Confluence: Unified Instruction Supply for Scale-Out Servers

Cansu Kaynak
EcoCloud, EPFL

Boris Grot
University of Edinburgh

Babak Falsafi
EcoCloud, EPFL

ABSTRACT

Multi-megabyte instruction working sets of server workloads defy the capacities of latency-critical instruction-supply components of a core; the instruction cache (L1-I) and the branch target buffer (BTB). Recent work has proposed dedicated prefetching techniques aimed separately at L1-I and BTB, resulting in high metadata costs and/or only modest performance improvements due to the complex control-flow histories required to effectively fill the two components ahead of the core's fetch stream.

This work makes the observation that the metadata for both the L1-I and BTB prefetchers require essentially identical information; the control-flow history. While the L1-I prefetcher necessitates the history at block granularity, the BTB requires knowledge of individual branches inside each block. To eliminate redundant metadata and multiple prefetchers, we introduce Confluence – a frontend design with unified metadata for prefetching into both L1-I and BTB, whose contents are synchronized. Confluence leverages a stream-based prefetcher to proactively fill both components ahead of the core's fetch stream. The prefetcher maintains the control-flow history at block granularity and for each instruction block brought into the L1-I, eagerly inserts the set of branch targets contained in the block into the BTB. Confluence provides 85% of the performance improvement provided by an ideal frontend (with a perfect L1-I and BTB) with 1% area overhead per core, while the highest-performance alternative delivers only 62% of the ideal performance improvement with a per-core area overhead of 8%.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles - cache memories

Keywords

Instruction streaming, Branch prediction

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

MICRO-48 December 05-09, 2015, Waikiki, HI, USA

Copyright 2015 ACM 978-1-4503-4034-2/15/12 ...$15.00.

DOI: http://dx.doi.org/10.1145/2830772.2830785

1. INTRODUCTION

With the slowdown in Dennard scaling [11, 16], improving server efficiency has become the primary challenge in large-scale IT infrastructure and datacenters. Many online services ranging from data and web serving to analytics are run in memory due to tight latency demands, requiring processors that can serve in-memory data with maximum throughput while maintaining low tail response latencies. Recent research identifies performance and efficiency bottlenecks in conventional server processors running scale-out online services, advocating for specialized manycore server processors [13, 27] that improve performance by an order of magnitude given the same silicon and power budgets and memory bandwidth with no modifications to software. Google has corroborated these results for their online services [20] with products announced (e.g., Cavium ThunderX [9]) as a first stepping stone in designing specialized server processors.

The key source of inefficiency in server processors is instruction supply [1, 13, 17, 20, 22, 30]. Modern server workloads, unlike desktop and scientific workloads, often run on deep software stacks of over a dozen layers of services. The result is multi-megabyte instruction working sets, commensurately large per-core control-flow state, incurring high latency on the execution's critical path and accounting for a large fraction of the overall silicon budget. Moreover, while there is much interplay among the instruction-supply mechanisms (e.g., instruction prefetching and branch prediction), these mechanisms are implemented standalone. Finally, all instruction-supply mechanisms are duplicated per core in manycore server processors leading to major silicon overprovisioning.

There is a myriad of techniques to improve instruction supply in servers. A brute-force approach to improve frontend performance is larger L1-I caches (e.g., Cavium ThunderX) or multi-level branch target buffers (BTB) [3, 5, 31], which exacerbate the silicon overprovisioning and are limited in effectiveness. Stream-based prefetchers to hide the latency of instruction fetches from lower levels of the cache hierarchy approach the performance of a perfect L1-I, but incur prohibitive storage costs due to storage-intensive control-flow metadata required to make predictions [14, 15]. Recent proposals advocate virtualizing [5] and/or sharing instruction-supply metadata [21] among multiple

cores. The state-of-the-art stream-based prefetcher, SHIFT [21], eliminates the majority of fetch stalls in servers by proactively streaming blocks that are likely to be accessed into the L1-I by leveraging shared instruction-supply metadata. Unfortunately, prior work falls short of providing a fully integrated approach to instruction supply within or across cores.

We present *Confluence*, an integrated architecture for instruction supply that exploits the interplay between instruction fetch and branch target prediction, the two most resource-intensive components in instruction supply in manycore server processors. We observe that timely proactive fetch of instructions into the L1-I paves the way for doing the same for the relevant instruction-supply metadata (i.e., branch targets) as there is a one-to-one correspondence between a branch instruction and its target metadata. Based on this insight, Confluence leverages a single stream-based prefetcher to fill both the L1-I and the BTB, effectively unifying the two sets of metadata and eliminating the associated storage redundancy.

An important challenge Confluence addresses is in managing the disparity in the granularity of control-flow metadata required by each of the structures it manages. Whereas an L1-I prefetcher operates at instruction-block granularity, and hence needs to track block-grain addresses, a BTB must reflect instruction-grain information of individual branches. Confluence overcomes this problem through AirBTB–a block-based BTB organization with an eager insertion policy. Whenever a block is prefetched into the L1-I, it is scanned for branch instructions and the entire set of targets associated with the branch instructions in the instruction block is inserted into AirBTB as a block. Our specific contributions are as follows:

- We show that conventional frontend designs for high-performance instruction supply incur massive per-core storage costs yet fall far from optimal due to their inability to eliminate both L1-I and BTB misses in a timely manner.

- We introduce Confluence, a frontend architecture that uses a single set of control-flow metadata shared across cores and virtualized in LLC, maintained at instruction-block granularity to fill both the L1-I and the BTB. Confluence leverages a stream-based prefetcher to proactively fill both the L1-I and the BTB ahead of the fetch stream, thus hiding the fill latency from the core.

- We propose AirBTB, a lightweight BTB design for Confluence that uses a block-based organization to reduce tag storage costs, lower BTB bandwidth requirements, and provide high hit rates by exploiting spatial locality in the instruction stream. Compared to a conventional BTB with minimal per-core storage, Confluence can eliminate 93% of the misses (32% higher than a state-of-the-art BTB prefetcher [5]), while a prohibitively large private BTB provides 95% miss coverage.

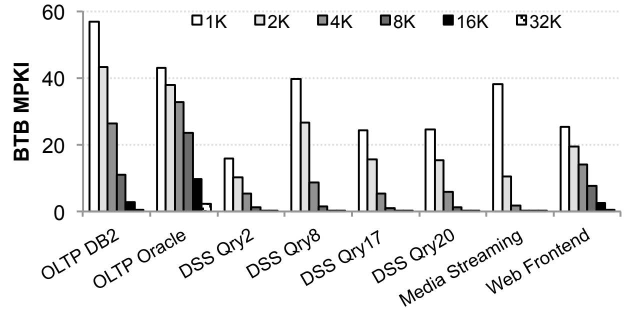

Figure 1: BTB MPKI as a function of BTB capacity (in kilo entries).

- We show that Confluence delivers 85% of the performance benefits of an ideal design (perfect L1-I and BTB combination) with only 1% area overhead per core, while the best alternative, a two-level private BTB with an 8% area overhead per core, delivers only 62% of the ideal performance improvement.

2. MOTIVATION

In this section, we briefly describe the state-of-the-art mechanisms to alleviate frequent misses in the BTB and L1-I. We quantify their performance benefits and associated storage overheads, and demonstrate that there is a need for an effective and low-cost unified mechanism for storing and managing instruction-supply metadata.

2.1 Conventional Instruction-Supply Path

Maximizing the core performance necessitates supplying the core with a useful stream of instructions to execute continuously. To do so, modern processors employ branch predictors accommodating conditional branch history and branch target history to predict the correct-path instructions to execute.

High-accuracy branch prediction necessitates capturing the prediction metadata of the entire instruction working set of a given application in the predictor tables. Unfortunately, server applications with large instruction working sets exacerbate the storage capacity requirements of these predictors. Figure 1 shows the BTB miss rate as a function of the total number of BTB entries per core. A BTB miss occurs when an entry for a predicted taken branch is not found in the BTB. Most server workloads used for this study (detailed in Section 4) require up to 16K BTB entries to fully capture all branches in their instruction working sets, while OLTP on Oracle benefits from even 32K entries, corroborating prior work [3, 5, 18]. The storage capacity requirement of a 32K-entry BTB is around 280KB (Section 4 details the cost).

Recent work has examined hierarchical BTBs that combine a small-capacity low-latency first level with a large-capacity but slower second level. The state-of-the-art proposals combine a two-level BTB with a dedicated transfer engine, which we refer to as a BTB prefetcher, that moves multiple correlated entries from the second level into the first level upon a miss in the first-level BTB. One approach, called PhantomBTB, uses tempo-

ral correlation, packing several entries that missed consecutively in the first level into blocks that are stored in the LLC using predictor virtualization [5]. Another approach, called Bulk Preload and implemented in the IBM zEC12, moves a set of spatially correlated regions (4KB) between a dedicated 24K-entry second-level BTB structure and the first level [3].

For both two-level designs, second-level storage requirements are more than 200KB per core. Moreover, accesses to the second level are triggered by misses in the first level, exposing the core to the latency of the second-level structure. For PhantomBTB, this latency is a function of NOC and LLC access delays, likely running into tens of cycles for a manycore CMP. In the case of bulk preload, this latency is in excess of 15 cycles [3].

While predicting the correct-path instructions to execute is essential for high performance, serving those instructions from the L1-I cache is also performance-critical in order not to expose the core to the long latencies of lower levels of the cache hierarchy. Doing so necessitates predicting the instructions that are likely to be fetched and proactively fetching the corresponding instruction blocks from the lower levels of the cache hierarchy into the L1-I (or prefetch buffers). To that end, fetch-directed prefetching (FDP) [32] decouples the branch predictor from the L1-I and lets the branch predictor run ahead to explore the future control flow. The instruction blocks that are not present in the L1-I along the predicted path are prefetched into the L1-I.

Although huge BTBs are effective at accommodating the target addresses for all taken branches in the instruction working set, when leveraged for FDP, they fall short of realizing the performance potential of a frontend with a perfect L1-I (i.e., L1-I that always hits) [15]. FDP's limitations are two-fold. First, because the branch predictor generates just one or two predictions per cycle, its lookahead is limited and is often insufficient to hide the long access latency to the lower levels of the cache hierarchy, which includes the round-trip time to the LLC and the LLC access itself. Second, because the branch predictor speculatively runs ahead of the fetch unit to provide sufficient prefetch lookahead, its miss rate geometrically compounds, increasingly predicting the wrong-path instructions. As a result, even with a perfect BTB, FDP significantly suffers from fetch stalls.

2.2 Covering L1-I Misses with Stream-Based Prefetching

To overcome FDP's lookahead and accuracy limitations, the state-of-the-art instruction prefetchers [14, 15, 21, 24] exploit temporal correlation between instruction cache references at block granularity. The control flow in server applications tends to be highly recurring at the request level due to serving the same types of requests perpetually. Because of the recurring control flow, the core frontend generates repeating sequences of instruction addresses, so-called temporal instruction streams. For example, in the address sequence $\underline{A,B,C},D,\underline{A,B,C},E$, the subsequence $\underline{A,B,C}$ is a temporal stream. The state-of-the-art instruction prefetchers exploit temporal correlation by recording and replaying temporal instruction streams consisting of instruction block addresses. This way, every prediction made by the prefetcher triggers the fetch of a whole instruction block into L1-I. Because of the high recurrence in the control flow, temporal instruction streams span several hundreds of instruction blocks [14]. As a result, stream-based instruction prefetchers can eliminate over 90% of the L1-I misses, providing near-perfect L1-I hit rates and performance [14].

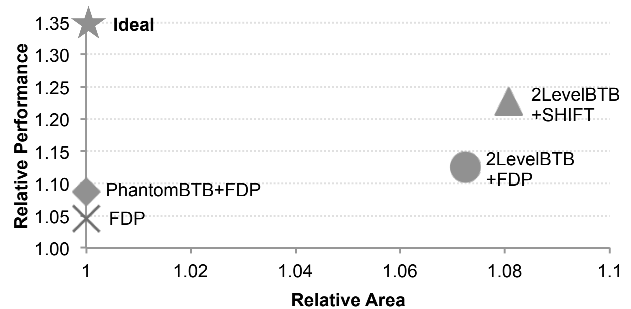

Figure 2: Relative performance & area overhead of conventional instruction-supply mechanisms.

However, the aggregate storage requirements of stream-based prefetchers scale with the application working set size and core count, commonly exceeding 200KB per core. To mitigate the storage overhead, the most recent work, SHIFT [21], proposes sharing the control-flow metadata across the cores running the same server application, thus eliminating inter-core metadata redundancy, and embedding the metadata into the LLC to eliminate dedicated storage for metadata.

2.3 Putting It All Together

We quantify the performance benefits and relative area overheads of all the instruction-supply mechanisms described above for a 16-core CMP running server workloads in Figure 2. Both the performance and area numbers are normalized to that of a core with a 1K-entry BTB without any prefetching. For all the BTB design points (except for the baseline and SHIFT), we leverage FDP for instruction prefetching.

We evaluate an aggressive two-level BTB design composed of a 1K-entry BTB in the first level (1-cycle access latency) and a 16K-entry BTB in the second level (4-cycle access latency). Such a BTB design necessitates around 140KB storage per core, corresponding to 7% of the core area footprint (for an ARM Cortex-A72). For a 16-core CMP, this totals more than 2MB of storage.

We also evaluate PhantomBTB, a state-of-the-art hierarchical BTB design with prefetching. PhantomBTB is comprised of a 1K-entry private BTB backed by a second-level BTB virtualized in the LLC. The second level features 4K temporal groups, each spanning a 64B cache line, for a total LLC footprint of 256KB. While not proposed in the original design, we take inspiration from SHIFT and share the second BTB level across cores executing the same workload in order to reduce the storage requirements. Without sharing, the storage overhead for PhantomBTB would increase by 16x

(i.e., to 4MB) for a 16-core CMP. Sharing the virtualized second-level BTB does not cause any significant reduction in the fraction of misses eliminated.

As Figure 2 shows, the 1K-entry BTB with FDP improves performance by just 5% over the baseline as it incurs frequent BTB misses, thus failing to identify control-flow redirects. PhantomBTB+FDP provides a 9% performance improvement over the baseline despite a large second-level BTB. The underwhelming performance of this configuration is attributed to its low miss coverage in the first-level BTB, which stems from the way PhantomBTB correlates branches (detailed analysis in Section 5), and the delays in accessing the second level of BTB storage in the LLC.

Compared to PhantomBTB+FDP, 2LevelBTB+FDP delivers better performance as the BTB metadata accesses from the second-level BTB are faster compared to the LLC access latency incurred by PhantomBTB. Among the evaluated designs, the highest performance is reached by 2LevelBTB+SHIFT, which combines a dedicated L1-I prefetcher (SHIFT) with a two-level BTB. This configuration improves performance by 22% over the baseline, demonstrating the importance of an effective L1-I prefetcher and underscoring the limitations of FDP. However, 2LevelBTB+SHIFT increases core area by 1.08x due to the high storage footprint of separate BTB and L1-I prefetcher metadata.

Finally, we observe that an Ideal configuration comprised of a perfect L1-I and a perfect single-cycle BTB achieves a 35% performance improvement over the baseline. 2LevelBTB+SHIFT delivers only 62% of the Ideal performance improvement. Since both the L1-I and the BTB in the 2LevelBTB+SHIFT design provide near-perfect hit rates, the performance shortfall relative to Ideal is caused by the delays in accessing the second level of the BTB upon a miss in the first level. Because of the high miss rate of the first-level BTB, these delays are frequent and result in multi-cycle fetch bubbles.

To summarize, the existing frontend designs are far from achieving the desired combination of high performance and low storage cost. Performance is limited by the delays caused in accessing the second BTB level. The high storage overheads arise from maintaining separate BTB and instruction prefetcher metadata. Even worse, because both sets of metadata capture the control-flow history, they cause redundancy within a core. Moreover, because the server cores run the same application, the metadata across cores overlap significantly, causing inter-core redundancy. Eliminating the intra- and inter-core redundancy necessitates the metadata to be unified within a core and shared across cores to maximize the performance benefits harvested from a given area investment.

3. CONFLUENCE: UNIFYING INSTRUCTION-SUPPLY METADATA

Achieving a high-performance instruction-supply path requires effective and timely L1-I and BTB prefetching. Existing L1-I prefetcher and BTB designs

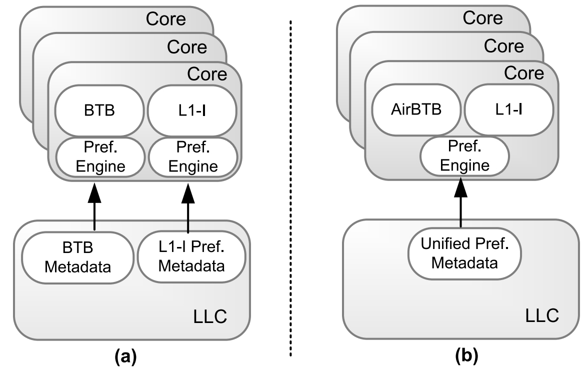

Figure 3: High-level organization of cores around (a) disparate BTB and L1-I prefetcher metadata (b) Confluence with unified and shared prefetcher metadata.

strive to capture the entire control-flow history of an application with their prediction metadata maintained independently as shown in Figure 3 (a).

We now describe Confluence, a specialized instruction-supply path that unifies the control-flow metadata and prefetch mechanisms for the L1-I and the BTB, and shares the metadata across cores running the same application as shown in Figure 3 (b). The single set of metadata maintained by Confluence eliminates the storage redundancy that plagues existing designs. Meanwhile, a stream-based prefetcher provides accurate and timely delivery of instructions and BTB entries ahead of the core's fetch stream.

The state-of-the-art stream-based prefetchers [14, 21] maintain a history of L1-I accesses at block granularity, which they leverage to anticipate the fetch stream well ahead of the core's fetch unit. However, the block-grain history presents a challenge for filling the BTB, which typically tracks individual branch PCs and their targets. To bridge the granularity gap, we introduce AirBTB, a lightweight BTB design whose content mirrors that of the L1-I, thus enabling a single control-flow history to be used for prefetching into both structures.

AirBTB avoids the need for local (i.e., intra-block) control-flow history that existing stream-based prefetchers lack, which would be expensive storage-wise to maintain. Instead, AirBTB inserts all targets of branch instructions in the instruction blocks brought into the L1-I (either by Confluence or on demand by the core) into the BTB. In doing so, AirBTB exploits spatial locality within instruction blocks (i.e., the likelihood of more than one branch instruction being executed in a block), which helps to reduce the number of misses. To minimize tag overheads and BTB write bandwidth, AirBTB adopts a block-based organization.

As Figure 4 depicts, Confluence synchronizes the insertions/evictions into/from AirBTB with the L1-I, thus guaranteeing that the set of blocks present in both structures is identical. As the blocks are proactively fetched from lower levels of the cache hierarchy as requested by the prefetch engine (step 1), Confluence generates the BTB metadata by predecoding the branch type and target displacement field encoded in

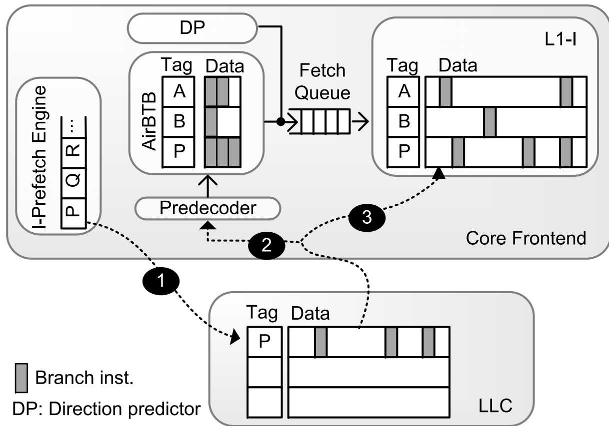

Figure 4: Confluence organization and instruction flow.

the branch instructions in a block and inserts the metadata into AirBTB (step 2) and the instruction block itself into the L1-I (step 3). Finally, Confluence relies on predictor virtualization [6] to store the control-flow metadata used by the prefetcher in the LLC, allowing all cores running a common workload to share metadata.

In the rest of this section, we describe the AirBTB organization, the insertion and replacement operations in AirBTB and how AirBTB operates within the branch prediction unit. We also briefly describe the state-of-the-art stream-based prefetcher, SHIFT [21], which enables Confluence through timely and accurate instruction supply with minimal storage overhead.

3.1 AirBTB Organization

AirBTB is organized as a set-associative cache. Because AirBTB's content is in sync with the L1-I, AirBTB maintains a *bundle* for each block in L1-I. Each bundle comprises a fixed number of *branch entries* that belong to the branch instructions in a block.

In a conventional BTB design, each entry for a branch instruction (or basic block entry) is individually tagged, necessitating to maintain a tag for each individual entry. Because the branches in a bundle in AirBTB belong to the same instruction block, the branch addresses share the same high-order bits, which constitute the address of the block. To exploit the commonality of high-order bits of the branch instruction addresses in a bundle, AirBTB maintains a single tag for a bundle, which is the instruction block address that contains the branches. We refer to this organization as *block-based organization*. The block-based organization amortizes the tag cost across the branches in the same block. Moreover, the block-based organization avoids conflict misses between the branch entries that belong to two different blocks resident in the L1-I.

Figure 5 depicts the AirBTB organization, where each bundle is tagged with the block address and contains entries of three branches, which fall into the same instruction block. The *branch bitmap* in each bundle is a bit vector that identifies the branch instructions in an instruction block. The branch bitmap maintains the knowledge of basic block boundaries within a block, allowing for providing the instruction fetch unit (L1-I), with multiple instructions to fetch in a single lookup. Each branch entry in a bundle contains the offset of the branch instruction within the cache block, the branch type (i.e., conditional, unconditional, indirect, return) and the branch target address (if the branch target is PC-relative, which is mostly the case).

Because each bundle maintains a fixed number of branch entries, instruction blocks with more branch instructions can overflow their bundles. Such overflows happen very rarely if bundles are sized correctly to accommodate all the branches in a cache block in the common case. To handle overflows, AirBTB is backed with a fully-associative *overflow buffer* consisting of a fixed number of entries. Each entry is tagged with full branch instruction address and maintains the branch type and target address. The branch bitmap in a bundle also keeps track of the branch entries in a block that overflowed to the overflow buffer.

3.2 AirBTB Insertions and Replacements

Insertions of the branch entries of a block into AirBTB take place in sync with the insertion of that instruction block into L1-I. By relying on spatial locality, Confluence inserts all the branch entries of an instruction block eagerly into AirBTB. This way, Confluence overprovisions for the worst case where each branch entry might be needed by the branch prediction unit, even though the control flow might diverge to a different block before all the entries in the current block are used by the branch prediction unit.

For each block fetched into the L1-I, Confluence necessitates identifying the branch instructions in a block and extracting the type and relative displacement field encoded in each branch instruction. Confluence relies on predecoding to generate the BTB metadata of the branches in a block before the block is inserted into the L1-I. The predecoder requires a few cycles to perform the branch scan within a cache block before the block is inserted into the L1-I [7, 34]. However, this latency is not on the critical path if the block is fetched into the L1-I earlier than it is needed with the guidance of the instruction prefetcher. Predecoding of an instruction block is performed even if the fetched block is a demand miss, adding a few cycles to the fetch latency. However, this is a rare event when an instruction prefetcher with high miss coverage and timeliness is employed.

As shown in Figure 5 on the left-hand side, for each instruction block fetched into the L1-I, Confluence allocates a bundle in AirBTB and inserts the branch entries into the bundle, while setting the bits of the corresponding branches in the branch bitmap, until the bundle becomes full. If the block overflows its bundle, the entries that cannot be accommodated by the bundle are inserted into the overflow buffer, while their corresponding bits are also set in the bitmap.

Upon the insertion of a new bundle due to a newly fetched instruction block, the bundle evicted from AirBTB corresponds to the instruction block evicted from the L1-I. This way, AirBTB maintains only the entries of the branch instructions resident in the L1-I.

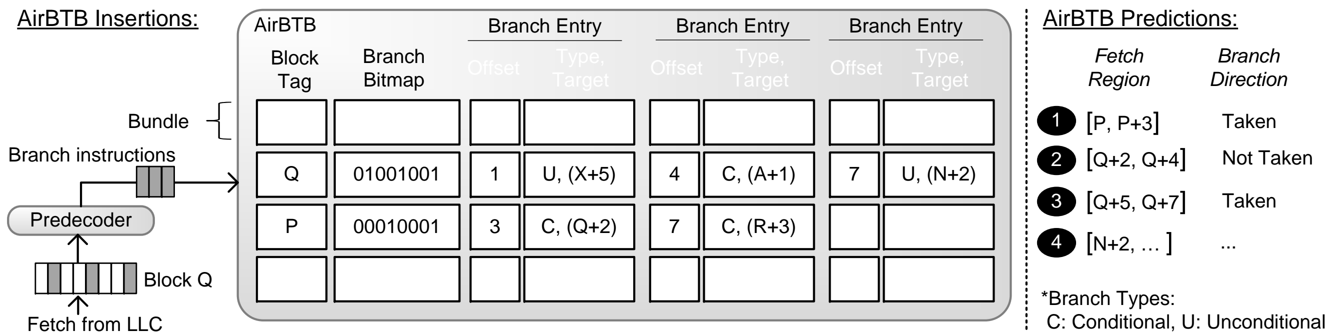

Figure 5: AirBTB organization.

Although the AirBTB organization described above is tailored to RISC ISAs, which have fixed-length instructions, the design can be easily extended to support CISC ISAs with variable-length instructions by maintaining a bit for each byte in an instruction block (instead of one bit per instruction in the bitmap) as the beginning of an instruction can be located in any byte within a block and extending the tag of each branch entry with the length of the instruction.

3.3 AirBTB Operation

Every lookup in AirBTB, in cooperation with the branch direction predictor, provides a *fetch region*, the addresses of the instructions starting and ending a basic block, to be fetched from the L1-I. In this section, we explain how AirBTB performs predictions along with the direction predictor in detail.

Figure 5 (right-hand side) lists the predictions made step by step. We assume that the instruction stream starts with address P. AirBTB first performs a lookup for block P and, upon a match, identifies the first subsequent branch instruction that comes after instruction P by scanning the branch bitmap. In our example, the first branch instruction after P is the instruction at address $P{+}3$. The fetch region, P to $P{+}3$, is sent to the instruction fetch unit and the target address for the branch instruction $P{+}3$ is read out. Next, a direction prediction is made for the conditional branch at address $P{+}3$ by the direction predictor and a lookup is performed for $P{+}3$'s target address $Q{+}2$ in AirBTB. Because the conditional branch is predicted taken, the next fetch region provided by the target address' bundle, $Q{+}2$ to $Q{+}4$, is sent to the fetch unit. Then, because the conditional branch $Q{+}4$ is predicted not taken, the next fetch region is $Q{+}5$ to $Q{+}7$.

If a branch is a return or indirect branch, the target prediction is made by the return address stack or indirect target cache respectively. If a branch indicated by the branch bitmap is not found in one of the branch entries in the bundle, AirBTB performs a lookup for that branch instruction in the overflow buffer. The rest of the prediction operation is exactly the same for branch entries found in the overflow buffer.

If AirBTB cannot locate a block or a branch entry indicated by a bitmap (e.g., due to evictions), it speculatively provides the fetch unit with a fetch region consisting of a predefined number of instructions following the last predicted target address, until it is redirected to the correct fetch stream by the core.

3.4 Prefetcher Microarchitecture

An accurate and timely prefetcher is the key enabler of Confluence. The goal of the prefetcher is to run ahead of the core's fetch stream and not be disturbed by misspeculation or misses in the core's frontend components (L1-I, BTB or branch direction predictor). For that reason, Confluence leverages SHIFT [21], the state-of-the-art stream-based instruction prefetcher. SHIFT relies on the history of previously observed instruction cache accesses, which it replays to fill the frontend with useful instructions. SHIFT stores the control-flow history at instruction-block granularity and amortizes its history storage cost across cores running the same application as described in Section 2.

SHIFT consists of two components to maintain the history of instruction streams; the history buffer and the index table. The history buffer, which is a circular buffer, maintains the history of the L1-I access stream generated by one core at block granularity. The index table provides the location of the most recent occurrence of an instruction block address in the history buffer for fast lookups. The content of these two components are generated by only one core and used by all cores running a common server application in a server CMP.

To enable sharing and eliminate the need for a dedicated history table, the history is maintained in the LLC, leveraging the virtualization framework [6]. Shared history is maintained in the LLC blocks that are reserved for the history. Read and write accesses to the shared history in the LLC compete with normal accesses, but normal accesses are prioritized. The history buffer size allocated (detailed in Section 4.2.1) is sufficient to capture the instruction working set of the server workloads evaluated in this study and hence provides the highest L1-I miss coverage.

Although more cores running a common workload better amortize the cost of history, SHIFT can be easily extended to support multiple workloads. Because the shared history is maintained in the LLC rather than dedicated storage, a disparate instance of history space can be easily allocated in the LLC for each workload in the case of workload consolidation. It has been shown that multiple instances of history provide performance benefits similar to that of a single shared history, as long as there is enough LLC capacity for history instance per workload [21].

A miss in the L1-I initiates a lookup in the index table to find the most recent occurrence of that block address in the history buffer. Upon a hit in the index table, the prefetch engine fetches prediction metadata from the history buffer starting from the location that is pointed to by the index table entry. The prefetch engine uses this metadata to predict future L1-I misses by prefetching the instruction blocks whose addresses are in the metadata. As predictions are confirmed to be correct (i.e., the predicted instruction blocks are demanded by the core), more block addresses are read from the history buffer and used for further predictions.

4. METHODOLOGY

4.1 Baseline System Configuration

We simulate a sixteen-core CMP running server workloads using Flexus [37], a Simics-based full-system multiprocessor simulator. We use the Solaris operating system and run the server workloads listed in Table 1. We run trace-based simulations for profiling and BTB miss coverage studies using traces with 16 billion instructions (one billion instructions per core; one instruction from each core is processed in round-robin fashion) in the steady state of workload execution. For the DSS queries, we use the traces of the full query executions. Our traces consist of both application and operating-system instructions.

For performance comparison, we leverage the SimFlex multiprocessor sampling methodology [37] extending the SMARTS sampling framework [38]. The samples are collected over 10-30 seconds of application execution (from the beginning to the completion of each DSS query). The cycle-accurate timing simulation for each measurement point starts from a checkpoint with warmed architectural state (branch predictors, caches, memory, prefetcher history), and then, runs 100K cycles in the detailed cycle-accurate simulation mode to warm up the queues and on-chip interconnect. The reported measurements are collected from the subsequent 50K cycles of simulation for each measurement point. Our performance metric is the ratio of number of application instructions retired to the total number of cycles (including the cycles spent executing the operating system instructions) as this metric has been shown to accurately represent the overall system throughput [37]. We compute the performance with an average error of less than 5% at the 95% confidence level.

We model a tiled server processor architecture whose architectural parameters are listed in Table 1. Today's commercial processor cores typically comprise 3-5 fetch stages followed by several decode stages [2, 7, 25, 34]. Similarly, we model a core with three fetch stages and fifteen stages in total, representative of an ARM Cortex-A72, which has an area of $7.2mm^2$ when scaled to the 40nm technology [2]. The branch prediction unit is decoupled from the fetch unit with a fetch queue of six basic blocks [31]. The branch prediction unit outputs a fetch region every cycle and enqueues the fetch region into the fetch queue to be consumed by the L1-I. Upon a miss in the BTB, a predefined number of instructions (eight) subsequent to the last fetch address predicted are enqueued in the fetch queue as the next fetch region. Misfetches due to BTB misses are identified right after the fetch stage, in the first decode stage, which corresponds to a misfetch penalty of 4 cycles.

4.2 Instruction-Supply Mechanisms

We compare Confluence (AirBTB coupled with SHIFT) against fetch-directed prefetching leveraging three different BTB designs, namely the *IdealBTB* with 16K entries and 1-cycle latency, realistic conventional *2LevelBTB*, and *PhantomBTB* as a two-level BTB design allowing for sharing the second-level BTB across cores. We also couple SHIFT with these three different BTB designs and compare with AirBTB to decouple the effects of instruction prefetching and BTB design. The area overheads of these different design points are determined with CACTI 6.5 [28] in 40nm technology assuming a 48-bit virtual address space.

4.2.1 Instruction Prefetchers

Shared History Instruction Fetch (SHIFT): SHIFT, described in detail in Section 3.4, tuned for maximum L1-I miss coverage, requires a 32K-entry history buffer (204KB) virtualized in the LLC and around 240KB of index storage embedded in the tag array of the LLC. With history buffer entries embedded in the existing LLC blocks, which results in a negligible performance overhead, SHIFT's only meaningful area overhead stems from the extension of the LLC tag array for the index pointers, which is estimated to be $0.96mm^2$. This corresponds to $0.06mm^2$ area overhead per core.

Fetch-Directed Prefetching (FDP): The branch prediction unit is decoupled from the L1-I with a queue that can accommodate six basic blocks (determined experimentally to maximize performance) and the branch prediction unit outputs a basic block every cycle, as described above. For each fetch region enqueued in the fetch queue, prefetch requests are issued for the instruction blocks that fall into the fetch region, if they are not already in the L1-I. Because FDP relies on the existing branch predictor metadata, it does not incur any additional storage overhead.

4.2.2 BTB Designs

AirBTB: The final AirBTB design maintains 512 bundles in total (same as the number of blocks in the L1-I) and 3 branch entries per bundle. Because the instruction size is 4B, each 64B block has 16 instructions in total. So, each bundle maintains a 16-bit branch bitmap. Each branch entry has a 4-bit offset, 2-bit branch type, and 30-bit target field. The overflow buffer has 32 entries. The final AirBTB design requires 10.2KB of storage, incurring $0.08mm^2$ area overhead per core (AirBTB's sensitivity to design parameters is evaluated in Section 5.3). AirBTB's area footprint is comparable to a 1K-entry conventional BTB with a victim buffer or PhantomBTB's first level. In total, Confluence, AirBTB backed by SHIFT ($0.06mm^2$ per core), incurs only $0.06mm^2$ area overhead per core.

Conventional BTB: To provide multiple instructions

Cores	UltraSPARC III ISA, ARM Cortex-A72-like: 3GHz, 3-way OoO, 128-entry ROB, 32-entry LSQ
Branch Prediction Unit	Hybrid branch predictor (16K-entry gShare, Bimodal, Meta selector) 1K-entry indirect target cache 64-entry return address stack 1 basic-block prediction per cycle
L1 I&D Caches	32KB, 4-way, 64B blocks 2-cycle load-to-use latency, 8 MSHRs
L2 NUCA Cache	512KB per core, unified, 16-way, 64B blocks, 16 banks, 6-cycle hit latency
Interconnect	4x4 2D mesh, 3 cycles per hop
Main memory	45ns access latency

OLTP - Online Transaction Processing (TPC-C)	
DB2	*IBM DB2 v8 ESE Database Server* 100 warehouses (10GB), 2GB buffer pool
Oracle	*Oracle 10g Enterprise Database Server* 100 warehouses (10GB), 1.4 GB SGA
DSS - Decision Support Systems (TPC-H)	
Qry 2, 8, 17, 20	*IBM DB2 v8 ESE* 480MB buffer pool, 1GB database
Media Streaming	
Darwin	*Darwin Streaming Server 6.0.3* 7500 clients, 60GB dataset, high bitrates
Web Frontend (SPECweb99)	
Apache	*Apache HTTP Server v2.0* 16K connections, fastCGI, worker threading model

Table 1: Architectural system and application parameters.

to be fetched with a single BTB lookup, prior work proposed organizing the BTB to provide a fetch range (i.e., basic-block range) [31, 39]. Each BTB entry is tagged with the starting address of a basic block (excluding the low-order bits used for indexing) and maintains the target address of the branch ending the basic block (30-bit PC-relative displacement; the longest displacement field in the UltraSPARC III ISA), the type of the branch instruction ending the basic block (2 bits), and a number of bits to encode the fall-through address (the next instruction after the branch ending the basic block). We found that the fall-through distance can be encoded with 4 bits for 99% of the basic blocks. We evaluate four-way set-associative BTB organizations as we did not see any additional benefits in hit rate from increasing the associativity. In our comparisons of conventional BTB with AirBTB, we augment the conventional BTB with a 64-entry victim buffer (around 0.6KB). The 1K-entry conventional BTB with a 64-entry victim buffer used as the baseline requires 9.9KB of storage in total ($0.08mm^2$) and has 1-cycle latency. For the conventional two-level BTB configuration, we use a 16K-entry BTB as the second level, which is 140KB and occupies $0.6mm^2$ area per core and has a latency of 4 cycles.

PhantomBTB: We tune the PhantomBTB design for our benchmark suite to maximize the number of misses eliminated. We use a 1K-entry conventional first-level BTB with a 64-entry prefetch buffer (9.9KB in total and $0.08mm^2$). The storage cost of PhantomBTB's first-level is almost the same as AirBTB.

For the virtualized prefetcher history, we pack six BTB entries (the maximum possible) in an LLC block and dedicate 4K LLC blocks (256KB assuming 64B blocks). We did not see any significant change in the percentage of misses eliminated with a bigger history. The region size used to tag each temporal group (i.e., LLC block) is 32 instructions.

Although the original PhantomBTB design maintains a private prefetcher history per core, we evaluate a shared prefetcher history as we run homogeneous server workloads where each core runs the same application code, thus is amenable to BTB sharing. This enables a fair comparison between PhantomBTB and Confluence, as Confluence relies on shared history for instruction prefetching. It is important to note that sharing the prefetcher history has negligible (i.e., less than 2%) effect on the percentage of misses eliminated by PhantomBTB as compared to the private prefetcher history that was originally proposed. Because the shared PhantomBTB prefetcher history is embedded in the existing LLC data blocks and its negligible performance impact due to aggregate capacity reduction is accounted for (like SHIFT), we assume that PhantomBTB does not incur any storage overhead.

5. EVALUATION

Confluence unifies the disparately maintained instruction-supply metadata for BTB and L1-I by relying on a stream-based prefetcher, whose effectiveness has already been demonstrated for L1-I [21]. Our focus is on demonstrating that while BTB and L1-I prefetcher designs have been proposed in isolation, their unification enables better performance and lower cost than configurations that maintain them separately.

5.1 Performance and Area Comparison

We first compare the performance benefits and associated area overheads of Confluence with conventional frontend designs discussed in Section 2 in Figure 6. All the performance and area numbers are normalized to a core with conventional BTB with 1K entries and 64-entry victim buffer as described in Section 4. As Figure 6 demonstrates, Confluence is the closest design point to Ideal by delivering 85% of the performance improvement delivered by the Ideal configuration (i.e., perfect L1-I and BTB achieving 35% performance improvement on average) with 1% storage area overhead per core (including private BTB's and SHIFT's per-core storage overhead). Confluence delivers higher performance as compared to all other design points detailed in Section 2.3 thanks to its timely and accurate insertions of instructions into the L1-I and branch entries into the BTB.

It is instructive to compare the 2LevelBTB+SHIFT and Confluence designs. Both feature SHIFT as the instruction prefetcher and a BTB with a high hit rate. By eliminating the redundant metadata in the 2LevelBTB+SHIFT design (~140KB for the second-level BTB), Confluence achieves a considerably lower storage footprint. Performance-wise, Confluence is 8% better, despite the fact that 2LevelBTB+SHIFT delivers a slightly higher hit rate (detailed analysis in Section 5.2). The reason for Confluence's superior performance is timeliness, as the BTB is filled proactively ahead of

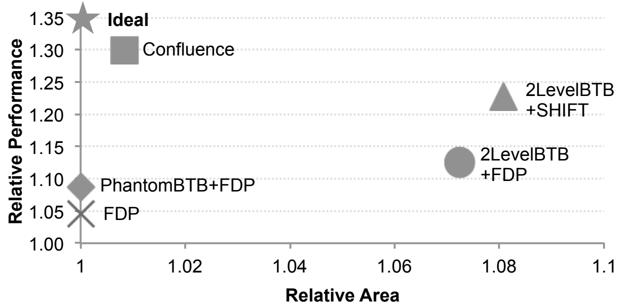

Figure 6: Confluence's performance benefits and area savings compared to conventional instruction-supply mechanisms.

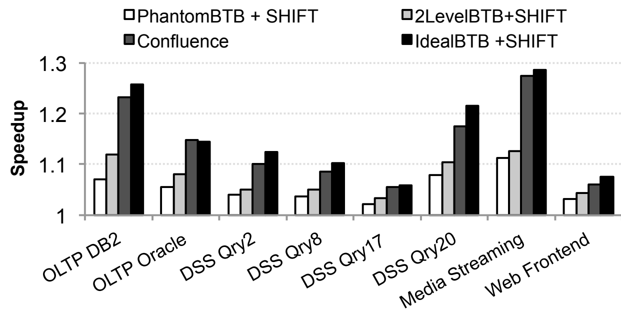

Figure 7: Speedup of various BTB designs over 1K-entry conventional BTB when coupled with SHIFT for instruction prefetching.

the fetch stream. In contrast, decoupled BTB designs (including both 2LevelBTB and PhantomBTB) trigger BTB fills only when a miss is discovered in the first-level BTB, thus exposing the core to the latency of the second level.

In order to isolate the benefits of timely insertions of entries into BTB from timely insertions of instruction blocks into L1-I, we also study the performance of the various BTB designs when coupled with SHIFT for instruction prefetching. We evaluate PhantomBTB, 2LevelBTB (which corresponds to 2LevelBTB+SHIFT in Figure 6), and Confluence, and compare their performance to an IdealBTB, which has the same capacity as the 2LevelBTB and 1-cycle access latency.

As Figure 7 shows, the lowest performance is achieved by PhantomBTB due to its low miss coverage in the first-level BTB as discussed in the next section. Although the 2LevelBTB design achieves the same hit rate as Ideal, it provides only 51% of Ideal's speedup due to frequent stalls caused by frequent L1-BTB misses, which expose the L2-BTB's access latency. In contrast, Confluence attains 90% of the speedup achieved by IdealBTB+SHIFT, highlighting Confluence's ability to not only provide high miss coverage in the frontend, but also to do so in a timely manner.

As noted in Section 2.1, OLTP on Oracle is the one workload that benefits from a BTB larger than 16K entries. As a result, IdealBTB with 16K entries exhibits some capacity misses on OLTP on Oracle. Because AirBTB eliminates more misses than IdealBTB (as shown in Figure 9), AirBTB provides slightly higher performance than IdealBTB on OLTP on Oracle.

In summary, the main reason why Confluence outperforms 2LevelBTB is because AirBTB's content is proactively populated by SHIFT in Confluence, whereas in 2LevelBTB, a first-level BTB miss is reactively served by the second-level BTB (i.e., SHIFT only populates the instruction cache, not the BTB). As a result, 2LevelBTB frequently exposes the core to the latency of the second-level BTB (4 cycles), while Confluence hides this latency with the help of the prefetcher. Hence, Confluence's performance advantage is due to serving most of the accesses by AirBTB with a 1-cycle latency.

5.2 Dissecting the AirBTB Benefits

To eliminate most of the misses within a given BTB storage budget, AirBTB modifies the baseline BTB design in several ways. Figure 8 shows how much each design decision helps to improve the miss coverage over a conventional BTB design with 1K entries by employing the mechanisms proposed for AirBTB step by step. First, AirBTB can afford more entries within a given storage budget as compared to the 1K-entry conventional BTB, because it amortizes the cost of tags across the entries in the same instruction block, eliminating 18% of the misses (*Capacity*). Second, because AirBTB eagerly identifies the branch instructions in a block upon a BTB miss in an instruction block and eagerly installs their entries before the branches are actually executed, it can eliminate 57% more misses on average (*Spatial Locality*). Third, by relying on the instruction prefetcher, AirBTB can eliminate 7% more misses by eliminating a BTB miss even if the first instruction touched in a missing block is a branch (*Prefetching*). Finally, the block-based organization employed by AirBTB guarantees that the blocks in the BTB are in sync with the L1-I, so that the BTB entries of two L1-I-resident blocks do not conflict, which provides 11% additional miss coverage (*Block-Based Organization*).

It is important to note that the 7% coverage benefit of prefetching understates the prefetcher's importance by focusing just on the hit rate of AirBTB and ignoring the timeliness aspect. In reality, prefetching is essential to hide the access latency to metadata and instruction blocks fetched from the lower levels of the hierarchy. Not hiding the long-latency accesses to lower levels of the cache hierarchy leads to frequent frontend stalls, significantly reducing the performance as quantified for other design points in Section 5.1.

Figure 9 shows the fraction of BTB misses eliminated by AirBTB, PhantomBTB and a 16K-entry conventional BTB over the 1K-entry conventional BTB. PhantomBTB eliminates only 61% of the misses on average, compared to AirBTB's 93% miss coverage. The discrepancy in the coverage is attributed to two major differences between the two designs.

First, because AirBTB amortizes the cost of the tags across the branch entries within the same instruction block, it can maintain more BTB entries as compared to PhantomBTB's first-level, which is a conventional BTB organization, within the same storage budget.

More importantly, PhantomBTB forms temporal

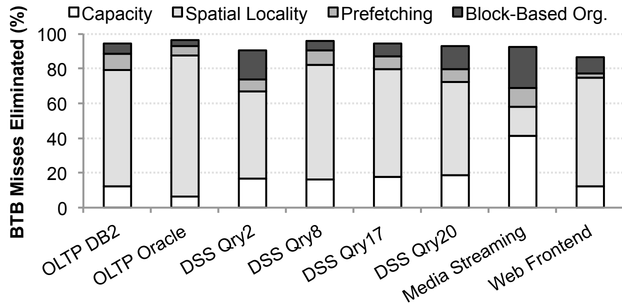

Figure 8: Breakdown of AirBTB miss coverage benefits over 1K-entry conventional BTB.

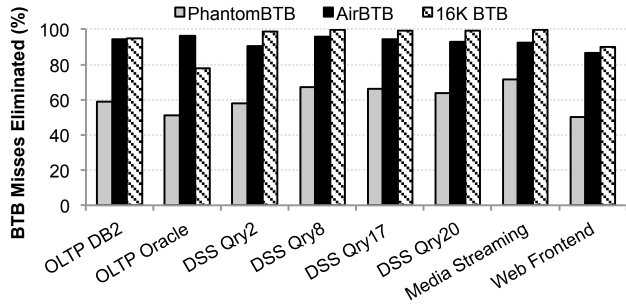

Figure 9: PhantomBTB, AirBTB and 16K-entry conventional BTB miss coverages over a 1K-entry conventional BTB.

groups of BTB entries that consecutively miss in the L1-BTB by packing a number of BTB entries in an LLC block and prefetching those entries into the first-level upon a miss in the first-level. In PhantomBTB, the BTB entries that fall into a temporal group depend heavily on the branch outcomes in the local control flow. Small divergences in the control flow significantly affect the content of the temporal groups and reduce the likelihood of same sets of branches always missing in the BTB consecutively. Moreover, because PhantomBTB maintains fixed-sized temporal groups of BTB entries, as opposed to long arbitrary-length streams as in SHIFT, its lookahead is limited to only a few BTB entries upon each L1-BTB miss.

In contrast, the stream-based prefetcher leveraged by Confluence is a better predictor of future control flow as it relies on coarse-grain temporal streams of instruction block addresses that often cover as many as a few hundred instruction blocks per stream. Hence, the stream prefetcher's highly accurate control-flow prediction at macro level coupled with AirBTB's eager insertion and block-based organization, which uncover spatial locality, provide a higher miss coverage than PhantomBTB. We conclude by noting that the coverage reported for PhantomBTB is the highest coverage we could attain and does not benefit from further increases in the size of its history storage.

Overall, AirBTB closely approaches the miss coverage of the 16K-entry conventional BTB, which provides 95% miss coverage on average, without incurring its high per-core storage overhead.

5.3 Sensitivity to Design Parameters

To determine the optimal number of branch entries to maintain in each bundle in AirBTB, we first characterize the distribution of branches within instruction blocks for each block that is demand-fetched into the L1-I during execution. Table 2 lists the average number of branch instructions in demand-fetched blocks (i.e., *static*) as well as the total number of branches actually executed (and taken) during the residency of cache blocks in the L1-I (i.e., *dynamic*). We show only the average values for the four DSS queries due to space constraints. Demand-fetched instruction blocks contain 3.5 static branch instructions and 1.5 dynamic branch instructions on average.

In light of the branch behavior characterization, we examine different AirBTB configurations by varying the bundle size (i.e., the number of branch entries in a bundle) and the size of the overflow buffer. The total number of bundles is fixed as AirBTB maintains only the instruction blocks resident in the instruction cache at any given point in time. Figure 10 shows AirBTB's miss coverage over the 1K-entry conventional BTB.

Workloads	OLTP DB2	OLTP Oracle	DSS Qrys	Media Streaming	Web Frontend
Static	3.6	2.5	3.4	3.5	4.3
Dynamic	1.4	1.6	1.4	1.5	1.5

Table 2: Branch density in blocks.

As the number of branch entries in a bundle increases, the overall miss coverage increases for all workloads as each bundle can accommodate more entries. 50% of instruction blocks contain up to three branches on average. As a result, an AirBTB configuration with three branch entries per bundle is able to capture the branch footprint of half of the instruction blocks at any given point in time. Unfortunately, such an AirBTB configuration (B:3, OB:0) has a higher miss rate than the baseline 1K-entry BTB for some of the workloads.

When AirBTB with three branch entries per bundle is backed with an overflow buffer (B:3, OB:32), it becomes effective at eliminating the misses as compared to a 1K-entry conventional BTB. For the AirBTB configuration with three branch entries per bundle, we found the optimal overflow buffer size to be 32 entries. We did not see any significant improvement in coverage beyond 32 overflow buffer entries. Such a configuration (B:3, OB:32) provides 93% miss coverage on average.

Finally, compared to the AirBTB configuration (B:3, OB:32), the AirBTB configurations with four entries per bundle (B:4, OB:32) requires more storage (around 2KB), while increasing the miss coverage by only 2%. For this reason, we use the AirBTB configuration with three branch entries per bundle and 32-entry overflow buffer (B:3, OB:32) as the final AirBTB design.

6. RELATED WORK

Branch target buffer is the key component allowing the branch prediction unit to run ahead of the core and provide the core with a continuous instruction stream [26, 29, 36]. Because the branch predictor is on the critical path, a large BTB with several cy-

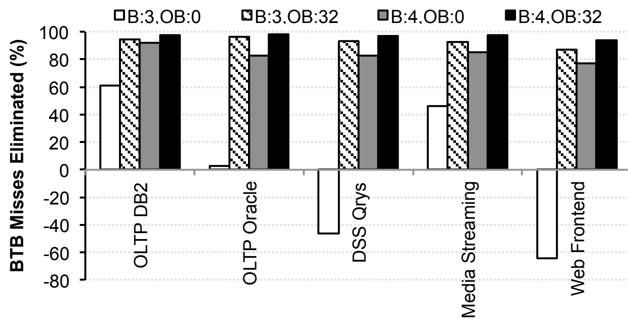

Figure 10: Miss coverage for various AirBTB configurations (B = branch entries in a bundle, OB = branch entries in the overflow buffer).

cles of latency greatly penalizes the instruction-delivery rate. One way of reducing BTB's capacity requirement is to maintain partial BTB tags instead of full tags [12], making BTB entries susceptible to aliasing. Another way is to maintain only the offsets of the fall-through and target addresses from the basic-block address instead of their full addresses as the distance between the basic-block address and the fall-through or target address is expected to be small [23, 31]. Although these compression techniques help to reduce the BTB capacity requirements, they cannot mitigate the number of individual entries that need to be maintained to capture the entire working set of an application, which is the fundamental problem for server workloads.

To mitigate the latency of large predictor tables, hierarchical branch predictors provide low access latencies with a smaller but less accurate first-level predictor and leverage a larger but slower second-level predictor to increase accuracy [7, 19, 29, 33]. The second-level table overwrites the prediction of the first-level table in the case of a disagreement at a later stage.

While hierarchical predictors provide a trade-off between accuracy and delay, they still incur high latencies to access lower levels. To hide the latency of lower-level predictor tables, several studies proposed prefetching predictor metadata from the lower-level table into the first-level table. PhantomBTB exploits the temporal correlation between BTB misses as misses to a group of entries are likely to recur together in the future due to the repetitive control flow [5]. Emma et al. also propose spilling groups of temporally correlated BTB entries to the lower levels of the cache hierarchy and tagging each group with the instruction block address of the first instruction in the group [10]. Upon a miss in the instruction cache, the corresponding BTB entry group can be loaded from the secondary table into the first level.

In a similar vein, bulk preload [3] maintains per-core two-level BTBs and fetches a group of BTB entries that belong to a fixed-size code region (i.e., 64 consecutive instruction blocks) upon a miss in that region from the second level into the first level. Bulk preload fails to eliminate the first miss in every region, because the first miss is the prefetch trigger. Moreover, because consecutive accesses to BTB do not frequently fall into large contiguous code regions because of divergences (the same reason why next-line prefetchers are ineffective at eliminating all instruction cache misses), there is only limited spatial locality that can be exploited.

Although various prefetching techniques for hierarchical BTBs aim to reduce the latency to the second level, they do not address the storage overhead associated with intra- or inter-core redundancy in metadata. To eliminate or mitigate the BTB storage overhead, cores with hardware multithreading [34, 35] employ predecoding to scan the branches in the instruction blocks that are fetched into L1-I, precompute the target addresses of the branches, and modify the branch instructions to store the lower bits of the target address before they are inserted into the instruction cache. This way, the target address of a taken branch is formed with a simple concatenation of the branch PC and the low order bits of the target address, right after the instruction is fetched from the L1-I. In the absence of multiple hardware threads to cover the latency, this scheme significantly hurts single-threaded performance by requiring several cycles after fetch to identify branches within a cache block and compute their targets. To mitigate the resulting fetch bubbles, some processors employ small BTBs [34]; however, such designs still expose the core to the high latency of target computation whenever the small BTB misses.

Prior proposals to unify BTB and L1-I [4] also suffer from multi-cycle latencies of instruction caches and have no flexibility in the number BTB entries that can be maintained per block. Finally, next cache line and set prediction [8] to predict the location of the target of a taken branch in the L1-I delivers low prediction accuracy in the presence of frequent L1-I misses.

7. CONCLUSION

Large instruction working sets of server workloads are beyond the reach of practical BTB and L1-I sizes due to their strictly low latency requirements. Frequent misses in BTB and L1-I result in misfetches and instruction fetch stalls, which greatly hurt the performance of server workloads. Existing proposals on mitigating frontend stalls rely on discrete prefetchers for BTB and L1-I, whose metadata essentially capture the same control-flow information.

This work proposed Confluence; a new frontend design, which synchronizes the BTB and L1-I content to leverage a single prefetcher and unified block-grain history metadata to fill both BTB and L1-I. By relying on an autonomous history-based prefetcher, Confluence avoids the timeliness problem of hierarchical BTB designs that expose the core to the latency of accessing the second level of BTB storage. Confluence eliminates 93% of BTB misses and 85% of L1-I misses, providing 85% of the speedup possible with a perfect L1-I and BTB at a cost of only 1% of the core area.

8. ACKNOWLEDGEMENTS

The authors would like to thank Joel Emer, André Seznec, Paolo Ienne, Djordje Jevdjic, Onur Kocberber, Stavros Volos, Alexandros Daglis, Javier Picorel, and

the anonymous reviewers for their insightful feedback on earlier drafts of this paper. This work was partially supported by the IBM Ph.D. Fellowship award.

9. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a modern processor: Where does time go?" in *The VLDB Journal*, Sep. 1999, pp. 266–277.

[2] ARM Processor Technology Update, www.arm.com.

[3] J. Bonanno, A. Collura, D. Lipetz, U. Mayer, B. Prasky, and A. Saporito, "Two level bulk preload branch prediction," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2013.

[4] B. K. Bray and M. J. Flynn, "Strategies for branch target buffers," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, 1991.

[5] I. Burcea and A. Moshovos, "Phantom-BTB: A virtualized branch target buffer design," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[6] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi, "Predictor virtualization," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[7] M. Butler, L. Barnes, D. Sarma, and B. Gelinas, "Bulldozer: An approach to multithreaded compute performance," *Micro, IEEE*, vol. 31, no. 2, pp. 6–15, March 2011.

[8] B. Calder and D. Grunwald, "Next cache line and set prediction," in *Proceedings of the International Symposium on Computer Architecture*, 1995.

[9] Cavium ThunderX ARM Processors, www.cavium.com.

[10] P. Emma, A. Hartstein, B. Prasky, T. Puzak, M. Qureshi, and V. Srinivasan, "Context look ahead storage structures," Feb. 26 2008, IBM, US Patent 7,337,271.

[11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the Annual International Symposium on Computer Architecture*, 2011.

[12] B. Fagin and K. Russell, "Partial resolution in branch target buffers," in *Proceedings of the Annual International Symposium on Microarchitecture*, 1995.

[13] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[14] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *Proceedings of the International Symposium on Microarchitecture*, 2011.

[15] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *Proceedings of the International Symposium on Microarchitecture*, 2008.

[16] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward dark silicon in servers," *Micro, IEEE*, vol. 31, no. 4, July 2011.

[17] ——, "Reactive NUCA: Near-optimal block placement and replication in distributed caches," in *Proceedings of the International Symposium on Computer Architecture*, 2009.

[18] R. B. Hilgendorf, G. J. Heim, and W. Rosenstiel, "Evaluation of branch-prediction methods on traces from commercial applications," *IBM J. Res. Dev.*, vol. 43, no. 4, pp. 579–593, Jul. 1999.

[19] D. A. Jiménez, S. W. Keckler, and C. Lin, "The impact of delay on the design of branch predictors," in *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*, 2000.

[20] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the International Symposium on Computer Architecture*, 2015.

[21] C. Kaynak, B. Grot, and B. Falsafi, "SHIFT: Shared history instruction fetch for lean-core server processors," in *Proceedings of the Annual International Symposium on Microarchitecture*, 2013.

[22] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker, "Performance characterization of a Quad Pentium Pro SMP using OLTP workloads," in *Proceedings of the International Symposium on Computer Architecture*, 1998.

[23] R. Kobayashi, Y. Yamada, H. Ando, and T. Shimada, "A cost-effective branch target buffer with a two-level table organization," in *Proceedings of the International Symposium of Low-Power and High-Speed Chips*, 1999.

[24] A. Kolli, A. Saidi, and T. F. Wenisch, "RDIP: Return-address-stack directed instruction prefetching," in *Proceedings of the Annual International Symposium on Microarchitecture*, 2013.

[25] K. Krewell and L. Gwennap, "Silvermont energizes Atom," *Microprocessor Report*, vol. 27, no. 5, pp. 12–17, May 2013.

[26] J. Lee and A. Smith, "Branch prediction strategies and branch target buffer design," *Computer*, vol. 17, no. 1, pp. 6–22, Jan 1984.

[27] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-out processors," in *Proceedings of the International Symposium on Computer Architecture*, 2012.

[28] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proceedings of the International Symposium on Microarchitecture*, 2007.

[29] C. Perleberg and A. Smith, "Branch target buffer design and optimization," *Computers, IEEE Transactions on*, vol. 42, no. 4, pp. 396–412, Apr 1993.

[30] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso, "Performance of database workloads on shared-memory systems with out-of-order processors," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[31] G. Reinman, T. Austin, and B. Calder, "A scalable front-end architecture for fast instruction delivery," in *Proceedings of the Annual International Symposium on Computer Architecture*, 1999.

[32] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*, 1999.

[33] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design tradeoffs for the Alpha EV8 conditional branch predictor," in *Proceedings of the Annual International Symposium on Computer Architecture*, 2002.

[34] M. Shah, R. Golla, G. Grohoski, P. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja, "Sparc T4: A dynamically threaded server-on-a-chip," *Micro, IEEE*, vol. 32, no. 2, pp. 8–19, March 2012.

[35] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams, "IBM POWER7 multicore server processor," *IBM Journal of Research and Development*, vol. 55, no. 3, pp. 1:1–1:29, May 2011.

[36] E. Sussenguth, "Instruction sequence control," Jan. 26 1971, US Patent 3,559,183.

[37] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, July-Aug. 2006.

[38] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the International Symposium on Computer Architecture*, 2003.

[39] T.-Y. Yeh and Y. N. Patt, "A comprehensive instruction fetch mechanism for a processor supporting speculative execution," in *Proceedings of the Annual International Symposium on Microarchitecture*, 1992.