Improving the Utilization of Micro-operation Caches in x86 Processors

Hyperthreads (**6**)
Sumon Nath
(21Q050007)

I. Summary

Micro-operation(uop) caches are used in modern x86 processors in order to bypass the high latency and power hungry fetch and decode units. The authors demonstrate the impact of uop caches on performance improvement and power reduction. They show that current design of uop caches are highly fragmented due to termination conditions(discussed in section II) and proposes two solutions to reduce the fragmentation.

II. Details

A. Fragmentation

- The uops decoded from the decoder unit is stored in a buffer called the accumulation buffer. Once a termination condition is met, all the uops in the accumulation buffer are combined together into a uop entry and filled into the uop cache.
- The first termination condition is the I-cache line boundary termination condition, which terminates a uop cache entry when the I-cache line boundary is reached. In case sequential code spans across I-cache line boundary, this termination condition unnecessarily creates two cache entries. Most often these entries are small and leads to uop cache fragmentation.
- The second termination condition is predicted taken branches. Similar to the first termination condition, this leads to small uop cache entries and leads to fragmentation.
- There are some other termination conditions, like the maximum number of immediate/displacement field per uop cache entry and the maximum number of micro-coded instructions per uop cache entry, which leads to uop cache fragmentation.

B. Solution I - CLASP

- Cache line boundary agnostic uop cache design(CLASP) is the first solution proposed by the authors to reduce uop cache fragmentation.
- CLASP targets the I-cache line boundary termination condition and basically relaxes it. This leads to merging of uop cache entries spanning across I-cache line boundary belonging to sequential code.
- As two uop cache entries are merged together by CLASP, it doubles the uop dispatch bandwidth.

C. Solution II - Compaction

- Compaction targets the termination conditions other than I-cache line boundary termination condition. As the name suggests, it tries to compact unrelated uop cache entries in the same uop cache line. As the entries are not merged together and are just placed in the same cache line, the dispatch bandwidth remains the same unlike CLASP
- The main challenge with compaction is which uop cache entries to compact together to get maximum uop cache utilization. The paper proposes three compaction techniques to address this issue.
- The first compaction technique they propose is the Replacement aware compaction(RAC). RAC tries to compact an incoming uop cache entry with the most recently used(MRU) cache line. This ensures that temporally correlated uop cache entries are compacted together in the same uop cache line.
- The second compaction technique is Prediction window aware compaction(PWAC) which tries to compact uop cache entries from the same prediction window.
- The last compaction technique is Forced-PWAC(FPWAC) which forces compaction an incoming uop cache entry with a cache entry from the same prediction window, in case the cache entry resident in the cache is previously compacted with a uop cache entry from some other prediction window.

III. Positives

- With all the mitigation techniques working together there is a performance improvement of 5.3% compared to a baseline with a uop cache which supports 2K entries.
- The average decoder power consumption decreases by 19.4% on average with all mitigation techniques working simultaneously considering the same baseline.

IV. Negatives

- The compaction technique FPWAC incurs an additional read and write penalty.

Review - 3 : Improving the utilization of Micro-operation Caches in X86 Processors

1st Pass:

→ why CISC? - reduce inst. fetch energy cost & bandwidth.
→ ~~why CISC instructions broken into fixed length uops.~~
→ High throughput CISC decoding requires energy hungry logic.
→ After decoding CISC insts are broken into uops.
→ uop cache is used to bypass costly decoding.

problems with uop cache:
→ heavily fragmented

solutions — 1) CLASP
— 2) uop cache compaction } these 2 are complimentary.

Background

A. Frontend.
→ Branch predictors ~~predi~~ generate prediction window which is a seq. of consecutive instructions predicted by the branch predictor.
→ PW can start or end anywhere in an I-cache line.
why is this info. useful?

* ~~instructions~~ 3 hardware structures are used:
I-cache, uop cache, loop cache.
→ inst. fetched from I-cache need to be decoded to get the uops.
→ uops are cached in the uop cache & loop cache.
→ In case of hit in uop/loop cache, ~~no need to~~ heavy penalty of decoding can be avoided.

→ loop cache contains uops of loops

B. uop cache

→ A uops are of fixed length CISC instr / partially decoded fixed length RISC operations.
This is implementation dependent
Key takeaway uops are fixed length.

→ uop cache entry = set of uops
- # of uops in a uop cache entry depends on the # of instructions in the PW.
- indirectly the size of uop cache entry thus depends on the PW terminating conditions (a) I cache line boundary (b) predicted taken branch
- Also the max # of uops allowed per uop cache entry is restricted by the uop cache line size.
- A uop cache entry can store ~~multiple~~ uops from multiple sequential PW.
- ~~A PW can also span~~
A PW (the uops in the PW) can span across two uop cache entries due to the ~~limit~~ limit on # of uops per cache entry, due to fixed cache line size.

→ uops are accumulated in a buffer till a termination condition is met, after which the ~~uops are uops~~ forming a uop cache entry is written into the uop cache.

→ uop cache is indexed by the starting address of the PW.

* A uop cache entry may not occupy an entire cache line, i.e., the # of ~~uops~~ in a cache entry varies.

Problem with trace caches.
→ trace cache builds up uop cache entries beyond taken branches
→ invalidating is complex
→ power & complexity overhead due to multi-branch predictio

Intro

Modern processors employ CISC ISA. ~~& uses a decoupled front-end~~.

↓

Decoding is high latency & power hungry due to variable length ISA

↓

~~Front~~ ~~Bottleneck in the~~ Restricts ~~front-end~~ high dispatch bandwidth

(~~For~~ For high exec. BW, high dispatch BW is necessary).

↓

* Also Instructions are decoded into fixed length uops to facilitate execution

→ In modern processors, Instruction are fetched & decoded in uops.

Problem: 1) Decode: high latency & power.

2) Ifetch miss: high penalty.

solution: uops are cached in uop cache.

* on hit, Ifetch & decode is bypassed saving latency & power.
* hit on entries containing predicted branches can reduce Branch misprediction. latency.

~~motivation~~:

problem with uop caches:

→ fragmentation (discussed later) → leads to lower cache utilization & thus low hit rate

solutions proposed:

1. CLASP: merges sequential uop cache entries into same cache line
   improves performance by 5.6%.
2. Compaction: with CLASP improves performance by 12.8%.

Motivation:

UPC ≈ uops committed per cycle. ~~(performance)~~ (proxy for frontend performance)

~~uop cache BW~~

baseline: uop cache with 2K uops.

uop cache with 64K uops.

- UPC improvement - 11.2%
- reduction in decoder power consumption - 39.2%
- Fetch ration (~ hit rate) - 69.7%
  improvement in.
  - increase in dispatch BW - 13.01%.
  - decrease in avg. branch misprediction penalty - 10.31%

~~Two~~ Two ways to ~~beat~~ ~~go ahead~~ improve performance

Baseline: 2K uops.

1) throw money ~~at the~~ & increase the cache size.
2) identify current design. ~~baseline~~ has some inefficiency & optimize.

Paper targets point - 2.

Sources of Inefficiency (fragmentation)

- ⊛ → cache lines are fragmented
  - ↳ reason: terminating conditions which govern uop cache entry formation.
- ⊛ 72% of cache entries are less than 40 B (whole is the size of a cache line).
- → 49.4% of cache entries are terminated due to

→ The second terminating condition causing fragmentation.

↳ uop cache entries terminated due to I-cache line boundary.

— uop cache entries corresponding to contiguous I-cache lines occupy different lines even though control flow between the uops / instructions is sequential.

Why this terminating condition?

→ To avoid building up traces in uop cache that necessitates a flush of the entire uop cache upon invalidation.

Solution 1 : CLASP

Cache Line boundary Agnostic uop cache design.

→ relaxes the I-cache line boundary constraint, & allows uop cache entries to build beyond I-cache line boundaries provided control flow is sequential across the I-cache line boundary.

→ About 35% of uop cache entries span across I-cache line boundaries after removing the boundary termination condition.

⊛ uop cache entries crossing I-cache line boundaries are merged together into an OC entry. This increases the dispatch BW, as more than one uop entries are dispatched on a single uop cache hit.

Solution-2: Compaction

→ Remaining termination conditions still lead to fragmentation

→ More than one uop cache entries are compacted together into a single uop cache line, provided the entries can be accomodated in the cache line.

→ Unlike CLASP uop cache entries are not merged into a single OC entry. Therefore, dispatch BW remains same.

→ Tags for both uop cache entries are stored for a single cache line.

Problem with compaction.

→ Selecting a victim according to LRU may not create enough space for the new uop cache entry.
Need to find a new entry.
This increases Ocache fill latency & can lead to decode stalls which negatively impacts performance.
The complication arises as size of new uop cache entry is only known until fill time.

Sol: Instead of having replacement state per cache entry, have it per cache line.
Evict entire Ocache line instead of just one entry.
Thus, there will be always space for new uop cache entry.

→ Crucial: which entries to compact together, as compacted entries will be evicted together.
And this will impact fetch ratio.

1) Replacement Aware compaction (RAC):

→ during uop cache fill RAC attempts to compact the new entry with most recently used OC entry.

→ This ensure compacted OCache entries are closed to each other temporally.

2) PWAC (PW aware compaction).

→ A single prediction window may lead to multip OC entries due to other terminating conditions.

→ Observed that almost 31.6% of PWs lead to multiple OC entries.

→ And normally these would go to different cache lines.

Sol: It is beneficial if these were in the same line as they are from same PW.

PWAC tries to compact entries from the same PW.

3) FPWAC. (Forced-PWAC).

→ In case PWAC can't compact OC entries from same PW, due to the fact that one entry (B) from PW was already compacted with an entry from PW (A).

And new entry from PW (B) can't be compacted with B.

→ FPWAC forces compaction of entries from same PW (B)

problem: additional read + write.

infrequent than PWAC.