Phantom-BTB: A Virtualized Branch Target Buffer Design

Ioana Burcea
University of Toronto
ioana@eecg.toronto.edu

Andreas Moshovos
University of Toronto
moshovos@eecg.toronto.edu

Abstract

Modern processors use branch target buffers (BTBs) to predict the target address of branches such that they can fetch ahead in the instruction stream increasing concurrency and performance. Ideally, BTBs would be sufficiently large to capture the entire working set of the application and sufficiently small for fast access and practical on-chip dedicated storage. Depending on the application, these requirements are at odds.

This work introduces a BTB design that accommodates large instruction footprints without dedicating expensive on-chip resources. In the proposed Phantom-BTB (PBTB) design, a conventional BTB is augmented with a virtual table that collects branch target information as the application runs. The virtual table does not have fixed dedicated storage. Instead, it is transparently allocated, on demand, in the on-chip caches, at cache line granularity. The entries in the virtual table are proactively prefetched and installed in the dedicated conventional BTB, thus, increasing its perceived capacity. Experimental results with commercial workloads under full-system simulation demonstrate that PBTB improves IPC performance over a 1K-entry BTB by 6.9% on average and up to 12.7%, with a storage overhead of only 8%. Overall, the virtualized design performs within 1% of a conventional 4K-entry, single-cycle access BTB, while the dedicated storage is 3.6 times smaller.

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

General Terms Design, Performance, Measurement

Keywords Predictor Metadata Prefetching, Predictor Virtualization, Branch Target Buffer

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'09, March 7–11, 2009, Washington, DC, USA.
Copyright © 2009 ACM 978-1-60558-215-3/09/03…$5.00

1. Introduction

Branch target buffers (*BTBs*) increase concurrency and hence performance by allowing the processor to fetch ahead in the instruction stream while previous instructions are still being fetched and processed. Without a mechanism like the BTB, the processor would have to first fetch a branch to calculate its target, stalling before it could fetch the target instructions. In a straightforward implementation, a BTB associates the address of a branch with its target address after the branch 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 is the same as the previous time, which is the case for direct branches and, occasionally, for indirect branches, the BTB prediction is correct.

Ideally, the BTB would be sufficiently large to capture the working set of taken branches. However, large BTBs consume considerable on-chip resources that may not be available or that may be used more effectively to implement other processor structures. Allocating a large chip area to BTBs becomes more of a challenge in chip-multiprocessors, since the cost of increasing each BTB multiplies by the number of cores on chip. Furthermore, increasing accuracy by using a large BTB does not necessarily lead to performance improvement because its access latency increases as well. As a result, depending on the application mix, it can be difficult to achieve a good balance among BTB accuracy, speed, and dedicated area. Commercial applications that repeatedly sequence through large instruction footprints challenge conventional BTB designs [2, 4, 13, 18].

Hierarchical BTBs [25], pipelining, and overriding predictors [27] can compensate, in part, for the longer access latency to larger BTBs. However, the issue of balancing the area cost of a larger BTB remains. The goal of this work is to design a BTB that achieves much of the accuracy of larger conventional BTBs without incurring their latency and area penalties.

This work introduces Phantom-BTB (PBTB), a novel hierarchical BTB design that exploits the current trend towards larger on-chip secondary caches. Building upon the recently proposed predictor virtualization [7], PBTB taps into transiently unused or under-utilized cache capacity for storing branch information to accommodate application demands

for a larger BTB. PBTB 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. The virtual table entries are transparently allocated and stored on demand, at cache line granularity, in the L2 cache, avoiding contention at the performance-critical L1 caches. For this purpose, the virtual table has associated a reserved portion of the physical address space. Experimental results show that allocating the virtual table in the L2 cache is not detrimental to cache performance even for applications with large instruction and data footprints that tax the on-chip memory hierarchy.

The key challenge for PBTB compared to hierarchical BTBs [25] is compensating for the higher latency of the virtual table. For this purpose, PBTB deviates from conventional hierarchical BTB designs, where the first level operates as a cache for a second level, larger predictor table. Instead, the virtual table is designed to facilitate timely prefetching of branch information into the first-level BTB. Accordingly, PBTB relies on repetition and temporal correlation in the BTB miss stream: the virtual table stores groups of temporally correlated branches that have missed in the first-level BTB.

Although PBTB shares the same goals with predictor virtualization (PV) [7], this work shows that a straightforward application of the PV technique is ineffective for BTBs. The original PV study demonstrated its applicability to a data prefetcher, which proved inherently tolerant to the higher prediction latency introduced by virtualization. In contrast, BTB prediction is latency-sensitive and a prefetch-based virtualized design proves to be an effective alternative. To the best of our knowledge, this work is the first to design and analyze a predictor metadata prefetcher for a latency-sensitive predictor such as the BTB.

Experimental results using full-system simulation of database and web server workloads show that Phantom-BTB performs better than conventional BTBs, with low hardware overhead. Specifically, a PBTB design 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%. Overall, PBTB performs within 1% of a 4K-entry, single-cycle access, conventional BTB, while requiring 3.6 times less dedicated storage. Experiments also demonstrate that: 1) the size of the virtual table can be tuned to achieve different levels of performance improvement, 2) PBTB improves application performance even when applied on top of larger conventional, dedicated BTBs for applications that benefit from increased BTB capacity, 3) PBTB is effective when used with a range of L2 cache sizes, and 4) the pressure PBTB places on the on-chip memory hierarchy is proportional to the demand of the application for BTB resources.

The rest of this paper is organized as follows. Section 2 motivates and presents the PBTB design. Section 3 compares PBTB with conventional BTBs. Section 4 presents the experimental methodology and analysis of PBTB. Section 5 reviews related work. Finally, Section 6 summarizes our motivation and findings.

2. Phantom Branch Target Buffer

Designing a *"one-size-fits-all"* BTB is difficult for general purpose processors that run a wide spectrum of applications, each with different runtime patterns and requirements. Instead of finding a compromise design point for BTBs, this work introduces a *"pay-as-you-go"* approach that expands the perceived BTB capacity according to the runtime demand of the application. Phantom-BTB augments a dedicated, conventional, first-level BTB with a larger virtual table. The virtual table does not have fixed dedicated storage: its entries are allocated on demand, at cache line granularity, in the second level cache, as shown in Figure 1.

The purpose of the virtual table is to capture branch information that can be brought into the dedicated BTB to improve its accuracy. In contrast to traditional hierarchical BTBs, the virtual table is not a larger version of the first-level BTB. The organization of the virtual table is decoupled from the dedicated BTB. The virtual table is structured to facilitate the timely prefetching of relevant BTB entries.

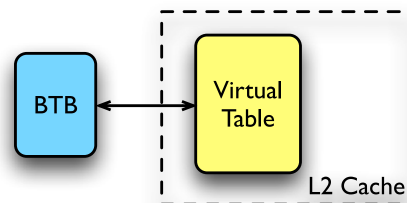

Figure 1. Phantom-BTB augments a dedicated BTB with a virtual table residing in the L2 cache

The main challenge for PBTB is compensating for the high access latency of the virtual table. Stalling the processor front-end while branch metadata is retrieved from the virtual table is not a practical option because application performance is highly sensitive to branch prediction latency [15]. To address this latency challenge, PBTB employs branch metadata prefetching, exploiting the temporal correlation in the BTB miss stream. It is well documented that the branch stream exhibits strong temporal correlation. For example, correlating branch direction predictors leverage this property to improve their accuracy [30]. While branch direction predictors rely on temporal correlation in the branch access stream, this work shows that temporal correlation persists even in the restricted BTB *miss* stream.

PBTB exploits this temporal correlation as follows. As branches miss in the dedicated BTB, they are grouped into

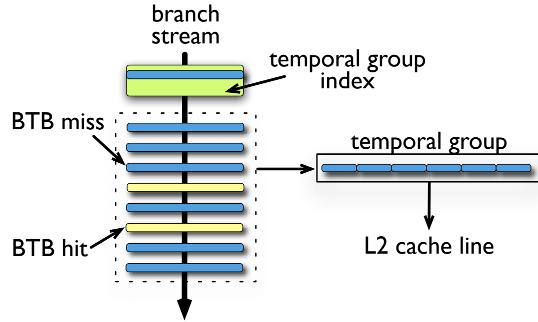

Figure 2. Temporal groups

temporal groups, as shown in Figure 2. In the current PBTB design, each temporal group represents a virtual table entry that maps to a single L2 cache line. A temporal group is associated with a *prefetch trigger*: a region of code surrounding the branch previous to the group that also missed in the BTB. Any subsequent miss for a branch within this region triggers the prefetch of the temporal group, such that identical repetition in the branch miss stream, while desirable, is not necessary for prefetching. This property is valuable for applications that exhibit only partial repetition in their control flow patterns; as long as the application follows a similar path through the code, prefetching will be triggered and some branches will be covered. Using a branch previous to the group to index in the virtual table allows PBTB to tolerate the latency of the L2 cache when temporal groups are prefetched.

BTB misses trigger prefetching of their associated temporal blocks into a prefetch buffer collocated with the dedicated BTB. The prefetch buffer and the BTB are accessed in parallel on each branch access. On a BTB miss and prefetch buffer hit, the corresponding branch is installed in the dedicated BTB, thus, increasing its perceived capacity.

Section 2.1 describes the architecture of PBTB and Section 2.2 discusses the implications for the on-chip memory hierarchy.

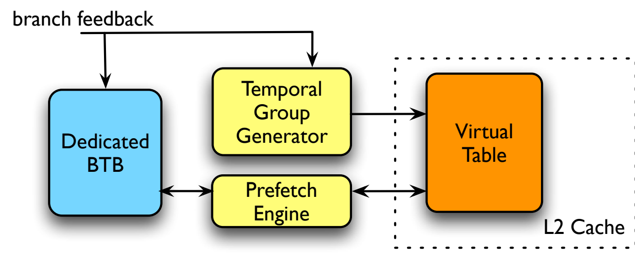

Figure 3. Phantom-BTB architecture

2.1 Phantom-BTB Architecture

Figure 3 shows the PBTB architecture. PBTB adds three components to a dedicated, conventional BTB: 1) a virtual table, 2) a temporal group generator, and 3) a prefetch engine. The next sections describe these components in detail.

2.1.1 Virtual Table

The main component that PBTB adds to a conventional BTB is the virtual table that contains branch target metadata collected as the application runs. This table is used to emulate a large BTB by prefetching its contents into the dedicated BTB.

The metadata table does not have dedicated physical storage, hence the name *virtual*. Instead, the virtual table resides in the L2 cache, allocated on demand at cache line granularity, competing for space with application data and instructions. The table has associated reserved physical address space (i.e., physical addresses that are not exposed to the operating system) for the sole purpose of installing its contents in the memory hierarchy non-intrusively. Given that the off-chip memory latency is prohibitively high for timely branch prefetches, the branch metadata is not propagated off-chip in the current design. This requires simple extensions to the cache controller, as detailed in Section 2.2.

A fundamental challenge for the Phantom-BTB design is how to organize the branch metadata such that it can be prefetched in a timely fashion, increasing the hit rate of the dedicated BTB. Inspired by previous research on branch and path correlation [24, 30, 29], the virtual table stores groups of temporally correlated branches.

A straightforward option for generating temporal groups is to aggregate target information for branches that are accessed closely in time. This would result in high information redundancy across groups, reducing the effective capacity of the virtual table. In addition, generating temporal groups based on all branches would lead to high traffic to the L2 cache for storing metadata. Instead, PBTB uses the BTB itself as a filter and generates temporal groups based on branches that miss in the dedicated BTB. The metadata corresponding to several consecutive branches that miss in the dedicated BTB is packed into a memory block equal in size to the L2 cache block. This memory block is sent and installed into the L2 cache, similarly to dirty-evict lines from the L1 cache. Later on, this memory block can be retrieved from the L2, and its information can be used to augment the dedicated BTB.

Conceptually, the virtual table is tag-less and direct mapped. Each temporal group is indexed with a prefetch trigger: a code region surrounding the last branch in the previous temporal group. Any branch within this region of code that misses in the dedicated BTB triggers the prefetch of the associated temporal group.

2.1.2 Temporal Group Generator

The temporal group generator (TGG) collects consecutive branches that missed in the dedicated BTB together with their target information and packs them into a contiguous memory block that is installed in the L2 cache. The number of branch entries in the TGG is dictated by the branch target metadata representation in the virtual table.

Each branch entry stores the full branch address, a two-bit branch type field and 30 bits from the target. This allows the formation of temporal groups comprising of six branches, assuming 48-bit virtual addresses and 64 byte L2 cache lines. As such, the TGG is a simple table that contains only six entries. Once the TGG's entries fill up, a temporal group is formed, and the corresponding memory request and data is sent to the L2. The address of the last branch in the group is used to create the virtual table index for the next temporal group.

2.1.3 Prefetch Engine

Prefetches are triggered by branches that miss in the dedicated BTB. The address of the branch is used to generate a memory request that retrieves from the L2 cache the temporal group associated with the branch. Once the metadata is received from the L2 cache, the temporal group is unpacked to generate BTB entries that are installed in a FIFO prefetch buffer. The prefetch buffer is accessed in parallel with the dedicated BTB. Upon a BTB miss and prefetch buffer hit, the corresponding branch information is installed in the dedicated BTB and removed from the prefetch buffer.

The prefetch buffer serves two roles. First, it reduces BTB pollution by only inserting in the BTB branches that are requested by the processor. This is important as the prefetched temporal group may only partially overlap with the current execution path. Second, it avoids multi-entry updates to the dedicated BTB. Entries are written into the dedicated BTB one-by-one as requested by the processor using the existing BTB ports.

2.2 Implications for the On-Chip Memory Hierarchy

The L2 cache is metadata oblivious: it receives memory requests from the PBTB and stores branch metadata the same way it stores application data and instructions. Simple hardware extensions are required to orchestrate the interaction between the L1 caches and the PBTB with the L2 cache, and to avoid propagating metadata off-chip.

Redirecting the replies from the L2 cache to the PBTB is performed by filtering the physical addresses associated with predictor metadata. Evictions of cache lines that contain metadata (i.e., cache lines with addresses corresponding to the virtual table) are dropped, without being forwarded to the off-chip memory hierarchy. In addition, upon a miss in the L2 cache for a virtual table address request, a reply that contains no data is sent back to the PBTB and the request is not propagated off-chip.

Finally, coherence is not necessary for predictor metadata, and thus, memory requests associated with the virtual table can bypass the coherence mechanism.

3. Phantom-BTB versus Conventional Branch Target Buffers

There are several differences between a traditional second-level BTB and the virtual table in PBTB. First, the virtual table contains additional information about the temporal correlation of BTB entries. This information is implicit in the way groups are formed. Second, the virtual table contains redundant information across groups (i.e., one BTB entry can be part of multiple temporal groups). Third, while the maximum size of the virtual table is fixed for indexing purposes, its entries are allocated on demand, at cache line granularity in the L2 cache. Depending on the application access patterns and demand, some virtual table indexes may not be used at all, while the temporal groups for others may be evicted from the L2 cache. Accordingly, the virtual table size is an upper bound on the L2 space it occupies at any given point. For all these reasons, a direct capacity comparison between the virtual table and a conventional BTB is not meaningful.

Although motivated by applications with large instruction footprints, PBTB naturally adapts to small branch working sets because all its activity is generated by BTB misses. If the dedicated BTB is sufficiently large to accommodate most branches from the working set of the application, temporal groups are built only for compulsory misses and phase changes. Once the dedicated BTB fills, it rarely incurs misses, and only few temporal groups are generated and prefetched. Assuming an LRU cache replacement policy, application demand for cache capacity naturally evicts infrequently accessed predictor metadata. Section 4.4 demonstrates that PBTB dynamically adjusts its request for L2 resources according to the application's demands.

The aforementioned point justifies a PBTB design even when it is possible to build a large and fast conventional BTB (which is not possible with today's technology). A conventional, large BTB dedicates resources for only one purpose. These resources are wasted for applications that do not need the large BTB. PBTB uses a practically-sized, fast dedicated BTB and relies on the on-chip memory hierarchy to expand the perceived BTB capacity on demand. The disadvantage of the static resource allocation in traditional BTB designs is more pronounced in homogeneous chip-multiprocessors where the cost of each BTB multiplies by the number of cores on chip. PBTB reduces the amount of storage per core and, thus, the area savings increase linearly with the number of cores. More importantly, Section 4.7 demonstrates that PBTB improves performance even when it uses a large, first-level BTB, as long as the application benefits from additional BTB capacity.

4. Experimental Analysis

This section is organized as follows: Section 4.1 describes the simulation infrastructure and the experimental methodology. Section 4.2 shows the accuracy of BTBs when vary-

ing their capacity. This experiment demonstrates that commercial applications benefit from large BTBs, which motivates the virtualization of BTBs. Section 4.3 compares the performance improvements of PBTB against that of several conventional BTB organizations. Section 4.4 demonstrates that the impact of BTB virtualization on the on-chip memory hierarchy is minimal. Sections 4.5 and 4.6 examine PBTB's sensitivity to the virtual table size and the L2 capacity. Section 4.7 demonstrates that PBTB improves performance even when applied to larger, conventional BTBs, as long as there is a need for additional BTB capacity. Finally, Section 4.8 compares PBTB with a straightforward application of predictor virtualization [7] to BTBs.

4.1 Methodology

The dynamically-scheduled, superscalar uniprocessor configuration described in Table 1 was simulated using Flexus, a full-system simulator based on Simics [12]. The processor has an eight-stage pipeline. A deeper pipeline would be more sensitive to improvements in branch prediction accuracy.

The branch direction predictor remains the same for all experiments to isolate the impact of the BTB on performance. The geometry of the conventional BTB designs is varied throughout the experiments, as specified in the text. Unless otherwise noted, the dedicated first-level BTB of the Phantom-BTB design contains 1K entries, and the virtual table consists of 4K temporal groups. A temporal group is designed to fit within a single L2 cache line and it contains metadata for six branches. Sizes are measured in number of branch entries for the dedicated BTBs and in number of temporal groups for the virtual table. The region of code that triggers prefetching of a temporal group spans 32 instructions. This value was experimentally found to perform the best.

Table 2 describes the commercial workloads used in our analysis: 1) The TPC-C v.3.0 online transaction processing workload running on IBM DB2 v8 ESE. 2) Two queries from the TPC-H decision support 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 separately simulated client systems; the results present the server activity.

For dedicated storage computations, we assume 48-bit virtual addresses [1]. The dedicated BTBs are indexed with the lower bits of the branch address. Each BTB entry contains the remaining branch tag, a 2-bit branch type field, and the 30 low-order bits of the target. This allows the BTB to capture all branch targets within a 4GB of contiguous virtual space. For the applications studied, this covers all branches. The target of the branch is obtained by concatenating the upper bits of the branch with the lower bits of the target.

All PBTB designs use a 64-entry prefetch buffer. The breakdown of the additional hardware overhead introduced by a PBTB design with a 4K-group virtual table and a 64-entry prefetch buffer is as follows: virtual table start address

Branch Predictor	Fetch Unit
8K GShare, 16K bi-modal 16K selector, 1 branch/cycle	8 instrs./cycle 64-entry fetch buffer
ISA & Pipeline	**Scheduler**
UltraSPARC III ISA, 4GHz 8 stage pipeline	256-entry 64-entry LSQ
Main Memory	**Issue/Decode/Commit**
3 GB, 400 cycles	8 instrs./cycle
L1D/L1I	**UL2**
64KB 2-way set-associative 64B blocks, LRU replacement 2 cycle latency	4MB, 16-way set-associative 64B blocks, LRU replacement 12 cycle latency

Table 1. Baseline system configuration

Online Transaction Processing (TPC-C)	
TPCC	100 warehouses (10GB), 16 clients, 1.4 GB SGA
Decision Support (TPC-H on DB2)	
Q2	Join-dominated, 450 MB buffer pool
Q17	Balanced scan-join, 450 MB buffer pool
Web Server	
Apache	16K connections, FastCGI, worker threading model
Zeus	16K connections, FastCGI

Table 2. Workloads

(40 bits; assuming 46-bit physical addresses and 64 byte cache lines); temporal group generator (12 bit index in virtual table, six branch entries of 78 bits each); prefetch engine (16 MSHRs, 12 bits each); prefetch buffer (64 entries of 78 bits each). The total overhead is 0.7 KB.

The results presented in Sections 4.2 and 4.4 are measured using functional simulation of two billion cycles during a steady state of the workload. The first billion cycles are used as warm-up and results are presented for the second billion cycles. All speedup experiments use the Flexus cycle-accurate timing simulator [12]. Each workload was run for 500 million cycles starting from a steady state of the application. To account for non-determinism in multithreaded workloads, small perturbations are artificially introduced in the main memory latency [3]. On each L2-cache miss, the memory latency is increased by a uniformly distributed pseudo-random number of up to 10 cycles. The average memory latency is the same for all runs. However, the timing perturbations lead to different execution paths and different runtime results. In all speedup graphs, runtime error bars are shown for 95% confidence intervals.

4.2 Branch Target Buffer Accuracy

To motivate the PBTB design, Figure 4 presents the number of mispredicted branches per 1K instructions (MPKI) for conventional branch target buffers as a function of BTB size. The size of the BTB is varied from 1K to 32K entries. All BTBs are four-way set-associative since further increasing the associativity shows marginal improvement. A branch

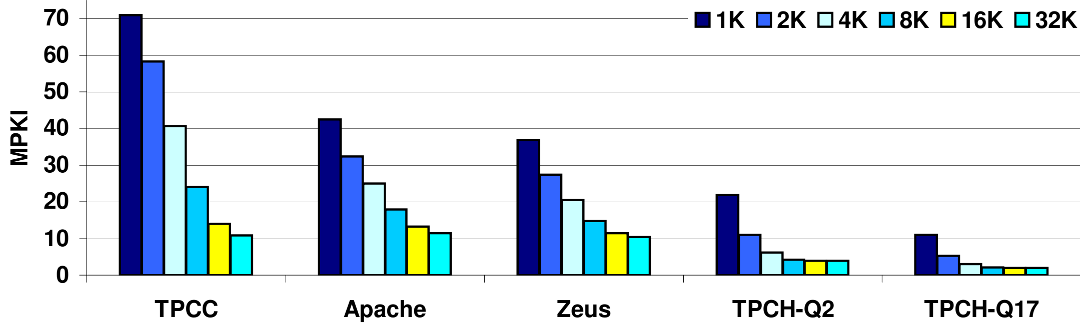

Figure 4. BTB accuracy as a function of BTB size

BTB Entries	1K	2K	4K	8K	16K
BTB Size (KB)	8.75	17.25	34	67	132

Table 3. BTB storage

Config.	1K	1K-64VB	1.25K	4K	16K	1K PBTB
Size (KB)	8.75	9.36	9.72	34	132	9.45

Table 4. Dedicated storage for different BTB configurations

is considered mispredicted when either its direction or its target is mispredicted. Increasing the number of BTB entries affects target prediction only, given that separate hardware structures predict branch directions.

The results show that these workloads have a high number of static branches and benefit from larger BTBs, albeit to different extents. On average, increasing the number of BTB entries from 1K to 16K results in 4.5 times fewer mispredicted branches. Further increasing the BTB size to 32K entries leads to marginal improvements. These results corroborate previous findings about instruction footprints of commercial workloads [2, 4, 13]. For example, Hilgendorf et al. show four different traces of commercial applications that benefit from up to 16K-entry BTBs [13].

Table 3 shows the size in kilobytes for each BTB configuration. Although the accuracy of a 16K-entry BTB is far higher than that of smaller BTBs, its storage overhead is considerable, close to twice the L1 cache size of current modern processors. These results expose the tradeoff between prediction accuracy and storage overhead. The Phantom-BTB design addresses this tradeoff by approximating the accuracy of a large BTB without the area and latency penalty.

4.3 Phantom-BTB Performance

This section compares the performance of PBTB against several conventional BTB designs. Figure 5 shows the percentage IPC increase relative to a baseline 1K-entry BTB for the following configurations: 1K-entry BTB with a 64 entry victim buffer (1K BTB-64VB), 1.25K-entry BTB (1.25K BTB), 4K-entry, single-cycle latency BTB (4K BTB 1-cycle), and 16K-entry BTB (16K Ideal) with an impractical single-cycle access latency. The last configuration, while not practical, shows the performance potential of a large conventional BTB ignoring latency effects. The PBTB configuration (1K Phantom) uses a 1K-entry dedicated BTB, a 64-entry prefetch buffer, and a 4K-group virtual table. The dedicated tables in all configurations are four-way set associative, except the 1.25K entry table which is five-way.

Since PBTB requires additional storage compared to the baseline, results are shown for a 1K-BTB with 64-entry victim buffer and for a 1.25K-entry BTB. Both these configurations use additional resources comparable to those introduced by PBTB. Table 4 shows the amount of dedicated storage for each configuration.

The results in Figure 5 show that PBTB makes good use of the hardware overhead it introduces on top of the 1K conventional BTB. The 1K-Phantom achieves an overall speedup of 6.9% and up to 12.7% for TPCC. The conventional approaches where the extra hardware is used for a victim buffer or for adding one more way to each set in the dedicated BTB show little improvement (less than 1.5% on average).

On average, the 1K-Phantom design performs within 1% of a 4K-entry single-cycle conventional BTB, while using 3.6 times less dedicated hardware. For TPCC, which is more sensitive to BTB accuracy, the 1K-Phantom outperforms the 4K-entry BTB.

Improving on the timeliness of the prefetcher and reducing the redundancy across temporal groups is key for approaching the performance potential of the 16K-Ideal BTB. On average, 55% of the prefetched entries are already present in the dedicated BTB when they arrive from the virtual table.

4.4 Impact on the On-Chip Memory Hierarchy

The PBTB competes with the application for L2 capacity and bandwidth. This section quantifies the impact of PBTB on the on-chip memory hierarchy by reporting the L2 misses and accesses per 1K instructions before and after virtualization. As Figure 6a shows, PBTB does not cause pollution in the L2 cache. The number of misses per 1K instructions that the application suffers due to BTB virtualization is negligi-

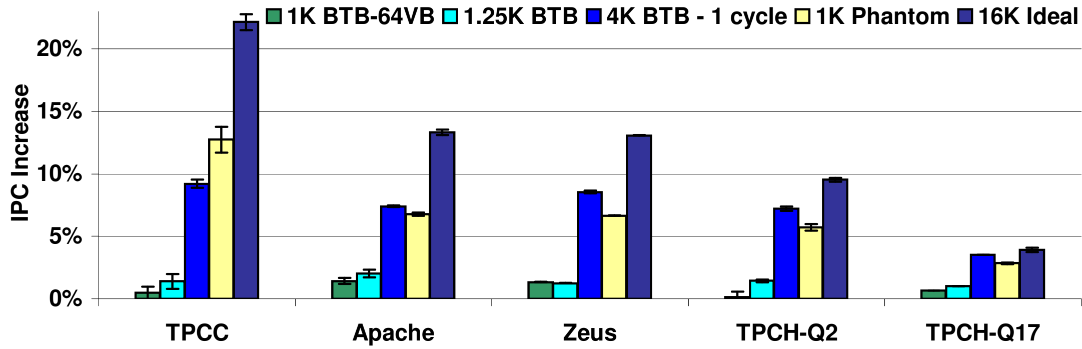

Figure 5. Phantom-BTB performance compared to conventional BTBs

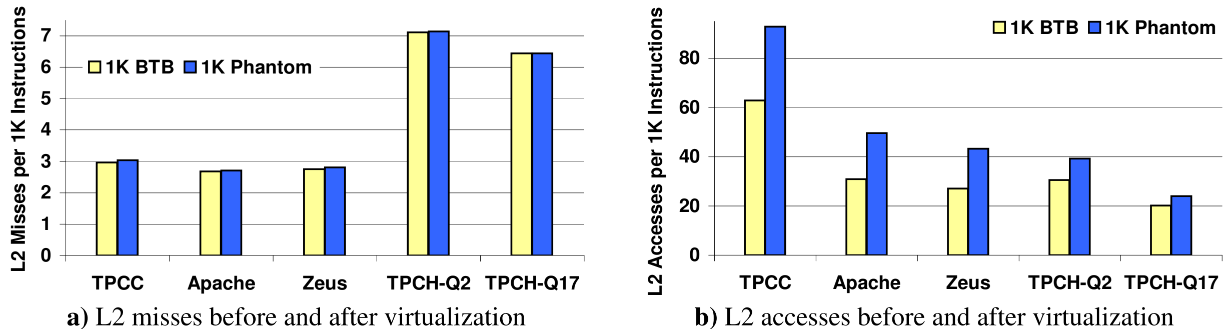

Figure 6. Phantom-BTB impact on the L2 cache

ble. The relative increase in the number of misses observed by the applications with PBTB is at most 2.4%, and 1.2% on average.

Figure 6b shows the number of L2 requests per 1K instructions with a 1K conventional BTB and 1K PBTB. As expected, Phantom-BTB increases the pressure on the L2 cache, since it stores and retrieves temporal groups from the virtual table. However, even in the worst case, for the TPCC workload, the number of accesses per 1K instructions is low, reaching only 90. It is interesting to note that, for the rest of the workloads, the L2 traffic with PBTB is lower than for TPCC with a conventional BTB.

The relative increase in the number of L2 requests with PBTB depends on the accesses to the virtual table. The results in Figure 6b indicate that PBTB dynamically adapts to the demands of the application; the L2 traffic varies according to the application's need for additional BTB capacity. For example, TPCH-Q7, which exhibits the smallest improvement from larger BTBs, incurs the lowest increase in L2 traffic.

4.5 Sensitivity to the Virtual Table Size

Figure 7 shows the performance improvement for PBTB compared to a 1K-entry conventional BTB when varying the size of the virtual table (i.e., 1K, 2K, or 4K temporal groups). All PBTBs use a 1K-entry first-level dedicated BTB. While all applications experience a higher speedup with the 4K-

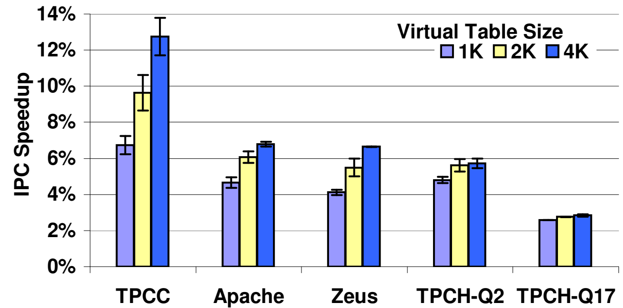

Figure 7. Phantom-BTB performance improvement with different virtual table sizes

group virtual table, the difference is more visible for TPCC. Even with a smaller virtual table, performance improves for all applications. This result suggests that the virtual table size can be tuned to obtain performance even with smaller caches, as long as there is excess capacity that can be used. The next section shows experimental results that confirm this point.

4.6 Phantom-BTB with Smaller L2 Caches

The Phantom-BTB design is motivated by increased on-chip caches that show diminishing returns. With smaller caches, PBTB may cause higher pollution when competing with the application for the cache capacity. Figure 8 shows the IPC speedup of a 1K-entry PBTB design over a 1K-entry conven-

tional BTB with two different L2 cache configurations: 1MB and 2MB (previous results were using a 4MB L2 cache). The virtual table in PBTB is configured to store up to 4K groups for the 2MB run, while for the 1MB configuration, the table is reduced to 1K temporal groups. In both configurations, the improvement obtained by higher branch prediction accuracy overcomes the negative effects of cache pollution, except for TPCC that incurs a slowdown of 2% with the 1MB L2 cache. With smaller caches, these workloads become even more memory bound and the improvements in branch misprediction become less important.

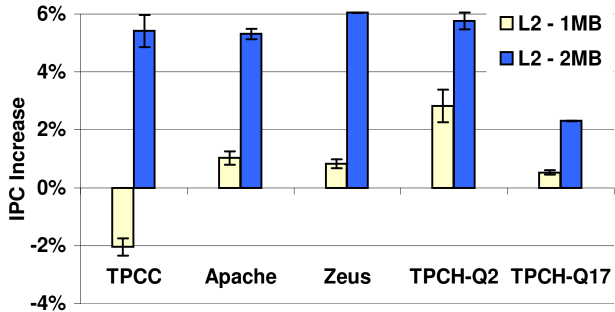

Figure 8. Phantom-BTB: Sensitivity to the L2 cache size

4.7 Phantom-BTB Applied to Larger Dedicated BTBs

All results so far analyzed the Phantom-BTB design with a dedicated 1K-entry BTB. This section demonstrates that Phantom-BTB can improve performance over larger dedicated BTBs, as long as the application benefits from additional BTB capacity. Figure 9 shows performance improvement for PBTB with 1K-, 2K-, and 4K-entry first-level BTBs compared to using the corresponding conventional BTBs alone. The speedup varies according to the application's need for a larger BTB. For example, adding Phantom on top of a 4K-entry dedicated BTB improves performance for TPCC by 7.1%. In contrast, for TPCH-Q17, which does not benefit from a larger than 4K-entry BTB, PBTB does not show any change in performance. This supports the argument that PBTB dynamically adapts to application demands.

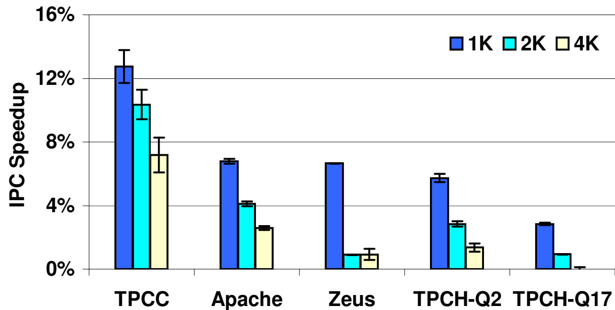

Figure 9. Phantom-BTB performance improvement over dedicated BTBs of different sizes

4.8 Phantom-BTB and Predictor Virtualization

This section demonstrates that a straightforward application of predictor virtualization [7] to BTBs does not improve performance. Figure 10 shows results for a virtualized BTB using a virtual table to store a large second level BTB (PV-BTB) and PBTB. The baseline is a conventional 1K-entry BTB. Both PBTB and PV-BTB use a virtual table that occupies up to 4K lines in the L2 cache. In PV-BTB, each entry in the virtual table stores a set from the dedicated BTB. The dedicated BTB in both schemes is a conventional 1K-entry BTB. As expected, branch target prediction is not tolerant to the L2 latency and, as a result, the PV-BTB incurs slowdowns for all benchmarks except TPCC. TPCC is a BTB-hungry application for which the extra latency to predict the correct path of execution offsets the branch misprediction penalty. In comparison, PBTB improves performance for all workloads.

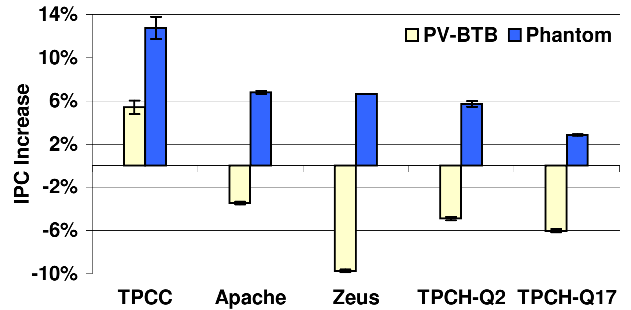

Figure 10. Phantom-BTB compared to PV-BTB

The PV-BTB design proves ineffective because the branch stream exhibits insufficient temporal and spatial locality. Temporal locality would amortize the L2 latency over multiple accesses to the same entry, while spatial locality would do so over accesses to other entries of the same set. Because of the large instruction footprints, consecutive instances of the same branch appear far apart in the branch stream. Due to BTB set indexing, spatially correlated branches end up in different BTB sets. Spatial correlation within a BTB set could be enforced by using a higher portion of the instruction address to access the BTB. This effectively compresses the instruction stream, forcing neighboring branches to fall onto the same set. Experiments, which are omitted in the interest of space, demonstrate that this approach is insufficient as well.

5. Related Work

Phantom-BTB is motivated by the need for larger BTBs in commercial workloads. Several previous studies have characterized commercial workloads finding that they exhibit large instruction footprints with high number of static branches [2, 4, 5, 18]. Annavaram et al. show that the high branch misprediction rate in commercial applications comes from aliasing due to high static branch count [4]. Hilgendorf

et al. characterize four traces of commercial applications that benefit from increased size BTBs up to 16K entries [13]. Our results show similar trends.

The concept of a branch target buffer has been studied extensively. Sussenguth describes one of the first BTB designs [28]. A detailed overview of BTB design alternatives is given by Lee and Smith [21] and by Perleberg and Smith [25], including hierarchical BTB designs. The Intel Nehalem processor incorporates a two-level BTB design; the details of this design are unknown at the time of this writing [1]. Similar to Phantom-BTB, hierarchical BTBs aim at offering much of the accuracy of larger BTBs. Phantom-BTB shares this goal and, in addition, it avoids the area cost of larger BTBs and the potentially higher access latency. Reducing the area cost of BTBs is possible by encoding its information [9, 11, 20]. In some cases, these savings come at the expense of increased latency as the BTB is split into several structures. These techniques are orthogonal to PBTB as they can be applied to both the dedicated BTB and the virtual table, reducing the cost of the dedicated predictor and the cache pressure of the virtual table.

The BTBs studied in this work make no effort to improve accuracy for indirect branches, except for returns where a return address stack was used [17]. Improving prediction accuracy for indirect branches is possible by using several approaches such as pattern or history based prediction [8, 9, 22, 16]. Using any of these techniques would increase the performance potential of Phantom-BTB as they would reduce indirect branch misses. Moreover, it may be possible to apply virtualization to these mechanisms reducing their cost. This investigation is left for future work.

Several studies proposed ways of utilizing larger and slower branch predictors. Seznec et al. describe an overriding predictor where a larger, slower, but more accurate branch direction predictor is used to correct the predictions of a smaller, faster, and less accurate one [27]. Jimenez et al. explain the tradeoff among performance, accuracy and latency in branch direction prediction and propose several techniques to compensate for larger and slower branch direction predictors, including using a small cache for fast access to recently accessed entries [15]. Their work relies on dedicated storage for the various tables. Phantom-BTB exploits the existing memory hierarchy and relies on prefetching to compensate for the longer latencies. Jimenez proposes a pipelined branch direction predictor that uses prefetching to fetch correlated pattern history entries into a small and fast first-level predictor [14]. Similarly, Phantom-BTB uses prefetching of branch target information of temporally correlated branches instead of branch direction entries. However, the second-level table (i.e., the virtual table) in Phantom-BTB does not use dedicated storage and it has higher access latency, since the table resides in the second level cache. As a result, the trade-offs and solutions are different.

Phantom-BTB indexes temporal groups of branches with an earlier branch that missed in the dedicated BTB to compensate for the long access latency to the virtual table. This technique is similar to the proposal by Yeh and Patt that indexes branch target addresses using an earlier instruction that dominates the branch [29]. In Yeh and Patt's mechanism, this allows the front-end to run ahead of the rest of the pipeline for wide-superscalar designs.

Ranganathan et al. also use the on-chip memory hierarchy to store predictor metadata [26]. In their proposal, parts of the memory hierarchy can be configured explicitly to store either data or predictor metadata. Phantom-BTB does not rely on explicit allocation of cache resources. The virtual table in PBTB is allocated at cache line granularity, competing for cache resources with the application. Moreover, except for not propagating metadata requests off-chip, the on-chip memory hierarchy is oblivious to the presence of PBTB.

Meixner and Sorin propose changes to the processor design such that all components that require SRAM for their implementation are unified into one storage component [23]. Components such as the L1 caches and the data/instruction TLBs are transformed in two-level structures, in which the last level is unified across all components. The last level structure must be sufficiently large to accommodate all components and must be tuned to accommodate the demands of different applications. This approach allows for some dynamic allocation of resources. The goal of PBTB, in addition to the dynamic allocation of resources, is to take advantage of the existing underutilized space in the memory hierarchy.

Emma et al. patented a processor implementation that stores branch metadata in the memory hierarchy [10]. Metadata corresponding to branches allocated in the same instruction cache line is packed together in a spatial group. These groups are spilled to the memory hierarchy when the corresponding instruction cache line is evicted from the L1 cache. The group is installed in a dedicated BTB when the instruction line is brought back to the L1 instruction cache. The possibility of using temporal correlation among branches in the same cache line is also mentioned. Phantom-BTB relies on prefetching of temporally correlated branches that missed in the BTB and it does not require any correlation between the content of the L1 caches and the dedicated BTB. Furthermore, the prefetch trigger is associated with a region preceding the temporal group facilitating partial coverage even when the code follows a slightly different path.

Burcea et al. proposed predictor virtualization where the memory hierarchy is used to store predictor metadata [7]. They demonstrated that predictor virtualization can be used to reduce the cost of an aggressive data prefetcher that is inherently tolerant to the high latency virtualized table. An open question with predictor virtualization is whether it is generally applicable to other predictors as well. This research addresses this question in part, as it virtualizes a latency-sensitive predictor. To the best of our knowledge,

Phantom-BTB is the first virtualized predictor design that demonstrates the benefits of metadata prefetching. An initial study of the PBTB design was presented in a technical report [6].

The AMD's Opteron is an example of a commercial processor implementation that uses a form of predictor virtualization: the L2 cache ECC bits store parts of the branch selector table [19].

6. Conclusions

Applications with large instruction footprints, such as commercial workloads, expose the tradeoff among accuracy, latency and hardware cost in BTB design. To address this tradeoff, this work introduced Phantom-BTB (PBTB), a novel virtualized BTB design. PBTB exploits the current trend towards larger on-chip caches and leverages it to expand the effective capacity of the BTB.

Full-system simulation of commercial workloads demonstrated that PBTB improves performance over a 1K-entry BTB by 6.9% on average, and up to 12.7%, while the storage overhead is only 8% of a conventional 1K-entry BTB. PBTB performs within 1% of a conventional 4K-entry, single-cycle access BTB, yet using 3.6 times less dedicated storage. Experimental results also showed that PBTB remains effective even when applied on top of larger dedicated BTBs, especially for applications with high BTB demands.

PBTB is a *"pay-as-you-go"* design that dynamically adapts to the application demand for BTB capacity, without incurring a latency penalty or imposing a fixed allocation of resources. To achieve this, PBTB employs several key techniques. First, PBTB augments a dedicated BTB with a larger, virtual table. This virtual table collects predictor metadata at run-time and is allocated in the on-chip caches, at cache line granularity. Second, the organization of the virtual table is decoupled from that of the first-level predictor and only the active entries of the virtual table are allocated in the second-level cache. In general, this decoupling allows metadata to be restructured to improve prediction accuracy or latency of the first-level predictor. It also opens up the possibility of using a sparsely populated virtual table, a property that offers further flexibility in the allocation and grouping of predictor metadata. In PBTB, metadata is organized to facilitate the prefetching of branch entries into the dedicated BTB, increasing its perceived capacity. Third, metadata collection and retrieval is triggered on misses incurred at the first level predictor. This allows the dynamic allocation of cache resources according to application's demand for predictor capacity. We believe that these techniques are general and can be applied for virtualizing other latency-sensitive predictors.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive feedback. Babak Falsafi, Jason Zebchuk, Myrto Papadopoulou and Kaveh Aasaraai commented on earlier versions of this paper. Dan Sorin and Vijayalakshmi Srinivasan provided valuable feedback regarding related work. Ioana Burcea would like to thank Livio Soares for engaging discussions and feedback at different stages of this work. This research was supported in part by an NSERC Discovery Grant, a Canada Foundation for Innovation New Opportunities Infrastructure Grant, and an Intel Research Council grant. Ioana Burcea was partially funded by an NSERC CGS-D3 Scholarship.

References

[1] First the tick, now the tock: Next generation Intel microarchitecture (Nehalem). White Paper, Intel Co., 2008.

[2] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999.

[3] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[4] Murali Annavaram, Trung Diep, and John Shen. Branch behavior of a commercial OLTP workload on Intel IA32 processors. In *Proceedings of the IEEE International Conference of Computer Design*, 2002.

[5] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[6] Ioana Burcea and Andreas Moshovos. Virtualizing branch target buffers. Technical report, University of Toronto, 2008. www.eecg.toronto.edu/∼ioana/papers/pbtb_tech_rep.pdf.

[7] Ioana Burcea, Stephen Somogyi, Andreas Moshovos, and Babak Falsafi. Predictor virtualization. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[8] Po-Yung Chang, Eric Hao, and Yale N. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[9] Karel Driesen and Urs Hölzle. The cascaded predictor: economical and adaptive branch target prediction. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[10] Philip G. Emma, Allan M. Hartstein, Brian R. Prasky, Thomas R. Puzak, Moinuddin K. A. Qureshi, and Vijayalakshmi Srinivasan. Context lookahead storage structures. IBM, U.S. Patent 7337271 B2, 2008.

[11] Barry Fagin and Kathryn Russell. Partial resolution in branch target buffers. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995.

[12] Nikolaos Hardavellas, Stephen Somogyi, Thomas F. Wenisch, Roland E. Wunderlich, Shelley Chen, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Simflex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Performance Evaluation Review*, 31(4), 2004.

[13] R. B. Hilgendorf, G. J. Heim, and W. Rosenstiel. Evaluation of branch-prediction methods on traces from commercial applications. *IBM Journal of Research and Development*, 43(4), 1999.

[14] Daniel A. Jiménez. Reconsidering complex branch predictors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[15] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, 2000.

[16] Jose A. Joao, Onur Mutlu, Hyesoon Kim, Rishi Agarwal, and Yale N. Patt. Improving the performance of object-oriented languages with dynamic predication of indirect jumps. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[17] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991.

[18] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance characterization of a Quad Pentium Pro SMP using OLTP workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[19] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(02), 2003.

[20] Ryotaro Kobayashi, Yuji Yamada, Hideki Ando, and Toshio Shimada. A cost-effective branch target buffer with a two-level table organization. In *Proceedings of the 2nd International Symposium of Low-Power and High-Speed Chips (COOL Chips II)*, 1999.

[21] Johnny K. F. Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1), 1984.

[22] Tao Li, Ravi Bhargava, and Lizy Kurian John. Adapting branch-target buffer to improve the target predictability of java code. *ACM Transactions on Architecture and Code Optimization*, 2(2), 2005.

[23] Albert Meixner and Daniel J. Sorin. Unified microprocessor core storage. In *Proceedings of the 4th International Conference on Computing Frontiers*, 2007.

[24] Ravi Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995.

[25] Chris H. Perleberg and Alan Jay Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4), 1993.

[26] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Reconfigurable caches and their application to media processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[27] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[28] Ed H. Sussenguth. Instruction sequence control. IBM, U.S. Patent 3559183, 1971.

[29] Tse-Yu Yeh and Yale N. Patt. Branch history table indexing to prevent pipeline bubbles in wide-issue superscalar processors. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, 1993.

[30] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.