small performance improvement and only a small additional bandwidth overhead.

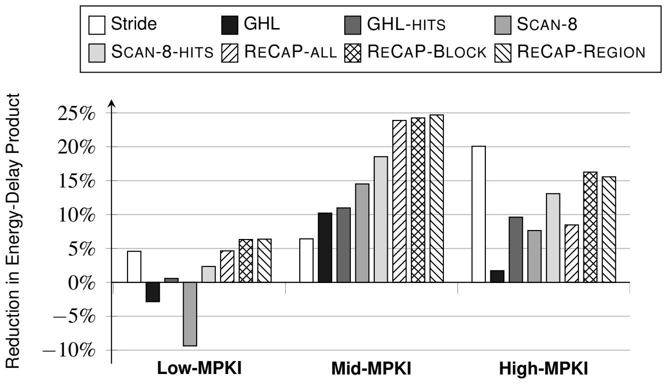

Figure 8. Reduction in energy-delay product for the L2 cache and memory.

For the High-MPKI workloads, high bandwidth leads to contention for the memory system and limits the potential performance improvements. For this workload, previously proposed techniques are not robust, incurring 1.3% to 2.7% performance losses compared to the stride prefetcher, even when enhanced with reuse-based filtering. However, RECAP-REGION is more robust and does not suffer from this behavior. RECAP-REGION offers the best performance for the Low-MPKI and Mid-MPKI workloads and is within 0.2% of the best performance for the High-MPKI workload.

6.5 RECAP Power Savings

The bandwidth savings from reuse-based filtering and the RECAP design result in power savings in the L2 and memory as illustrated by Figure 8. The L2 cache and prefetcher power were estimated using CACTI 6.5 and we used DRAMSim2 to estimate memory power.

Figure 8 shows the reduction in energy-delay product (EDP) of the L2 cache and memory for all prefetchers compared to the baseline system. Again, RECAP-REGION and RECAP-BLOCK are the best designs, with both reducing EDP by 15%. Past works are not robust and increase the EDP for the Low-MPKI workload. Our enhanced versions of these prefetchers do not suffer from this behavior, but RECAP-REGION and RECAP-BLOCK reduce EDP even further. Given the speedups shown in Figure 7, we expect even larger reductions in EDP for the system as a whole.

7 Storage Overheads

Table 3 describes the hardware overheads of the three prefetcher configurations compared to the baseline L2 cache with five state bits per cache line. We assume 46-bit physical addresses and 3-bit partition IDs, where necessary, and we ignore the small write-back buffer and prefetch FIFO common to all designs (these structures increase the total storage to 14.125 KB). CACTI 6.5 [19] was used to estimate the area, dynamic energy and leakage power for all structures assuming a 32nm feature size, including the different configurations of L2 cache tag arrays. We only show the estimates for RECAP-REGION since the three RECAP designs only differ in their use of the reuse bit, so RECAP-ALL and RECAP-BLOCK would have 128-bytes less storage and slightly less energy and power consumption. For the GHL, the dynamic energy assumes that updating the LRU structure requires six accesses to read and modify all the necessary entries in the doubly-linked list.

The RECAP design presented here requires about one fifth the area of GHL and only slightly more area than the SCAN-8 prefetcher. RECAP uses significantly less leakage power than the other designs, and less dynamic energy than GHL. In addition to these power savings, RECAP saves more energy by reducing the number of accesses to off-chip memory .

Table 3. Hardware Overheads for Prefetchers

	GHL	SCAN-8	RECAP
Storage (KB)	75.25	12	11.625
Area (mm^2)	0.143	0.0246	0.0290
Dynamic Energy (pJ)	49.4	3.33	6.66
Leakage Power (mW)	28.6	16.6	4.80

8 Related Work

Early research studying the effects of context-switch induced cache pollution showed that significant performance impact can occur; it also concluded that these detrimental effects will get worse with increasing cache sizes and levels of cache hierarchy [1, 18, 22]. More recently, a thorough characterization of the cache misses due to context switches demonstrated that a restored thread is likely to evict its own lines from the previous context, because such remaining lines are typically closer to the LRU position in the stack [17]. To mitigate the impact of context switches on cache misses, more sophisticated OS scheduling algorithms were studied [15], especially for context switches due to I/O, which are predominant in commercial applications.

Prior work on reducing activation and deactivation latency of the threads focuses on restoring register state on thread migration, and does not address the impact of allowing caches to remain "cold" on thread restoration [4, 23].

Section 4 described two cache prefetchers that are specifically targeted at the cold cache problem caused by multiprogrammed virtualization [9], and other similar scenarios [8]. Brown et al. have recently proposed a related prefetcher that focuses on short running threads that can result from new execution models such as speculative multithreading and helper threads [3]. They proposed prefetching a small number of recently accessed cache blocks and regions of memory near the current stack pointer and program counter. This approach was effective at prefetching sufficient data, "to roughly transfer enough data to cover the first 1,000 instructions" [3] after a thread migration, but showed little benefit for execution intervals of a million instructions or more, making this work ill-suited to multiprogrammed virtualization environments.

RECAP is not the first work to use coarse-grain information to improve prefetcher performance. Prior works have proposed prefetchers based on tracking and predicting the foot prints of "spatial regions" [5, 7, 16, 21]. However, unlike RECAP, the spatial regions are not tracked with region addresses, and in fact, partition switches will interrupt such spatial region tracking for a thread.

9 Conclusion

We propose the RECAP prefetcher to reduce the cold cache effects caused by multithreaded virtualization. RECAP exploits coarse-grain patterns in an application's memory access stream to restore the cache working set after a partition switch with very little bandwidth overhead. Compared to a system with only a stride prefetcher, RECAP-REGION improves performance by a mean of 4.5%, and up to 42% for some workloads. These performance benefits come with average increases in bandwidth of only 14% and result in an average reduction of 16% in the energy-delay product of the L2 cache and main memory. RECAP's hardware structures requires roughly 14.125 KB of on-chip storage.

Compared to other cache restoration prefetchers, RECAP-REGION uses 12% and 27% less total bandwidth than GHL and a CRP-like prefetcher, respectively, and RECAP-REGION reduces the energy-delay product by 13% and 12%.

We also propose two modified versions of existing prefetchers that use block reuse filtering to significantly reduce bandwidth overheads. These modified designs have little or no hardware overhead and achieve speedups within 1% of the original designs. Our proposed GHL-HITS design uses 16% less bandwidth and reduces the energy-delay product by 3.6% compared to original GHL design. Our proposed SCAN-8-HITS design uses 32% less bandwidth and reduces the energy-delay product by 7% compared to a SCAN-8 prefetcher.

One of the key shortcomings of all of the cache restoration prefetchers discussed in this paper is that none of them adapt dynamically to workload behaviors. Some applications receive very little benefit from any amount of prefetching, and other applications experience increasing speedups with increasing amounts of prefetching. For example, a SCAN-12 prefetcher that scans twelve cache ways has a 5% speedup over SCAN-8 for vpr, but for most other applications SCAN-12 has lower performance than SCAN-8 due to the cache pollution and memory contention caused by prefetching too many blocks. Thus, future work might explore the potential for dynamic mechanisms to tune how many blocks to prefetch for each application, rather than simply prefetching all blocks that are predicted to be useful.

Additional future work might examine ways for software to exploit the RECAP mechanism. The RECAP mechanism relies on the hypervisor to reserve a portion of physical memory for each prefetch list and to control when to prefetching occurs. Other software could potentially leverage RECAP to act as a prefetcher for environments that do not use multiprogrammed virtualization. In particular, compilers might be able to leverage RECAP to perform prefetching for applications that do not benefit from any known prefetching techniques.

10 Acknowledgements

The authors would like to thank Hillery Hunter, Jose Moreira, and Pratap Pattnaik at IBM Research for supporting this work. This work was also partially supported by an NSERC Discovery grant and an IBM Faculty Award, and Jason Zebchuk was partially supported by an IBM PhD Fellowship. We thank the anonymous reviewers for their helpful comments and suggestions. We also thank the Impetus group at Carnegie Melon University and the PARSA group at Ecole Polytechnique Fédérale de Lausanne for their assistance with the Flexus simulator and with the commercial workloads. Finally, we thank WindRiver for the Simics simulator.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst.*, 6(4):393–431, Nov. 1988.

[2] 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/.

[3] J. Brown, L. Porter, and D. Tullsen. Fast thread migration via cache working set prediction. In *Proc. of the 17th Annual Symp. on High Performance Computer Architecture*, Feb. 2011.

[4] J. Brown and D. Tullsen. The shared-thread multiprocessor. In *Proc. of the 22nd Annual Int'l Conf. on Supercomputing*, 2008.

[5] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Stealth prefetching. In *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[6] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proc. of the 21st Annual Int'l Conf. on Supercomputing*, ICS '07, New York, NY, 2007.

[7] C. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos. Accurate and complexity-effective spatial pattern prediction. In *Proc. of the 10th Int'l Symp. on High Performance Computer Architecture*, 2004.

[8] H. Cui and S. Sair. Extending data prefetching to cope with context switch misses. In *Proc. of the IEEE Int'l Conf. on Computer Design*, 2009.

[9] D. Daly and H. Cain. Cache restoration for highly partitioned virtualized systems. In *Proc. of the 18th Annual Symp. on High Performance Computer Architecture*, Feb. 2012.

[10] J. Engblom, D. Aarno, and B. Werner. Full-system simulation from embedded to high-performance systems. In R. Leupers and O. Temam, editors, *Processor and System-on-Chip Simulation*, pages 25–45. Springer US, 2010.

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

[12] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):31–35, Mar. 2004.

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

[14] S. M. Khan, Y. Tian, and D. A. Jimenez. Sampling dead block prediction for last-level caches. In *Proc. of the 43rd Annual IEEE/ACM Int'l Symp. on Microarchitecture*, Washington, DC, 2010.

[15] 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*, Oct. 2005.

[16] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proc. of the 25th Annual Int'l Symp. on Computer Architecture*, June 1998.

[17] 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, Oct. 2008.

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

[19] N. Muralimanohart, R. Balasubramonian, and N. Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *Proc. of the 40th Annual IEEE/ACM Int'l Symp. on Microarchitecture*, Dec. 2007.

[20] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. 10(1):16 –19, Jan. 2011.

[21] S. Somogyi, T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *Proc. of the 33rd annual Intl. Symp. on Computer Architecture*, June 2006.

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

[23] E. Tune, R. Kumar, D. Tullsen, and B. Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *Proc. of the 37th annual IEEE/ACM Intl. Symp. on Microarchitecture*, 2004.

[24] 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.