Improving the Utilization of Micro-operation Caches in x86 Processors

Jagadish B. Kotra, John Kalamatianos
AMD Research, USA.
Email: Jagadish.Kotra@amd.com, John.Kalamatianos@amd.com

***Abstract*—Most modern processors employ variable length, Complex Instruction Set Computing (CISC) instructions to reduce instruction fetch energy cost and bandwidth requirements. High throughput decoding of CISC instructions requires energy hungry logic for instruction identification. Efficient CISC instruction execution motivated mapping them to fixed length micro-operations (also known as uops). To reduce costly decoder activity, commercial CISC processors employ a micro-operations cache (uop cache) that caches uop sequences, bypassing the decoder. Uop cache's benefits are: (1) shorter pipeline length for uops dispatched by the uop cache, (2) lower decoder energy consumption, and, (3) earlier detection of mispredicted branches.**

In this paper, we observe that a uop cache can be heavily fragmented under certain uop cache entry construction rules. Based on this observation, we propose two complementary optimizations to address fragmentation: Cache Line boundary AgnoStic uoP cache design (CLASP) and uop cache compaction. CLASP addresses the internal fragmentation caused by short, sequential uop sequences, terminated at the I-cache line boundary, by fusing them into a single uop cache entry. Compaction further lowers fragmentation by placing to the same uop cache entry temporally correlated, non-sequential uop sequences mapped to the same uop cache set. Our experiments on a x86 simulator using a wide variety of benchmarks show that CLASP improves performance up to 5.6% and lowers decoder power up to 19.63%. When CLASP is coupled with the most aggressive compaction variant, performance improves by up to 12.8% and decoder power savings are up to 31.53%.

***Index Terms*—Micro-operations Cache, CPU front-end, CISC, X86, Micro-ops.**

I. Introduction

Most commercial processors achieve high performance by decoupling execution from instruction fetch [50]. Maintaining high execution bandwidth requires ample supply of instructions from the processor's front-end. High bandwidth, low latency decoders play a critical role in enabling high dispatch bandwidth of instructions to the back-end. However, achieving high dispatch bandwidth requires high throughput instruction decoding which is challenging due to variable length of x86 instructions and their corresponding operands [32]. As a result, x86 instruction decoding is a multi-cycle operation that serializes the identification and decoding of the subsequent instruction and thus falls on the critical path. To reduce the serialization latency, x86 processors employ multiple decoders operating in parallel which increases the decoder power.

Traditional x86 instructions are translated into fixed length uops during the decode stage. These fixed length uops make issue and execution logic simpler [5, 51][1]. These uops are hidden from the Instruction Set Architecture (ISA) to ensure backward compatibility. Such an ISA level abstraction enables processor vendors to implement an x86 instruction differently based on their custom micro-architectures. However, this additional step of translating each variable length instruction in to fixed length uops incurs high decode latency and consumes more power thereby affecting the instruction dispatch bandwidth to the back-end [5, 34, 22].

To reduce decode latency and energy consumption of x86 decoders, researchers have proposed caching the decoded uops in a separate hardware structure called uop cache[2] [51, 36]. An uop cache stores recently decoded uops anticipating temporal reuse [17, 51]. The higher the percentage of uops fetched from the uop cache, the higher the efficiency, because:

- The uop cache fetched uops can bypass the decoder thereby saving the decoder latency from the critical pipeline path.
- The uop cache fetched uops containing branches can benefit from early detection of mispredicted branches thereby lowering the branch misprediction latency.
- The complex x86 decoders can be shut down when uops are fetched from the uop cache resulting in power savings.

In this paper, we propose various simple and practical optimizations for the uop cache that improves its effectiveness without increasing its overall area. More specifically, this paper makes the following contributions:

- We note that in a decoupled front end processor, the fetch unit (called prediction window) and the uop cache entry may differ in the way they are constructed to satisfy the design constraints of each block (OpCache vs Fetcher). That may lead to uop cache fragmentation, resulting in severely under-utilized capacity and thus lower uop cache hit rate. Based on this observation, we propose two optimizations that improve overall uop cache utilization.
- The first optimization, CLASP, merges uop cache entries mapped to sequential PWs, into the same uop cache line, improving performance by up to 5.6% via higher uop cache utilization.
- The second optimization is motivated by the observation that uop cache entries are often smaller than each uop cache line, due to predicted taken branches and other constraints imposed by the uop cache entry build logic. To alleviate such fragmentation, we propose different methods for merging uop cache entries from different, non-sequential

[1]These uops resemble decoded RISC instructions, i.e. consist of pipeline control signals of a RISC-like operation.

[2]Note that Opcache, uop cache and OC all refer to a micro-operations cache and are used interchangeably in this paper.

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00025

PWs in the same uop cache line. We evaluate three novel Compaction variants, viz., (a) Replacement policy Aware Compaction (RAC), (b) Prediction Window (PW) Aware (PW-Aware) Compaction (PWAC), and (c) Forced Prediction Window (PW) Aware Compaction (F-PWAC). F-PWAC encompassing CLASP and the other compaction techniques improves the performance by up to 12.8% over the baseline uop cache design.

The rest of this paper is organized as follows. In Section II, we describe the background of an x86 processor front-end covering the intricacies of uop cache management. In Section III, we demonstrate the performance and power benefits of an uop cache in a modern x86 processor design. We present our methodology in Section IV before presenting our optimizations, viz., CLASP and Compaction in Section V. The experimental evaluation is covered in Section VI, while related work is covered in Section VII before concluding in Section VIII.

II. Background

A. Modern x86 processor front-end

Figure 1 shows a typical x86 processor front-end. As depicted in the figure, there are three hardware structures: (a) Instruction cache (I-cache), (b) Uop cache [51, 34], and, (c) loop cache (buffer) [5, 21, 19, 41, 52] that can feed the back-end engine with uops. The I-cache stores x86 instructions, whereas the uop cache and loop cache hold already decoded uops. Hence, instructions fetched from the I-cache need to be decoded while uops fetched from either the uop cache or the loop cache bypass the instruction decoders thereby saving the decoder pipeline latency and energy. The loop cache stores uops found in loops small enough to fit while the remaining uops are stored in the uop cache. Consequently, any techniques that increase the percentage of uops fed to the back-end from the uop cache or loop cache improves performance and energy-efficiency.

The front-end branch predictors generate Prediction Windows in a decoupled front end architecture. Each Prediction Window (PW) dictates a range of consecutive x86 instructions (marked by start and end address) that are predicted to be executed by the branch predictor. PW addresses are sent to the Icache, uop cache and loop cache and in the case of a hit uops are dispatched to the back end from the most energy-efficient source. A PW generated by the branch predictor can start anywhere in an I-cache line and can terminate at the end of an I-cache line or anywhere in the middle of an Icache line if a predicted taken branch or a predefined number of predicted not-taken branches have been encountered. Figures 2(a), 2(b), 2(c) present various examples demonstrating where a PW can start and end with respect to an I-cache line boundary. The PW in Figure 2(a) starts at the beginning and terminates at the end of an I-cache line. The PW in Figure 2(b) starts in the middle and terminates at the end of the I-cache line. This happens when the previous PW included a predicted taken branch whose target address falls in the middle of an I-cache line. The PW is terminated at the end of the I-cache line. The

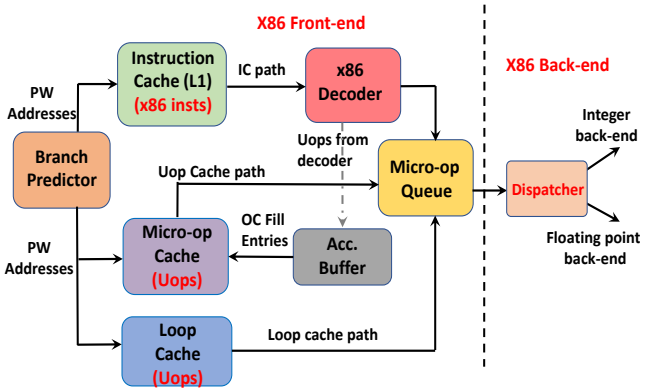


Fig. 1: x86 Front-end (example)

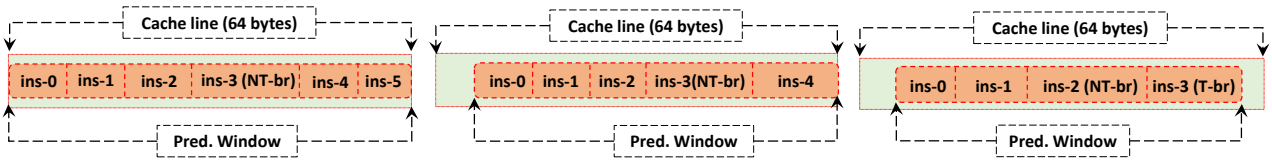
PW in Figure 2(c) starts in the middle and ends before the end of the I-cache line due to a predicted taken branch instruction (ins-3).

B. Micro-operations Cache (Uop Cache)

As depicted in Figure 1, the primary source of uops that bypass the complex x86 decoders is the uop cache. In this subsection, we discuss the uop format, the uop cache entry construction rules and the uop cache management including the uop cache lookup and fill logic.

1) Uop Format: Uop cache holds uop sequences. Uops demand a fixed format to avoid variable length decoding logic, however the uop format remains highly implementation dependent and may change across different processor generations. On one side of the spectrum, uops can be encoded as a fixed length CISC instructions, saving uop cache area at the expense of decoding logic required to identify the fields needed to rename, issue and execute each uop. On the other end, uops can be encoded as partially decoded, fixed length RISC operations (similar to that in [35]), to ease decoding logic complexity at the expense of additional metadata storage. We assume each uop to occupy 56-bits in our studies (Table I). Note that optimal uop encoding is implementation dependent and is out of the scope of this paper.

2) Uop Cache Entries: In the baseline design, an uop cache line comprises of a single uop cache entry. In this paper, an uop cache line represents the physical storage while the uop cache entry represents the set of uops stored in the uop cache line. Given that a PW drives the creation of uop cache entries, most of the PW terminating conditions apply to uop cache entry creation (except for the one breaking a PW due to maximum number of predicted not-taken branches). However, due to the fixed size uop cache line, each uop cache entry has additional terminating conditions: a maximum number of uops and a maximum number of immediate/displacement fields. Each uop cache entry also includes metadata that enable identification of these fields when reading out uops on an uop cache hit [17, 1]. Such metadata include the number of uops and imm/disp fields per entry. In order to simplify decoding logic that identifies uops and imm/disp fields, we place all uops on the left side and all imm/disp fields on the right side of each uop cache line. There may be empty space between



(a) (b) (c)

Fig. 2: PWs depicted with respect to I-cache lines. (T/NT-br: Taken/Not Taken branch instructions)

them. Summarizing the uop cache entry terminating conditions include: (a) I-cache line boundary, (b) predicted taken branch, (c) maximum number of uops allowed per uop cache entry, (d) maximum number of immediate/displacement fields allowed per uop cache entry, and (e) maximum number of micro-coded uops allowed per uop cache entry [17].

By terminating an uop cache entry at the I-cache line boundary, only uops corresponding to instructions from the same I-cache line are stored in a single uop cache entry. Similarly, terminating an uop cache entry due to a predicted taken branch prevents caching the same uops in multiple uop cache entries (like a trace cache implementation would allow). In order to invalidate those uops we must either search or flush the entire uop cache. Prior trace-cache designs required flushing of the entire trace cache for self-modifying code invalidations [4]. Both of these constraints simplify uop cache probing and invalidation logic. Such logic supports self-modifying code and inclusion of the uop cache with either the I-cache or the L2 cache.

Please note that under the above termination conditions, an uop cache entry can store uops corresponding to multiple PWs because it can include uops from any number of sequential PWs. Alternatively, a PW can span uops across two uop cache entries because of the limits on the number of uops per uop cache entry due to fixed uop cache line size. In this work, we assume a baseline uop cache that can contain a maximum of 2K uops [17]. The uops decoded by the x86 decoder are stored in an accumulation buffer (Figure 1) until one of the above termination conditions is met. Once a termination condition is met, the group of uops forming the uop cache entry is written in to the uop cache.

3) Uop Cache Operations (lookup/fills): Similar to traditional caches, the uop cache is organized as a set-associative structure indexed by the starting physical address of the PW. Unlike traditional caches, each uop cache line stores uops and their associated metadata. Even though an uop cache physical line has a fixed size, the uop cache entries may not occupy all available bytes and the number of uops per uop cache line varies. The uop cache is byte addressable. Therefore, an uop cache entry tag and set index is generated by the entire PW starting physical address. If an uop cache lookup incurs a hit, then the entire uop cache entry is sent to the micro-op queue (Figure 1) in a single clock cycle. However, in scenarios where a PW spans across two uop cache entries, the uop cache entries are dispatched in consecutive clocks. Upon an uop cache miss, x86 instructions are fetched from the I-cache, decoded and fed to the uop queue as depicted in Figure 1. The decoded uops from the I-cache path are also written to the uop cache via the accumulation buffer.

4) Why are trace-based uop caches impractical?: As mentioned in Section II-B2, trace cache builds uop cache entries beyond a taken branch. For example, with a trace-cache based implementation, a basic block "B" can be part of multiple uop cache entries (traces) such as "CB", "DB", "EB" etc, if the branches in blocks "C", "D" and "E" jump to "B". Aside from the wasted uop cache space occupied by the multiple instances of "B", invalidating "B" by a Self-Modifying Code (SMC) access complicate the uop cache design. This is because "B" can be in any number of uop cache entries across different uop cache sets. Thus, the SMC invalidating probe has to either search the entire uop cache or simply flush it to ensure "B" is invalidated. Several runtime languages like Java employ dynamic compilers which can modify instructions at run-time using SMC support.

Additional overheads include power and complexity because trace caches demand multi-branch prediction mechanisms to predict beyond a taken branch and achieve high fetch bandwidth. Such overheads have motivated researchers to adopt uop cache designs limited to basic block-based uop cache entry organizations [51] as also evidenced in modern commercial x86 processors [3, 6]. In our work, we model a baseline uop cache that is a limited and practical form of trace cache, where entries are built beyond predicted non-taken branches and terminate upon encountering a predicted taken branch. Consequently our baseline design limits uop cache entry redundancy and facilitates SMC invalidations by allowing the basic block "B" of the previous example to be installed only within the same uop cache set. This can occur in two scenarios: (a) Uop cache entry starting from "B" if reached by a predicted taken branch or (b) Uop cache entry starting with "AB" if control flow falls through from "A" to "B". Both instances of "B" can exist in the same uop cache set if the number of uop cache and I-cache sets match (assumption in baseline Uop cache). Both instances of "B" can be invalidated in one tag lookup by a single SMC invalidating probe to the I-cache line normalized address where both "A" and "B" are mapped to.

III. Motivation

In this section, we demonstrate the importance of uop cache in the design of the modern x86 processor. Please refer to Table I for our evaluation setup and Table II for our evaluated workloads.

A. Performance and decoder power impact

As discussed in Sections I and II, uop cache fetched uops directly bypass the x86 decoder thereby resulting in

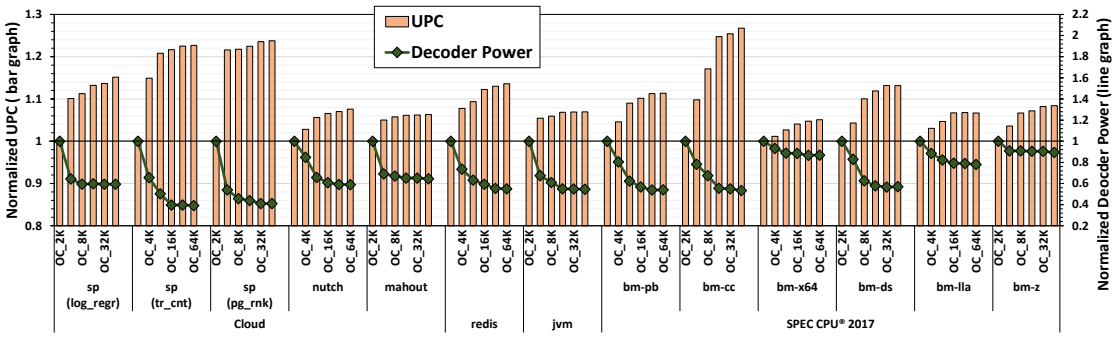


Fig. 3: Normalized UPC and decoder power results with increase in uop cache capacity.

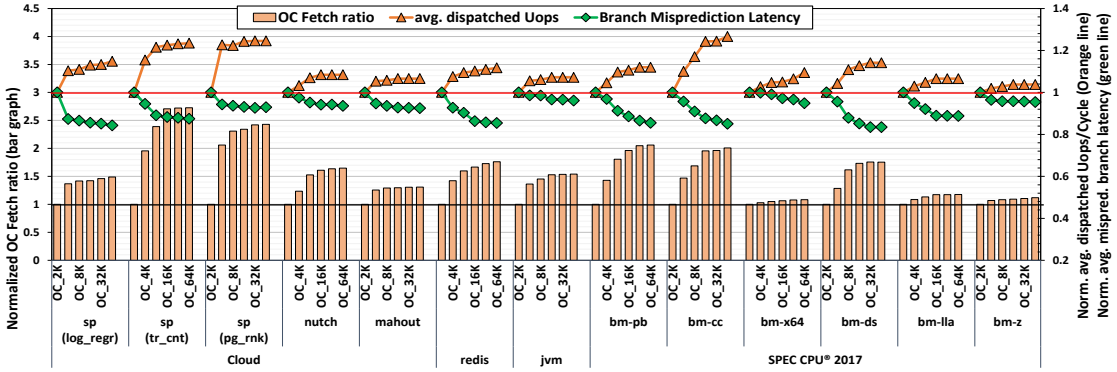


Fig. 4: Normalized uop cache fetch ratio (bar graph), average dispatched uops/cycle (orange line graph), and, average branch misprediction latency (green line graph) results with increase in uop cache capacity.

shorter pipeline latencies and lower energy consumption. Figure 3 shows the normalized UPC (uops committed Per Cycle) improvement (bar graph) on the primary Y-axis and the corresponding normalized x86 decoder power (line graph) on the secondary Y-axis as we increase the uops residing in uop cache from 2K to 64K. As can be observed from Figure 3, an uop cache with 64K uops improves the performance by 11.2% with a maximum gain of 26.7% for 502.gcc_r (bm-cc). This performance improvement correlates directly to the higher number of uops dispatched to the back-end from the uop cache. Correspondingly, x86 decoder power consumption reduces by 39.2%.

Figure 4 shows the improvement in uop cache fetch ratio with a larger uop cache. The uop cache (OC) fetch ratio is calculated as the ratio of uops dispatched from the uop cache to the total uops dispatched. The total uops dispatched include those from the uop cache as well as the I-cache. The average improvement in uop cache fetch ratio is 69.7% for a 64K uop cache over one with 2K uops baseline. For Sparkbench (sp) uop cache containing 64K uops improves the uop cache hitrate by 172.7% over an uop cache with 2K uops.

B. Front-end dispatch bandwidth impact

Figure 4 (orange line graph) on secondary Y-axis shows the improvement in average dispatch bandwidth as we increase the uops residing in uop cache from 2K to 64K. The dispatch bandwidth is defined as the average number of uops fed to the back-end per cycle. An uop cache hosting 64K uops dispatches 13.01% more uops per cycle to the back-end compared to a baseline uop cache having 2K uops. This improvement is highest for 502.gcc_r (bm-cc) application which dispatches 26.6% more uops per cycle. The improvement in dispatch bandwidth is due to increased number of uops bypassing the decoder and avoiding pipeline bubbles (stalls) due to the complexities in decoding of x86 instructions.

C. Branch misprediction penalty impact

Figure 4 (green line graph) on secondary Y-axis shows how the average branch misprediction latency changes with the uop cache size. The branch misprediction penalty is measured as the number of elapsed cycles between branch fetch cycle and branch misprediction detection and pipeline redirection. The average branch misprediction penalty drops by 10.31% on an average with larger uop cache sizes. This reduction is as high as 16.54% for the 531.deepsjeng_r (bm-ds). The bigger uop cache enables more uops to bypass the x86 decoder. Consequently, more mispredicted branches can be detected earlier in the pipeline thereby reducing the average branch misprediction penalty. Due to earlier detection of mispredicted branches, less instructions are incorrectly fetched and executed resulting in additional performance and energy savings. Please note that the reduction in branch misprediction penalty with increased uop cache utilization is a second-order effect due to the reduced pipeline depth because of more uop cache hits.

D. Sources of fragmentation in uop caches

Unlike an I-cache line whose capacity is fully utilized with instruction bytes, the uop cache can not fully utilize its physical lines because of the terminating conditions that

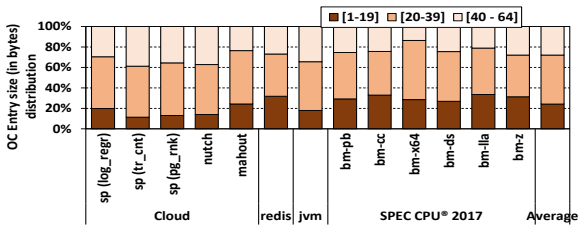


Fig. 5: Uop cache entries size distribution.

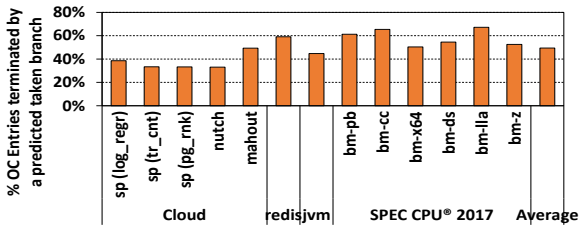


Fig. 6: Percentage of uop cache entries terminated by a predicted taken branch.

govern the construction of uop cache entries. Figure 5 shows the distribution of uop cache entry size in bytes. On an average, 72% of the uop cache entries installed in the uop cache are less than 40 bytes in size assuming a 56-bit uop size and a 64B uop cache line.

Figure 6 shows that 49.4% of the uop cache entries are terminated by a predicted taken branch, with the maximum being 67.17% for 541.leela_r (bm-lla) from the SPEC CPU® 2017 suite.

Another terminating condition is the I-cache line boundary as explained in Section II-B2. We notice that this constraint severely reduces the front-end dispatch bandwidth and increases Ocache fragmentation. Figure 7 depicts uop cache entries where entries-1, 3 are terminated because of crossing I-cache line (64-Byte) boundaries, while uop cache entries-2 and 4 are terminated by other conditions described in Section II-B2. Uop cache entries-1 and 3 are terminated even though the control flow between entries-1 and 2 (and similarly between entries-3 and 4) is sequential; there are no predicted taken branch instructions. Since uop cache entries-1 and 2 (and similarly entries-3 and 4) correspond to contiguous I-cache lines, they occupy two different lines in uop cache sets, set-3 and set-0, respectively.

As explained in Section II-A, a PW can start anywhere within an I-cache line (Figures 2(a), 2(b) and 2(c)). Thus, depending on where the PWs corresponding to entries-1, 3 in Figure 7 start, these uop cache entries can include only a few uops because they are terminated when crossing the I-cache line boundary. I-cache line boundary termination can also result in smaller entries-2 and 4, especially if they are terminated by a predicted taken branch. As we can observe, I-cache line boundary termination can split sequential code into smaller uop cache entries, which are mapped to more physical lines in different uop cache sets.

Overall, smaller uop cache entries hurt performance and energy efficiency for the following reasons:

- Increased fragmentation caused by smaller uop cache entries leave uop cache space under-utilized which lowers hit rate.

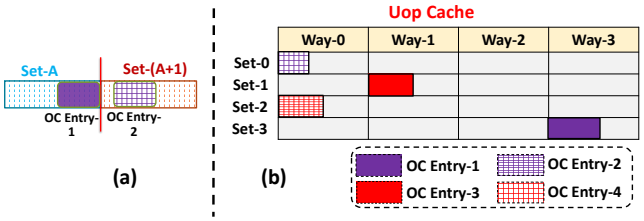


Fig. 7: Fragmented uop cache containing entries that are terminated by I-cache line boundaries (depiction).

- Reduced front-end dispatch bandwidth when hitting on smaller uop cache entries.

Summarizing the results from Sections III-A, III-B, and, III-C, we demonstrate that uop caches are instrumental in improving performance via higher front-end dispatch bandwidth and lower average branch misprediction latency and reducing dynamic power via lower use of x86 decoders. In Section III-D, we show that installed uop cache entries are smaller and severely fragment the uop cache when we terminate uop cache entries crossing the I-cache line boundary, negatively effecting front-end bandwidth and therefore performance. To that end, we present two optimizations CLASP and Compaction in Section V that address various sources of fragmentation.

Core	3 GHz, x86 CISC-based ISA Dispatch Width: 6 instructions Retire Width: 8 instructions Issue Queue: 160 entries Decoder Latency: 3 cycles Decoder Bandwidth: 4 insts/cycle
Uop Cache	32-sets, 8-way associative True LRU replacement policy Bandwidth: 8 uops/cycle Uop Size: 56-bits; ROB: 256 Uop Queue Size: 120 uops Max uops per uop cache entry: 8 Imm/disp operand size: 32 bits Max imm/disp per OC entry: 4 Max U-coded insts per OC entry: 4
Branch Predictor	Tage Branch Predictor [49] 2 branches per BTB entry; 2-level BTBs
L1-I	32KB, 8-way associative, 64Bytes cache line; True LRU Replacement branch prediction directed pretcher Bandwidth: 32 Bytes/cycle
L1-D	32KB, 4-way associative, 64Bytes; True LRU replacement policy
L2 Cache	512KB (private), 8-way associative, 64Bytes cache line, unified I/D cache
L3 Cache	2MB (shared), 16-way associative, 64Bytes cache line, RRIP repl. policy
Off-chip DRAM	2400 MHz

TABLE I: Simulated Processor Configuration.

IV. Methodology

A. Experimental Setup

We used a trace-based in-house cycle-accurate simulator to evaluate the performance of our proposed uop cache optimizations. Our simulator correlated to synthesized RTL, models a x86 out-of-order processor with a three-level cache hierarchy. Our simulator is correlated with a commercial x86 industry grade processor with over 3000 traces and has an R^2 value (correlation coefficient) of 0.98 with silicon and RTL. We accurately model various aspects of out-of-order processor including AVX-128/256 and 512-bits along with the x86 micro-code. Table I summarizes the architectural details of our simulated processor. The L1 I-cache employs branch directed

Suite	Application	Input	Branch MPKI
Cloud	SparkBench[12] (sp)	Logistic Regression [13] (log_regr)	10.37
		Triangle Count (tr_cnt)	7.9
		Page Rank (pg_rnk)	9.27
	Nutch [8]	Search Indexing	5.12
	Mahout [7]	Bayes Classification	9.05
Redis [9]	redis	Benchmarked using memtier [10]	1.01
SPECjbb 2015-Composite [15]	jvm	Inputs include point-of-sale requests, online purchases, and data-mining operations	2.15
SPEC CPU® 2017 [14]	500.perlbench_r (bm-pb)	reference input	2.07
	502.gcc_r (bm-cc)	reference input	5.48
	525.x264_5 (bm-x64)	reference input	1.31
	531.deepsjeng_r (bm-ds)	reference input	4.5
	541.leela_r (bm-lla)	reference input	11.51
	557.xz_r (bm-z)	reference input	11.61

TABLE II: Workloads Evaluated.

prefetching of instructions, while each level in the data cache hierarchy employ different prefetchers. The decoder power results reported are collected using Synopsys PrimeTime PX (PTPX) methodology [16] and is estimated by running tests on synthesized RTL in prior commercial core designs and using PTPX tool chain.

B. Workloads

We experimented with a wide variety of suites covering Cloud, Server and SPEC CPU® 2017 suites. From the Cloud domain, we used Sparkbench [12], Mahout [7] and Nutch [8] applications, while the Server suite included SPECjbb®2015-Composite [15] and redis [9]. Finally, we also included the SPEC CPU® 2017 [14] benchmarks. These workloads exhibit significant under-utilization of micro-op cache due to fragmentation. The Branch MPKI shown in Table II indicates the branch misses per kilo instructions encountered in our baseline. The traces are collected using SimNow [11] which includes full-system/operating systems activity and the binaries are compiled using gcc with -O2 flag. The workloads along with inputs used in the evaluation are summarized in Table II. The trace snippets simulated capture the original program behavior using the methodology described in [40].

V. Optimized Uop Cache Design

A. Cache Line boundary AgnoStic uoP cache design (CLASP)

In Section III-D, we demonstrated with an example how the uop cache entries terminated by I-cache line boundary crossing are mapped into different uop cache sets 'A' and 'A+1' as depicted in Figure 7-(b). As explained before, uop cache entries are terminated when crossing I-cache line boundaries, independent of the actual control flow. This is done conservatively to avoid building traces of uops that might warrant flushing of the entire uop cache upon invalidation.

I-cache line boundary termination is conservative and results in smaller uop cache entries, causing lower performance and higher energy consumption. In our Cache Line boundary AgnoStic design (CLASP), we relax this constraint and allow

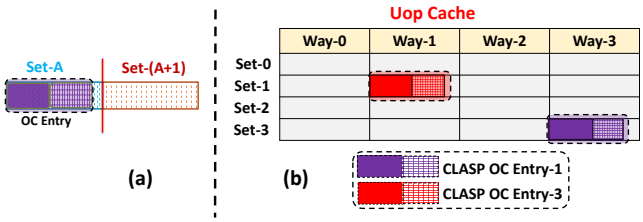


Fig. 8: CLASP design.

building uop cache entries beyond an I-cache line boundary, only when the control flow is sequential when crossing the I-cache line boundary.

Figure 8 shows how CLASP enables forming larger OC entries compared to the ones in Figure 7. OC entry-1 (entry-3) in Figure 8-(b) comprises of uops from entries-1 and 2 (entries-3 and 4) in Figure 7-(b) and occupies only one uop cache set (Set-3, Set-1). Since the meta-data for each uop cache entry requires keeping the start and end address covered by that entry, we can reuse the baseline uop cache lookup mechanism with the start address of the uop cache entry to index the correct set. However in CLASP, unlike the baseline design, an entire OC entry-1 (comprising of uops from entry-1 and entry-2) is now dispatched to the back-end in a single cycle, thereby improving dispatch bandwidth. In addition, the fetch uop cache logic, that determines the next fetch address, will not change with CLASP because the next fetch address calculation in the baseline also uses the end address provided by the uop cache entry.

The major challenge that arises with CLASP is to efficiently probe and invalidate uop cache entries because they are remapped to a different uop cache set without flushing the entire uop cache. We achieve that by merging only uop cache entries mapped to sequential control flow with CLASP. This guarantees that the split uop cache entries are mapped to *consecutive* uop cache sets. In other words, CLASP only remaps uop cache entries mapped to consecutive uop cache sets. The number of sets to be probed is a function of the number of contiguous I-cache lines whose uops map to a single uop cache entry. If we only allow uops corresponding to instructions from two contiguous I-cache lines to be part of a single uop cache entry, the number of possible uop cache sets that must be searched for invalidation is two. Accessing two consecutive uop cache sets in a single cycle can be achieved by mapping consecutive sets to different uop cache banks. For example, in Figure 8-(b), which depicts a CLASP implementation supporting a maximum of two I-cache lines, searching Sets-'A' and 'A-1' for an instruction address that originally correspond to Set-'A' should suffice to support probing.

CLASP enables forming larger uop cache entries that better utilize the available uop cache space and reduces fragmentation while boosting front-end dispatch bandwidth. Figure 9 shows the percentage of uop cache entries that span the I-cache line boundaries. CLASP improves the uop cache fetch ratio on an average by 11.6%, while the maximum improvement is as high as 24.9% for 502.gcc_r (bm-cc) from the SPEC

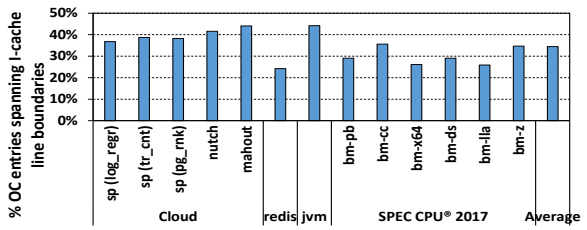


Fig. 9: Uop cache entries spanning I-cache line boundaries after relaxing the I-cache line boundary termination constraint.

CPU® 2017 suite as will be shown in Section VI-A along with the other results that include: improvements in performance, decoder power savings, average dispatch bandwidth and reduction in average branch misprediction penalty.

B. Compaction

CLASP addresses the fragmentation caused by the I-cache line boundary termination by allowing uops mapped to sequential code to participate in the same uop cache entry (while still allowing one uop cache entry per uop cache line). The remaining termination conditions still lead to smaller uop cache entries severely fragmenting the uop cache space. To address the uop cache fragmentation due to those constraints, we propose Compaction which opportunistically tries to place more than one uop cache entry in a single uop cache line depending on the available free space.

Figure 10 demonstrates the idea of Compaction of two uop cache entries in a single uop cache line. In the baseline OC entries - 1, 2 occupy two lines, line-1 and line-2, even though there is enough space to accommodate both entries in a single uop cache line. Compacting more than one uop cache entry per line improves uop cache space utilization and results in higher uop cache hit rate. Please note that Compaction is opportunistic as it can only be performed if the compacted uop cache entries together fit entirely in a single uop cache line. In our design, Compaction is performed only when an uop cache entry is being written to the uop cache.

> Note that the proposed compaction technique only groups multiple uop cache entries in a single uop cache line and does not fuse multiple uop cache entries into one. Fusing multiple uop cache entries in the same uop cache line would violate the termination conditions of an uop cache entry. Uop cache bandwidth does not increase with Compaction unlike CLASP. That is, only one uop cache entry will be sent to the uop queue per cycle irrespective of how many other uop cache entries are resident in the same uop cache line.

Compaction allows more entries to be resident in the uop cache. To enable compaction, each uop cache line tag is widened to accommodate the compacted uop cache entries tag information. Figure 11 presents the details on how various fields in a compacted uop cache line can be tracked and accessed. For example, since the uops from one uop cache entry needs to be separated from another, the uop cache tags

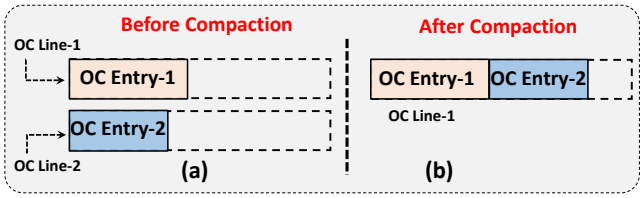


Fig. 10: Compaction Depiction.

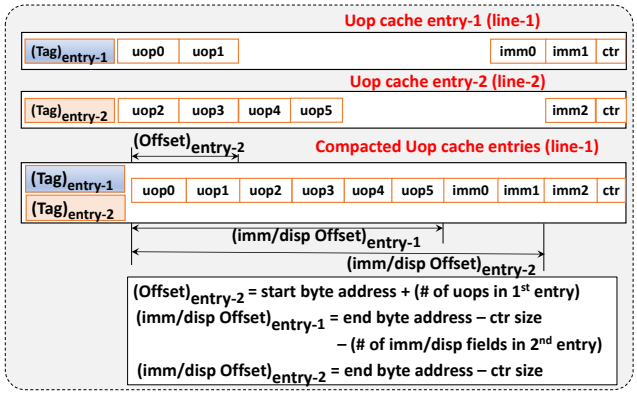


Fig. 11: Compacted uop cache entries.

need to be augmented with various offset fields. These offset fields include, $\text{offset}_{(\text{entry-2})}$, imm/displacement $\text{offset}_{(\text{entry-2})}$. The $\text{offset}_{(\text{entry-2})}$ points to starting byte of the uops of entry-2. Similarly, imm/displacement $\text{offset}_{(\text{entry-2})}$ points to the offset of where the immediate and displacement fields of uop cache entry-2 are stored. Figure 11 shows how these offsets are computed upon compaction. These augmented offset fields in the uop cache tag enable fetching the correct uops and their corresponding immediate/displacement fields upon a tag hit. Upon compaction, the "ctr" field, which contains error protetion bits for all bytes in the line, is also updated. Also, compacting more than one uop cache entry necessitates comparing multiple uop cache tags (though in parallel) of entries compacted in the same way during an uop cache lookup operation.

While compaction minimizes the fragmentation caused by smaller uop cache entries, maintaining fast uop cache fill time is critical to performance. Since the accumulation buffer (Figure 1) is of limited capacity, the x86 decoder will have to stall if the entries from the accumulation buffer are not freed in time to accommodate newly decoded uops. Freeing accumulation buffer entries can be delayed by uop cache fill time which in turn can be complicated by the eviction time. Victim selection can be critical in defining fill latency because evicting an uop cache entry may not leave enough space to accommodate the new uop cache entry if the victim uop cache entry's size is smaller than that of the new uop cache entry. One solution is to find a victim uop cache entry that can accommodate the new one. This solution can also be suboptimal in terms of performance because it can evict a different uop cache entry from the one selected by the replacement policy that tracks temporal reuse (e.g. LRU). In the worst case, the chosen uop cache entry can be the most recently accessed one. This complexity arises from the fact

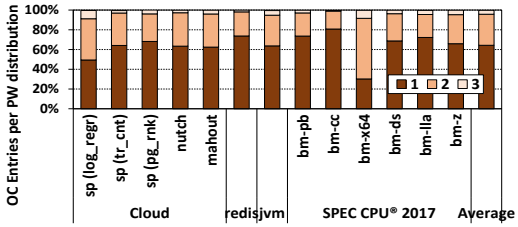


Fig. 12: Uop cache entries per PW distribution.

that the uop cache controller does not know the size of an entry being filled until fill time.

To address this issue, we propose maintaining replacement state per line, independent of the number of compacted uop cache entries. The replacement logic is similar to that of a non-compacted, baseline uop cache design. Upon a hit to a compacted line, the shared replacement state is updated. During an uop cache entry fill, all uop cache entries from the victim line are evicted, thereby ensuring the uop cache entry to be filled will have enough space. This scheme enables Compaction to have the same uop cache fill latency as the baseline. Because all compacted uop cache entries are treated as a unit by the replacement policy, the decision on which uop cache entries to compact together plays a crucial role in the overall fetch ratio of the compacted uop cache. To that end, we propose three different allocation (Compaction) techniques: Replacement Aware Compaction (RAC), Prediction Window Aware Compaction (PWAC), and, Forced Prediction Window-Aware Compaction (F-PWAC).

1) Replacement Aware Compaction (RAC): As the compacted uop cache entries are treated as a unit by the replacement policy, it is important to compact entries that will be accessed close to each other in time. During an uop cache fill, RAC attempts to compact the uop cache entry with the uop cache entries touched recently. That is, it tries to compact the new OC entry with the Most Recently Used (MRU) one.

RAC improves uop cache fetch ratio and performance as we compact uop cache entries that are temporally accessed close to each other. Though RAC allows compacting uop cache entries accessed closely in time, the replacement state can be updated by another thread because the uop cache is shared across all threads in a multithreaded core. Hence, RAC cannot guarantee compacting OC entries of the same thread together.

2) Prediction Window-Aware Compaction (PWAC): In this subsection, we propose another compaction policy called, Prediction Window-Aware Compaction (PWAC). In PWAC, we only compact uop cache entries belonging to the same PW (of a thread). Figure 12 shows the number of uop cache entries present per PW. As can be observed, on the average, 64.5% of the PWs contain only one uop cache entry, while 31.6% of the PWs contain two uop cache entries and 3.9% of the PWs contain three uop cache entries. A PW may contain multiple uop cache entries due to multiple OC entry termination conditions that differ from the PW ones (Section II-B2).

Figure 13 depicts our proposed PWAC scheme. Figure 13-(a) shows the PW agnostic compaction scheme where uop

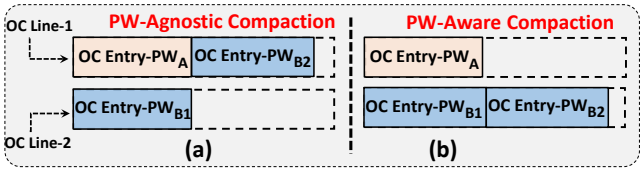


Fig. 13: PWAC (Example).

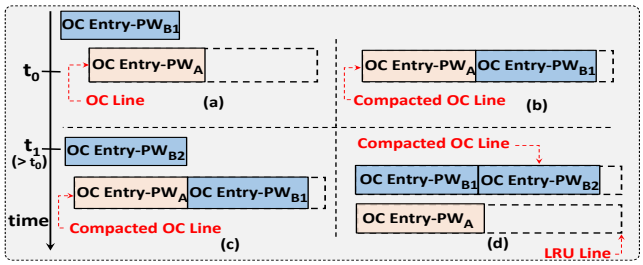


Fig. 14: F-PWAC (Example).

cache entry-PW$_{B1}$ is compacted with uop cache-PW$_A$ in line-1 while uop cache entry-PW$_{B2}$ is filled in line-2. Uop cache entry-PW$_{B1}$ and uop cache entry-PW$_{B2}$ belong to the same PW-B, however they are compacted in two separate lines. It is beneficial to compact uop cache entries belonging to the same PW in the same uop cache line because they are going to be fetched together as they belong to the same basic block and were only separated due to uop cache entry termination rules. Figure 13-(b) depicts the PW-Aware compaction scheme where uop cache entry-PW$_{B1}$ and uop cache entry-PW$_{B2}$ are compacted into line-2. To enable PWAC, each uop cache entry must be associated with a PW identifier (PW-ID) in the accumulation buffer and each uop cache entry tag now contains an additional PW-ID field.

3) Forced Prediction Window-Aware Compaction (F-PWAC): While PWAC, tries to compact uop cache entries from the same PW in the same line, it is still best-effort. There exists scenarios where PWAC is unsuccessful. Figure 14 presents such scenarios. Consider an uop cache entry-PW$_{B1}$, the first uop cache entry in PW-B is to be filled at time t_0 as shown in Figure 14-(a). Because the uop cache line containing uop cache entry-PW$_A$ has enough free space to accommodate uop cache entry-PW$_{B1}$, it is compacted with uop cache entry-PW$_A$ as in Figure 14-(b). However, at time t_1 ($> t_0$), uop cache entry-PW$_{B2}$, the second uop cache entry in PW-B needs to be written to the uop cache. In this case, it is not possible to compact uop cache entry-PW$_{B2}$ in the same line as uop cache entry-PW$_{B1}$ as it is compacted with uop cache entry-PW$_A$ and does not contain enough space to accommodate uop cache entry-PW$_{B2}$. PWAC misses out on such opportunities due to timing. Please note that PWAC could have successfully compacted uop cache entry-PW$_{B2}$ with uop cache entry-PW$_{B1}$ had uop cache entry-PW$_{B1}$ been written to a victim LRU line at time t_0. However, since uop cache entry-PW$_{B1}$ is compacted with another uop cache entry (of PW-A in this case), PWAC cannot compact uop cache entry-PW$_{B2}$ with uop cache entry-PW$_{B1}$ though they belong to same PW-B.

F-PWAC addresses such lost opportunities by forcing the uop cache entries of the same PW into the same line as

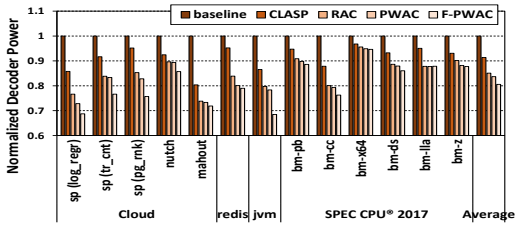


Fig. 15: Normalized decoder power.

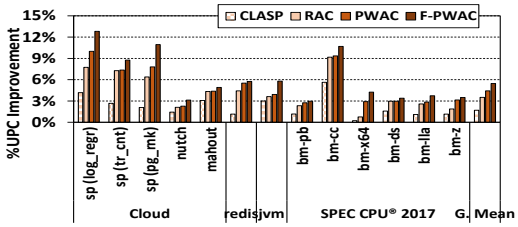


Fig. 16: Compaction (maximum of two uop cache entries per line) performance improvement.

depicted in Figure 14-(d) at time t_1. In F-PWAC, uop cache entry-PW_{B1} is read out and compacted with uop cache entry-PW_{B2}, while uop cache entry-PW_A is written to the LRU line after the victim entries (or a single entry if not compacted) are evicted. Uop cache entry-PW_A is written to the current LRU line to preserve hits to uop cache entry-PW_A because it is the most recently written entry to the uop cache if RAC is enabled. After writing to the LRU line, the replacement information of the line containing uop cache entry-PW_A is updated. Though F-PWAC requires an additional read of the previously compacted entry and an additional write during an uop cache entry fill, it is infrequent in steady state because it improves uop cache fetch ratio. F-PWAC compacts more uop cache entries belonging to the same PW compared to PWAC.

VI. Experimental Results

In this section, we present the results and analysis of our optimizations. All studies except for the sensitivity studies in Section VI-B1 assume a maximum of two compacted uop cache entries per uop cache line.

A. Main Results

Figure 15 shows the normalized decoder power for all proposed optimizations. CLASP reduces decoder power consumption over the baseline by 8.6% as more uops bypass the decoder. RAC, PWAC and F-PWAC further reduce the decoder power on an average by 14.9%, 16.3% and 19.4%, respectively. Figure 16 presents the improvement in performance (Uops Per Cycle or UPC) over the baseline for all our optimizations. Figure 17 shows uop cache fetch ratio (bar graph), average dispatch bandwidth (orange line graph) and average branch misprediction latency (green line graph) for the proposed mechanisms: CLASP, RAC, PWAC and F-PWAC, all normalized to a baseline uop cache system with 2K uops. In RAC, PWAC and F-PWAC, a maximum of two uop cache entries are compacted per line.

CLASP improves the uop cache fetch ratio by 11.6% on the average, with a maximum improvement as high as 24.9% for 502.gcc_r (bm-cc). CLASP also improves the average dispatch bandwidth over all workloads by an average of 2.2%, with a maximum improvement of 5.6% for 502.gcc_r (bm-cc). The average reduction in branch misprediction latency across all the workloads by CLASP is 2.01%, with 502.gcc_r (bm-cc) showing the maximum reduction at 4.75%. CLASP improves performance by 1.7%, with a maximum improvement of 5.6% for 502.gcc_r (bm-cc). CLASP only optimizes for the fragmentation due to uop cache entries spanning multiple sets while still containing a maximum of one uop cache entry per line. It does not address the fragmentation per uop cache line due to small OC entries.

RAC, PWAC and F-PWAC improve the uop cache fetch ratio across all the workloads by 20.6%, 22.9%, and, 28.77%, respectively. They also improve the average dispatch bandwidth by 4.3%, 5.14%, and, 6.3%, respectively. These compaction based optimizations reduce the branch misprediction latency by 3.45%, 4.25%, and, 5.23%, respectively. The corresponding performance improvements caused by RAC, PWAC and F-PWAC across all workloads are 3.5%, 4.4%, and, 5.45%, respectively. Note that in F-PWAC compaction technique, we first apply PWAC. If an uop cache entry can be compacted with another uop cache entry of the same PW, there is no need to use F-PWAC. F-PWAC is performed only when an uop cache entry with the same PW Id as that being installed in the uop cache is compacted with an uop cache entry of a different PW. When an uop cache entry being compacted belongs to only one PW, F-PWAC and PWAC cannot find another uop cache entry that belongs to the same PW, the fall-back compaction technique is RAC. Figure 18 shows the percentage of uop cache entries compacted. On an average, 66.3% of the entries written to uop cache are compacted without evicting any other entries. Figure 19 shows the distribution of allocation techniques used during compaction. On an average, 30.3% of the uop cache entries are compacted with RAC, 41.4% are compacted with PWAC, while the remaining 28.3% are compacted with F-PWAC. All results on compaction enable CLASP.

B. Sensitivity Results

1) Maximum Compacted Entries per line: The maximum number of allowable compacted uop cache entries per line is statically defined because the number of additional offset fields etc. in the uop cache tag array need to be provided at design time. While Figures 16 and 17 present results for a maximum of two allowable uop cache entries per line, Figures 20 and 21 presents the performance improvement and uop cache fetch ratio for an uop cache that can compact a maximum of three uop cache entries. Increasing the maximum compacted entries to three improves the performance by 6.0% as opposed to 5.4% for a maximum of two uop cache entries. The uop cache fetch ratio allowing a maximum of three compacted uop cache entries per line is 31.8% while that for two compacted uop cache entries is 28.2%. The small improvement in performance and uop cache fetch ratio over the two compacted uop cache entries is due to the fact that not many lines could compact

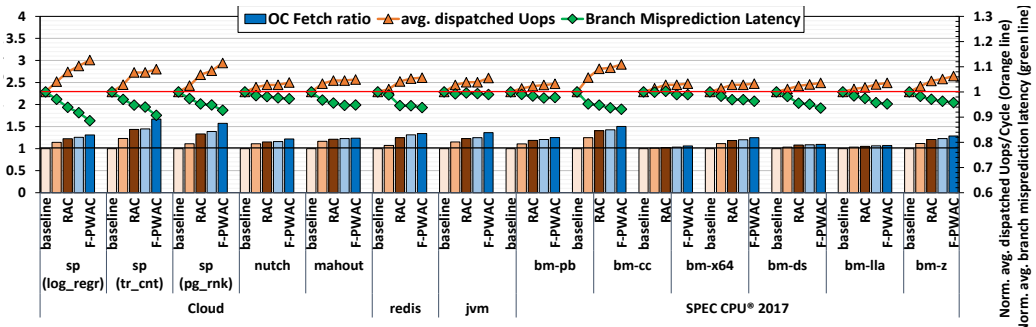


Fig. 17: Normalized uop cache fetch ratio (bar graph), average dispatched uops/cycle (orange line graph), and, average branch misprediction latency (green line graph) results for baseline, CLASP, RAC, PWAC, and F-PWAC.

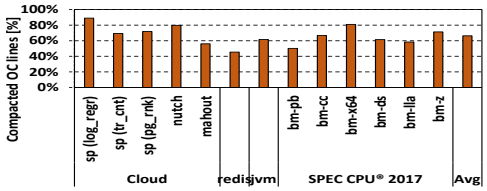


Fig. 18: Compacted uop cache lines ratio.

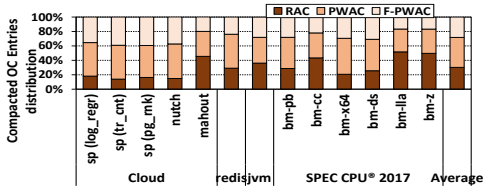


Fig. 19: Compacted uop cache entries distribution.

three uop cache entries. This is because, in most cases, two compacted uop cache entries do not leave enough free space to compact one additional uop cache entry.

2) Uop Cache Size: Figure 22 shows the performance improvement over a baseline uop cache containing 4K uops. F-PWAC improves the performance by 3.08% over a baseline uop cache. Maximum improvement is for 502.gcc_r (bm-cc) at 11.27%. The corresponding overall improvement in uop cache fetch ratio is 13.5% with the maximum for redis at 32.47%, while 502.gcc_r (bm-cc) incurs an uop cache fetch ratio increase of 25.3%. The dispatch bandwidth is improved by 3.75% with maximum 11.2% for 502.gcc_r (bm-cc). The branch misprediction latency is reduced by 4.43% while the decoder power is reduced by 13.94% overall.

VII. Related Work

Code compression has been investigated as an optimization that targets code reduction. It is typically performed at compile-time, link-time and at the binary level. Code reduction has been achieved by exploiting ISA-level redundancy

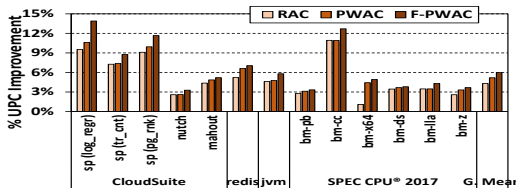


Fig. 20: Compaction (maximum of three uop cache entries per line) performance improvement.

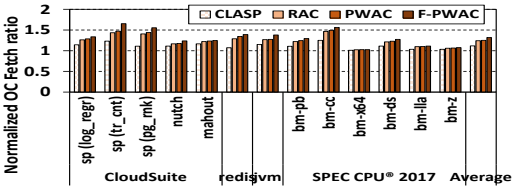


Fig. 21: Uop cache fetch ratio change with a maximum of three compacted uop cache entries per line.

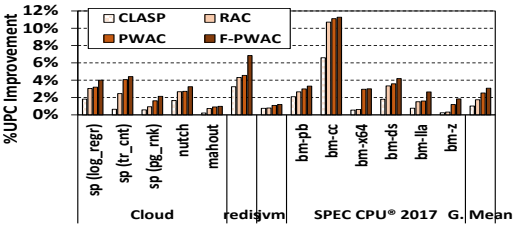


Fig. 22: Performance improvement over a baseline uop cache with capacity for up to 4K uops.

such as register reuse per instruction, narrow displacement values and shorter instruction opcodes for frequently used instructions with new ISA extensions by Waterman in [54]. Common instruction sequence repetition and identification of functionally equivalent code sequences has been proposed by Debray et al. in [26]. In [25], Cheung et al. extend code compression with predication and different set of flags to allow procedures represent different code sequences. In [24], Chen et al. compact repeating code sequences that form single-entry, multiple-exit regions. In [30], Fisher describes a code compaction technique that reduces the size of microcode generated in VLIW (statically scheduled) machines. In [28], Drinic et al. use Prediction Partial Matching (PPM) to compress code by constructing a dictionary of variable-length super-symbols that can identify longer sequences of instructions. In [53], Wang et al. propose modifications to the BitMask algorithm that records mismatched values and their positions in code sequences, to enable compression of longer code sequences. With the exception with the work on compressed ISA extensions in [54], all code compression methods described above operate at some level of the software stack with no hardware support. The work by Waterman in [54] describes new compressed ISA-level instructions to reduce code size. Our technique does not attempt to compress micro-ops and can be combined with

code compression to expand the scope of compaction in the uop cache.

Cache compaction has been introduced as support for cost-effective cache compression. Yang et al. in [37], Lee et al. in [38] and Kim et al. in [43] propose storing two logical compressed cache lines into one physical one only if each logical line is compressed to at least half its original size. Our compaction scheme does not have this limitation and can compact two uop groups as long as they meet the conditions required to construct an uop cache entry. Variable sub-block compaction techniques such as the ones used in the Variable Size Compression [18], Decoupled Compressed Cache [48] and Skewed Compressed Cache [47] are tailored for data accesses because they take into account the compressibility of data sub-blocks to reduce the overhead or avoid expensive recompaction. Our technique focuses on compacting together uncompressed code blocks only, therefore avoiding the need for recompaction due to dynamically changing compression ratios. Our technique also differs from other cache compaction techniques in that it targets a L0 small cache as opposed to large L2 and L3 caches. Thus, additional tag space needed to support compaction has a much smaller area overhead compared to that of the L2 and L3 caches and can be used to serve the compaction mechanism. Moreover, unlike work in [18] and [20] which require back pointers to ensure proper indexing of data sub-blocks, our approach does not require such metadata because we duplicate tags to index uop cache entries. Finally, uop cache compaction exploits code spatial locality to achieve better hit rates by compacting sequentially fetched uop groups. Similarly, data spatial locality across cache lines is explored for cache compaction in [48] by sharing a super-block tag across four data lines. However our compaction design does not utilize super-block tags, avoiding complications in the uop cache replacement policy.

Trace caches improve instruction fetch bandwidth by grouping statically non-contiguous but dynamically adjacent instructions [29, 27, 42]. The uop cache design we consider is a limited form of a trace cache because it stores uops beyond a predicted not-taken branch but not beyond a (predicted) taken branch. This design decision greatly simplifies uop cache tag design and avoids the complexity of traditional trace cache designs which require multiple tags per cache line. In addition, our proposed compaction policies are different than that of trace caches in that we allow statically non-contiguous, dynamically non-contiguous uops to be placed in the same uop cache line, based on their size and metadata. Finally, the goal of our compaction methods is to increase the uop cache hit rate (which also lowers energy consumption) and not to increase the peak uop fetch bandwidth by fetching either multiple uop groups per cycle or beyond (predicted) taken branches.

Another set of techniques that target improved cache efficiency, similar to our proposed uop compaction approach, is code reordering. In [39, 31, 33, 44] authors present different compile-time or link-time techniques that exploit subroutine-level temporal locality to adopt a different placement of subroutines in the binary and reduce I-cache misses. Authors in [2, 23, 45, 55] exploit basic block reordering to convert more branches into not-taken and improve I-TLB and I-cache hit rate. Ramirez et al. in [46] employ profile guided code reordering to maximize sequentiality of instructions without hurting I-cache performance.

Code-reordering work differs from our proposals because it (a) operates at the compiler or linker level and (b) can not dynamically detect compaction opportunities. Moreover, our technique does not change the binary code layout, does not need profile information or software stack modifications, operates transparently to the user and improves both performance and power. In addition, our technique takes into consideration the fragmentation of code due to the presence of taken branches to improve the placement of sequential code fragments in uop cache. Finally, please note that our optimizations are orthogonal to prior code reordering proposals and can work on binaries that have been rewritten with optimized code layouts.

VIII. Conclusion

In this paper, we demonstrate that x86 uop caches are fragmented because of the uop cache entry terminating conditions that are used to simplify the uop cache baseline architecture. To address this fragmentation, we propose two optimizations: CLASP and Compaction. CLASP reduces uop cache fragmentation by relaxing the I-cache line boundary termination condition. Compaction reduces uop cache fragmentation by exploiting empty space across possibly unrelated uop cache entries in the same uop cache set. We evaluate three compaction policies: RAC, PWAC, and F-PWAC. RAC compacts temporally correlated uop cache entries in the same uop cache line, while PWAC compacts uop cache entries from the same PW in the same uop cache line. F-PWAC optimizes the best-effort PWAC by forcing the compaction of uop cache entries from the same PW to the same uop cache line. CLASP and Compaction (a) improve uop cache utilization/fetch ratio, dispatch bandwidth, average branch misprediction penalty and overall performance, and (b) reduce decoder power consumption. These optimizations combined improve performance by 5.3%, uop cache fetch ratio by 28.8% and dispatch bandwidth by 6.28%, while, reducing the decoder power consumption by 19.4% and branch misprediction latency by 5.23% in our workloads.

Acknowledgment

The authors would like to thank the AMD internal reviewers, Gabriel H. Loh, Sreenivas Vadlapatla, and Nuwan Jayasena, whose reviews helped improve the quality of this paper. Additionally, the authors would also like to thank the AMD product team members for their help with answering several questions about the front-end design and simulation infrastructure. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] "AMD Family 17h Processors," https://urlzs.com/rwTW9.

[2] "Improved Basic Block Reordering," https://arxiv.org/pdf/1809.04676.pdf.

[3] "Intel Haswell Uop Cache," https://www.anandtech.com/show/6355/intels-haswell-architecture/6.

[4] "Intel Pentium-4/Xeon Software Dev. Manual," https://cutt.ly/1eZu9H3.

[5] "Intel Performance Optimization Manual," https://cutt.ly/PX7i5v.

[6] "Intel Sandybridge Uop Cache," https://www.realworldtech.com/sandy-bridge/4/.

[7] "Mahout: Benchmark for creating scalabale performant machine learning applications," https://mahout.apache.org.

[8] "Nutch Benchmark," https://nutch.apache.org/.

[9] "Redis Benchmark," https://redis.io/.

[10] "Redis benchmarked using Memtier," https://github.com/RedisLabs/memtier_benchmark.

[11] "SimNow," https://developer.amd.com/simnow-simulator/.

[12] "SparkBench IBM Benchmark," https://codait.github.io/spark-bench/workloads/.

[13] "SparkBench IBM Benchmark, logistic regression input," https://codait.github.io/spark-bench/workloads/logistic-regression/.

[14] "Standard Performance Evaluation Corporation, "SPEC CPU® 2017,"," https://www.spec.org/cpu2017/.

[15] "Standard Performance Evaluation Corporation, "SPECjbb®"," https://www.spec.org/jbb2015.

[16] "Synposys Prime Time PX (PTPX)," https://cutt.ly/X9wCgU.

[17] "X86 Processor Microarchitectures," https://www.agner.org/optimize/microarchitecture.pdf.

[18] A. Alameldeen and D. Wood, "Adaptive Cache Compression for High Performance Processors," in *International Symposium on Computer Architecture (ISCA)*, 2004.

[19] T. Anderson and S. Agarwala, "Effective hardware-based two-way loop cache for high performance low power processors," in *Proceedings 2000 International Conference on Computer Design (ICCD)*, 2000.

[20] A. Arelakis and P. Stenstrom, "SC2: A Statistical Compression Cache Scheme," in *International Symposium on Computer Architecture (ISCA)*, 2014.

[21] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis, "Energy and performance improvements in microprocessor design using a loop cache," in *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, 1999.

[22] M. Brandalero and A. C. S. Beck, "A mechanism for energy-efficient reuse of decoding and scheduling of x86 instruction streams," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2017.

[23] B. Calder and D. Grunwald, "Reducing branch costs via branch alignment," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[24] W. K. Chen, B. Li, and R. Gupta, "Code compaction of matching single-entry multiple-exit regions," in *Proceedings of the 10th International Conference on Static Analysis (SAS)*, 2003.

[25] W. Cheung, W. Evans, and J. Moses, "Predicated instructions for code compaction," in *Software and Compilers for Embedded Systems (SCOPES)*, 2003.

[26] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 2000.

[27] Y. P. D.H. Friendly, S. Patel, "Alternative Fetch and Iissue Policies for the Trace Cache Fetch Mechanism," in *Proceedings of the International Symposium on Microarchitecture*, ser. Micro, 1997.

[28] M. Drinić, D. Kirovski, and H. Vo, "Ppmexe: Program compression," *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 2007.

[29] J. S. E. Rotenberg, S. Bennett, "Trace Cache: A Low Latency Approach to High Bandwith Instruction Fetching," in *Proceedings of the International Symposium on Microarchitecture*, ser. Micro, 1996.

[30] Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers (TC)*, 1981.

[31] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder, "Procedure placement using temporal ordering information," in *Proceedings of 30th Annual International Symposium on Microarchitecture (MICRO)*, 1997.

[32] A. González, F. Latorre, and G. Magklis, "Processor microarchitecture: An implementation perspective," *Synthesis Lectures on Computer Architecture*, 2010. [Online]. Available: https://doi.org/10.2200/S00309ED1V01Y201011CAC012

[33] A. H. Hashemi, D. R. Kaeli, and B. Calder, "Efficient procedure mapping using cache line coloring," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI)*, 1997.

[34] M. Hirki, Z. Ou, K. N. Khan, J. K. Nurminen, and T. Niemi, "Empirical study of the power consumption of the x86-64 instruction decoder," in *USENIX Workshop on Cool Topics on Sustainable Data Centers (CoolDC)*, 2016.

[35] S. Hu, I. Kim, M. H. Lipasti, and J. E. Smith, "An approach for implementing efficient superscalar cisc processors," in *The Twelfth International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

[36] G. Intrater and I. Spillinger, "Performance evaluation of a decoded instruction cache for variable instruction-length computers," in *Proceedings the 19th Annual International Symposium on Computer Architecture (ISCA)*, 1992.

[37] Y. Z. J. Yang and R. Gupta, "Frequent Value Compression in Data Caches," in *33rd International Symposium on MicroArchitecture (Micro)*, 2000.

[38] W. H. J.S. Lee and S. Kim, "An On-Chip CacheCompression Technique to reduce Decompression Overhead and Design Complexity," *Journal of Systems Architecture: the EuroMicro Journal*, 2000.

[39] J. Kalamationos and D. R. Kaeli, "Temporal-based procedure reordering for improved instruction cache performance," in *Proceedings of International Symposium on High-Performance Computer Architecture(HPCA)*, 1998.

[40] R. Kumar, S. Pati, and K. Lahiri, "Darts: Performance-counter driven sampling using binary translators," in *IEEE Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017.

[41] L. H. Lee, W. Moyer, and J. Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," in *In Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 1999, pp. 267–269.

[42] G. T. M. Postiff and T. Mudge, "Performance limits of trace caches," *Journal of Instruction Level Parallelism*, 1999.

[43] T. A. N.S. Kim and T. Mudge, "Low-Energy Data Cache using Sign Compression and Cache Line Bisection," in *Proceedings of the 2nd Workshop on Memory Performance Issues*, ser. WMPI, 2002.

[44] G. Ottoni and B. Maher, "Optimizing function placement for large-scale data-center applications," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017.

[45] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[46] A. Ramírez, J.-L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero, "Software trace cache," in *ACM International Conference on Supercomputing (ICS)*, 2014.

[47] S. Sardashti, A. Seznec, and D. A. Wood, "Skewed compressed caches," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[48] S. Sardashti and D. A. Wood, "Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[49] A. Seznec, "A new case for the tage branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[50] L. M. Shen J.P, *Modern processor design: Fundamentals of Superscalar processors*. McGraw-Hill, 2012.

[51] B. Solomon, A. Mendelson, R. Ronen, D. Orenstien, and Y. Almog, "Micro-operation cache: a power aware frontend for variable instruction length isa," *IEEE Transactions on Very Large Scale Integration Systems (VLSI)*, 2003.

[52] M. Taram, A. Venkat, and D. Tullsen, "Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[53] W. J. Wang and C. H. Lin, "Code compression for embedded systems using separated dictionaries," *IEEE Transactions on Very Large Scale Integration Systems (VLSI)*, 2016.

[54] A. Waterman, "Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed," Master's thesis, University of California, Berkeley, 2011.

[55] C. Young, D. S. Johnson, M. D. Smith, and D. R. Karger, "Near-optimal intraprocedural branch alignment," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI)*, 1997.