Mitigating Front-End Stalls in Warehouse-Scale Computers

Grant Ayers*, Nayana Prasad Nagendra**,†, Heiner Litz*,†, Svilen Kanev†, Hyoun Kyu Cho†, Tipp Moseley†, Christos Kozyrakis*, Partha Ranganathan†

Stanford University*
Princeton**
UC Santa Cruz†
Google‡
CPU Dataflow

Instructions
(front-end)

Data
(back-end)
Google web search CPU performance

Front-end stalls (39%)
- 13.8%
- 9.7%

Back-end stalls (29%)
- 15.4%
- 8.5%

Web search leaf node CPU utilization

- Retiring: 32%
- Back-End: Core: 20.5%
- Back-End: Memory: 15.4%
- Bad Speculation: 8.5%
- Front-End: Latency: 9.7%
- Front-End: Bandwidth: 13.8%
Google web search performance

“Memory Hierarchy for Web Search [1]”

Google web search performance

- Front-end stalls (39%)
- Back-end stalls (29%)

Web search leaf node CPU utilization

- Retiring: 32%
- Back-End: Core: 8.5%
- Back-End: Memory: 20.5%
- Bad Speculation: 15.4%
- Front-End: Latency: 9.7%
- Front-End: Bandwidth: 13.8%
Research objective

Understand and mitigate front-end stalls for server-class workloads

1. Characterize instruction behavior in large data center workloads
2. Understand what has already been done
3. Propose a new “code prefetch” instruction and feedback-directed optimization strategy to reduce instruction cache misses

Results show up to 96% instruction cache reduction with minimal overhead
Instruction Characterization
Instruction working sets

Instruction working sets are often 100x larger than instruction caches

- Web search, bigtable are ~4 MiB
- SPEC 400.perlbench is ~172 KiB
- 10x higher MPKI than SPEC, 1.5-8x higher than CloudSuite
- 15-30% potential lost across Google fleet (2-3x higher than SPEC)

Instruction growth is as high as 20% per year
Fleet-wide execution profile

Misses initially rise faster, suggesting some pointwise optimizations are left.

More significantly, the long tail means automation is necessary to significantly reduce cache misses.
What causes instruction cache misses?

Instructions are normally fetched sequentially ⇒ Simple next-line prefetchers (NLP) handle this well

Misses are all caused by control-flow-changing instructions such as calls, returns, branches, and exceptions

```
<__libc_fcntl>:
...
cmpl  $0x2,0x18(%rsp)
mov   0x1c(%rsp),%eax
jne   512aca <__libc_fcntl+0x5a>
neg   %eax
jmp   512aca <__libc_fcntl+0x5a>
nopl  0x0(%rax)
mov   %edi,0xc(%rsp)
mov   %rdx,(%rsp)
callq  515780 <__libc_enable_asynccancel>
mov   (%rsp),%rdx
mov   %eax,%r8d
mov   $0x7,%esi
mov   0xc(%rsp),%edi
mov   $0x48,%eax
syscall
cmp   $0xfffffffffffff000,%rax
```
Fleet-wide instruction composition

On average, 15% of all executed instructions change control flow

- Mostly direct branches (>80%)
- Relatively little indirection (<0.5%)
Which instructions cause misses?

On average, 15% of all executed instructions change control flow

- Mostly direct branches (>80%)
- Relatively little indirection (<0.5%)

Which types cause cache misses?

Intuitively, calls span larger distances and are therefore more likely than branches to miss
Miss-causing instructions

Most misses are caused by calls (53%), followed by branches (46%)

⇒ Branches are more inherently cacheable
⇒ Call optimization has more impact

Indirection causes significant (24%) misses despite being the least-executed (<0.5%)

⇒ Difficult to address indirection statically
Instruction prefetchability

**What to prefetch**
- Most misses (76%) are caused by instructions with statically-encoded targets
- Most (90%) indirect instructions have fewer than five targets

**When to prefetch**
- Dynamic profiling gives accurate and relevant call graphs for calculating injection sites and indirect targets
- E.g., GWP, Intel PEBS/PT/LBR, Pin/DynamoRio
## Existing approaches

<table>
<thead>
<tr>
<th></th>
<th>Invasive hardware</th>
<th>Large on-chip storage</th>
<th>Dynamic program behavior</th>
<th>Commercial viability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming / history</td>
<td>Red</td>
<td>Red</td>
<td>Green</td>
<td>Red</td>
</tr>
<tr>
<td>Fetch/Branch-directed</td>
<td>Red</td>
<td>Yellow</td>
<td>Green</td>
<td>Red</td>
</tr>
<tr>
<td>Static software</td>
<td>Green</td>
<td>Green</td>
<td>Red</td>
<td>Red</td>
</tr>
</tbody>
</table>
### Existing approaches

<table>
<thead>
<tr>
<th></th>
<th>Invasive hardware</th>
<th>Large on-chip storage</th>
<th>Dynamic program behavior</th>
<th>Commercial viability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming / history</td>
<td>![Red]</td>
<td>![Red]</td>
<td>![Green]</td>
<td>![Red]</td>
</tr>
<tr>
<td>Fetch/Branch-directed</td>
<td>![Red]</td>
<td>![Yellow]</td>
<td>![Green]</td>
<td>![Red]</td>
</tr>
<tr>
<td>Static software</td>
<td>![Green]</td>
<td>![Green]</td>
<td>![Red]</td>
<td>![Red]</td>
</tr>
<tr>
<td>Dynamic software</td>
<td>![Green]</td>
<td>![Green]</td>
<td>![Green]</td>
<td>![Green]</td>
</tr>
</tbody>
</table>
Dynamic Software Instruction Prefetching
Instruction prefetching overview

Program binary

Miss profile

Control flow profile

Prefetch Injection

Optimized program binary
This is easy, right?!
Challenge: Fan-in

Insert everywhere?
Challenge: Fan-in

Cardinality of paths generally increases as you move back in time

- Requires additional prefetch injections
- Each prefetch site has reduced impact

Solution: Leverage profiling information to only inject useful prefetches
Challenge: Fan-out

time ➔ instructions

miss

Prefetch H

A B C D E F G H I
Challenge: Fan-out

Injection locations can lead to a high cardinality of future code paths

- Prefetches are useless and waste resources
- Performance can degrade due to cache pollution

Solution: Filter out high-fan-out paths and leverage a *dynamic window injection* (possible that we won't execute H at all)
Dynamic window injection

Candidate injection sites:

A. ?
Dynamic window injection

Candidate injection sites:

A. **No** (high fan-out)
B. ?
Dynamic window injection

Candidate injection sites:

A.  No (high fan-out)
B.  Maybe (might reduce A fan-out)
C.  ?
Dynamic window injection

Candidate injection sites:

A. No (high fan-out)
B. Maybe (might reduce A fan-out)
C. Yes (min. fan-in, no fan-out)
D. ?
E. ?
Dynamic window injection

Candidate injection sites:

A. No (high fan-out)
B. Maybe (might reduce A fan-out)
C. Yes (min. fan-in, no fan-out)
D. No (high fan-out loop)
E. No (high fan-out loop)
F. ?
Dynamic window injection

Candidate injection sites:

A. No (high fan-out)
B. Maybe (might reduce A fan-out)
C. Yes (min. fan-in, no fan-out)
D. No (high fan-out loop)
E. No (high fan-out loop)
F. Yes (helps D,E, conflicts with C?)
Hardware requirements

**Code prefetch instruction**

- Loads data into L1-I (optionally L2/L3)
- Utilizes the iTLB
- Loads data into S state (for MESI systems)

**Very similar to existing instructions:**

- x86: `prefetcht*`, `prefetchnta`
  - Data only, DTLB
- ARM: `pli`
  - Implementation-defined behavior
  - Insufficient level of control
- POWER: `icbt`
  - Insufficient level of control
Prefetching results

Google Web Search

- Up to 96% miss coverage
- Less than 1% execution overhead
- Minimal code growth (< 0.01%)
- Timing simulations in zsim indicate 11% performance improvement (11.6% cap)
Ongoing work

1. Full evaluation with a code prefetch instruction
2. Improved path selection and filtering
3. Extend to branch target preloading
Conclusions

1. The CPU front-end is a critical bottleneck for data-center-scale workloads
2. Effective solutions need to be automated and commercially-viable
3. With profile-guided optimization and a single code prefetch instruction, we can improve even simple front-end architectures and capture nearly all instruction cache misses
Questions?