Predictor Virtualization

Ioana Burcea[*] Stephen Somogyi[†] Andreas Moshovos[*] Babak Falsafi[†‡]

[*]Department of Electrical and Computer Engineering, University of Toronto

[†]Computer Architecture Laboratory (CALCM), Carnegie Mellon University

[‡]School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne

{ioana, moshovos}@eecg.toronto.edu ssomogyi@ece.cmu.edu babak@cmu.edu

Abstract

Many hardware optimizations rely on collecting information about program behavior at runtime. This information is stored in lookup tables. To be accurate and effective, these optimizations usually require large dedicated on-chip tables. Although technology advances offer an increased amount of on-chip resources, these resources are allocated to increase the size of on-chip conventional cache hierarchies.

This work proposes *Predictor Virtualization*, a technique that uses the existing memory hierarchy to emulate large predictor tables. We demonstrate the benefits of this technique by virtualizing a state-of-the-art data prefetcher. Full-system, cycle-accurate simulations demonstrate that the virtualized prefetcher preserves the performance benefits of the original design, while reducing the on-chip storage dedicated to the predictor table from 60KB down to less than one kilobyte.

Categories and Subject Descriptors B.3.2 **[Memory Structures]:** Design Styles — Cache memories

General Terms Performance, Design.

Keywords Predictor virtualization, caches, memory hierarchy, metadata.

1. Introduction

Architecture research has produced a wide variety microarchitectural, predictor-based optimizations, including value prediction [16,17,24] instruction reuse [26], hardware prefetching [4,7,13,14,18,25,31,27,32], dynamic program optimization [1,19,23,35], pointer caching [8], and cache replacement policies, e.g., [20]. These techniques collect *metadata* information at runtime about the application's behavior and store it in on-chip buffers or lookup tables. The information stored in these tables (referred to as *predictors* hereafter) is used to anticipate future program behavior and to apply optimizations accordingly. Despite considerable effort to understand, design and tune these optimizations, general-purpose processors implement only very few of them. For example, in the case of prefetching, currently, only the simplest proposals are implemented in hardware, e.g., [28].

Adopting prediction techniques is difficult due to several factors. First, the intrusive nature of some optimizations calls for significant changes in the processor design. Second, these techniques often require a considerable amount of dedicated on-chip resources that are too expensive to provide. Even worse, several such optimizations have been shown to achieve increasing performance with larger table sizes [8,26]. As applications data and instruction footprints grow, often predictor tables need to scale accordingly to remain effective. Moreover, in a world moving towards chip-multiprocessors, where several cores are packaged together on the same chip, the resources required by these techniques multiply by the number of cores. Third, general-purpose processors are used, as their name suggests, for a wide spectrum of application domains. These applications often exhibit different runtime characteristics that require different optimizations. Dedicating considerable chip resources to implement a specific set of optimizations may not be the best design option. Instead, the extra chip resources are usually dedicated to increase the capacity of conventional cache hierarchies, a trend motivated by large footprint applications and high off-chip memory latency. This research addresses the latter two factors.

This work introduces *Predictor Virtualization (PV)*, a technique that aims at breaking on-chip limitations for storing predictor information. PV considerably reduces the dedicated on-chip resources necessary to implement predictor-based optimizations, while preserving their effectiveness. PV takes advantage of the existing memory hierarchy, including caching capabilities, without requiring intrusive modifications. PV relies on the memory hierarchy to transparently store predictor metadata and employs a small dedicated on-chip cache to record the active entries of the predictor. Whenever this cache is not enough to store the content of the predictor, the predictor metadata is spilled to the memory hierarchy. The spilling process uses normal memory requests, injected on the backside of the L1. Thus, predictor data is naturally stored in caches and can optionally be stored in main memory. As discussed in Section 2.4 some predictors can be virtualized easily. For other predictors it is necessary to revisit their design with virtualization in mind. Fortunately, predictor virtualization offers several opportunities for compensating for the variable access latency it introduces.

Existing technology trade-offs suggest that PV can be meaningful and feasible. Specifically, off-chip and on-chip memory hierarchy capacities have become sufficiently large to accommodate allocating, on demand, a small percentage of their resources for caching predictor information. Moreover, depending on the application, increased on-chip cache sizes show diminishing returns, calling for a better utilization of the available capacity.

The main contribution of this work is twofold. First, it introduces predictor virtualization as a technique meant to break on-chip resource limitations in hardware optimizations and to promote the implementation of such optimizations. Second, it presents an example of predictor virtualization, applying the method to a state-of-the-art data prefetcher. In more detail, the contributions of this work are:

- It presents *Predictor Virtualization*, a technique that uses the existing memory hierarchy to emulate large predictor tables. PV considerably reduces the amount of dedicated on-chip resources, while preserving the effectiveness of the original optimization. PV does not require any modification of the memory hierarchy other than the ability to inject memory requests to last level caches.
- It demonstrates the benefits of PV by virtualizing the spatial memory streaming prefetcher (SMS) [27]. Using Flexus [12], a full-system, cycle-accurate simulator to model a quad-core architecture, and a set of eight commercial benchmarks, this work shows that the virtualized prefetcher matches the performance of the original scheme, while reducing the dedicated on-chip predictor table from 60 kilobytes to less than one kilobyte.

The rest of this paper is organized as follows. Section 2 introduces predictor virtualization by presenting a general architecture for virtualized predictors. Section 3 discusses a virtualized design for SMS. Section 4 presents the experimental analysis of this design. Section 5 reviews related work, while Section 6 concludes the paper.

2. Predictor Virtualization

PV enables hardware optimizations to use large predictor tables without dedicating precious on-chip resources. Instead of storing all predictor information in a large, dedicated table, PV stores the predictor table in the memory space. To maintain the illusion of a large, dedicated table, PV delivers predictor entries on-demand to a small cache that is tightly-coupled to the processor.

Figure 1 presents the PV architecture where we start with a hardware optimization technique comprising an *optimization engine,* and a *predictor table,* as shown in part (a). Under PV, as shown in part (b), the optimization engine remains unchanged but the predictor table is replaced by two components: the *PVTable*, and the *PVProxy*. The next two subsections describe these two components in detail.

2.1 PVTable

The *PVTable* is the predictor table stored in the main memory address space. There are multiple alternative design options for storing the predictor table. One option is to reserve a portion of the physical address space to be used by the predictor table. A second option is to rely on operating system support to manage a set of either virtual or physical addresses to map the *PVTable*. This work assumes the first option where a small chunk of the physical memory space is reserved for the *PVTable*. Each core has its own predictor table, and, hence, its own chunk of reserved physical memory address space. This approach does not require OS support. Alternatively, multiple cores can share the same virtualized PVTable.

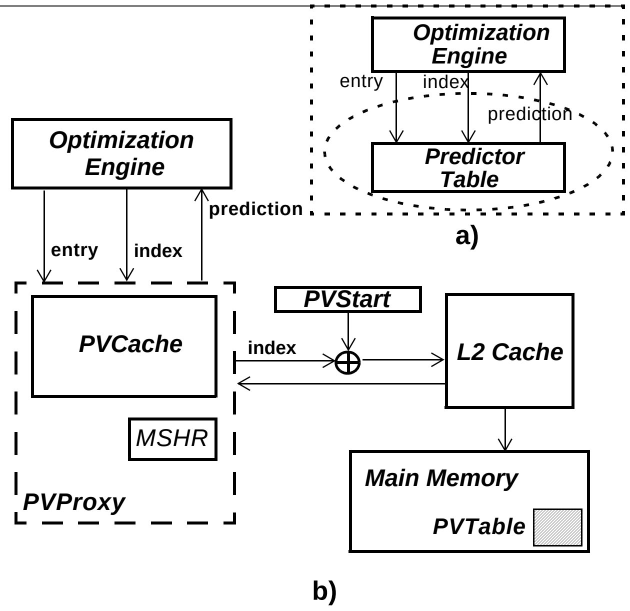

Figure 1. a) Optimization engine and non-virtualized predictor table; b) The architecture of the virtualized predictor table

The start address for each PVTable is fixed and the physical memory space dedicated to the predictor is reserved, without declaring the corresponding physical addresses to the OS. Thus, the OS is oblivious to the physical space dedicated to the predictor.

In our design, we store the start address of the table in a special per-core control register (*PVStart*). Depending on how the predictor space is managed, this register may be included in the architectural state of the processor and saved on context switches. For example, if sharing the predictor table among applications is detrimental, independent tables can be preserved by allocating different chunks of main memory to different applications via the *PVStart* registers. In this work, we assume a shared predictor table per core and the *PVStart* registers are not part of the architectural state.

2.2 PVProxy

The interface between the optimization engine and the original predictor table is preserved in the virtualized architecture. The predictor table stores information for future reference which is retrieved by the optimization engine to make predictions. The *PVProxy* provides the optimization engine with two basic operations: 1) store an entry in the prediction table, and 2) retrieve an entry. For both operations, the optimization engine provides an index to the *PVProxy*. This is the index normally used to identify the entry requested from the predictor table.

The *PVProxy* contains a small structure used to store a few of the predictor entries (*PVCache*). Upon receiving a request from the optimization engine, the *PVProxy* uses the requested index to check whether the corresponding entry is in the *PVCache*. If it is, the operation is performed as requested. If it is not, the *PVProxy* initiates a request to bring the entry. The address for this memory request is computed using the start address of the in-memory predictor table (*PVStart*) and the index provided by the optimization engine. The request is stored in an MSHR-like structure. In this work, the *PVProxy* sends its requests to the L2 cache. When the reply from the memory system arrives, the predictor entry is installed in the *PVCache* and the request from the optimization engine is completed as usual. Alternatively, the PVProxy may report a predictor miss to the optimization engine.

Upon installation of a new entry in the *PVCache*, another entry needs to be evicted, if all entries are occupied. The *PVCache* keeps a dirty bit per entry to record whether the information stored has been modified. Upon eviction, a modified entry is sent to the memory hierarchy. Non-modified entries are discarded.

The memory requests generated by the *PVProxy* do not differ from the requests generated by the conventional L1 caches and, thus, the rest of the memory hierarchy is oblivious to the data it transfers and stores. Moreover, as a natural consequence, predictor entries are stored in the L2 or latter caches leading to reduced latency for delivering the prediction to the optimization engine. The only required modification to the memory hierarchy is to allow the *PVProxy* to communicate with the L2 cache. Arbitration is necessary between the L1 caches and the newly introduced *PVProxy*. Application requests initiated by the L1 caches can be prioritized over *PVProxy* requests. In this work, we did not prioritize application requests over PV requests.

It is desirable that several entries of the predictor be packed in a memory block equal to the size of the L1 cache block for two reasons. First, the footprint of the PV data in the caches is minimized. Second, multiple predictor entries can be brought in with one memory request. Consequently, the latency of fetching a predictor entry from memory can be amortized over several predictions, provided spatial locality in the predictor space exists.

As a design option, the main memory backend storage can be completely eliminated by making the caches virtualization aware (e.g., via a bit per block or by exposing the *PVStart* register to the hierarchy). Upon eviction, dirty lines that belong to the predictor are propagated only on chip, but not off-chip. Thus, the predictor entries that are not hot enough to stay on chip are lost. Since the prediction mechanism is of advisory nature, this approach affects only the effectiveness of the optimization, not its correctness. This design option avoids any increase in off-chip memory bandwidth. In this work entries are propagated off-chip.

2.3 Advantages of Virtualization

Beside reducing the on-chip storage dedicated to predictors, PV presents several advantages when compared to traditional on-chip lookup tables. In particular:

- The size of the predictor table can be configured at runtime, based on the observed behavior of the application, as long as it does not exceed the space reserved in physical memory. Using virtual memory to store the predictor table alleviates this requirement.
- If a feedback mechanism is in place, the optimization can be turned on and off, depending on the application's behavior. The predictor entries that are stored in caches are going to be silently replaced if they are not accessed. In the traditional approach, when the optimization is turned off, the space occupied by the large on-chip predictor is entirely wasted.
- The predictor table can be configured to belong to a process, a set of applications, a core, or a set of cores. Since predictor entries are accessed based on the start address of the in-memory table, it is sufficient to provide the proper start address in the control register to enable shared or independent tables. The traditional approach usually provides per core tables, without the ability of having per-process or per set of applications predictors. Per-process predictor tables eliminate inter-process interference in multi-programmed environments. However, adding this feature needs OS support. Since the required OS support is minimal, adding more flexibility to the virtualization may prove a better alternative. We defer the analysis of this trade-off to future work.
- Recently, virtual machine (VM) research [2] and technology [29] received a lot of attention. Live virtual machine migration [6] is a vital aspect of this technology. VM migration involves transparently moving a virtual machine on a different hardware platform. As part of the migration process, the physical memory content is moved from one host to another. PV enables transparently moving predictor information as part of VM migration. In the traditional design, on a VM migration, all information from the predictor tables would be lost, suffering a performance penalty since these tables may require long training periods.
- PV offers a way for applications to influence predictor behavior provided the virtualized predictor lives in the process's virtual space. The process can update predictor entries by simply writing to the corresponding memory locations. The *PVCache* needs to be coherent for guaranteed delivery of these updates.
- Because virtualized tables live in the memory space it may be possible to make them semi-persistent, thus having subsequent invocations of an application benefit from previously collected predictor metadata.

2.4 To Virtualize or Not To Virtualize?

A virtualized predictor exhibits non-uniform access latency. For virtualization to be effective the optimization engine must tolerate this non-uniform latency. This may not be possible for all predictors. The latency of the predictor depends on the *PVCache* hit-rate and whether the PV data remains stored on chip, in the cache hierarchy. Predictor virtualization offers several opportunities for compensating for the non-uniform access latency.

First, some predictors are inherently tolerant of longer access latencies for two reasons: Either a single prediction entry generates several speculative actions, or because the prediction is for a much longer latency operation. Section 3 discusses one such predictor that associates multiple prefetching predictions with a single predictor entry.

Second, PV can exploit temporal locality just as caches do. As predictors rely on repetition in program behavior (e.g., program counter values or data addresses), some predictors will exhibit short-term repetition which will lead to reuse of predictor data. Timely reuse of predictor data makes the PV entries hot in the caches, and, thus, predictor data remains stored on chip.

Third, since PV communicates with the memory hierarchy using cache blocks it should be possible to fetch multiple entries with a single cache transfer. This provides us with an

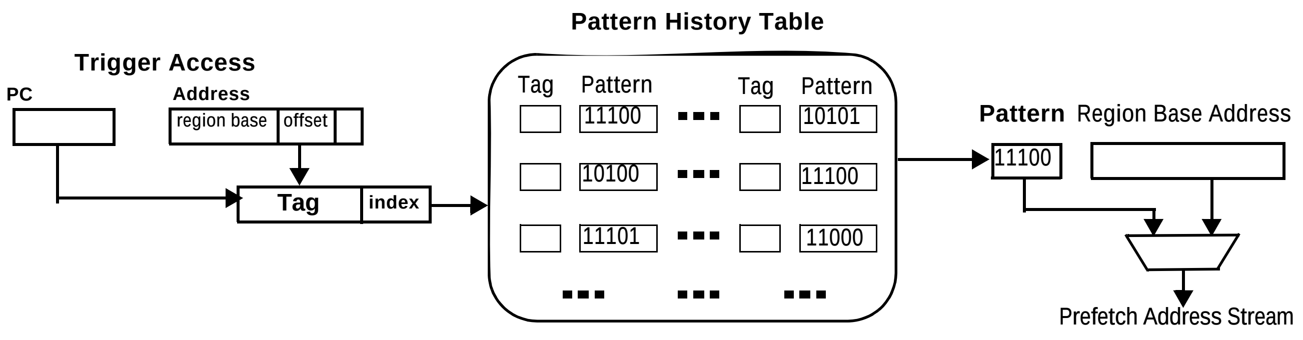

Figure 2. Pattern History Table: prediction and prefetching process.

opportunity to benefit from spatial locality in the predictor access stream. Spatial locality is determined by the way the predictor information is stored and accessed, and is highly influenced by the predictor design: what information is stored in each entry, how the table is indexed, how entries are clustered and brought together into the *PVCache*. Any *a priori* knowledge about predictor entry correlation can be exploited by packing entries in the virtualized table such that a single access brings in several temporally correlated predictor entries, amortizing the extra incurred latency.

Finally, any *a priori* knowledge about predictor entry correlation can also be exploited by anticipating future prediction accesses. Since multiple entries are packed into a single cache block, it is sufficient to guess only the appropriate cache block and not necessarily the exact predictor entry.

If the *PVCache* hit-rate is high, PV is preferable over a large dedicated predictor also because it reduces predictor lookup latency for the common case. Because the *PVCache* is much smaller than a large dedicated predictor, its latency will also be smaller.

The details of the virtualization design are particular to each individual hardware optimization. This includes design decisions such as the *PVCache* geometry and the packing of entries in contiguous memory blocks. In order for the PV to effective, these design decisions have to be carefully considered for each hardware optimization.

3. Virtualizing a state-of-the-art data prefetcher

This section reviews the Spatial Memory Streaming data prefetcher [27], and explain how it can be virtualized.

3.1 Spatial Memory Streaming

Spatial memory streaming (SMS) is a data prefetcher that extracts spatially-correlated memory access patterns over large regions of memory at runtime and uses them to predict future access patterns. SMS streams the predicted data blocks via the level two cache and into the level one cache as fast as available bandwidth and resources allow. This section briefly explains how SMS works.

Applications exhibit spatial relationships among the cache blocks they access. At runtime, SMS records the memory accesses of the application, discovers the spatial relationships among them, and uses these spatial patterns to predict the cache blocks accessed by the application in the near future. The memory accessed by the application is split into contiguous regions called *spatial regions.* A spatial region contains a fixed number of blocks. A spatial pattern is a bit vector that records which blocks within a region were accessed. The first access in a spatial region generation is called a triggering access. A spatial region generation ends when any block within the region accessed during the generation is removed from the cache by replacement or invalidation. The next access to the same spatial region will start a new generation for that region (i.e., the next access is a triggering access).

SMS uses two main hardware structures: the active generation table (AGT), and the pattern history table (PHT). The PHT consumes the bulk of the physical resources needed by SMS. The PHT stores the spatial patterns discovered by the AGT. The AGT records spatial patterns for active regions as the processor accesses memory. A region is active as long as all blocks that have been accessed since the triggering access are still in the cache. For each spatial pattern, the AGT stores the PC of the triggering access and the offset within the spatial region of the block referred to by this access. At the end of a generation (eviction or invalidation of any block accessed during the generation), the spatial pattern from the AGT is transferred to the PHT and the AGT entry is freed.

Figure 2 depicts the PHT and the prediction process. The table is indexed by a combination of the PC and the block offset of the triggering access. The PHT is used to predict the memory blocks accessed within a generation. Thus, the PHT is accessed at the start of each spatial generation. If the PHT contains a pattern corresponding to the triggering access, the spatial pattern is used to predict the memory blocks that are going to be accessed during that generation. The region base address and the bitvector pattern are used to generate the stream of prefetch addresses.

Although the AGT is conceptually a single table, in practice, it is implemented as two tables: the accumulation table and the filter table. The filter table records the spatial region tag, the PC and the block offset for all triggering accesses. Once a spatial region records an access different than its triggering one, the corresponding entry from the filter table is transferred to the accumulation table where the spatial pattern is being built. Thus, the filter table reduces the pressure on the accumulation table by filtering out all spatial regions that have only one access during a generation. The accumulation table contains all the active regions that have at least two different blocks set in their pattern.

The PHT is implemented as a set-associative table indexed by a combination of the PC and the block offset of the triggering access. This table is the predictor table that we are virtualizing, together with the two operations supported by the table: store and retrieve patterns. The two tables of the AGT and their functionality remain the same as in the original SMS design.

3.2 Virtualizing SMS

To provide accurate predictions, the PHT needs to be large. The original SMS study [27] uses a 16-way set-associative table with 16K entries (i.e., 1K sets) for the PHT. Considering spatial regions of 32 blocks and 16 bits from the PC used as index, the space required for the PHT is 86KB (64KB of data and 22KB of tag space). Virtualization can achieve the same performance improvements with less than a kilobyte worth of dedicated on-chip space. The virtualization is applied to the PHT table only. The PHT is by far the most resource-hungry among the tables used in the SMS prefetcher (the AGT needs less than one kilobyte of storage). The rest of the prefetcher remains identical to the original design.

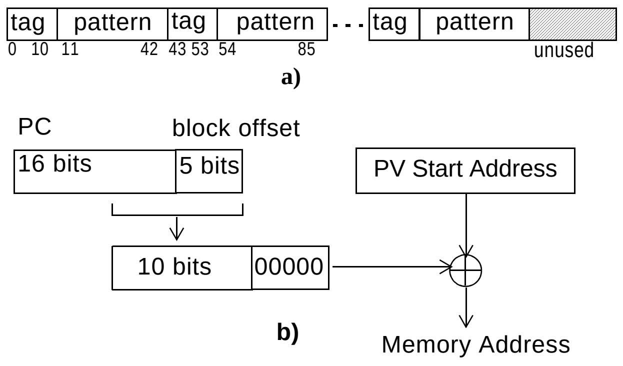

Figure 3. a) One set of the PHT is packed in 64 bytes of contiguous memory in the *PVTable*. b) Memory address calculation for the *PVProxy* memory request.

In this section we describe the virtualization of the SMS prefetcher. The virtualization is implemented following the architecture discussed in Section 2. In the virtualized version of the SMS, the PHT is stored in main memory (*PVTable)*. The on-chip mechanism (*PVProxy*) insures proper delivery of the PHT predictions to the prefetching engine. In our design, we assume a two-level on-chip cache hierarchy. Next, we detail the implementation of the two main components of the virtualized prefetcher: *PVTable* and *PVProxy*.

3.2.1 PVTable

There are multiple alternatives for mapping the content of the PHT to memory storage. We choose to pack all entries within one set (both tags and data) of the PHT into a contiguous block of memory equal in size to the L1 cache block (64 bytes), such that we can bring into the *PVCache* an entire set of the predictor table with only one L2 request. By adjusting the PHT's associativity, we are able to fit a PHT set to a cache block. Grouping the entries within a set into a cache block helps reducing the footprint of the PV in the caches. In Subsection 4.2, we show that adjusting the associativity of the PHT as required for virtualization does not reduce the prefetching performance potential. In the case of SMS, the entries within a cache block have little spatial or temporal correlation.

Figure 3a shows how entries within a set of the predictor are packed into a 64 byte contiguous memory block. The PHT table chosen for virtualization contains 1K sets, each set with 11 ways. The index used for the table is composed from concatenating 16 bits from the PC with five bits representing the block offset of the triggering access in a spatial region of 32 blocks. Since the table contains 1K sets, 10 bits from the 21-bit table index are used for the set index. The remaining 11 bits are stored as tag in the table. A spatial pattern needs 32 bits to record the access to the 32 blocks in a spatial region. Thus, there are 43 bits to be stored per predictor entry. After packing 11 entries of 43 bits each in a 64 byte block, there are some trailing unused bits. These bits could be used for LRU information, or could be reserved for further optimizations.

3.2.2 PVProxy

The *PVProxy* acts as a mediator between the prefetcher and the in-memory *PVTable*. All the requests that would normally go to the PHT table in the original design are sent to the *PVProxy.*

The *PVProxy* contains a dedicated cache (*PVCache*) in which a few sets from the *PVTable* are stored at a time. In the implementation evaluated the *PVCache* is fully associative. A tag identifies the *PVTable* set that is stored in each entry. Upon receiving a request, the *PVProxy* checks its *PVCache*. On a hit, the request is satisfied as in the non-virtualized case. On a miss, the *PVProxy* sends a memory request to the L2 cache to bring the PHT set in its *PVCache.*

Figure 3b shows how the memory address for the request is computed. The *PVProxy* receives an index that uniquely identifies the entry in the PHT. This index contains 21 bits, 16 bits from the PC of the triggering access and 5 bits from the block offset. The 10 lower bits are used to compute the set index within the 1K entry PHT table. To obtain the memory address, the set index is padded with six zeros (since each set is stored in a 64-byte contiguous memory block) and added to the start address of the *PVTable*. The *PVProxy* requests sent to the memory hierarchy are similar to the ones generated by the L1 cache. The L2 is oblivious to the existence of a different entity besides the L1 cache. Several optimizations regarding these memory requests are possible. The priority of these requests could be lower so that they do not interfere with the normal execution of the application. Given the advisory nature of the prefetcher, the requests could bypass the cache coherence. We have not implemented any of these optimizations in our current system.

Once the data is brought in the *PVCache*, the request is completed as in the original design. If the predictor set contains a pattern, the pattern is used to generate the stream of prefetches as in the non-virtualized design.

4. Experimental Analysis

This section quantitatively shows that SMS benefits from predictor virtualization. Our goal is to demonstrate the feasibility of virtualization and its benefits, and not that SMS is the best possible prefetcher.

This section is organized as follows: Subsection 4.1 describes the simulation methodology. Subsection 4.2 compares the prefetching potential with different predictor table sizes demonstrating that large predictors outperform small

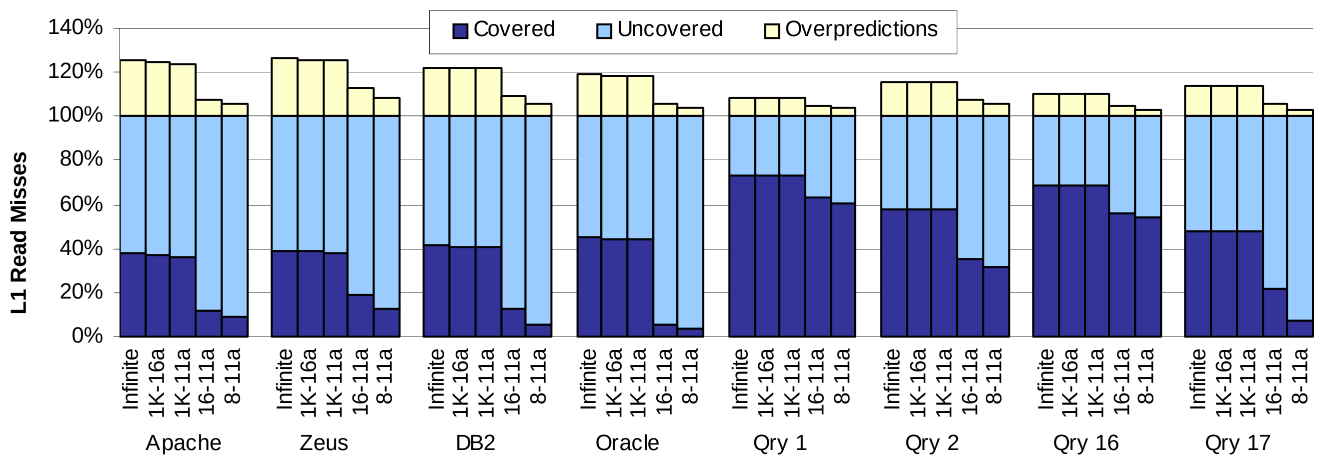

Figure 4. SMS performance potential.

predictors by a great margin. This result motivates the virtualization of large predictors. Subsection 4.3 discusses the impact of PV on the memory system. Subsection 4.4 demonstrates that applying PV to SMS offers virtually identical performance compared to the original design, considerably reducing the dedicated on-chip resources. Finally, Subsection 4.5 discusses the sensitivity of virtualization with respect to L2 cache size and latency.

4.1 Methodology

We simulate a four-core CMP with a shared L2 cache based on the Piranha cache design [3] using the Flexus simulator [12]. Table 1 shows the core and system configuration used in most experiments. We explicitly mention in the text whenever one of these parameters is varied.

Each core implements a next-line instruction prefetcher. The baseline configuration does not contain any form of data prefetching. This allows us to isolate the performance improvements obtained by the original SMS prefetcher and its virtualized design.

The SMS prefetcher was simulated using an AGT with 64 entries in the accumulation table and 32 entries in the filter table. Spatial regions contain 32 blocks. These parameters are the tuned values from the original SMS study [27]. We also verified that these values are still the best choice for the chip-multiprocessor architecture we simulate. The geometry of the predictor table (PHT) is varied throughout the experiments as specified. Each core has its own SMS prefetcher. Prefetching is performed directly into the L1 cache, without any intermediate buffering. For the virtualized configurations, each core has its own *PVProxy* and *PVTable*.

Table 2 describes the simulated workloads. These include: (1) The TPC-C v.3.0 online transaction processing workload running on both IBM DB2 v8 ESE and Oracle 10g Enterprise Database Server. (2) Four queries from the TPC-H DSS workload running on IBM DB2 v8 ESE. (3) The SPECweb99 benchmark running with Apache HTTP Server v2.0 and Zeus Web Server v4.3, respectively. The web servers were driven with separate simulated client systems; however, client activity is not included in the reported results.

All results except the execution time results presented in Subsection 4.4 and Subsection 4.5 are measured using functional simulation of two billion cycles. We use the first billion cycles as warm-up period and present measurements for the second billion cycles. Two workloads, TPC-H queries 2 and 17, complete in less than two billion cycles; thus, for these queries, we take measurements for 209 million and 259 million cycles, respectively, after the warm-up period of one billion cycles.

The speedup results presented in Subsection 4.4 and Subsection 4.5 use the Flexus cycle-accurate, full-system simulator with the SMARTS sampling methodology [34]. Each sample measurement involves 100K cycles of detailed warming followed by 50K cycles of measurement collection. Results are reported with error bars for a 95% confidence interval. We used matched-pair sampling [9] to measure performance variation. Performance is measured as the aggregate number of user instructions committed each cycle (i.e., committed user instructions summed over the four processor cores divided by the number of total elapsed cycles). This aggregate measures total application throughput [32].

Table 1. Base processor configuration

Branch Predictor	**Fetch Unit**
8k GShare + 16K bi-modal + 16K selector 2 branches per cycle	Up to 8 instr. per cycle 64-entry Fetch Buffer
	Scheduler
	256-entry/64-entry LSQ
ISA & Pipeline	**Issue/Decode/Commit**
UltraSPARC III ISA, 4GHz 8-stage pipeline out-of-order execution	any 8 instr./cycle
	Main Memory
	3 GB, 400 cycles
L1D/L1I	**UL2**
64kB 4-way set-associative 64B blocks LRU replacement 2 cycle latency	8MB, 16-way set-associative 8 banks, 64B blocks LRU replacement 6/12 cycle tag/data latency

4.2 Prefetching Potential vs. Predictor Table Size

This section studies the change in SMS performance potential as a function of predictor table size. The results show that, to

Table 2. Workloads

Online Transaction Processing (TPC-C)	
Oracle	100 warehouses (10GB), 16 clients, 1.4 GB SGA
DB2	100 warehouses (10GB), 64 clients, 450 MB buffer pool
Decision Support (TPC-H on DB2)	
Qry 1	Scan-dominated, 450 MB buffer pool
Qry 2	Join-dominated, 450 MB buffer pool
Qry 16	Join-dominated, 450 MB buffer pool
Qry 17	Balanced scan-join, 450 MB buffer pool
Web Server	
Apache	16K connections, FastCGI, worker threading model
Zeus	16K connections, FastCGI

exploit the full potential of the prefetcher, the predictor tables have to be large. This result motivates predictor virtualization.

Figure 4 shows SMS' performance potential for different predictor table configurations. The performance potential is plotted as the percentage of L1 read misses that are eliminated by the prefetcher, i.e., *covered* misses. The misses that are not eliminated by SMS are presented in the graph as *uncovered*. The *overpredictions* represent the prefetched data blocks that are evicted or invalidated before they are used.

The first bar in the graph corresponds to infinite predictor tables. In this case, the predictor table stores all the spatial patterns discovered. The second configuration (1K-16a) is the configuration that was found the best in the original SMS study for a 16 node, shared-memory architecture [27]. In this configuration, the predictor is organized as a 16-way set-associative table with 1K sets. Decreasing the number of sets beyond 1K results in lower coverage. We experimented with different configurations for the predictor and we found that reducing the table associativity from 16 to 11 (1K-11a) does not decrease the performance potential noticeably. The smaller associativity conveniently allows us to pack an entire predictor table set in a contiguous memory block of equal size to the cache block size (64 bytes). We also show results for smaller 11-way set-associative tables that have only 16 and 8 sets, respectively (16-11a and 8-11a).

SMS' behavior varies across applications when the predictor size is decreased. For example, for Oracle coverage drops from 44% with 1K sets down to less than 4% with eight sets. Other workloads, such as some TPC-H queries are less sensitive to the predictor size. For example, for Query 1, the coverage decreases from 73% with the infinite tables to 62% with a 16-set tables. Subsection 4.4 demonstrates that even this relatively small decrease in coverage can lead to important performance loss.

In the interest of space, Figure 5 presents all intermediate table sizes (i.e., from 1K entries to eight entries) for three representative workloads. While the curve describing the decrease in covered misses is different for each of them, all benchmarks experience a significant drop in coverage when varying the number of dedicated entries in the predictor table.

Overall, the results show that reducing predictor sizes leads to a significant drop in performance potential for most workloads. Thus, naively restricting the size of the predictor table is not a viable solution for decreasing the on-chip resources dedicated to the predictor.

Table 3 lists the storage requirements per configuration. In its original proposal, the predictor needs more than 80 kilobytes of on-chip space to achieve its potential (1K-16a

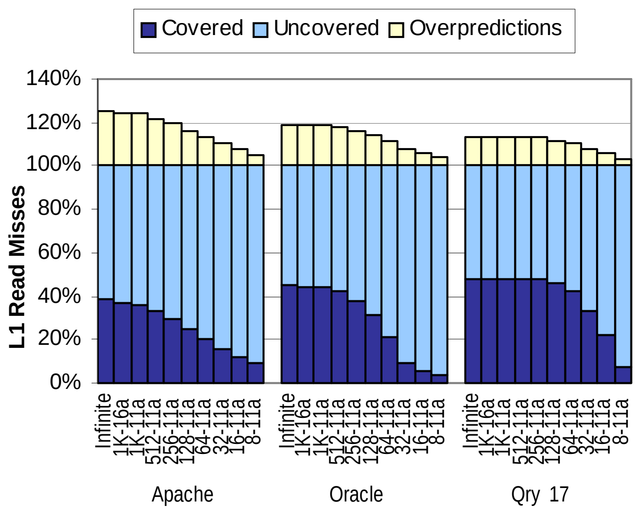

Figure 5. SMS potential - representative behavior.

configuration). Each core uses its predictor, hence the space dedicated on chip is multiplied by the numbers of cores on chip. While the small predictors require approximately one kilobyte of space, their performance potential is much lower for most workloads. PV aims at achieving the best area vs. performance potential trade-off.

We choose to virtualize the table with 1K sets and associativity of 11, since its performance is close to that of the infinite table. The difference in coverage between the 1K-11a and the infinite tables is at most 3% across all applications. Each *PVTable* has 1K sets, and each set is mapped to a 64 byte block, for a total of 64KB main memory storage per core.

Table 3. Storage for different predictor configurations

Configuration	Tags	Patterns	Total
1K-16	22KB	64KB	86KB
1K-11	15.125KB	44KB	59.125KB
16-11	374B	880B	1.225KB
8-11	198B	440B	0.623KB

4.3 Impact of the Predictor Virtualization on the Memory System

The performance of the virtualized predictor depends on two factors: 1) the interaction with the rest of the memory system, and 2) the timeliness of the prefetcher after virtualization. In this subsection, we study the impact of prefetcher virtualization on the number of the L2 requests and on the off-chip memory traffic. The timeliness of the prefetcher is reflected in the results presented in Subsection 4.4.

Figure 6 reports the increase in L2 memory requests due to virtualization as a function of the number of *PVCache* sets. For an eight set *PVCache,* the increase in memory requests varies between 25% and 44%, with an average of 33%. Increasing the number of dedicated sets for the *PVCache* does not lead to noticeably different results. This implies that before an entry gets evicted from the PVCache it is either used only once or exhibits very short term temporal locality. Only for two

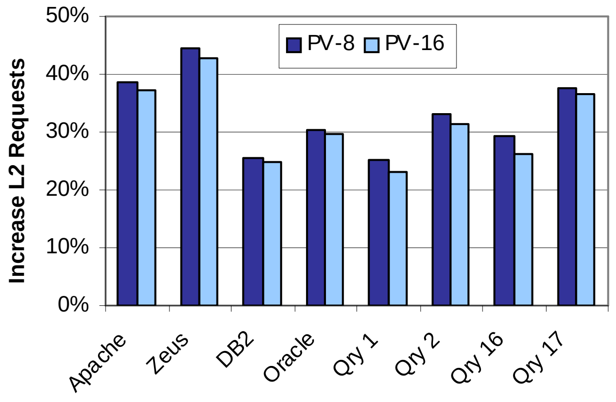

Figure 6. Percentage increase of L2 memory requests due to virtualization.

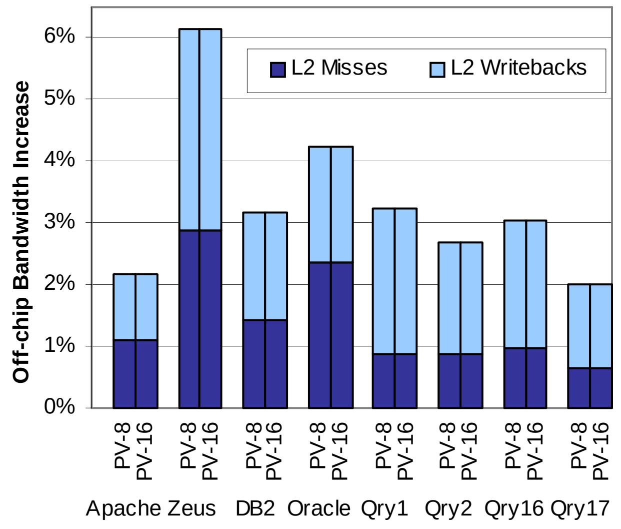

Figure 7. Impact of the virtualization on the off-chip bandwidth split in L2 misses and L2 write backs.

benchmarks, Query 1 and Query 16, increasing the number of sets to 32 (results not shown due to space limitations) leads to a decrease in the additional memory traffic of more than 5%. While the increase in L2 memory requests is significant, Subsection 4.4 demonstrates that it has a minimal impact on performance.

Figure 7 shows the percentage increase in the number of L2 cache misses and write backs for virtualized predictors with eight and 16 sets in the *PVCache*, respectively, compared to the non-virtualized SMS prefetcher. For five out of the eight simulated workloads, the increase in number of misses is less than 1%. For the rest, the increase in L2 cache misses is less than 3%. The percentage increase in the number of L2 cache misses does not change when the number of dedicated entries is increased to 16 or even 32 entries (results not shown due to space limitations). The L2 write backs experience minimal increases as well, with a maximum of 3.2% increase for Zeus.

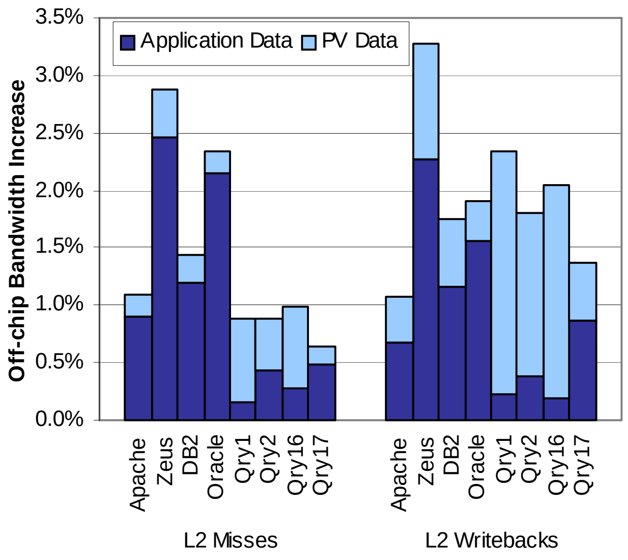

Figure 8. Percentage increase of off-chip memory traffic split in application and predictor data for PV-8 configuration.

The increase in off-chip bandwidth due to virtualization is 3.3% on average, with a maximum of 6.5% for Zeus.

Overall, there is little benefit from increasing the number of dedicated on-chip resources from eight sets to 16 or even 32, therefore, we decide to use only eight sets for the *PVCache* in our final design.

Figure 8 further splits the increase of L2 misses and writebacks into application data and predictor data for a configuration with eight dedicated entries in the *PVCache*. These results demonstrate two important points. First, the predictor entries that are naturally cached in the L2 do not lead to cache pollution. L2 application data misses increase by less than 2.5% for all studied benchmarks, with an overall average increase of 1%. The L2 cache space is better used to store predictor information rather than exclusively storing application data. Second, most of the memory requests generated by the *PVProxy* are satisfied by the L2 cache. Across all applications, more than 98% of the *PVProxy* memory requests are filled in L2 (results not shown due to space limitations). The predictor entries are hot enough to be kept in the L2 cache. This considerably reduces the latency of the virtualized predictor compared to the case where its entries would be delivered from main memory most of the time.

4.4 Performance of the Virtualized Predictor

This subsection presents cycle-accurate performance results for the virtualized predictor compared to the non-virtualized counterparts. Figure 9 presents the percentage speedup for four configurations. The first three bars represent the percentage speedup increase of the non-virtualized prefetcher with different predictor table sizes: 1K sets, 16 sets and eight sets respectively, all with a set-associativity of 11. The last bar in the graph shows the speedup for the virtualized predictor with just eight dedicated on-chip sets. The baseline uses no prefetching. On average, the smaller dedicated prefetchers achieve only half of the performance of the 1K sets prefetcher.

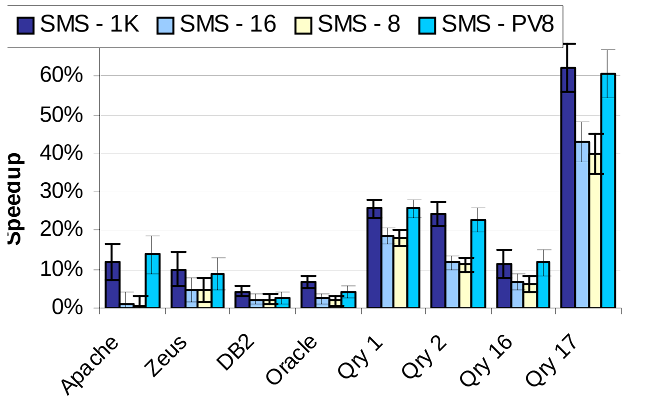

Figure 9. Performance of the virtualized predictor.

For Apache, the small predictor tables are entirely inefficient, leading to no speedup compared to the baseline.

Overall, the virtualized prefetcher achieves similar performance to the non-virtualized one. On average, the original prefetcher improves performance by 19%, while the virtualized improves performance by 18%. In the worst case, for Oracle, the improvement of the non-virtualized prefetcher is 6.7%, while it is 4.2% with the virtualized prefetcher. For the same benchmark, the small, non-virtualized prefetchers improve performance only by 2%.

4.5 Sensitivity to the L2 Cache Size and Latency

Predictor virtualization is motivated by the diminishing returns in application performance improvements with increased cache sizes. Thus, we expect that the space occupied by the PV data becomes less of a concern for larger caches. Accordingly, we studied the original prefetcher and the virtualized design in functional simulation, while varying the L2 cache size from 512KB to 2MB per core, for a total of 2MB to 8MB. Figure 10 shows the increase in off-chip bandwidth split between L2 misses and L2 write backs. As expected, PV interferes less with the application as the L2 cache capacity increases. The interference is minimal in the case of an L2 cache size of 2MB per core, for a total of 8MB.

We also verify that PV remains effective even with a longer L2 latency. For this experiment, we measured performance with the L2 tag and data latencies increased to eight and 16 cycles respectively (previous runs assumed latencies of six and 12 cycles). The first bar in Figure 11 represents the speedup with the original data prefetcher, while the second bar shows speedup with virtualization. On average, the difference is less than 1.5%.

4.6 PVProxy Space Requirements

The space required by virtualization can be broken down as follows: space for the *PVCache* - 473 bytes, tags that uniquely identify the sets within the *PVTable* that are stored in the *PVCache* - 11 bytes, dirty bits - 1 byte, MSHRs - 84 bytes. We also include an Evict Buffer with 4 entries (256 bytes) and a 16-entry pattern buffer (64 bytes) that stores the patterns while the corresponding sets are brought from the lower cache. The total space required by the PVProxy adds up to 889 bytes per core. The corresponding non-virtualized prefetcher requires

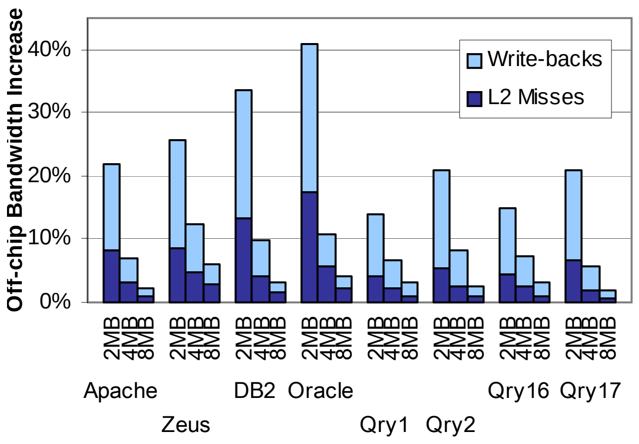

Figure 10. Off-chip bandwidth increase for different L2 cache sizes split into L2 misses and write backs.

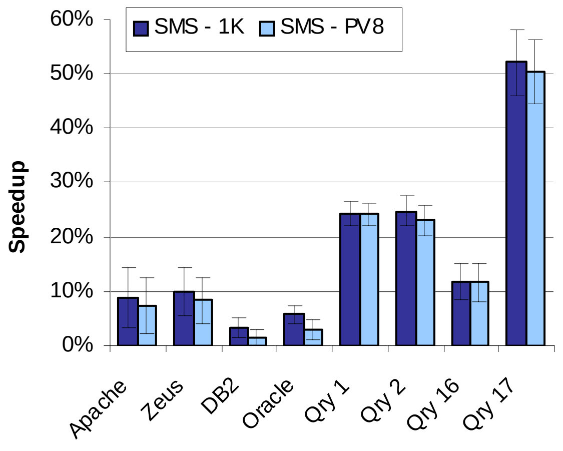

Figure 11. Performance of the virtualized prefetcher with increased L2 latency.

59.125KB of on-chip storage per core. The dedicated on-chip space is reduced by virtualization by a factor of 68, while preserving performance.

5. Related work

Using the memory hierarchy to store information other than instructions of data is not a new concept. Techniques such as LimitLESS directories [5], SC++lite [11] and VTM [21], spill machine state to the memory hierarchy. One fundamental difference between these schemes and PV is the nature of the information spilled to the memory hierarchy. In PV this information is advisory in nature and it can be silently discarded without affecting correctness.

Reconfigurable caches, introduced by Ranganathan et al. [22], replace conventional caches with a reconfigurable design that allows the on-chip SRAM storage to be divided in different partitions. These partitions can be used as conventional caches or as lookup structures for hardware optimizations. Although the cache latency does not increase due to additional logic, the cache design needs to be modified. The partitioning is done at a coarse granularity, and the sizes and the number of partitions are fixed at design time. In Predictor Virtualization the on-chip SRAM is transparently used for storing predictor information. Moreover, the allocation of cache space for storing predictor information is done on demand, at very fine granularity (i.e., cache block).

Developed independently, LT-cords [10] uses main memory to store prefetching metadata that could not be practically stored on-chip (e.g., tens to hundreds of megabytes). LT-cords makes no attempt to minimize the dedicated on-chip resources. LT-cords relies on large dedicated on-chip predictors (e.g., 200KB). PV shares some of the goals of LT-cords in that it seeks to enable the implementation of otherwise impractical, due to storage requirements, predictors. However, PV is a general framework for emulating otherwise impractical to implement predictors, and for reducing the on-chip storage required in as much as possible for other predictors.

The AMD Opteron processor uses the L2 ECC bits to store branch history information upon eviction of an instruction cache line to L2 [15]. This can be viewed as a form of predictor virtualization. However, this approach is not transparent to the memory hierarchy.

6. Conclusion

This paper introduced predictor virtualization, a technique meant to break the on-chip limitations for predictor storage in dynamic hardware optimizations. PV can be used either to reduce the cost of existing predictors or to enable the implementation of otherwise impractical predictors. PV uses the existing conventional cache hierarchy to transparently store predictor entries, instead of dedicating valuable on-chip resources.

Using full-system, cycle-accurate simulation and a set of eight commercial applications, this work demonstrated the benefits of PV by virtualizing a state-of-the-art data prefetcher. The virtualized prefetcher preserves the performance improvements of the original design, while reducing the on-chip space requirements by two orders of magnitude. The virtualized predictor needs less than 900 bytes of on-chip storage to achieve the same performance as the original prefetcher that requires approximately 60KB of dedicated space.

Moving forward we expect that there are other existing predictors, such as for example, branch target prediction, that will naturally benefit from predictor virtualization. These are predictors that naturally exhibit both temporal and spatial locality in the predictor access stream. There are other predictors, such as branch direction predictors, that may not benefit as-is by virtualization. However, we expect that it will be possible to virtualize these predictors by revisiting their design with virtualization in mind. In addition, virtualization may provide a natural way for sharing predictor state across processes, making predictor state semi-persistent, and exposing predictors to software via regular memory operations. Finally, future work can investigate the several options that exist on how to manage metadata in the memory hierarchy, whether to spill this metadata to main memory, and whether this metadata should be exposed to the operating system and applications.

Acknowledgements

We thank the anonymous reviewers for their time and insightful comments. Jason Zebchuk, Myrto Papadopoulou and Livio Soares provided useful feedback on earlier drafts of this paper. Ioana Burcea is supported by an NSERC Canada Graduate Scholarship D3. This work was supported in part by an NSERC Discovery Grant, an Intel research grant, an Intel equipment donation, and a Canada Foundation for Innovation equipment grant.

References

[1] Almog, Y., Rosner, R., Schwartz, N., and Schmorak, A. Specialized Dynamic Optimizations for High-Performance Energy-Efficient Microarchitecture. In Proc. of the Intl' Symposium on Code Generation and Optimization, 2004.

[2] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In Proc. of the 19th Symposium on Operating Systems Principles, 2003.

[3] Barroso, L. A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., and Verghese, B. Piranha: a scalable architecture based on single-chipu multiprocessing. In Proc. Intl' Symposium on Computer Architecture, 2000.

[4] Cantin, J. F., Lipasti, M. H., and Smith, J. E. Stealth prefetching. In Proc. of the 12th Intl' Conference on Architectural Support For Programming Languages and Operating Systems, 2006.

[5] Chaiken, D., Kubiatowicz, J., and Agarwal, A. LimitLESS directories: A scalable cache coherence scheme. In Proc. of the Intl' Conference on Architectural Support For Programming Languages and Operating Systems, 1991.

[6] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In Proc. of the 2nd Symposium on Networked Systems Design & Implementation, 2005.

[7] Cooksey, R., Jourdan, S., and Grunwald, D. A stateless, content-directed data prefetching mechanism. In Proc. of the 10th Intl' Conference on Architectural Support For Programming Languages and Operating Systems, 2002.

[8] Collins, J., Sair, S., Calder, B., and Tullsen, D. M. Pointer cache assisted prefetching. In Proc. of the 35th Intl' Symposium on Microarchitecture, 2002.

[9] Ekman, M., and Stenström, P. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. Proc. Intl' Symp. on the Performance Analysis of Systems and Software, 2005.

[10] Ferdman, M., and Falsafi, B. Last-Touch Correlated Data Streaming. In Proc. of the Intl' Symposium on Performance Analysis of Systems and Software, 2007.

[11] Gniady, C. and Falsafi, B. Speculative sequential consistency with little custom storage. In Proc. of the Intl' Conference on Parallel Architectures and Compilation Techniques, 2002.

[12] Hardavellas, N., Somogyi, S., Wenisch, T. F., Wunderlich, R. E., Chen, S., Kim, J., Falsafi, B, Hoe, J. C., and Nowatzyk, A. G. SimFlex: A fast, accurate, flexible full-system

simulation framework for performance evaluation of server architecture. SIGMETRICS Performance Evaluation Review, 2004.

[13] Hu, Z., Martonosi, M., and Kaxiras, S. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In Proc.of the 29th Intl' Symposium on Computer Architecture, 2002.

[14] Jerger, N., Hill, E., and Lipasti, M. Friendly Fire: Understanding the Effects of Multiprocessor Prefetching. In Proc. of the International Symposium on Performance Analysis of Systems and Software, 2006.

[15] Keltcher, C.N., McGrath, K.J., Ahmed, A., Conway, P. The AMD Opteron processor for multiprocessor servers. IEEE Micro, 23(2): 66-76, 2003.

[16] Lipasti, M. H. and Shen, J. P. Exceeding the dataflow limit via value prediction. In Proc. of the 29th Intl' Symposium on Microarchitecture, pages 226-237, 1996.

[17] Lipasti, M. H., Wilkerson, C. B., and Shen, J. P. Value locality and load value prediction. In Proc. of the Seventh Intl' Conference on Architectural Support For Programming Languages and Operating Systems, 1996.

[18] Nesbit, K. J., and Smith, J. E. Data Cache Prefetching Using a Global History Buffer. In the Proc. of the 10th Intl' Symposium on High Performance Computer Architecture, 2004.

[19] Patel, S.J., and Lumetta, S.S. rePLay: A hardware framework for dynamic optimization. Transactions on Computers, 50(6): 590-608, 2001.

[20] Qureshi, M.K., Lynch, D.N., Mutlu, O., Patt, Y. N., A Case for MLP-Aware Cache Replacement, In Proc. of the 33rd Intl' Symposium on Computer Architecture, 2006.

[21] Rajwar, R., Herlihy, M., and Lai, K. Virtualizing Transactional Memory. In Proc. of the 32nd Intl' Symposium on Computer Architecture, 2005.

[22] Ranganathan, P., Adve, S., and Jouppi, N. P. Reconfigurable caches and their application to media processing. In Proc. of the 27th Intl' Symposium on Computer Architecture 2000.

[23] Rosner, R., Almog, Y., Moffie, M., Schwartz, N., and Mendelson, A. Power awareness through selective dynamically optimized traces. In Proc. of the 31th Intl' Symposium on Computer Architecture, 2004.

[24] Sazeides, Y. and Smith, J. E.The predictability of data values. In Proc. of the 30th Intl' Symposium on Microarchitecture,1997

[25] Sherwood, T., Sair, S., and Calder, B. Predictor-directed stream buffers. In Proc. of the 33rd Intl' Symposium on Microarchitecture, 2000

[26] Sodani, A. and Sohi, G. S. Dynamic instruction reuse. In Proc. of the 24th Intl' Symposium on Computer Architecture, 1997

[27] Somogyi, S., Wenisch, T. F., Ailamaki, A., Falsafi, B., Moshovos, A. Spatial Memory Streaming. In Proc. Intl' Symposium on Computer Architecture, 2006.

[28] Tendler, J., Dodson, S., and Fields, S. IBM eServer Power4 System Microarchitecture, Technical White Paper, IBM Server Group, 2001

[29] VMWare - http://www.vmware.com

[30] Wang, K. and Franklin, M. Highly accurate data value prediction using hybrid predictors. In the Proc. of the 30th Intl' Symposium on Microarchitecture, 1997.

[31] Wang, Z., Burger, D., McKinley, K. S., Reinhardt, S. K., and Weems, C. C. Guided region prefetching: a cooperative hardware/software approach. In Proc. of the 30th Intl' Symposium on Computer Architecture, 2003

[32] Wenisch, T. F., Somogyi, S., Hardavellas, N., Kim, J., Ailamaki, A., and Falsafi, B. Temporal Streaming of Shared Memory. In Proc. of the 32nd Intl' Symposium on Computer Architecture, 2005.

[33] Wenisch, T.F., Wunderlich, R. E., Ferdman, M., Ailamaki, A., Falsafi, B., and Hoe, J. C. SimFlex: statistical sampling of computer system simuation. IEEE Micro, 26(4): 18-31, 2006.

[34] Wunderlich, R. E., Wenisch, T. F., Falsafi, B., Hoe, J. C. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In Proc. of the 30th Intl' Symposium on Computer Architecture, 2003.

[35] Zhang, W., Calder, B., and Tullsen, D. M. An Event-Driven Multithreaded Dynamic Optimization Framework. In Proc. of the 14th Intl' Conference on Parallel Architectures and Compilation Techniques, 2005.