Weeding out Front-End Stalls with Uneven Block Size Instruction Cache

Roman Brunner
Norwegian University of Science and Technology (NTNU), Norway

Rakesh Kumar
Norwegian University of Science and Technology (NTNU), Norway

***Abstract*—The core front-end remains a critical bottleneck in modern server workloads owing to their multi-MB instruction footprints stemming from deep software stacks. Prior work has mainly investigated instruction prefetching and cache replacement policies to mitigate this bottleneck. In this work, we take an orthogonal approach and analyze instruction cache storage efficiency. Our analysis shows that, on average, about 60% of the bytes in a cache block are never accessed before the block is evicted from the instruction cache. This represents a huge storage inefficiency that more than halves the effective cache capacity. We observe that this inefficiency is caused by the fixed cache block sizes which are unable to accommodate the varying spatial locality inherent in the instruction stream. To mitigate this inefficiency, we propose Uneven Block Size (UBS) instruction cache, which supports different cache block sizes in a cache set. Our evaluation shows that UBS cache improves the storage efficiency by 32 percentage points over the baseline instruction cache. Further, by supporting uneven block sizes, UBS cache accommodates more than twice the number of blocks than a conventional cache within a given storage budget. Overall, the additional blocks combined with the better storage efficiency result in UBS cache approaching the performance of a 64KB conventional cache on a set of server workloads while requiring a storage budget similar to a 32KB conventional cache.**

I. Introduction

Server applications have seen a continuous demand to provide increasingly more and complex functionality. Managing this complexity necessitates implementing the functionality through deep software stacks. For example, a typical server request may need to touch a web server, database, storage and network I/O, logging and monitoring code etc. Such deep software stacks result in application code footprints reaching tens of megabytes.

At the microarchitectural level, such massive code footprints overwhelm the capacity of private per-core front-end structures, such as instruction cache (L1-I), which are optimized for low latency rather than high storage capacity. Consequently, L1-I experiences frequent misses that stall the instruction fetch for tens of cycles while the miss is filled from lower-level caches. Such frequent front-end stalls severely limit instruction delivery to the core, thus resulting in poor performance.

Though the front-end bottleneck is a well-established problem [1]–[3], prior work has mostly explored prefetching [4]–[9] and replacement policies [10] to address it. In this work, we investigate an orthogonal aspect, namely cache storage efficiency, to mitigate this bottleneck. We define storage efficiency as the fraction of bytes in the L1-I that are actually accessed by the core. At a cache block level, storage efficiency is the fraction of bytes in a cache block that are accessed before the block is evicted from the L1-I.

We analyze the storage efficiency of a 32KB L1-I with 64-byte blocks on a set of server workloads. Our analysis reveals that for about 61% of cache blocks, only 32 or fewer bytes are accessed before they are evicted. Further, for 11% (up to 30%) of the cache blocks, a meagre 8 or fewer bytes are accessed before eviction. Finally, there are only 12% of cache blocks that see all bytes accessed by the core. Overall, these results show that a large fraction, 60% on average, of L1-I storage space is occupied by unused bytes, thus drastically reducing the effective L1-I capacity.

We observe that the poor storage efficiency of L1-I is a consequence of the fixed cache block size, typically 64 bytes. Such fixed-sized cache blocks are unable to accommodate the spatial locality in the instruction stream, which varies based on branch directions and basic block sizes. This varying spatial locality results in some cache blocks experiencing 100% utilization while others seeing only 6.25%. Further, high L1-I MPKI (misses per kilo instructions) exaggerates the storage inefficiency as it reduces the duration a cache block stays in L1-I, thus reducing the probability of a byte being accessed before the block gets evicted.

One possibility for improving L1-I storage efficiency is to increase spatial code locality, and researchers have proposed a number of compiler optimizations such as hot/cold splitting [11], partial inlining [12], code layout optimizations [13], [14], feedback directed optimizations [15], etc. in that direction. However, a recent study from Google [4] shows that, despite all these optimizations, even among their hottest and well-optimized functions, more than 50% of the code is completely cold. Further, the study shows that the hot and cold regions of code are frequently tightly mixed with each other. At the microarchitectural level, it means that the cold code is brought into the L1-I along with the hot code. However, the core rarely fetches the cold code before the cache block containing it is evicted, thus resulting in high storage inefficiency.

In this work, instead of optimizing the code layout to increase spatial locality, we seek to design an instruction cache that can gracefully accommodate varying spatial locality, thus enabling high storage efficiency. To that end, we propose to abandon the idea of having fixed-size cache blocks, i.e., the root cause of the storage inefficiency. Instead, based on our

observation of varying spatial locality, we propose to adopt uneven cache block sizes and size different ways of a n-way set associative cache to hold progressively larger blocks. For example, cache blocks in way-1 may hold 4 bytes, way-2 cache blocks may hold 8 bytes, and so on. In doing so, we match the spatial locality to the way that provides the best fit.

Unlike a conventional L1-I where an incoming 64-byte block from L2 cache can be placed anywhere in a set, an L1-I with uneven block sizes must first predict the useful bytes in the incoming block and then choose an appropriate way to place them. To do so, we propose a simple predictor that filters out the bytes that are unlikely to be accessed during the lifetime of a block in L1-I while the remaining bytes are installed in the cache. The predictor is implemented as a small direct-mapped cache, and an incoming block from L2 cache is always first placed here. Further, the predictor has a bit-vector per block to track whether a byte or instruction, depending on ISA, in the block has been accessed. When a block is evicted from the predictor, the accessed bytes are moved to one of the cache ways, which is selected based on the number of accessed bytes, while the unaccessed bytes are discarded.

This paper introduces Uneven Block Size (UBS) cache to maximize storage efficiency by placing only the useful bytes in L1-I. UBS Cache is a set associative L1-I with unevenly sized ways to accommodate the varying spatial locality of the instruction stream. It is aimed at keeping only the hot code (useful bytes) in the cache while weeding out the cold code (unused bytes) that a conventional L1-I keeps due to its rigid block size. Our evaluation shows that, for a given storage budget, UBS cache supports more than twice the number of cache blocks than a conventional L1-I. Further, it achieves 32 percentage points better storage efficiency than a conventional cache on a set of server workloads. This work makes the following key contributions:

- We show that the fixed size cache blocks result in poor storage efficiency as 60% of the bytes in a cache block are never accessed before the block is evicted from L1-I.
- Our analysis shows large spatial locality variation in the instruction stream, implying that a fixed-sized cache block is not a good fit for capturing this locality.
- We introduce UBS cache, a simple and highly storage-efficient cache organization with unevenly sized ways that gracefully accommodate the varying spatial locality.
- We present a useful byte predictor that weeds out the cold code and places only the hot code in L1-I.
- We demonstrate that, by accommodating more than 2x cache blocks and providing better storage efficiency, UBS cache approaches the performance of a 64KB conventional L1-I while requiring a storage budget similar to a 32KB L1-I.

II. Background

A. Front-end Bottleneck

Front-end stalls are a long-standing bottleneck in servers, with the first characterizations appearing in late 90's [1]–[3]. While the earlier work showed that databases [1] and online transaction processing [2] workloads suffer from front-end stalls due to instruction cache misses, recent work [4], [16] shows that modern scale-out workloads such as web search, media streaming, in-memory analytics, web and data serving etc. are inflicted with the same bottleneck. Nearly a decade ago, Google published a study [17], profiling their live datacenter with more than 20,000 machines and showing that the core front-end was stalled for 15-30% of the execution time even in their most optimized applications. Their recent results [4] further emphasize that the front-end bottleneck will continue to be a serious performance limiter as they show the front-end stalls to be responsible for nearly 25% of the execution time.

The front-end remains a bottleneck irrespective of whether the server applications are implemented as monolithics or a collection of microservices [18], [19]. A recent study [19] from Meta shows that their microservices are stalled at the front-end for up to a whopping 37% of execution time. Furthermore, even the short-running serverless functions are also severely bottlenecked by the front-end stalls as a recent study [20] demonstrates that more than 50% of execution time is spent on them.

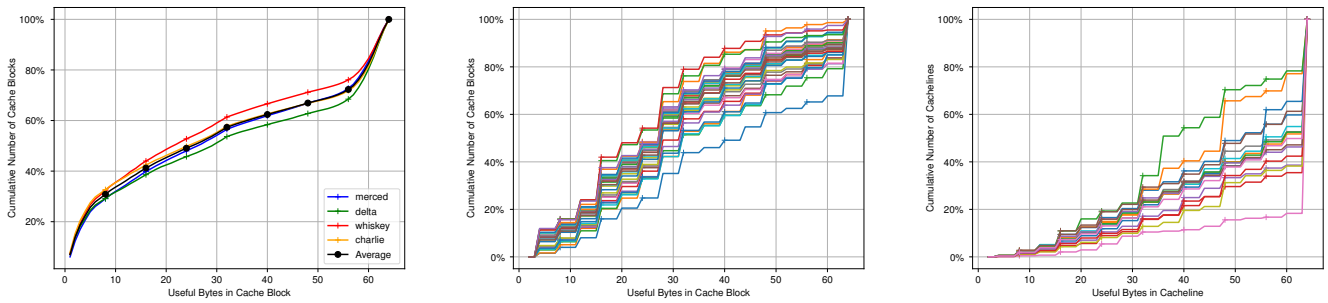
B. Sizing a Cache Block

A cache block is the granularity at which information is stored in the cache and moved between different cache levels. By moving information at block granularity, rather than individual bytes or words as requested by the core, caches exploit spatial locality as the nearby bytes brought in with the requested bytes enjoy a hit if accessed by the core. However, the cache block *size* offers a trade-off between exploitable spatial locality, tag overhead, and cache storage efficiency. A larger block size reduces tag overhead and will likely exploit more spatial locality as more nearby bytes are brought in with the requested bytes. However, it might also reduce storage efficiency if the spatial locality in the cache block turns out to be low and most of the nearby bytes go unaccessed during the block's lifetime in cache. In contrast, small cache block size increases tag overhead and cannot exploit much spatial locality as very few nearby bytes are brought in with the requested bytes. However, it also does not hurt storage efficiency much if the spatial locality in a cache block turns out to be low.

Our analysis of modern server applications shows large variations in the spatial locality of their instruction streams. Therefore, a cache with a fixed block size either misses opportunities in exploiting spatial locality or exhibits poor storage efficiency.

III. Motivation

As L1-I offers a very limited capacity to keep the access latency low, we analyze how efficiently its limited storage is utilized. To do so, we study the number of bytes accessed in a cache block during its lifetime in L1-I. The results of this study are plotted in Figure 1a for the server traces released by Google [21] and Figure 1b for the Qualcomm server traces



(a) Google traces [21] (b) Qualcomm IPC-1 **server** traces [22] (c) Qualcomm IPC-1 **client/SPEC** traces [22]

Fig. 1: Cumulative number of bytes accessed in cache blocks before eviction. Each line in the graph represents a workload.

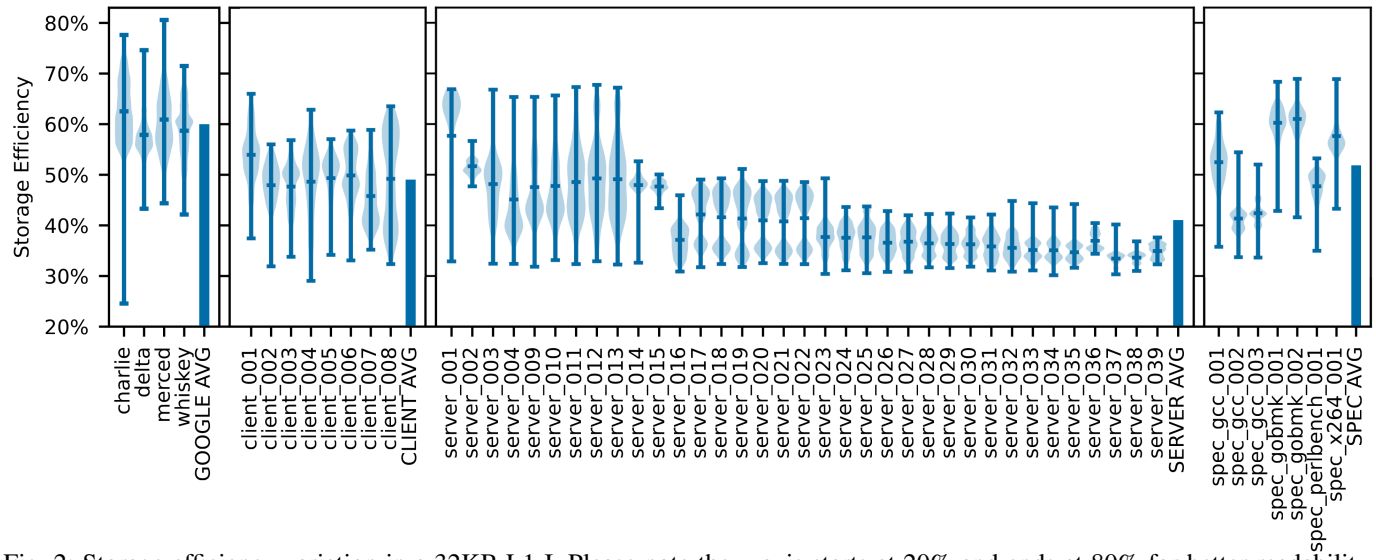


Fig. 2: Storage efficiency variation in a 32KB L1-I. Please note the y-axis starts at 20% and ends at 80% for better readability. The bars show average storage efficiency for each workload category.

released for the first instruction prefetching competition (IPC-1) [22]. The X-axis shows the number of bytes fetched from the cache block over its lifetime in L1-I, while the Y-axis shows the cumulative fraction of cache blocks. The Figure 1b traces are compiled for ARM ISAs which has a fixed 4-byte instruction size; therefore, the increase in used bytes is in step of four bytes. In contrast, Figure 1a traces are compiled for x86 ISA.

Both Figure 1a and Figure 1b show similar trends, i.e., for nearly 60% of cache blocks, more than 50% of bytes are never fetched by the core. This represents a huge underutilization of limited L1-I storage capacity. In fact, this result implies that the effective storage capacity of a 32KB L1-I is actually less than 16KB as the rest of it is occupied by useless bytes that are never accessed.

High L1-I MPKI (misses per kilo instruction) is only partially responsible for this storage underutilization since it reduces the lifetime of a cache block thus lowering the probability of a byte being accessed. However, this is not the only contributor as we observe similar underutlization for client and SPEC workloads as well, Figure 1c, despite their low MPKI since their instruction working sets mostly fit in L1-I. The fundamental reason for this behaviour is that the hot and cold code are frequently tightly mixed together [4]. It's implication at microarchitectural level is that the cold code is brought in to L1-I along with the hot code in an attempt to exploit spatial locality. However, the cold code is rarely accessed, thus resulting in huge storage inefficiency.

Zooming into the results of Figure 1a further highlights the extent of storage underutilization as it shows that nearly 30% of cache blocks use barely 8 or fewer bytes from a 64 byte cache block in Google workloads. Further, 16 bytes or fewer are accessed for nearly 40% of cache blocks. Figure 1a and Figure 1b also show that there are very few cache blocks, around 12% on average, for which all 64 bytes are fetched by the core. Further, for about 20% of the cache blocks, 60 or more bytes are fetched by the core.

To further understand the extent of underutilization, we plot the L1-I storage efficiency distribution in the violin chart of Figure 2. To generate this data, we sample the L1-I at a fixed interval of 100K cycles and record the number of bytes in the cache that have been accessed at least once.

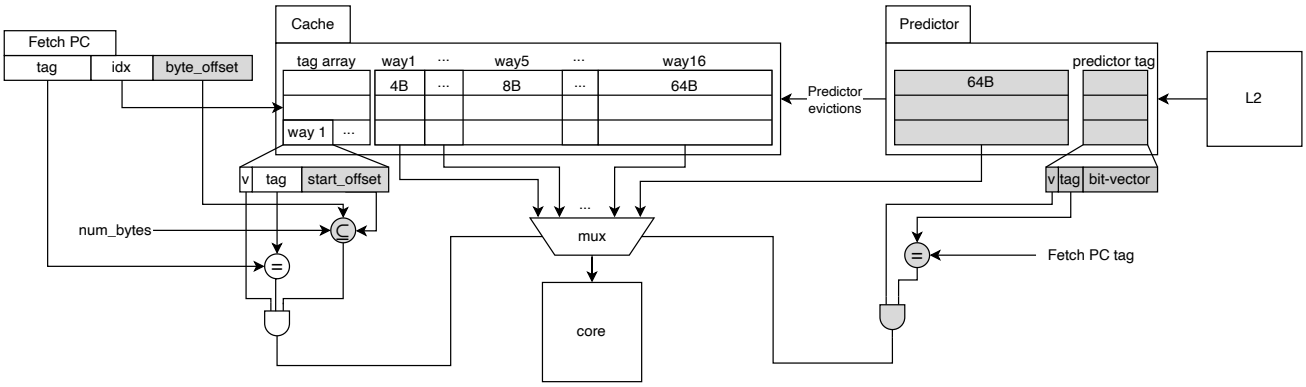


Fig. 3: The UBS cache microarchitecture.

As the figure implies, the storage efficiency varies over time; however, overall, it remains very low. In fact, there are some periods when the storage efficiency drops as low as 24%. Also, Google workloads show better storage efficiency than the other workloads as they employ profile guided code layout optimizations. Overall, these results emphasize the massive storage underutilization in L1-I.

We draw the following two **key insights** based on the analysis presented in this section:

Large variability in spatial locality: The results in Figure 1a and Figure 1b imply that there is a large variability in the spatial locality in the instructions stream. For example, in Google workloads, 30% of the cache blocks see fewer than 8 bytes accessed, while more than 60 bytes are accessed for 20% of the cache blocks.

One (block) size does not fit all: A single fixed size cache blocks are inherently unable to capture the large variability in spatial locality, therefore resulting in massive storage underutilization. Therefore, we need some sort of variability in block sizes as well to match the spatial locality variability.

IV. Uneven Block Size (UBS) Cache

Building on the insights gained in Section III, we introduce Uneven Block Size (UBS) cache to increase the storage efficiency. Different ways of UBS cache are sized to hold different number of bytes so that an incoming cache block from L2 cache can be placed in an appropriate way based on its predicted spatial locality, thereby minimizing the storage underutilization. There are two key differences in the internal organization of the UBS cache, as shown in Figure 3, that set it apart from conventional caches.

- While a conventional L1-I has a homogeneous structure with each way holding 64-bytes; in UBS cache, the ways are unevenly sized and can hold a different number of bytes with respect to each other.
- UBS cache employs a *Predictor* to identify which bytes or sub-blocks from a 64-byte block to insert in the cache.

A. UBS cache interface

UBS cache maintains the existing interface to the L2 cache, i.e., it requests/receives a 64-byte cache block from the L2. The interface to the core fetch engine, however, is slightly modified. This is because many existing designs always fetch aligned 16- or 32-byte blocks from L1-I even though many of these bytes are not on the execution path as predicted by the branch prediction unit (BPU). As a result, these bytes are discarded without ever being decoded. In contrast, our design fetches only those bytes that are on the predicted execution path. Therefore, the fetch engine needs to provide the UBS cache with a *start byte address* and the *number of bytes* to be fetched, just like the load-store unit provides similar information to the data cache. This information is already produced in the baseline core by the BPU. Concretely, whenever the BPU predicts a branch to be taken, its target becomes the *start byte address* and the number of bytes until the next predicted taken branch become the *number of bytes*. If the fetch range, i.e., *number of bytes* from the *start byte address*, is wider than the fetch bandwidth, the fetch engine (or cache controller) splits the request into multiple fetch requests.

B. Predictor design

One of the key aspects of UBS cache is to predict the spatial locality in the 64-byte cache blocks received from L2 cache for inserting only the potentially useful bytes into the cache. For that, an intuitive approach would be to predict spatial locality based on history. However, tracking the history in multi-MB instruction working sets of server applications would result in a high storage cost for the predictor. Therefore, we explore a design that, instead of using history, monitors the first few accesses to the block to predict its spatial locality. To determine how many accesses are sufficient, we study how long it takes to touch (i.e. the first access) the bytes that are accessed during the lifetime of a block in the cache. Figure 4 shows the fraction of bytes accessed during the lifetime of a block in the cache that are touched between insertion of the block to cache and the next n misses in the same set. As the figure shows, about 94.6% (Google), 90.4% (client), 93.3% (server), and 89.8% (SPEC) of accessed bytes are touched between the insertion of the block and the very next miss in the same set. Waiting for more misses only marginally increases the touched bytes. These results imply that a predictor that defines useful bytes as the ones that are touched from a block's

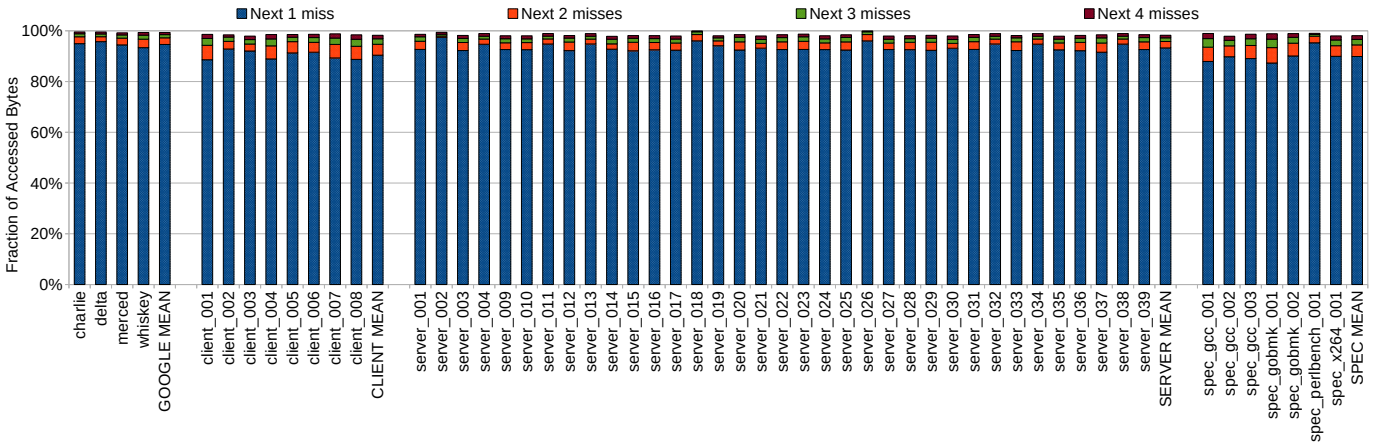


Fig. 4: Fraction of all accessed bytes that are touched (i.e., accessed at least once) between insertion of the block to cache and the next 1, 2, 3, and 4 misses in the same set.

insertion to the next miss in the set will provide 94.6%, 90.4%, 93.3%, and 89.8% accuracy on Google, client, server, and SPEC workloads respectively.

Based on the analysis above, we employ a small cache to act as a locality predictor, and all incoming cache blocks are first placed here. The predictor maintains a bit-vector per cache block to record the bytes fetched by the core during a 64-byte block's lifetime in the predictor. Notice that for ISAs with fixed instruction length, such as ARM, RISC-V, etc., recording the instructions, instead of bytes, fetched by the core is sufficient; thus reducing the storage requirements of the bit-vector. When the core fetches a byte, the corresponding bit in the bit-vector is set. When a block is evicted from the predictor, its bit-vector is examined to place only the accessed bytes into one of the UBS cache ways.

Since the predictor is simply a cache, it can take any of the potential cache organizations. However, based on the results of Figure 4, a simple direct-mapped organization with the same number of sets as in the UBS cache suits well. Therefore, logically, the predictor can be seen as one additional way in the UBS cache.

C. UBS organization

Several aspects of UBS organization are similar to a conventional cache, while some others differ. Concretely, UBS cache still stores the tags for 64-byte aligned cache blocks even though it might not store a full 64-byte block. Further, it maintains the same number of sets as in a conventional 32KB instruction cache; however, each set contains more number of ways. Therefore, the indexing function/logic of the cache stays the same.

The main difference compared to the conventional cache is that UBS needs to record what sub-block(s) of a 64-byte block are stored in the cache. For that, UBS stores the offset of the first byte, called *start_offset*, of the sub-block in the 64-byte block as shown in Figure 3. We need 6 bits to represent the *start_offset* as any of the 64 bytes can be the starting byte of a sub-block. However, in fixed instruction size ISAs such as ARM or RISC-V, we need only 4 bits to represent the *start_offset* assuming 4-byte instructions. Also, note that we do not need to store the size of the sub-block, as it is implied by the way where a request hits. For example, a hit in way-1 implies a sub-block size of 4 bytes.

D. Sizing UBS ways

The key idea of UBS cache is to size the cache ways to match the spatial locality in the instruction stream. Therefore, we leverage the data in Figure 1 to determine the size of the ways. The data in these figures suggest that some of the ways need to hold much smaller blocks than 64 bytes. Therefore, for a given storage budget per set, UBS cache provides more ways than the conventional L1-I. Based on the data in Figure 1a and Figure 1b, we design UBS cache to distribute the storage budget of a set in 16 ways with the way-sizes of 4-, 4-, 8-, 8-, 8-, 12-, 12-, 16-, 24-, 32-, 36-, 36-, 52-, 64-, 64-, and 64-bytes. The way sizes are chosen to evenly distribute the pressure on the ways.

E. UBS cache lookups

On lookups, both the UBS cache and predictor are accessed in parallel, and a request can hit in only one of these two. The UBS cache lookup mechanism itself is very similar to a conventional L1-I lookup. It is indexed with the *index* bits of the fetch address, and *tag* bits are compared with the tags stored in each way of the selected set as shown in Figure 3. However, unlike conventional L1-I, a tag match does not guarantee that the requested instructions are present in UBS cache. This is because a tag match only indicates that some (or possibly all) bytes from the 64-byte aligned block are present in the cache. To identify whether the requested bytes are present or not, the *byte_offset* bits of the fetch address are compared with the *start_offset*s stored in the cache. If the comparison reveals that the requested fetch range is a subset of the bytes in a UBS cache way, this indicates a hit in the cache. For example, *Fetch Range 1* in Figure 5 is a subset of one of the sub-blocks present in UBS cache, thus indicating a hit. Further, it is important to note that the tag comparison and *start_offset* comparison happen in parallel.

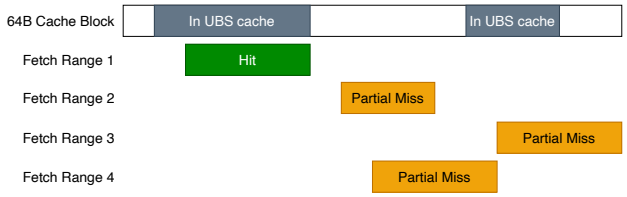


Fig. 5: UBS cache hit and partial miss

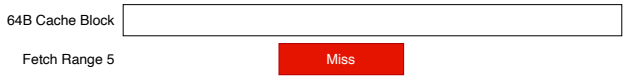


Fig. 6: UBS cache miss

If a sub-block of a 64-byte block is present in UBS cache but does not contain all the requested bytes, we consider it a partial miss. There are three scenarios under which a partial miss can occur as depicted in Figure 5. First, if none of the requested bytes are present in the cache, as represented by *Fetch Range 2* in Figure 5. We categorize such partial misses as *missing sub-block* as the whole sub-block containing the requested bytes is missing from the cache. Second, if the starting bytes of the requested bytes are present but the last bytes are not, as represented by the *Fetch Range 3* in Figure 5. We call such partial misses *overruns* because the requested bytes overrun the block present in the UBS cache. Finally, if the starting bytes of the requested address are not present but the ending bytes are present, as represented by *Fetch Range 4* in Figure 5. Such partial misses are categorised as *underruns*. It is also possible that there is no tag match in any of the ways, indicating that none of the bytes of the 64-byte block is present in the cache. We categorize such misses as full misses as shown in Figure 6.

Finally, notice that there might be a tag match in more than one way on a UBS cache lookup. This is because multiple sub-blocks from the same 64-byte block can be stored in different ways. However, there can be a hit in only one of the ways, as we ensure that sub-blocks are non-overlapping and non-contiguous.

F. Handling UBS cache misses

On cache misses, including the partial ones, UBS cache fetches the missing cache block from the lower levels of cache hierarchy (L2, LLC, etc.) just like a conventional L1-I. The incoming cache block is placed into the predictor, and the predicted useful sub-blocks from the victim block are inserted into the cache.

While moving a sub-block of 64-byte block from the predictor, the size of the sub-block determines the potential candidate ways where the sub-block can be placed. One simple strategy would be place the sub-block in the way that closely matches its storage requirements. For example, a sub-block with 16 bytes will always be placed in the way-8 as its storage capacity matches to the storage requirements of the sub-block. However, such a strategy would cause unnecessary 'conflict' misses if many sub-blocks with a similar size are seen in a short interval of time. An alternative strategy would be to place a sub-block in any of the ways that have enough storage capacity to accommodate the sub-block. For example, a sub-block with four or fewer bytes can be placed anywhere as all the ways in a set can accommodate it. However, such a policy would put more pressure on the ways with larger storage capacity compared to the ways with lower capacity.

To balance the pressure across ways while also avoiding 'conflict' misses, we choose to restrict the number of candidate ways for placing a sub-block to four. Specifically, if way-n offers the storage capacity closest to the storage requirements of the sub-block, we consider ways from way-n to way-n+3 for placing the sub-block. For example, a sub-block with 16 bytes can be placed in one of the ways from way-8 to way-11. To choose one of these four candidate ways, we use a slightly modified LRU policy that compares the LRU counters of only the four candidate ways and replaces the least recently used among them. The selected candidate way might have capacity for more bytes than are present by the sub-block. In such cases, we fill the remaining capacity with the bytes following the sub-block, even though they had not been accessed.

G. Avoiding duplication in UBS cache

As multiple sub-blocks of a 64-byte block may reside in different ways of UBS cache, it is important to ensure that there are no duplicate bytes in these sub-blocks as it will waste cache capacity. To understand the source of byte duplication, consider the situation when we see a partial miss in UBS cache. In this case, since not all the requested bytes are present in the cache, the full 64-byte block is retrieved from the lower-level caches. At this point, the full 64-byte block is present in the predictor, while some of its sub-blocks are already present in the cache. Later, when the used bytes are moved from the predictor to the cache, they might overlap with the bytes which were already present in the cache when the partial miss occurred; thus resulting in duplicated bytes.

To avoid this duplication, we invalidate the existing sub-blocks from UBS cache before inserting the useful bytes from the predictor. The invalidation is done as soon as the partial miss is detected because the fetch is stalled until the miss is resolved. Also, the subsequent accesses for the requested block will be served from the predictor.

Further, before invalidating the sub-blocks, we mark the corresponding bytes in the bit-vector of the predictor as useful. This ensures that the useful bytes are not lost if the 64-byte block stays in the predictor only for a short interval due to a subsequent miss.

V. Methodology

To evaluate the performance of the UBS cache, we use ChampSim [23], [24], a trace-drive simulator that provides a detailed implementation of the core front-end and the cache hierarchy. We model an 8-way 32KB baseline instruction cache with a LRU replacement policy. Furthermore, we model a

TABLE I: Microarchitectural parameters.

Core	4 wide fetch, decode and commit, 224 entry ROB, 97 entry scheduler, 128 entry load queue, 72 entry store queue
Branch Prediction Unit	4K entry BTB, Hashed Perceptron
Instruction Prefetcher	FDIP, 128 entry Fetch Target Queue
L1-I	32KB, 8 ways, 4 cycles latency, LRU, 8 MSHR
L1-D	48KB, 12 ways, 5 cycles latency, LRU, 16 MSHR
L2	512KB, 8 ways, 12 cycles latency, LRU, 32 MSHR entries
L3	2MB, 16 ways, 30 cycles latency, LRU 64 MSHR
DRAM	3200MHz, 1 channel, 1 rank, 8 banks, 12.5 (tRP), 12.5 (tRCD), 12.5 (tCAS)

TABLE II: UBS cache parameters.

Predictor	64-sets, direct-mapped
Cache	64-sets, 16-ways
Cache Way Sizes	4, 4, 8, 8, 8, 12, 12, 16, 24, 32, 36, 36, 52, 64, 64, 64
Replacement Policy	Modified LRU
Fetch latency	4 cycles
MSHR	8 entries

cache block size of 64-bytes across the entire cache hierarchy. In addition, the front-end is equipped with a Fetch Directed Instruction Prefetcher (FDIP) [7]. The microarchitectural parameters of the modelled processor are listed in Table I. We warm up microarchitectural structures 50 million instructions and run the simulation for another 50 million instructions.

A. Workloads

To assess the L1-I storage efficiency, we use the traces recently released by Google for a subset of their server workloads [21], as well as Qualcomm server traces provided for the first instruction prefetching championship (IPC-1) [22]. Further, we use the Qualcomm traces with fixes made by Feliu et. al. [25]. Google traces, however, do not contain instruction dependency information; therefore, there is no reliable way to generate the performance results for them. Consequently, we analyze the performance benefits of UBS cache only on IPC-1 traces. We expect similar performance on Google traces as well since they show similar L1-I storage efficiency as the IPC-1 traces as shown in Figure 1a and Figure 1b.

In addition to server traces, IPC-1 traces also include some client and a subset of SPEC benchmark traces. These workloads experience much lower L1-I pressure, thus showing lower L1-I MPKI. We use them in our analysis to understand how UBS cache behave under low pressure.

B. UBS Cache Configuration

The UBS cache configuration is presented in Table II. Both the predictor and the cache have 64 sets. The predictor is direct mapped, whereas the cache is 16-way set associative with differently sized ways. The replacement policy is LRU with the modification of that the least recently used UBS cache block within 4 ways, starting from the way that offers the storage capacity closest to the required capacity, is evicted.

TABLE III: Conv-L1I and UBS storage requirements

	32KB Conv-L1I	UBS Cache
Predictor bit-vector	-	2B
Start Offsets	-	4b × 10 + 3b × 2 + 2b × 1 + 0b × 3 = 6B
Tag (26b), LRU (3b/4b), Valid (1b)	8×(26b+3b+1b) = 30B	16 ×(26b + 4b + 1b) (data tag) + 27b (predictor tag) = 65.375B
Data Array	8×64B = 512B	$\sum$ way_sizes = 508B
Total per Set	542B	581.375B
Total Cache	64 × 542B = 33.875KB	64×581.375B = 36.34KB
Overhead UBS	-	2.46KB

VI. Evaluation

A. Storage Requirements

Table III presents the storage requirements for a conventional L1-I (Conv-L1I) and UBS. The additional storage requirements for UBS stem from the bit-vectors in the predictor, the start_offset bits, and additional tags (and other metadata) due to higher associativity. Their storage needs for a fixed instruction size (4-byte) ISA is listed in Table III.

The bit-vector of the predictor needs a single bit per instruction; therefore, requiring a total of 16 bits or 2B per set. The bits required for start_offset vary based on way size. Concretely, 64B byte ways do not need start_offset, 52B way needs 2 bits for the start_offset as only one of the first four instructions in the 64B block can be the start_offset, 36B/32B ways needs 3 bits, and the rest of the ways need 4 bits.

Due to higher associativity, UBS stores more tags than Conv-L1I. Further, higher associativity means more bits for replacement policy are needed. With 16 ways, UBS needs 4 bits for replacement. Also, since the predictor is direct-mapped, it does not need LRU bits. Overall, UBS contains 16 ways with 31-bit metadata and one predictor with 27-bit metadata.

Based on these storage requirements, UBS needs 2.46KB of additional storage over Conv-L1I.

B. Storage Efficiency

As UBS cache is aimed at boosting storage efficiency, we first evaluate its effectiveness in doing so. Figure 7 presents its storage efficiency across Google, client, server, and SPEC workloads. On average, as the figure shows, UBS cache achieves a storage efficiency of 72%, 75%, 73%, and 74% for Google, client, server, and SPEC workloads respectively. This is a significant improvement over the storage efficiency of the conventional L1-I which is limited to 60%, 49%, 41%, and 52% respectively for Google, client, server, and SPEC as shown in Figure 2. These results highlight the effectiveness of uneven block sizes employed by UBS cache in improving the storage efficiency.

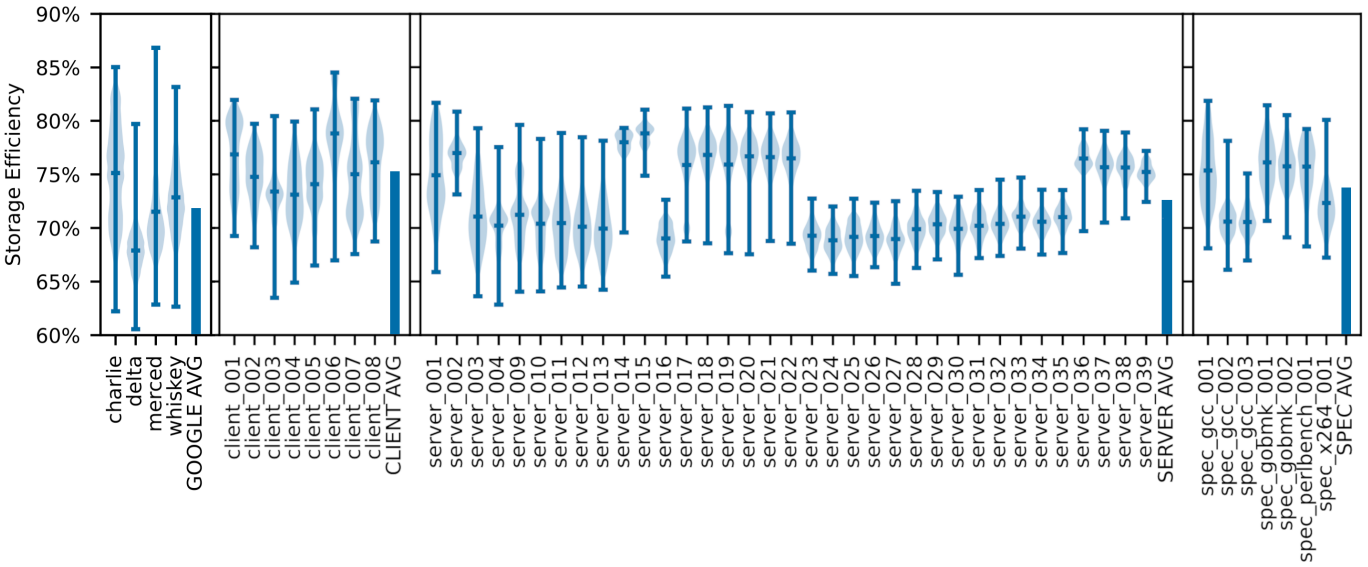


Fig. 7: Storage efficiency of UBS. Please note the y-axis starts at 60% and ends at 90% for better readability. The bars show the average storage efficiency for each workload category.

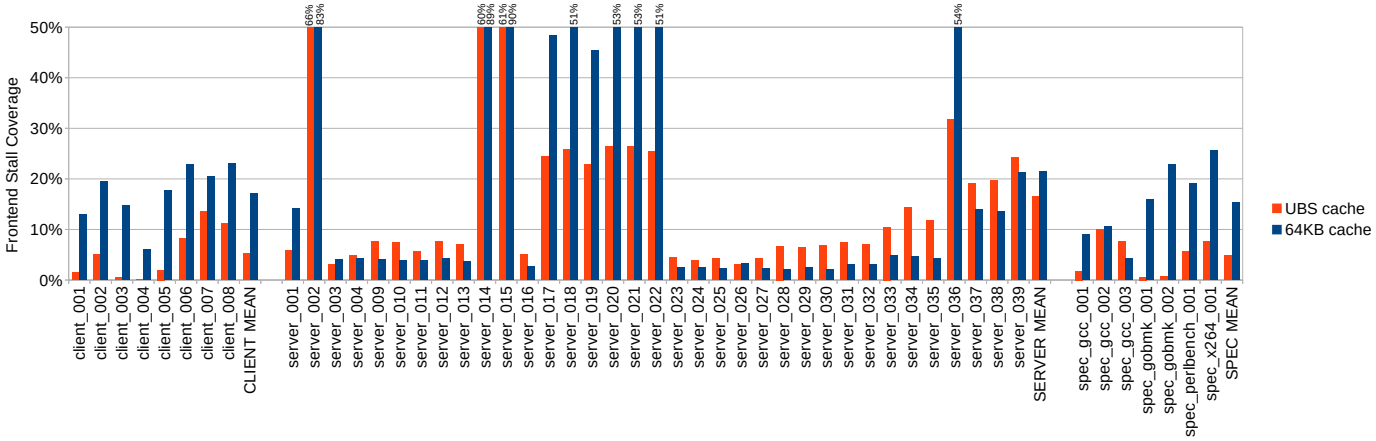


Fig. 8: Front-end stall cycles covered by UBS and 64KB L1-I over the baseline 32KB L1-I (higher is better).

The storage efficiency varies over time, and Figure 2 shows that it goes as low as 24% for the conventional L1-I. In comparison, the minimum storage efficiency for UBS cache is 60%, thus UBS cache improves the minimum storage efficiency experienced during the course of execution by 36 percentage points. The maximum storage efficiency seen by the conventional cache is around 80%, whereas UBS cache achieves a storage efficiency of as high as 87%.

C. Front-end Stall Cycle Coverage

To understand the effectiveness of UBS cache, we assess its ability to reduce front-end stall cycles across client, server, and SPEC workloads. We use the *stall cycles covered* metric over *misses covered* metric to precisely capture the impact of in-flight prefetches, i.e., the ones which have been issued but the requested block has not arrived to L1-I when needed by the fetch unit.

Figure 8 plots the fraction of front-end stall cycles covered by UBS and 64KB L1-I over the baseline 32KB L1-I . The figure shows that, on average, UBS cache covers 5.3%, 16.5%, and 4.8% of the front-end stalls in client, server, and SPEC workloads respectively. The average stall cycle coverage of UBS is especially pronounced in server workloads as they put higher pressure on L1-I. Looking at individual workloads, a number of workloads such as *server_002*, *server_014*, *server_015*, *server_017* to *server_022*, etc. show a significantly high stall cycle coverage. In fact, UBS cache eliminates more than 60% of the front-end stalls on *server_002*, *server_014* and *server_015*.

For comparison, Figure 8 also plots the stall cycle coverage achieved by a conventional 64KB L1-I. On average, the 64KB cache achieves slightly higher coverage than UBS. This is because even though UBS cache (including the predictor) has slightly more blocks than 64KB L1-I, the average blocks sizes are smaller which leads to partial misses as discussed in Section IV-E. These partial misses limit the overall stall cycle coverage. However, there are some server workloads on which UBS cache provides better coverage than 64KB L1-I because of fewer partial misses as we present in the next section.

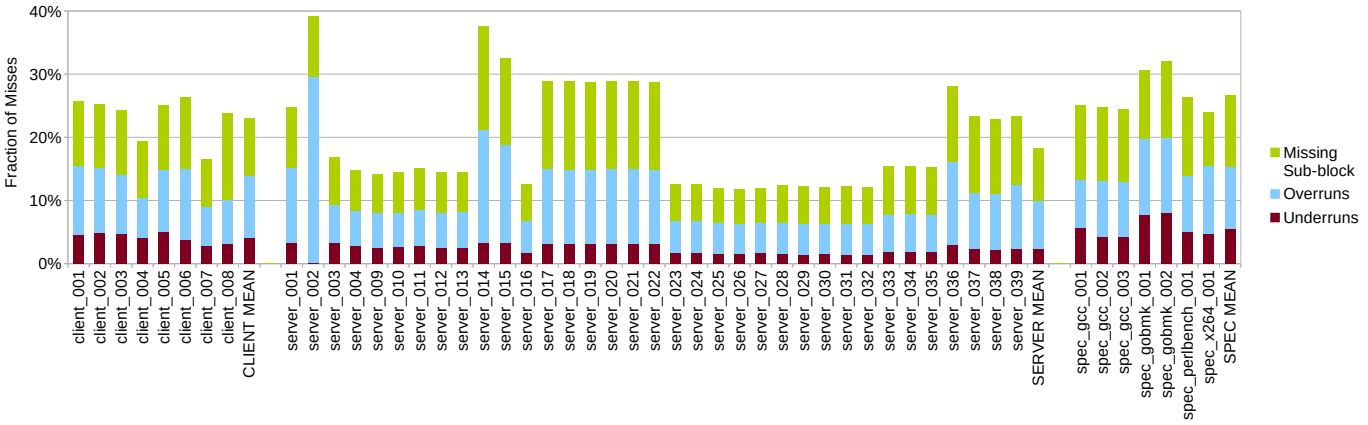


Fig. 9: Partial misses in UBS cache, split up by type of partial miss.

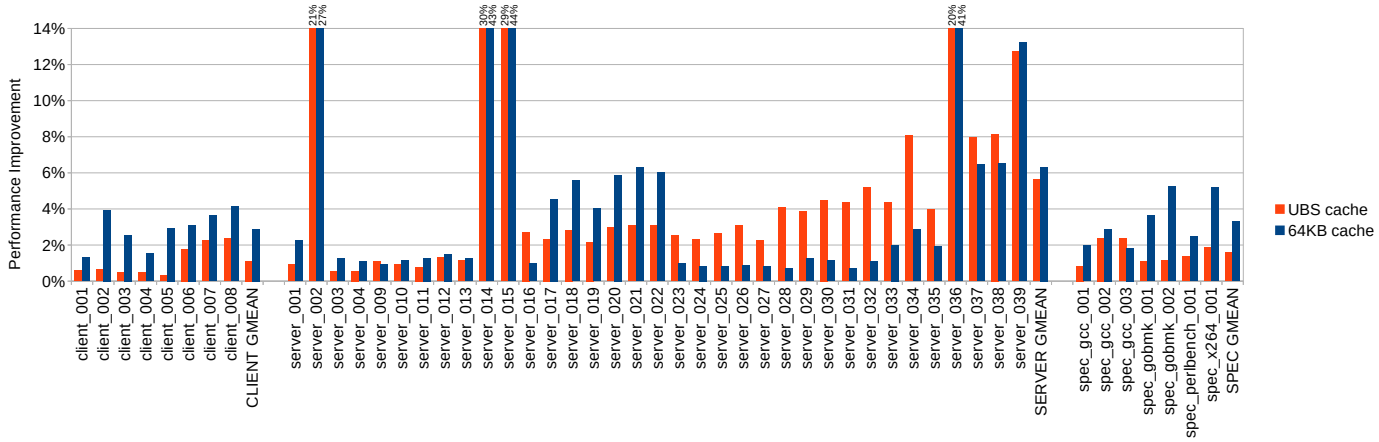


Fig. 10: Performance improvement by UBS and 64KB L1-I over the baseline 32KB L1-I

As shown in Figure 8, there are some applications, such as from *server_003* to *server_013*, for which the front-end stall coverage is very low even with UBS cache and 64KB conventional cache. This is likely because of the large reuse distances, in addition to large instruction footprints, that even UBS cache and 64KB conventional cache cannot capture.

D. Understanding Partial Misses

As described in Section IV-E, we categorize partial misses as: missing sub-block, overruns, and underruns. Figure 9 presents the distribution of partial misses among these categories. On average, about 23%, 18.2%, and 26.6% of all misses are partial misses in client, server, and SPEC workloads respectively. Partial misses are higher in client and SPEC workloads because of their lower MPKI. As a consequence, in the conventional cache, cache blocks stay longer in the cache and the nearby bytes brought in along with the requested bytes get much more time to accessed. However, the lifetime of a block in the predictor of UBS cache is less than the lifetime of a block in conventional cache. Thus, if the nearby byte are not accessed during the lifetime of a block in the predictor, they might result in a partial miss later. The figure also shows that the partial misses are dominated mainly by *missing sub-blocks* and *overruns*, whereas *underruns* are comparatively less.

E. Performance Analysis

To understand the performance benefits delivered by the reduced front-end stalls, we plot the performance gain achieved by UBS and 64KB conventional L1-I over a 32KB conventional L1-I in Figure 10. The performance improvement largely follows the stall cycle coverage trends presented in Figure 8. Specifically, we see significant performance improvement on several server workloads whereas client and SPEC workloads show relatively small improvements.

On server workloads, on geometric average, UBS cache provide about 5.6% performance gain, compared to 6.3% of 64KB L1-I, over the baseline 32KB L1-I. This emphasizes the effectiveness of UBS cache in improving storage efficiency as it provides 89% of the performance achievable by doubling the size of the conventional cache. Notice that on some of the server workloads, such as from *server_023* to *server_035*, UBS cache significantly outperforms 64KB L1-I. However, on some other workloads 64KB L1-I outperforms UBS cache. These results are in line with the results on front-end stall cycle

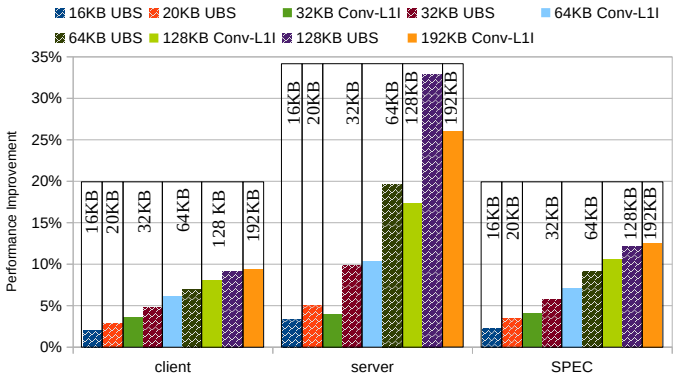


Fig. 11: Geomean performance improvement of UBS and Conv-L1I for different cache sizes over a 16KB Conv-L1I.

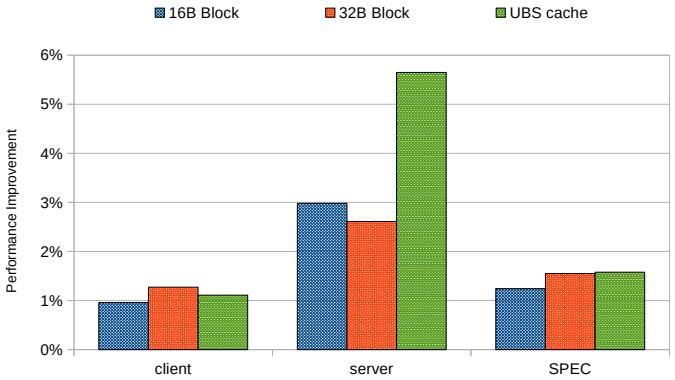


Fig. 12: Geomean performance improvement of 16B and 32B block size L1-I and UBS over a 64B block Conv-L1I.

coverage and fraction of partial misses. Concretely, UBS cache sees higher partial misses on the workloads where 64KB L1-I performs better.

F. UBS at different L1-I sizes

To assess how well UBS performs at different storage budgets, we plot UBS and conventional L1-I (Conv-L1I) performance gain at different sizes over a 16KB Conv-L1I in Figure 11. There are two interesting points to note in this figure. First, UBS needs significantly less storage than Conv-L1I to provide similar performance. For example, a 20KB UBS outperforms 32KB Conv-L1I on server workloads, and nearly matches its performance on client and SPEC workloads. Similarly, a 128KB UBS outperforms a much larger 192KB Conv-L1I on server workloads. Second, for similar storage budget, UBS always outperforms Conv-L1I. As the figure shows, this is case for all the cache sizes considered, i.e., 16KB, 32KB, 64KB, and 128KB.

G. Comparison with smaller block size cache

Reducing the L1-I block size is also likely to improve storage efficiency as likelihood of unused bytes decreases with smaller block sizes. To understand their effectiveness, we compare UBS against L1-I designs with block sizes of 32B and 16B instead of the default 64B. Despite reducing the block size, we still fetch the entire 64B block from L2 to L1-I in both designs. Further, the 64B blocks prefetched by FDIP

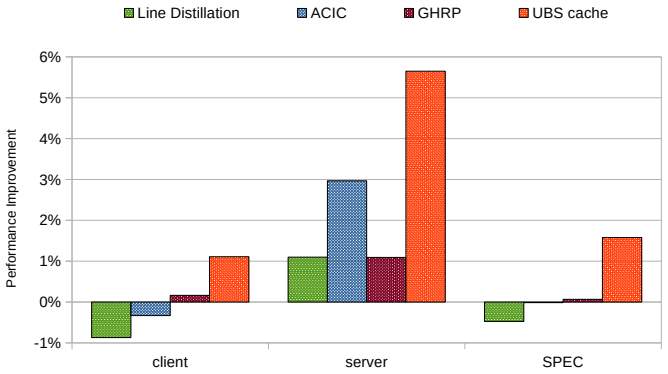


Fig. 13: Geomean performance improvement of UBS and prior work over Conv-L1I.

are placed into a prefetch buffer and only the requested 16B or 32B chunks are placed in L1-I.

As reducing the block size while maintaining the cache capacity results in more tag overhead, we size UBS, 16B block, 32B block caches to have similar storage budget. Specifically, 16B block cache, 32B block cache, and UBS require 37.5KB, 35.75KB, and 36.34 of total storage.

Figure 12 shows that the UBS provides about twice the performance gain of 16B and 32B block caches on server workloads which are the target of this work. On client and SPEC workloads, all three designs provide very similar performance. These results show the effectiveness of uneven block sizes of UBS on applications with high L1-I pressure.

H. Comparison with prior work

Prior work has targeted improving several aspects of L1-I design. This section compares UBS against a state-of-the-art cache replacement policy called GHRP [10] and an insertion policy called ACIC [26]. Further, we compare against a spatial locality aware data cache design, called Line Distillation [27], by adapting it to instruction cache.

The results presented in Figure 13 show that all three techniques improve performance on server workloads, albeit not as much as UBS. Among the three, ACIC provides the best performance as it filters out the cache blocks that are unlikely to see reuse. GHRP improves performance by keeping those blocks in the cache that are likely to see reuse. However, both of these techniques work at cache block granularity. In contrast, UBS captures performance opportunities at a finer sub-block granularity. More importantly, UBS can work in congruence with ACIC and GHRP since insertion policy, replacement policy, and block size are complementary aspects of a cache design.

On client and SPEC workloads, these techniques barely provide any performance gain. In fact, adapting Line Distillation to L1-I shows a small performance drop in both workload categories. UBS, in contrast, is still able to find some opportunities and deliver a noticeable performance gain.

I. Latency Analysis

There are two factors that can affect the access latency of UBS cache compared to a conventional L1-I:

TABLE IV: Tag and data array access latencies.

#ways	#sets	Block size	tag-array latency	data-array latency
8	64	64	0.09 ns	0.77 ns
17	64	64	0.12 ns	1.71 ns

- UBS cache features more than twice the number of ways (including the predictor).
- UBS cache requires additional logic to detect a hit as it needs to check if the requested bytes are within the sub-blocks present in the cache.

We assess the impact of both of these factors on the access latency.

Table IV presents the access latency of tag and data arrays for different cache designs as reported by CACTI 7.0 [28] at 22nm technology node. As the table shows, for a 32KB conventional L1-I, the tag array access latency is only a fraction of the data array access latency as the tag array takes 0.09ns to access in comparison to 0.77ns of the data array latency. As the tag and data array are accessed in parallel in L1 caches, the overall cache access latency is determined by the data array latency.

1) UBS tag array access latency: Since UBS cache features 17-ways (including the predictor) and 64-sets, we configure CACTI to model a conventional cache configuration with 17-ways and 64-sets. However, we keep the block size at 64-bytes because CACTI does not supports unevenly sized ways. Despite the fixed 64-byte block size, the tag array of this configuration faithfully mimics the tag array of UBS cache. As the table shows, the tag array access latency of this configuration is only 0.12ns, still much lower than the access latency of the data array of a conventional 32KB L1-I. These results show that increasing the number of ways to 17 does not make the tag-array latency a limiting factor in overall cache access latency.

The second factor that can affect the tag array access latency of UBS cache is the additional logic to check if the requested bytes are within the sub-blocks present in the cache. One implementation of such a circuit is depicted in Figure 14. This circuit checks if the offset (in 64-byte block) of the first requested byte is greater than or equals to the *start_offset* of the sub-block present in the cache. Further, it also checks if the offset (in 64-byte block) of the last requested byte is less than or equal to the offset of the last byte in the sub-block which is computed by adding way_size to *start_offset*.

Notice that these comparisons start in parallel with the tag comparison. Also, assuming a 38-bit physical address space (i.e., 256GB physical memory), a 32KB 8-way set associative cache with 64-byte blocks will require 26-tag bits. In contrast, we only need 6-bit arithmetic for checking if the requested bytes are a subset of the sub-block since *byte_offset*s are only 6-bits. To assess the latency requirements of the additional logic and compare it against the latency of tag comparison, we implement them in RTL and synthesize using Cadence Genus synthesizer with 28nm technology library from ST Microelectronics. Our analysis shows that the latency of the added logic is 1.6x of the tag comparison latency. CACTI

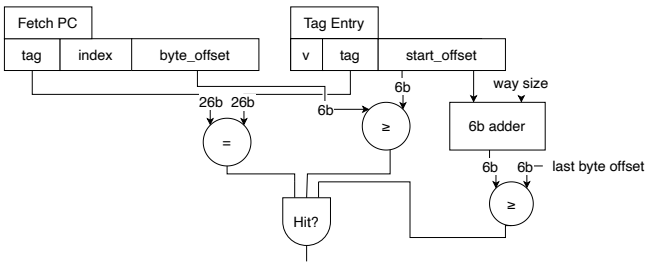


Fig. 14: Circuit for detecting UBS cache hits.

reports a comparator latency of 0.018ns, thus the added logic of UBS will have a latency of 0.028ns. This implies that the tag array access latency increases from 0.12ns to 0.13ns (0.12ns - 0.018ns + 0.028ns), for a 17-way cache, which is still significantly lower than the data array latency of a 32KB conventional cache.

Overall, this analysis shows that the extra ways and the additional logic do not cause the UBS tag array access latency to become a bottleneck, as it remains well below the data array latency.

2) UBS data array access latency: As Table IV shows, the data array access latency of the 17-way cache is more than twice than that of an 8-way cache. However, all the ways in this 17-way cache hold 64-bytes, meaning that the cache capacity is also more than twice. However, this is not the case for the UBS cache where some of the ways offer much lower storage capacity than 64-byte. In fact, the overall storage capacity of UBS cache is slightly less than 32KB.

Since the overall storage capacity of all the ways of UBS is similar to the storage capacity of 8-ways of a conventional 32KB L1-I, we can group multiple logical ways of UBS into a single 64B physical way. For example, one possibility is to consolidate the ways of sizes [8, 8, 12, 32], [4, 8, 16, 36], [4, 24, 36], [52, 12], [64], [64], [64] together in seven physical ways, with the predictor being the 8th physical way. Consequently, UBS cache will have the same number of physical data ways as a conventional 32KB cache. Therefore, its data array access latency will also be the same as that of the baseline 32KB conventional cache.

Note that even though the consolidation results in eight physical ways in the data array, we still have 16-ways in the tag array. A side effect of this consolidation is that a tag array hit in any of the ways that are consolidated together will select the same data array physical way. For example, based on the consolidation scheme of the previous paragraph, a tag array hit in way-3, way-4, way-6, or way-10 will select the physical way-1 in the data array.

To read the desired bytes out of a 64B physical way, UBS calculates the shift amount slightly differently compared to a conventional L1-I because of the logical way consolidation. The shift amounts in a conventional L1-I can be 0, 16, 32, or 48 bytes, if the core front-end always fetches aligned 16-bytes. In UBS, however, the shift amount depends on where exactly the logical way resides in a 64B physical way. Therefore, the

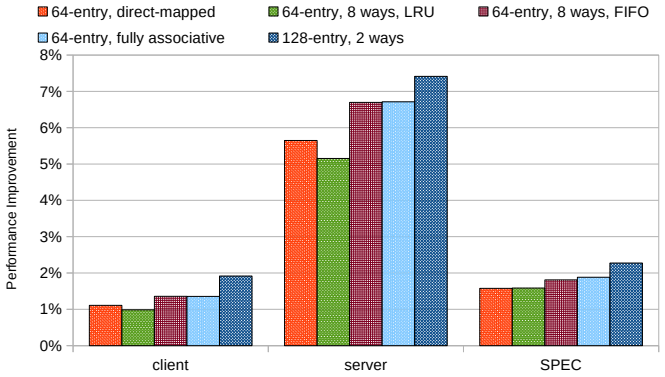


Fig. 15: Geomean performance improvement of UBS with different predictor designs over Conv-L1I.

Fig. 16: Geomean performance gain of UBS and Conv-L1I for different way configurations over a 32KB Conv-L1I.

shift amount, i.e., *fetch_byte_offset*[1] in the logical UBS block, needs to be adjusted based on the logical way that sees the hit. For example, if there is a hit in the logical way-2, the size of way-1 will be added to the *fetch_byte_offset* to get the shift amount, as that is where the first byte to be fetched from the way-2 resides in the 64B physical block.

Regarding the latency of calculating the shift amount, *fetch_byte_offset* calculation happens in parallel with the tag comparison; however, the size of the preceding logical way(s) is added to it once we know which way sees the hit. This requires a 6-bit addition; hence, the shift amount is available in 0.13ns (i.e., the hit detection time, Section VI-I1) + 0.01ns (6-bit adder latency) = 0.14ns. Thus, the shift amount calculation is not on the critical path as it is available well before the data array access completes, i.e. latency of 0.77ns(Table IV).

Further, notice that a conventional L1-I with aligned 16-byte fetch accesses needs to support only four shift amounts, i.e., 0, 16, 32, or 48 bytes. In contrast, UBS potentially needs to support all possible shift amounts, i.e. 0-63 and 0-15 in variable and fixed (4-byte) instruction length ISAs respectively. However, supporting more shift amounts is unlikely to increase the cache access latency as data caches already support such shift amounts while providing similar latency to conventional instruction caches.

To summarize, the analysis in this section shows that the tag array latency is much lower than the data array latency. Further, even with more ways and additional hit detection logic of UBS cache, the tag array latency does not become a limiting factor. In addition, by consolidating multiple logical ways of UBS cache into a single 64B physical way, we fit all UBS ways into eight 64B physical ways; thereby maintaining the access latency of the data array. Therefore, the access latency of UBS cache stays the same as that of the baseline 32KB conventional L1-I.

[1]*fetch_byte_offset* is the offset of the first byte to be fetched from a cache block. In UBS, *fetch_byte_offset* is calculated as *byte_offset* - *start_offset*, following the nomenclature of Figure 14; whereas *fetch_byte_offset* in a conventional cache is the same as the *byte_offset*.

J. Impact of predictor organization and size

Figure 15 presents UBS performance gain over Conv-L1I with different predictor organizations and sizes (64-entry direct-mapped predictor is the default predictor). Overall, all the predictors provide similar performance. Even doubling the predictor size to 128 entries (with additional storage for the extra 64 entries) does not show drastic performance gain. It is interesting to see that an 8-way set-associative predictor with LRU performs slightly worse than the direct-mapped predictor. This is because the frequently accessed cache blocks stay in the predictor for longer which effectively reduces the capacity of the predictor. Therefore, replacing LRU with a FIFO policy improves performance and nearly matches the performance of a fully-associative predictor.

K. UBS's sensitivity to number/size of ways

Figure 16 presents UBS speedup with different numbers of ways over a 32KB Conv-L1I. Further, the figure plots two configurations, i.e., config1 and config2, that differ in how the ways are sized. For example, for 14-way UBS, config1 way sizes are [4, 4, 8, 12, 16, 24, 28, 28, 32, 36, 36, 64, 64, 64] and config2 way sizes are [4, 4, 8, 16, 24, 28, 32, 36, 40, 44, 52, 60, 64, 64]. The results show only small performance variation for UBS with 12 or more ways, especially on server workloads. For example, 12-way (config2), 14-way (config2), 16-way (config2), and 18-way UBS provide 5.2%, 5.26%, 5.85%, and 5.76% performance gain while the default 16-way configuration provides a speedup of 5.65%. For the 10-way configurations, storage efficiency does not improve as much as for other configurations because the block sizes stay large for most of the ways. However, it still provides considerable performance improvement, 3.53% and 4.18%, respectively, over the baseline on server workloads. The figure also shows that doubling the number of ways to 16 (and halving the number of sets to maintain the cache capacity) in a Conv-L1I provides negligible speedup, especially on the server workloads (0.26%).

L. Analyzing UBS on More Traces

To further assess the effectiveness of UBS, we evaluate UBS on traces that were not used in the design process of

UBS. We ran UBS on Championship Value Prediction (CVP-1) [29] traces: 45 integer, 13 floating-point, and 77 server traces. Our results show that UBS outperforms a conventional 64KB L1-I on these traces as it achieves 2.6%, 1.5%, and 0.29% performance gain over baseline 32KB L1-I compared to 1.9%, 0.9%, and 0.26% of 64KB conventional L1-I on server, floating-point, and integer traces respectively.

VII. Related work

Front-end bottleneck is a well established performance limiter in server applications. Large scale studies from Google [4], [17] and Meta [19] show that the front-end stalls are responsible for increasingly large fraction of CPU cycles in their application fleet. Further, recent studies [20], [30] show that the core front-end is a critical performance bottleneck even in short running serverless functions.

Over the years, researchers have proposed several prefetching and replacement approaches to mitigate the front-end bottleneck. Reinman et. al [7] proposed fetched directed instruction prefetcher (FDIP) more than two decades ago. Recent work [8], [31] shows that when coupled with a large enough branch target buffer (BTB), FDIP approaches the performance of an ideal L1-I even on modern server applications. Consequently, researchers have been looking at reducing BTB misses. Boomerang [8] explicitly identifies BTB misses and fills them by predecoding corresponding cache blocks. Shotgun [9], [32] observes that BTB misses for unconditional branches, i.e. calls, returns, etc., severely limit prefetching opportunities compared to short taken branches. Therefore, it reserves bulk of BTB storage for unconditional branches. UDP [33] aims to improve accuracy and timeliness of prefetches. PDIP [34] assists FDIP in cases where it struggles. Further, new BTB designs [35]–[38] have been proposed to maximize the number of branches in a fixed BTB storage budget, thus assisting FDIP. BTB prefetching techniques [39]–[41] have also been explored to further reduce BTB misses.

Apart from FDIP, many other prefetchers have been investigated. Temporal stream prefetchers [42]–[45] record the instruction stream and replay it for prefetching; however, they incur high storage overhead. EIP [46], [47] is targeted at improving prefetch timeliness. Further, many software and profile guided optimization techniques [4], [15], [48]–[54] has also been proposed. In addition, cache and BTB replacement policies [10], [55]–[57] have been explored to mitigate the front-end bottleneck.

As the past research has primarily focused on prefetching and replacement policies, the instruction cache design itself has stayed the same with all cache blocks being the same size. In contrast, there has been some work on spatial locality aware data caches. However, these designs are either unable to accommodate the large variability in instruction stream spatial locality or highly complex to implement. One of these designs, called Line distillation [27], splits the cache into a word-organized cache (WOC) and a line-organized cache (LOC) and moves individual words into the WOC from LOC if the cacheline exhibits poor spatial locality. However, given the large variability in the spatial locality, cache blocks with only two different sizes fall well short of capturing the variability.

Another design, Amoeba [58], chooses a more flexible approach than Line Distillation by merging the tag and storage array into a unified storage area. This enables dynamically allocating as many tags as the workload requires. It uses a predictor to identify the words that are likely to be used and fetches only those into the cache. This dynamic behavior enables a relatively flexible adjustment to the application requirements. However, this flexibility also makes the design more complex. For example, the location of tags in the cache is not fixed; therefore, Amoeba needs to first locate the tags before doing a tag match. Further, the replacements are complex, and the cache is prone to fragmentation because an incoming block might not fit in any of the available spaces in the set. Further, multiple evictions might be required to make enough space for the incoming block. UBS cache avoids these issues by keeping the tag locations fixed. Also, finding a way where the incoming block fits in UBS cache is easier due to the fixed size of each way. Further, the spatial locality predictor of Amoeba requires dedicated storage which is likely to lead to significant storage overhead given the massive instruction footprints of server workloads. UBS cache, in contrast, uses a very simple locality predictor, whose storage requirements are carved out of the cache storage budget.

VIII. Conclusion

Front-end stalls are a long-standing bottleneck in server workloads, and researchers have proposed several instruction prefetching and replacement mechanisms to alleviate it. The paper looked into a largely unexplored aspect for mitigating the bottleneck. Specifically, we analyzed the cache storage efficiency, i.e., the fraction of used bytes in the cache. Our analysis revealed a huge under-utilization of cache capacity that effectively halves the available storage capacity.

To improve the storage efficiency, this paper proposed a new cache design called Uneven Block Size (UBS) cache. Each way of the UBS cache is sized to hold blocks of different sizes to that they match the spatial locality in the instruction stream. Our evaluation shows that UBS cache improves the storage efficiency by 32 percentage points compared to a conventional cache. Further, by supporting uneven block sizes, UBS cache more than doubles the number of blocks in the cache at a given storage budget. Consequently, compared to a 32KB conventional instruction cache, it reduces front-end stalls by 16.5%, and provides comparable performance to what is achieved by a 64KB conventional L1-I, on a set of server workloads.

Acknowledgment

We thank the reviewers for their valuable feedback. This work is partially supported through the Research Council of Norway (NFR) grant 302279 to NTNU.

References

[1] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 266–277, 1999.

[2] 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*, pages 15–26, 1998.

[3] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V Adve, and Luiz André Barroso. Performance of Database Workloads on Shared-Memory Systems With Out-Of-Order Processors. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 307–318, 1998.

[4] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 462–473, 2019.

[5] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. I-SPY: Context-driven conditional instruction prefetching with coalescing. In *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, volume 2020-October, pages 146–159. IEEE Computer Society, 10 2020.

[6] Aasheesh Kolli, Ali Saidi, and Thomas F Wenisch. RDIP: Return-Address-Stack Directed Instruction Prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 260–271, 2013.

[7] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 16–27, 1999.

[8] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. Boomerang: A metadata-free architecture for control flow delivery. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 493–504, 2017.

[9] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. Blasting through the front-end bottleneck with shotgun. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 30–42, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Samira Mirbagher Ajorpaz, Elba Garza, Sangam Jindal, and Daniel A. Jiménez. Exploring predictive replacement policies for instruction cache and branch target buffer. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 519–532, 2018.

[11] Robert Cohn and P Geoffrey Lowney. Hot Cold Optimization of Large Windows/NT Applications. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 80–89. IEEE, 1996.

[12] Tom Way and Lori L Pollock. Evaluation of a Region-Based Partial Inlining Algorithm for an ILP Optimizing Compiler. In *Proceedings of the 10th Annual IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 698–705, 2002.

[13] Rahman Lavaee, John Criswell, and Chen Ding. Codestitcher: Inter-procedural Basic Block Layout Optimization. In *Proceedings of the 28th International Conference on Compiler Construction*, pages 65–75. ACM, 2 2019.

[14] Yuxuan Zhang, Tanvir Ahmed Khan, Gilles Pokam, Baris Kasikci, Heiner Litz, and Joseph Devietti. Ocolos: Online Code Layout Optimizations. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 530–545. IEEE, 2022.

[15] Dehao Chen, Tipp Moseley, and David Xinliang Li. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 12–23. IEEE, 2016.

[16] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48. ACM, 3 2012.

[17] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[18] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An Open-Source Benchmark Suite for Microservices and their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[19] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. SoftSKU: Optimizing Server Architectures for Microservice Diversity@Scale. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, pages 513–526, 2019.

[20] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. Lukewarm serverless functions: Characterization and optimization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 757–770, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Parthasarathy Ranganathan and Victor Lee. Advancing Systems Research with Open-Source Google Workload Traces, 5 2022. https://cloud.google.com/blog/topics/systems/workload-traces-for-google-warehouse-scale-computers.

[22] 1st Instruction Prefetching Championship Traces. https://research.ece.ncsu.edu/ipc/infrastructure/#Traces.

[23] ChampSim Simulator. https://github.com/ChampSim/ChampSim.

[24] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. The Championship Simulator: Architectural Simulation for Education and Competition. 2022.

[25] Josué Feliu, Arthur Perais, Daniel A. Jiménez, and Alberto Ros. Rebasing microarchitectural research with industry traces. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*, pages 100–114, 2023.

[26] Yunjin Wang, Chia Hao Chang, Anand Sivasubramaniam, and Niranjan Soundararajan. Acic: Admission-controlled instruction cache. In *Proceedings - International Symposium on High-Performance Computer Architecture, HPCA*, volume 2023-February, pages 165–178. IEEE Computer Society, 2023.

[27] Moinuddin K. Qureshi, M. Aater Suleman, and Yale N. Patt. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 250–259. IEEE, 2007.

[28] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 694–701, 2011.

[29] First Championship Value Prediction. https://www.microarch.org/cvp1/cvp1online/contestants.html.

[30] Truls Asheim, Tanvir Ahmed Khan, Baris Kasicki, and Rakesh Kumar. Impact of microarchitectural state reuse on serverless functions. In *Proceedings of the Eighth International Workshop on Serverless Computing*, WoSC '22, page 7–12, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. Re-establishing fetch-directed instruction prefetching: An industry perspective. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 172–182, 2021.

[32] Rakesh Kumar and Boris Grot. Shooting down the server front-end bottleneck. *ACM Trans. Comput. Syst.*, 38(3–4), jan 2022.

[33] Surim Oh, Mingsheng Xu, Tanvir Ahmed Khan, Baris Kasikci, and Heiner Litz. Udp: Utility-driven fetch directed instruction prefetching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 1188–1201, 2024.

[34] Bhargav Reddy Godala, Sankara Prasad Ramesh, Gilles A. Pokam, Jared Stark, Andre Seznec, Dean Tullsen, and David I. August. Pdip: Priority directed instruction prefetching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 846–861, New York, NY, USA, 2024. Association for Computing Machinery.

[35] Truls Asheim, Boris Grot, and Rakesh Kumar. A storage-effective btb organization for servers. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1153–1167, 2023.

[36] Truls Asheim, Boris Grot, and Rakesh Kumar. BTB-X: A storage-effective BTB organization. *IEEE Computer Architecture Letters*, 20(2):134–137, 2021.

[37] Niranjan K Soundararajan, Peter Braun, Tanvir Ahmed Khan, Baris Kasikci, Heiner Litz, and Sreenivas Subramoney. Pdede: Partitioned, deduplicated, delta branch target buffer. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 779–791, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Vishal Gupta and Biswabandan Panda. Micro btb: A high performance and storage efficient last-level branch target buffer for servers. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*, CF '22, page 12–20, New York, NY, USA, 2022. Association for Computing Machinery.

[39] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. Twig: Profile-guided btb prefetching for data center applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 816–829, New York, NY, USA, 2021. Association for Computing Machinery.

[40] Ioana Burcea and Andreas Moshovos. Phantom-btb: a virtualized branch target buffer design. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, pages 313–324, 2009.

[41] James Bonanno, Adam Collura, Daniel Lipetz, Ulrich Mayer, Brian Prasky, and Anthony Saporito. Two level bulk preload branch prediction. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 71–82, 2013.

[42] Michael Ferdman, Thomas F Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal Instruction Fetch Streaming. In *International Symposium on Microarchitecture*, 2008.

[43] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. Proactive Instruction Fetch. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[44] Cansu Kaynak, Boris Grot, and Babak Falsafi. SHIFT: Shared History Instruction Fetch for Lean-core Server Processors. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[45] Cansu Kaynak, Boris Grot, and Babak Falsafi. Confluence: Unified instruction supply for scale-out servers. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 166–177, 2015.

[46] Alberto Ros and Alexandra Jimborean. A cost-effective entangling prefetcher for instructions. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 99–111, 2021.

[47] Alberto Ros and Alexandra Jimborean. Wrong-path-aware entangling instruction prefetcher. *IEEE Transactions on Computers*, 73(2):548–559, 2024.

[48] David Xinliang Li, Raksit Ashok, and Robert Hundt. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 53–61, 2010.

[49] Guilherme Ottoni and Bertrand Maher. Optimizing function placement for large-scale data-center applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 233–244. IEEE, 2017.

[50] C-K Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. Ispike: a post-link optimizer for the intel/spl reg/itanium/spl reg/architecture. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 15–26. IEEE, 2004.

[51] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2019.

[52] Chi-Keung Luk and Todd C Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 182–193. IEEE, 1998.

[53] Murali Annavaram, Jignesh M Patel, and Edward S Davidson. Call graph prefetching for database applications. *ACM Transactions on Computer Systems (TOCS)*, 21(4):412–444, 2003.

[54] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. I-spy: Context-driven conditional instruction prefetching with coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 146–159. IEEE, 2020.

[55] Nayana Prasad Nagendra, Bhargav Reddy Godala, Ishita Chaturvedi, Atmn Patel, Svilen Kanev, Tipp Moseley, Jared Stark, Gilles A. Pokam, Simone Campanoni, and David I. August. Emissary: Enhanced miss awareness replacement policy for l2 instruction caching. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[56] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. Ripple: Profile-guided instruction cache replacement for data center applications. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 734–747, 2021.

[57] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A. Jiménez, Heiner Litz, and Baris Kasikci. Thermometer: profile-guided btb replacement for data center applications. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 742–756, New York, NY, USA, 2022. Association for Computing Machinery.

[58] Snehasish Kumar, Hongzhou Zhao, Arrvindh Shriraman, Eric Matthews, Sandhya Dwarkadas, and Lesley Shannon. Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy. In *Proceedings - 2012 IEEE/ACM 45th International Symposium on Microarchitecture, MICRO 2012*, pages 376–388, 2012.