

Teaching Old Caches New Tricks: RegionTracker and Predictor Virtualization

Ioana Burcea, Jason Zebchuk and Andreas Moshovos
Electrical and Computer Engineering Department
University of Toronto
{ioana,zebchuk,moshovos}@eecg.toronto.edu

Abstract

On-chip last-level caches are increasing to tens of megabytes to accommodate applications with large memory footprints and to compensate for high memory latencies and limited off-chip bandwidth. This paper reviews two on-going research efforts that exploit such large caches: coarse-grain cache management, and predictor virtualization. Coarse-grain cache management collects and stores cache information at a large memory region granularity (e.g., 1KB to 8KB). This coarse view of memory access behaviour enables optimizations that were not previously possible with conventional caches. Predictor virtualization is motivated by the observation that on-chip storage has become sufficiently large to accommodate allocating, on demand, a small percentage of its capacity for purposes other than storing program data and instructions. Predictor virtualization uses conventional caches to store program metadata, i.e., information about program behaviour. Such metadata information can be used for several optimizations that improve performance and power. This paper summarizes the progress made and the on-going activity in these two research efforts.

1. Introduction

Several technology and application trends favor chip multiprocessor (CMP) architectures which integrate multiple processor cores, a memory hierarchy and interconnect onto the same chip. CMPs are used for commercial servers and for end-user systems as they can support both multi-program and parallel/multithreaded workloads. Designing high-performance and power-aware memory hierarchies and interconnects is imperative for CMPs in order to meet the memory demands of multiple processors and applications while not exceeding power constraints. Continuing application trends towards larger memory footprints, multi-program workloads, the poor scaling of off-chip bandwidth, and the speed gap between on-chip and off-chip memories

combine to put further pressure on the on-chip memory hierarchy and interconnect. Caches have been one of the most effective mechanisms in combating these pressures, making it no surprise that caches in current high-performance processors can store up to 24MB [6]. Future processors will only continue this trend of growing on-chip cache capacities. Such unprecedented on-chip cache capacities present an opportunity for revisiting their management and use.

Existing caches are used to store program instructions and data using block-centric management techniques, where a typical block is relatively small (e.g., 64 bytes). Our work takes the position that as cache capacities grow, another *coarse-grain management layer* becomes useful, if not necessary. Moreover, we argue that abundant cache capacity can be used to store not only program instructions and data, but also program *metadata* that can be used to improve various aspects of system performance and behaviour.

This paper reviews work in these two fronts. Section 2 reviews coarse-grain tracking and its applications. The focus is on *RegionTracker*, a cache design that has been built from the ground-up with coarse-grain tracking in mind [15]. *RegionTracker* is a practical design that serves as the building block for coarse-grain memory systems optimizations. Section 3 reviews *Predictor virtualization*, a technique that uses a small portion of on-chip cache capacity to store, on-demand, program *metadata* (i.e., information about program behaviour). As demonstrated, predictor virtualization can greatly improve the cost and performance of two metadata-based performance enhancing techniques: memory prefetching [3] and branch target address prediction [2].

2. Coarse-Grain Cache Management

For chips with an ever-growing cache capacity, a coarse-grain view of cache contents offers many potential benefits: exposing new patterns, reducing redundancy, and predicting future behaviour. To adapt an old saying, coarse-grain cache management allows optimizations to see the forest, and not just the trees; in other words, it exposes behaviour patterns that would be hard to detect by exam-

ining only fine-grain blocks of memory. Even when fine-grain behaviour is easy to detect, due to the spatial locality prevalent in many applications, adjacent fine-grain blocks frequently experience similar behaviours. Tracking such behaviours once for a large region of memory removes the redundancy of tracking the same behaviour in multiple fine-grain memory blocks. In addition, after observing the behaviour of a single block, optimizations can predict that other blocks in the same coarse-grain region will exhibit similar behaviour. As Sections 2.1.1 through 2.1.4 demonstrate, many recent works have exploited these advantages of coarse-grain cache management.

Despite the many benefits of coarse-grain management, practical concerns still require using fine-grain caches. For example, bandwidth limitations make it impractical for caches to operate only on coarse-grain memory regions. As a result, modern caches commonly use small cache lines, and this trend is likely to continue.

To exploit the benefits of coarse-grain cache management while maintaining fine-grain caches, many proposed coarse-grain optimizations use large supplementary structures. While the relative overhead of such structures may be small – only 5% to 15% of total cache area – the large size of on-chip caches makes the absolute costs of such structures very high – 5% of a 24MB cache is over 1MB. Such large structures consume important portions of tightly constrained chip area and power budgets.

2.1 RegionTracker and its Applications

RegionTracker provides a dual-grain cache structure that both manages fine-grain blocks to maintain low bandwidth, and also tracks coarse-grain regions to enable many optimizations that exploit coarse-grain behaviours [15]. Without changing the cache data array, RegionTracker replaces a conventional cache tag array with a new structure with similar size. The resulting cache occupies approximately the same area and achieves similar performance to the traditional design, while adding new functionality to enable coarse-grain optimizations.

RegionTracker replaces a conventional cache tag array with three new structures: the region vector array (*RVA*), the evicted region buffer (*ERB*), and the block status table (*BST*). The *RVA* combines the functions of a conventional tag array with new functionality to enable coarse-grain optimizations. As shown in Figure 1, the *RVA* is a set-associative structure where each entry contains a region tag that identifies a coarse-grain region of memory, some state information, and a vector of block information fields (*BLOFs*) that indicate which blocks the cache contains and in which ways they are stored.

On a normal cache lookup the region tag and *BLOF* are used to locate the requested block in the data array. The

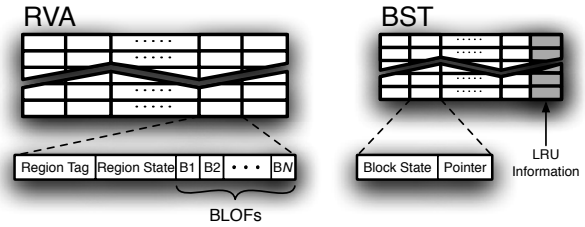


Figure 1. RegionTracker’s *RVA* and *BST* structures

way indicated in the *BLOF* also indicates the location of the block status (e.g., dirty, etc.) in the *BST*. On a cache miss a new block needs to be allocated in the data array and a new *RVA* entry may need to be allocated if none exists for the region containing the new block. The LRU information in the *BST* is used to select a victim block to replace, and the pointer in the *BST* is used to help locate the *RVA* entry that identifies the victim block. To allocate a new *RVA* entry a victim entry is chosen and moved to the *ERB*, a small, fully-associative structure. The information in the victim entry stays valid as it moves to the *ERB*, avoiding the need to evict any blocks that might be identified by the victim *RVA* entry.

This organization allows RegionTracker to provide the same coarse-grain functionality as a conventional cache tag array. At the same time, the *RVA* allows coarse-grain optimizations to easily determine which blocks in a region are currently cached, as well as providing a location to store information about each cached region.

Our recent work has demonstrated that for an 8MB shared L2 cache in a four-core CMP, a RegionTracker cache requires 3-9% less area than a conventional tag array and results in no statistically significant change in performance [15]. The next few sections demonstrate the benefits that RegionTracker offers by enabling or improving many different coarse-grain cache optimizations.

2.1.1 Snoop Elimination

Many multiprocessor systems use snoop-based coherence protocols to maintain coherence among multiple caches. Requests that miss in a processor’s local cache are broadcast to all other processors and to main memory. Each processor then searches its local cache to guarantee that the requesting processor receives an up-to-date copy of the requested data. Previous work has shown that an average of 80% of broadcasts in scientific applications are unnecessary [10], and between 15% and 94% of broadcasts are unnecessary across a broad range of workloads [4]. These unnecessary broadcasts waste tag array and interconnect bandwidth and energy.

Due to spatial locality, when a block is not shared, it is likely that other blocks in the same coarse-grain memory region are also not shared and could avoid broadcasts.

RegionScout and *coarse-grain coherence tracking (CGCT)* exploit these coarse-grain patterns to avoid unnecessary snoops [9, 4]. When replying to a snoop broadcast, processors indicate not only whether they have a copy of the requested block, but also whether they have copies of any other blocks from the same coarse-grain region. This allows processors to identify non-shared regions and subsequent requests to these regions avoid sending broadcasts.

Instead of searching the cache on each snoop request, both of these optimizations use supplementary structures to easily identify whether a processor has any cached blocks in the region, and to track non-shared regions. RegionTracker can provide the same functionality: its RVA naturally reports whether the cache contains any blocks for a region, and RVA entries can track where regions are shared. RegionTracker can implement both snoop optimizations while reducing, if not eliminating, their area overhead.

For example, RegionScout uses counting Bloom filters [10] to track which regions are present in each processor’s cache, and uses small lookup tables to track non-shared regions. For a four core CMP with 512MB private L2 caches running various commercial workloads, RegionScout can eliminate 19% of all snoops when tracking 8KB regions. This implementation requires 22KB of storage overhead per core. Implementing a similar optimization using RegionTrack with 1KB regions doubles the effectiveness to eliminate 41% of all snoop broadcasts, while eliminating the storage overhead.

2.1.2 Stealth Prefetching

Cantin et al. proposed *Stealth Prefetching*, a prefetching mechanism that tracks which blocks within a region the processor has accessed, and prefetches these same blocks if the processor subsequently has a cache miss for one of these blocks [5]. This optimization exploits the observation that large data structures frequently have combinations of fields that are commonly accessed with significant temporal locality. Such coarse-grain patterns would be difficult to detect without using a coarse-grain view of memory.

Stealth prefetching uses a modified form of the RCA structure used by CGCT [4]. Stealth prefetching increases the size of the RCA structure with bit vectors to track individual block presence history. Considering 1KB regions, this RCA structure requires 122KB of overhead for a 1MB L2 cache. In a four processor system, stealth prefetching provides a 20% speedup on average. RegionTracker can replace this RCA structure; however, the RCA used by stealth prefetching tracks more regions than RegionTracker and benefits from information about regions that were previously stored in each cache. A RegionTracker implementation can either use a larger RegionTracker, or use a supplementary RCA-like structure to achieve the same speedup

with area overheads of 78KB and 56KB respectively. Alternatively, the size of RegionTracker can be adjusted to trade prefetcher effectiveness for area overhead.

2.1.3 DRAM Speculation Throttling

In snoop multiprocessors, speculatively overlapping the main memory access with snoop processing can improve performance. Speculation succeeds when no other processor has a copy. Otherwise, the memory request is superfluous, thus wasting power and hurting performance.

Power-efficient DRAM speculation (PEDS) exploits the same coarse-grain patterns as CGCT to reduce DRAM energy consumption by 16-21% by avoiding speculative DRAM accesses for regions that are likely to find data in remote caches [1]. Namely, when a request for one block finds a dirty copy in a remote cache, subsequent requests within the same region are also likely to find dirty copies in remote caches. PEDS adds a 6% area overhead to each last-level cache. RegionTracker can implement the same optimization with significantly less, or possibly no, overhead.

2.1.4 Virtual Tree Coherence

Virtual Tree Coherence exploits coarse-grain memory behaviour to provide a scalable coherence protocol that has limited area and bandwidth overhead and provides fast cache to cache transfers [7]. Each cached memory region of memory cached is assigned a root node that provides an ordering point for all requests within the region. The root node for each region is assigned to a node that shares at least some of the blocks from that region increasing the possibility for fast cache-to-cache transfers. For a 16-core CMP running scientific and commercial applications, VTC reduces execution time by 11% compared to a greedy-order snoop-based protocol, and by 25% compared to a directory-based protocol. RegionTracker serves as an enabling technology that makes VTC practical to implement.

2.1.5 Coherence Delegation

Directory coherence protocols offer a low-bandwidth mechanism for maintaining cache coherence. However, by ordering all requests through the directory, they introduce an extra indirection for cache-to-cache transfers, increasing their latency and often hurting performance.

Coarse-grain cache management could be used to reduce the overhead of indirections through the directory. Similar to VTC, the ordering point for a coarse-grain region can be dynamically re-assigned to a node that is actively sharing that region. This can improve latency and, thus, performance while maintaining its low bandwidth overhead.

Our current work proposes just such a scheme: a RegionTracker-based directory structure dynamically *del-*

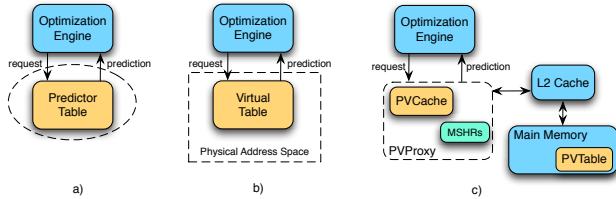


Figure 2. a) Conventional predictor-based optimization. b) Virtualized predictor-based optimization. c) The architecture of the virtualized predictor

egates control of individual regions of memory to different cores in a CMP. This *coarse-grain coherence delegation* scheme tracks fine-grain coherence information for all blocks, but this information can be stored either in a statically mapped directory, or in a delegated directory entry at a dynamically chosen core in the CMP.

A RegionTracker style directory maintains coherence information for fine-grain blocks of memory, while simultaneously tracking coarse-grain regions of memory. Similar to RegionTracker caches, a RegionTracker directory provides similar performance to a conventional directory and occupies a similar area, but adds the capability to track and store coarse-grain information in the directory.

We are studying three different types of policies that trade increasing coherence protocol complexity for increasing performance benefits. At the simplest level, the directory can choose to delegate only regions of memory that are private to an individual CMP core. Accesses to such delegated regions bypass the directory and directly access the memory controller. A slightly more complex policy allows the directory to delegate control of a region that is shared on-chip. In this case, the directory transfers the coherence information for all blocks in that region to a delegate core. Subsequent requests by that core to the delegated region can bypass the directory and retrieve data directly from either an on-chip sharer, or from off-chip memory. As a final optimization, a region that has been delegated to a core can allow accesses from *other* nodes to be ordered through that core. The RegionTracker cache at each core can track which core each region has been delegated to and send requests for blocks in that region to the delegate core. When the delegate core has a copy of the requested block, it can directly supply the data to the requester and minimize the latency of the on-chip transfer.

Our current work explores the trade-offs for performance benefits and complexity for these various types of coarse-grain coherence delegation schemes. All of these schemes, however, leverage a RegionTracker style directory and RegionTracker caches to facilitate collecting and maintaining coarse-grain information without the need for large supplementary structures.

3. Predictor Virtualization

Modern processors rely on predictor-based hardware optimizations that collect application *metadata* and store it in on-chip lookup tables (i.e., *predictor tables*) [11, 12, 13, 8]. This metadata is used to discover patterns in application execution, anticipate future application behaviour, and apply optimizations accordingly. As application footprints grow, predictor tables need to scale to remain effective. This requirement is at odds with existing trends of increasing the number of cores on chip, since the space dedicated to the predictor tables multiplies accordingly.

Traditionally, processor designers use a one-size-fits-all approach for hardware predictors, allocating a fixed portion of the on-chip resources to the predictor table. In consequence, the designer has to strike a balance between including different hardware optimizations and deciding on the resources to be dedicated to each optimization. This often results into a suboptimal design where: 1) the predictor resources are wasted for applications that do not benefit from the particular prediction or that require a smaller predictor table, or 2) the predictor under performs because the application would benefit from a much larger predictor table. In the second case, sometimes it is not possible to just use larger predictor tables because of latency constraints.

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

Figure 2 depicts the architecture of a hardware, predictor-based optimization before and after virtualization. The hardware optimization logic is separated into the optimization engine and the predictor table. The predictor table is stored in main memory address space. There are multiple design options for where to store the predictor table, but storing the predictor in a reserved portion of the physical address space is the least intrusive as it allows the OS to be oblivious to the physical space dedicated to the predictor.

The interface between the optimization engine and the original predictor table is preserved in the virtualized design. A small PV cache intercepts all the requests that would normally go to the predictor table. Upon receiving a request from the optimization engine, the dedicated PV cache checks whether it already contains the requested entry. If it does, the data is delivered to the optimization engine, as in the original design. Upon missing in the dedicated cache, a request is sent to the memory hierarchy (L2 in our system) to retrieve the entry from the predictor table. When the reply from the memory system arrives, the pre-

dictor entry is installed in the cache and the request from the optimization engine is completed as usual.

Several entries of the predictor metadata are 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 into the dedicated cache with only one request. Consequently, the latency of fetching a predictor entry from memory can be amortized over several predictions, provided the predictor exhibits spatial and temporal locality.

The memory requests generated by PV are similar to the requests generated by the L1 caches and, thus, the rest of the memory hierarchy is oblivious to the predictor metadata it transfers and stores. Moreover, as a natural consequence, predictor entries are stored in the L2 or lower level caches, leading to reduced latency for delivering the prediction to the optimization engine. The only required modification to the memory hierarchy is to allow PV to communicate with the L2 cache. Arbitration is necessary between the L1 caches and the newly introduced component.

The next sections show two examples of how PV can be used to considerably reduce the resources allocated to predictor tables or to emulate large predictor tables that are impractical to build otherwise.

3.1. Prefetching

Spatial memory streaming (SMS) is a runtime data prefetcher that extracts spatially-correlated memory access patterns over large regions of memory and uses them to predict future access patterns [14]. SMS streams the predicted data blocks into the level one cache as fast as available bandwidth and resources allow. The discovered patterns are stored in a pattern history table (PHT).

To provide accurate predictions, the PHT needs to be large. The original SMS study uses a 16-way set-associative PHT with 16K entries (i.e., 1K sets). The space required for the PHT is 86KB (64KB of data and 22KB of tag array). The virtualized predictor design achieves the same improvements with less than 1KB of dedicated on-chip resources. Virtualization is applied only to the PHT, since the PHT is by far the most resource-hungry among the tables used in SMS. The rest of the prefetcher remains unchanged.

Figure 3 presents the percentage speedup obtained by three different configurations for a set of commercial applications. The first bar represents the percentage speedup of the non-virtualized data prefetcher with a 1K-set PHT. The second bar shows the performance obtained by naively reducing the number of PHT sets to eight. The last bar shows the speedup for the virtualized predictor with just eight dedicated sets. The space requirements of the original design with eight sets is comparable to the space required by the virtualized design. The small dedicated prefetcher

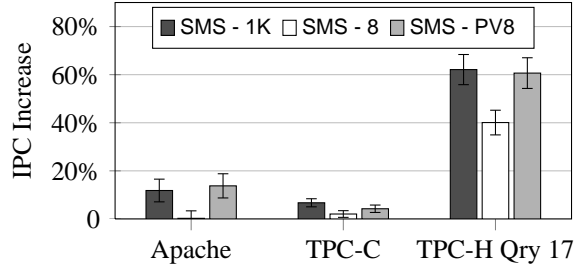


Figure 3. Performance improvement of a virtualized SMS design compared to original designs

achieves on average only half of the performance of the 1K-set prefetcher, while the virtualized design matches the performance of the large predictor; the original prefetcher improves performance by 19% on average, while the virtualized design achieves a 18% performance improvement. The virtualized predictor uses less than 900 bytes of storage to achieve within 1% performance to the original design that requires approximately 60KB of storage.

3.2. Phantom Branch Target Buffers

Processors use *Branch Target Buffers (BTBs)* to predict the target of taken branches. Using the BTB, the processor can fetch and execute instructions starting from the target address without waiting for the branch to execute. Performance improves since the processor can execute more instructions concurrently. BTBs store the target address of a branch after it is taken for the first time. Upon subsequent encounters of the same branch, the processor can predict its target address by looking it up in the BTB. If the target address did not change, the BTB prediction is correct.

A processor cannot execute instructions at a faster rate than it fetches them. Accordingly, BTBs have to produce predictions fast enough in order to keep the processor’s front-end busy; a BTB that is twice as slow can greatly reduce performance as it could, in principle, half the instruction execution rate. For this reason, BTB latency is often limited by the processor’s clock cycle. While larger BTBs could improve BTB coverage and accuracy they would reduce instruction fetch rate and, thus, hurt performance.

PV can be used to emulate larger dedicated BTBs without incurring an area or a latency penalty. *Phantom BTB (PBTB)* is a virtualized BTB design that complements a reasonably tuned, first-level, conventional BTB with a second-level, large virtual table. While the first-level BTB uses dedicated storage and operates the same way a conventional BTB does, there is no dedicated storage for the second-level table. Instead, the virtual table lives in a reserved portion of the physical address space. The virtual table entries are transparently allocated and stored on demand, at cache line granularity, in the L2 cache. *PBTB* relies on repetition and

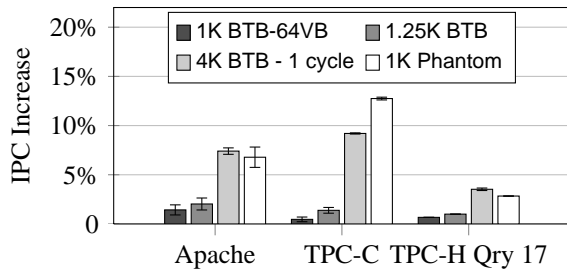


Figure 4. Phantom-BTB performance compared to conventional BTB designs

temporal correlation in the BTB miss stream: the virtual table stores groups of temporally correlated branches that have missed in the first-level BTB.

Experimental results show that allocating the virtual table in the L2 is not detrimental to overall performance even for applications with large instruction and data footprints that tax the on-chip memory hierarchy. Figure 4 shows the performance improvement of different BTB configurations compared to a 1K-entry conventional BTB baseline. A PBTB configuration that adds only 8% storage overhead to a conventional 1K-entry BTB improves IPC performance by an average of 6.9% and up to 12.7%. If this additional hardware is employed in conventional approaches, such as implementing a 64-entry victim buffer (1K BTB-64VB) or to add one way to the BTB table (1.25K BTB), the performance gain is marginal. On average, PBTB performs within 1% of a 4K-entry, single-cycle access, conventional BTB, while it requires 3.6 times less dedicated storage.

Virtualization could offer further benefits. For example, the virtualized resources be shared among cores so that one core benefits from the information collected by another. The virtualized tables can be exposed to software via regular loads and stores allowing the software to influence predictor behaviour. Finally, virtualized tables can be made persistent, thus allowing prediction information to be maintained across application invocations.

4. Conclusions

This paper reviewed two techniques that exploit the ever increasing capacity of last-level on-chip caches. The first technique uses coarse-grain cache management to more effectively use the on-chip capacity or to optimize memory system behaviour. The second technique uses conventional caches to dynamically emulate large, dedicated predictors. The proposed RegionTracker provides a dual-grain cache organization that provides a key building block to support coarse-grain cache optimizations that have been shown to provide significant performance and power improvements. RegionTracker not only reduces the overhead of previously

proposed optimizations, but also enables and encourages new coarse-grain optimizations that might otherwise be impractical. Predictor virtualization uses a small percentage of the large on-chip cache capacity to store program metadata instead of just storing program data and instructions. Predictor virtualization uses the abundant cache capacity to replace large dedicated lookup structures used by predictor-based hardware optimizations. This technique has been used to reduce the on-chip area of memory prefetchers and branch target address prediction, without significantly reducing the performance benefits of these optimizations.

We believe that both coarse-grain tracking and predictor virtualization are technologies that will enable many other optimizations beyond what was described here.

References

- [1] N. Aggarwal, J. F. Cantin, M. H. Lipasti, and J. E. Smith. Power-efficient DRAM speculation. In *Proc. HPCA-14*, Feb. 2008.
- [2] I. Burcea and A. Moshovos. Phantom-BTB: Improving branch target buffer performance by leveraging the on-chip memory hierarchy. In *Proc. ASPLOS XIV*, Feb. 2009.
- [3] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi. Predictor virtualization. In *Proc. ASPLOS XIII*, Feb. 2008.
- [4] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *Proc. ISCA-32*, Jun. 2005.
- [5] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Stealth prefetching. In *Proc. ASPLOS XII*, Oct. 2006.
- [6] B. Davis. Intel server update. Presentation, available at <http://download.intel.com/pressroom/pdf/nehalem-ex.pdf>, May 2009.
- [7] N. D. Enright Jerger, L.-S. Peh, and M. H. Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast for scalable cache coherence. In *Proc. MICRO-41*, Dec. 2008.
- [8] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, 1984.
- [9] A. Moshovos. RegionScout: Exploiting coarse grain sharing in snoop-based coherence. In *Proc. ISCA-32*, Jun. 2005.
- [10] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi. Jetty: Filtering snoops for reduced energy consumption in smp servers. In *Proc. HPCA-7*, Jan. 2001.
- [11] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proc. HPCA-10*, Feb. 2004.
- [12] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proc. MICRO-30*, Dec. 1997.
- [13] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proc. MICRO-33*, 2000.
- [14] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *Proc. ISCA-33*, Jun. 2006.
- [15] J. Zebchuk, E. Safi, and A. Moshovos. A framework for coarse-grain optimizations in the on-chip memory hierarchy. In *Proc. MICRO-40*, Dec. 2007.