CONSTABLE

Improving Performance and Power Efficiency by Safely Eliminating Load Instruction Execution

Rahul Bera*  Adithya Ranganathan*  Joydeep Rakshit  Sujit Mahto
Anant V. Nori  Jayesh Gaur  Ataberk Olgun  Konstantinos Kanellopoulos
Mohammad Sadrosadati  Sreenivas Subramoney  Onur Mutlu

SAFARI
SAFARI Research Group
safari.ethz.ch

ETH zürich

intel®
Key Problem

Load instructions are a key limiter of instruction-level parallelism (ILP)

Data Dependence
Stall load-dependent instructions due to long load execution latency

Resource Dependence
Stall other loads due to contention in load execution resources

(e.g., address generation unit, load port, ...)

SAFARI
Prior Works on Tolerating Load Latency

• Load value prediction (LVP) [Lipasti+, ASPLOS’96; Sazeides+, MICRO’96; ...]
• Memory renaming (MRN) [Moshovos+, ISCA’97; Tyson+, MICRO’97; ...]

Mitigate Data Dependence
By speculatively executing load-dependent instructions using a predicted load value

Do Not Mitigate Resource Dependence
Predicted load still gets executed to verify speculation, consuming execution resources
Motivation

Safely breaking load data dependency without executing a load instruction may provide additional performance benefits.

How do we start?

By finding load instructions that repeatedly produce identical results across dynamic instances.
Key Finding I: Global-Stable Loads

• Some loads repeatedly fetch **the same data value from same load address** across entire workload

  - Both operations, **address generation & data fetch**, produce identical results across all dynamic instances

  - Prime targets for **breaking data dependency without execution**

Global-Stable Load
Key Finding I: Global-Stable Loads

Nearly 1 in every 3 dynamic loads is a global-stable load

Across a wide range of 90 workloads
In the Paper: Analysis of Global-Stable Loads

• Why do these loads even exist in well-optimized real-world workloads?
  - Accessing **global-scope variables**
  - Accessing **local variables of inline functions**
  - **Limited** set of architectural registers

• **Can increasing architectural registers help?**
  - **Very small change** even after doubling x64 registers

• **Deeper characterization** of global-stable loads
  - Which **addressing mode** do they use?
  - How **far away** do they appear in a workload?
In the Paper: Analysis of Global-Stable Loads

- Why do these loads even exist in well-optimized real-world workloads?
  - Accessing global scope variables
  - Accessing local variables of inline functions
  - Limited set of architectural registers

- Can increasing architectural registers help?
  - Very small change even after doubling x64 registers

- Deeper characterization of global-stable loads
  - Which addressing mode do they use?
  - How far away do they appear in a workload?

But do they limit ILP even when using load value prediction and memory renaming?
Key Finding II: Global-Stable Loads Cause Resource Dependence

All execution cycles where
at least one load port is utilized

In an aggressive OoO processor with 6-wide issue, 3 load ports, a load value predictor (EVES [Seznec, CVP'18]), and memory renaming enabled
Key Finding II: Global-Stable Loads Cause Resource Dependence

All execution cycles where at least one load port is utilized

In an aggressive OoO processor with 6-wide issue, 3 load ports, a load value predictor (EVES [Seznec, CVP'18]), and memory renaming enabled.
Key Finding II: Global-Stable Loads Cause Resource Dependence

All execution cycles where at least one load port is utilized.

Even when using load value prediction and memory renaming, global-stable loads limit ILP due to resource dependence.

What’s the performance headroom of mitigating the resource dependence?
Key Finding III: High Performance Headroom

Mitigating both data and resource dependence has more than 2x the performance benefit of mitigating only data dependence of global-stable loads.

Ideal elimination of global-stable loads exceeds performance of a processor with 2x wider load execution width.
Load Execution Resources Lag Behind

![Graph showing parameter scaling and load execution ports]

- **Load execution ports**
  - Sandy Bridge (2010)
  - Haswell (2014)
  - Skylake (2015)
  - Sunny Cove (2019)
  - Golden Cove (2021)
  - Lion Cove (2024)

Parameter scaling:
- 3.4x
- 3x
- 1.5x
Mitigating load resource dependence has high performance potential in recent and future generation processors.
Our Goal

To improve instruction-level parallelism by mitigating both load data dependence and resource dependence.
Mitigates both load data dependence and load resource dependence

By safely eliminating the entire execution of a load instruction
Constable: Key Insight

```
add rax, 0x10
mov r8, [rbp+0x8]
sub rax, r8
cmp rsi, rax
jle 0x40230e
add rax, 0x10
mov r8, [rbp+0x8]
sub rax, r8
cmp rsi, rax
jle 0x40230e
add rax, 0x10
```

Dynamic instruction stream

LD₁

Two successive dynamic instances of the same static load instruction

LD₂
If the source register `rbp` has not been modified

- `add rax, 0x10` (LD₁)
- `mov r8, [rbp+0x8]` (LD₁)
- `sub rax, r8`
- `cmp rsi, rax`
- `jle 0x40230e`
- `add rax, 0x10`

LD₂ would have the same address as LD₁

Address generation of LD₂ can be eliminated

If no store or snoop request to address `[rbp+0x8]`

- `ld 1` (LD₂)
- `ld 2` (LD₂)
- `add rax, 0x10` (LD₂)
- `sub rax, r8` (LD₂)
- `cmp rsi, rax` (LD₂)
- `jle 0x40230e` (LD₂)

LD₂ would fetch the same data as LD₁

Data fetching of LD₂ can be eliminated
**Constable: Key Steps**

- **Dynamically identify** load instructions that have historically fetched the same data from the same load address (i.e., *likely-stable*)

- **Eliminate execution** of likely-stable loads by tracking modifications to their source registers and their load addresses
Prior Related Literature

Rich literature on skipping redundant computations by **memoizing previously-computed results**

[Michie, Nature’68; Harbison+, ASPLOS’82; Richardson, SCA’93; Sodani+, ISCA’97; González+, ICPP’99; ...]

**Aim to memoize every instruction**

*including multiple dynamic instances of each instruction*

**Require large memoization buffer**

*Often bigger than the size of L1 data cache*
Key Improvements over Literature

Rich literature on skipping redundant computations by memoizing previously-computed results

[Michie, Nature’68; Harbison+, ASPLOS’82; Richardson, SCA’93; Sodani+, ISCA’97; González+, ICPP’99; ...]

1 Focus only on loads that are likely stable

- Lower storage overhead with high load elimination coverage
- Lower design complexity Fewer port requirements, lower power
Key Improvements over Literature

Rich literature on skipping redundant computations by memoizing previously-computed results

[Michie, Nature'68; Harbison+, ASPLOS’82; Richardson, SCA’93; Sodani+, ISCA’97; González+, ICPP’99; ...]

1. **Focus** only on loads that are likely stable

2. Eliminate loads *early* in the pipeline

   **Elimination at rename stage**
   by explicitly monitoring changes to the source registers and load address of a likely-stable load
Key Improvements over Literature

Rich literature on skipping redundant computations by **memoizing previously-computed results**

[Michie, Nature’68; Harbison+, ASPLOS’82; Richardson, SCA’93; Sodani+, ISCA’97; González+, ICPP’99; ...]

1. **Focus** only on loads that are likely stable

2. Eliminate loads **early** in the pipeline

3. Ensure **correctness** in today’s processors

- Maintain correctness in presence of **out-of-order load issue**
- Maintain coherence in **multi-threaded & multi-core execution**
Design Overview
Constable: Key Steps

Identify likely-stable loads

Eliminate by tracking modifications
Identify a Likely-Stable Load

- Using a **stability confidence** counter per load instruction

```assembly
add rax, 0x10
mov r8, [rbp+0x8]
sub rax, r8
cmp rsi, rax
add rax, 0x10
mov r8, [rbp+0x8]
sub rax, r8
cmp rsi, rax
ret
mov r8, [rbp+0x8]
sub rax, r8
cmp rsi, rax
jle 0x40230e
```

**Stability Confidence**

- **Same data & address as last dynamic instance**
  - +1

- **Different data or different address**
  - /2
Eliminate a Likely-Stable Load

```assembly
jle 0x40230e
add rax, 0x10
mov r8, [rbp+0x8]
sub rax, r8
cmp rsi, rax
add rax, 0x10
sub rax, r8
mov r8, [rbp+0x8]
sub rax, r8
cmp rsi, rax
add rax, 0x10
mov r8, [rbp+0x8]
```

Stability confidence crosses threshold

- Register Monitor
  - rbp  PC
  - Insert
- Address Monitor
  - 0x4200e0  PC
  - Insert
- Elimination Table
  - PC  0x2ae
  - Insert
  - Eliminate flag
  - Last value

- No reservation station
- No address generation unit
- No load port
- Still takes ROB and load buffer

SAFARI

to handle correct elimination of in-flight loads
Stop Elimination of a Likely-Stable Load

```assembly
jle 0x40230e
add rax, 0x10
mov r8, [rbp+0x8]
sub rax, r8
cmp rsi, rax
pop rbx
add rbp, 0xd8
jle 0x40230e
mov r8, [rbp+0x8]
sub rax, r8
cmp rsi, rax
jle 0x40230e
```

Elimination flag not set.

Gets executed
More in the Paper

• **Ensuring safe and correct elimination** in presence of
  - Out-of-order load issue
  - Multi-threaded & multi-core execution
  - Wrong-path execution

• **Integration of Constable** into the processor pipeline

• **Microarchitecture for breaking data** dependence on the eliminated loads

• **Microarchitecture of Constable’s own structures**
  - Read and write port requirements
More in the Paper

• Ensuring safe and correct elimination in presence of
  Out-of-order load issue

• Integration of Constable into the processor pipeline

• Microarchitecture for breaking data dependence on the
  eliminated loads

• Microarchitecture of Constable's own structures

- Read and write port requirements

Constable: Improving Performance and Power Efficiency by Safely Eliminating Load Instruction Execution

*Rahul Bera*¹  *Adithya Ranganathan*²  *Joydeep Rakshit*²  *Sujit Mahto*²
*Anant V. Nori*²  *Jayesh Gaur*²  *Ataberk Olgun*¹  *Konstantinos Kanellopoulos*¹
*Mohammad Sadrosadati*¹  *Sreenivas Subramoney*²  *Onur Mutlu*¹

¹ETH Zürich  ²Intel Processor Architecture Research Lab

Load instructions often limit instruction-level parallelism (ILP) in modern processors due to data and resource dependences they cause. Prior techniques like Load Value Prediction (LVP) and Memory Renaming (MRN) mitigate load data dependence by predicting the data value of a load instruction. However, they fail to mitigate load resource dependence as the predicted load instruction gets executed nonetheless (even on a correct prediction), which consumes hard-to-scale pipeline resources that otherwise could have been used to execute other load instructions.

Our goal in this work is to improve ILP by mitigating both load data dependence and resource dependence. To this end, we propose a purely-microarchitectural technique called Constable, that safely eliminates the execution of load instructions. Constable dynamically identifies load instructions that have re-stall for multiple cycles, which can limit ILP. On the other hand, a load instruction consumes multiple hard-to-scale hardware resources (e.g., reservation station entry, port to address generation unit, L1-data cache read port) which often causes resource dependence in the pipeline, also limiting ILP.

Prior works propose many latency tolerance techniques to improve ILP by mitigating load data dependence. Load Value Prediction (LVP) [32,42,43,71,98,107,114,139–143,151,153–155, 159,160] and Memory Renaming (MRN) [120,121,147,177,178] are two such widely-studied techniques that mitigate load data dependence via data value speculation. LVP and MRN speculatively execute load-data-dependent instructions using the predicted value of the load instruction, thus improving ILP.

Evaluation
Methodology

• Industry-grade x86-64 simulator modeling aggressive OoO processor
  - 8-wide fetch, 6-wide issue to 3 load ports, 512-entry ROB
  - With memory renaming, zero/constant/move elimination, branch folding
  - Five prefetchers throughout cache hierarchy

• 90 workloads of wide variety
  - All from SPEC CPU 2017
  - Client (SYSMark, DaCapo, ...)
  - Enterprise (SPECjbb, SPECjEnterprise, ...)
  - Server (BigBench, Hadoop, ...)

Mechanisms compared against

• EVES, the state-of-the-art load value predictor [Seznec, CVP’18]
• Early Load Address Resolution [Bekerman+, ISCA’00]
• Register File Prefetching [Shukla+, ISCA’22]

Configurations

• No simultaneous multi-threading (SMT)
• 2-way SMT
Performance Improvement in noSMT

Constable alone provides similar performance as EVES with only $\frac{1}{2}$ of EVES’ storage overhead.

Constable on top of EVES outperforms EVES alone.
Performance Improvement in 2-way SMT

<table>
<thead>
<tr>
<th></th>
<th>Client</th>
<th>Enterprise</th>
<th>FSPEC17</th>
<th>ISPEC17</th>
<th>Server</th>
<th>GEOMEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geomean speedup over the baseline</td>
<td>1.00</td>
<td>1.05</td>
<td>1.10</td>
<td>1.15</td>
<td>EVES</td>
<td>11.3%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Constable</td>
<td></td>
<td></td>
<td>EVES + Constable</td>
<td>8.8%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EVES (the state-of-the-art load value predictor)</td>
<td></td>
<td></td>
<td>Constable</td>
<td>3.6%</td>
</tr>
</tbody>
</table>

SAFARI
Performance Improvement in 2-way SMT

Constable provides higher performance benefits in a 2-way SMT processor
Improvement in Resource Efficiency

**Reduction in L1 Data Cache Accesses**
- 26% average

**Reduction in Reservation Station Allocation**
- 8.8% average
Improvement in Resource Efficiency

Reduction in L1 Data Cache Accesses

- 26% average

Reduction in Reservation Station Allocation

- 8.8% average

Constable significantly improves resource efficiency by eliminating load instruction execution
Reduction in Dynamic Power

Memory Execution Unit Power

- L1-D
- DTLB
- Others

OoO Unit Power

- RS
- RAT
- ROB
- Others

9.1% average reduction
5.1% average reduction

SAFARI
Reduction in Dynamic Power

By eliminating load instruction execution, Constable reduces dynamic power consumption.
Area and Power Overhead of Constable’s Own Structures

- **12.4 KB**
  - Storage overhead per core

- **0.232 mm²**
  - 0.0061% area of Intel Alderlake-S processor

- **Low Energy**
  - Up to 10.8 pJ/read and 16.7 pJ/write
More in the Paper

• **Load elimination coverage** of Constable
  - 23.5% of all dynamic loads are eliminated

• **Per-workload performance** analysis
  - Up to 31.2% over baseline
  - 60/90 workloads outperforms EVES by more than 5%

• **Performance contribution per load category**
  - Stack loads contribute the highest

• **Performance improvement over prior works**
  - 4.7% over Early load address resolution
  - 3.6% over Register file prefetching

• **Performance sensitivity**:  
  - Higher performance in every configuration up to 2X load execution width  
  - Higher performance in every configuration up to 2X pipeline depth
More in the Paper

• Load elimination coverage of Constable - 23.5% of all dynamic loads are eliminated

Constable: Improving Performance and Power Efficiency by Safely Eliminating Load Instruction Execution

*Rahul Bera¹  *Adithya Ranganathan²  Joydeep Rakshit²  Sujit Mahto²
Anant V. Nori²  Jayesh Gaur²  Ataberk Olgun¹  Konstantinos Kanellopoulos¹
Mohammad Sadrosadati¹  Sreenivas Subramoney²  Onur Mutlu¹

¹ETH Zürich  ²Intel Processor Architecture Research Lab

Load instructions often limit instruction-level parallelism (ILP) in modern processors due to data and resource dependences they cause. Prior techniques like Load Value Prediction (LVP) and Memory Renaming (MRN) mitigate load data dependence by predicting the data value of a load instruction. However, they fail to mitigate load resource dependence as the predicted load instruction gets executed nonetheless (even on a correct prediction), which consumes hard-to-scale pipeline resources that otherwise could have been used to execute other load instructions.

Our goal in this work is to improve ILP by mitigating both load data dependence and resource dependence. To this end, we propose a purely-microarchitectural technique called Constable, that safely eliminates the execution of load instructions. Constable dynamically identifies load instructions that have re-stall for multiple cycles, which can limit ILP. On the other hand, a load instruction consumes multiple hard-to-scale hardware resources (e.g., reservation station entry, port to address generation unit, L1-data cache read port) which often causes resource dependence in the pipeline, also limiting ILP.

Prior works propose many latency tolerance techniques to improve ILP by mitigating load data dependence. Load Value Prediction (LVP) [32,42,43,71,98,107,114,139–143,151,153–155, 159,160] and Memory Renaming (MRN) [120,121,147,177,178] are two such widely-studied techniques that mitigate load data dependence via data value speculation. LVP and MRN speculatively execute load-data-dependent instructions using the predicted value of the load instruction, thus improving ILP.

To Summarize...
Our Key Findings

1. A large fraction (34%) of dynamic loads fetch the same data from the same address throughout the entire workload.

2. These global-stable loads cause significant ILP loss due to resource dependence.

3. Eliminating global-stable load execution provides more than 2x the performance benefit of just breaking their load data dependency.
Our Proposal

**Constable**

Identifies and eliminates loads that repeatedly fetch same data from same address

- **High performance benefit**
  - over a strong baseline system
  - without (5.1%) and with SMT (8.8%)

- **Improves resource efficiency**
  - L1-D access reduction by 26%
  - RS allocation reduction by 8.8%

- **Reduces dynamic power**
  - L1-D power by 9.1%
  - RS power by 5.1%

- **Low storage overhead**
  - Only 12.4 KB/core,
  - 0.232 mm² in 14-nm technology
There’s Still Headroom...

Constable successfully eliminates **57%** of all global-stable loads at runtime.

**43%** of global-stable loads do not get eliminated.

We need to understand more software primitives that generate global-stable loads.
Open-Source Tool

Load Inspector

A tool to analyze load instructions in any off-the-shelf x86(-64) program

https://github.com/CMU-SAFARI/Load-Inspector
Open-Source Tool

Load Inspector

A tool to analyze load instructions in any off-the-shelf x86(-64) program

Study global-stable loads

Study the effects of increasing architectural registers using APX extension to x64 ISA

https://github.com/CMU-SAFARI/Load-Inspector
CONSTABLE

Improving Performance and Power Efficiency by Safely Eliminating Load Instruction Execution

arXiv

Load Inspector

SAFARI
SAFARI Research Group
safari.ethz.ch

ETH Zürich

Intel®
BACKUP
<table>
<thead>
<tr>
<th>Motivation &amp; Design</th>
<th>Methodology &amp; Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why do global-stable loads exist?</td>
<td>Simulation parameters</td>
</tr>
<tr>
<td>Effect of increasing registers</td>
<td>Evaluated workloads</td>
</tr>
<tr>
<td>Characterization of global-stable loads</td>
<td>Workload-wise performance</td>
</tr>
<tr>
<td>Resource dependence by global-stable loads</td>
<td>Load-category-wise performance</td>
</tr>
<tr>
<td>Performance headroom</td>
<td>Comparison with prior works</td>
</tr>
<tr>
<td>Constable overview</td>
<td>Elimination coverage</td>
</tr>
<tr>
<td>Architecting elimination table</td>
<td>Coverage of global-stable loads</td>
</tr>
<tr>
<td>Effect of wrong-path update</td>
<td>Reduction in power</td>
</tr>
<tr>
<td>Example of Constable’s operation</td>
<td>Scaling width</td>
</tr>
<tr>
<td>Ensuring coherence</td>
<td>Scaling depth</td>
</tr>
<tr>
<td>Storage overhead</td>
<td>Violating memory ordering</td>
</tr>
<tr>
<td>Area/energy overhead</td>
<td></td>
</tr>
</tbody>
</table>
Why Do Global-Stable Loads Exist?

```cpp
Random* Random::s_rng = 0;

Random* Random::get_Rng(void)
{
    if (s_rng == 0)
    
        s_rng = new Random();
    
    return s_rng;
}
```

Example code from 541.leela_r
Why Do Global-Stable Loads Exist?

Random* Random::s_rng = 0;

Random* Random::get_Rng(void)
{
    if (s_rng == 0)
        s_rng = new Random();
    return s_rng;
}

Example code from 541.leela_r

Disassembly (compiled by GCC* with –O3)

624: mov rax, [rip+0x1f4ac5]
62b: test rax, rax
62e: je 0x638
630: ret
631: nop
638: sub rsp,0x8
63c: mov edi,0xc
641: call 0x460
Why Do Global-Stable Loads Exist?

Random* Random::s_rng = 0;

Random* Random::get_Rng(void)
{
    if (s_rng == 0)
    {
        s_rng = new Random();
        return s_rng;
    }
}

Example code from 541.leela_r

Disassembly (compiled by GCC* with –O3)

624: mov rax, [rip+0x1f4ac5]
62b: test rax, rax
62e: je 0x638
630: ret
631: nop
638: sub rsp, 0x8
63c: mov edi, 0xc
641: call 0x460

*GNU GCC 13.2

Gets initialized only once and never changes

Global-stable load
Why Do Global-Stable Loads Exist?

```c
static inline bool rc_shift_low(lzma_range_encoder *rc, uint8_t *out,
                       size_t *out_pos, size_t out_size)
{
    ...
    do
    {
        if (*out_pos == 0x0f)
            return true;
        out[*out_pos] = rc->cache + (uint8_t)(rc->low >> 32);
        ++*out_pos;
        rc->cache = 0xFF;
    } while (--rc->cache_size != 0);
    ...
}
```

Global-stable loads accessing function arguments. Function is repeatedly called by the same caller with the same arguments.
Why Do Global-Stable Loads Exist?

Global-stable loads exist for many reasons:

• Accessing **global variables**
• Accessing **local variables** of an **inline function**
• **Limited architectural registers**
• ...
Effects of Increasing Architectural Registers

Fraction of all dynamic loads that are global-stable are nearly same without or with APX
Effects of Increasing Architectural Registers

Fraction of global-stable loads (i.e., Constable’s opportunities) are much higher than reduction in loads by APX

Compiled with Clang 18.1.3 with and without -mapxf
The profile of global-stable loads stays largely unchanged after doubling registers using APX.
Characterization of Global-Stable Loads (I)

Fraction of global-stable loads

- PC-relative
- Stack-relative
- Reg-relative

Client
Enterprise
FSPEC17
ISPEC17
Server
AVG
Characterization of Global-Stable Loads (II)
Characterization of Global-Stable Loads (III)

Fraction of global-stable loads that use respective addressing mode

- PC-relative
- Stack-relative
- Reg-relative

Categories:
- [0–50)
- [50, 100)
- [100, 250)
- 250+

SAFARI
Global-stable loads cause significant resource dependence
Performance Headroom Analysis

![Graph showing performance headroom analysis for different benchmarks and scenarios. The x-axis represents different benchmarks: Client, Enterprise, FSPEC17, ISPEC17, Server, and GEOMEAN. The y-axis represents geometric speedup over the baseline. The graph compares Ideal Stable LVP, Ideal Stable LVP + data fetch elimination, 2X load execution width, and Ideal Constable.]
**Constable Overview**

(a) Newly added structures in the pipeline

1. New load: Look up with load PC
2. If can_eliminate flag is set
   - Break load data dependence using last-fetched value and eliminate load execution
   - 1. Mark the load “likely-stable” if confidence is above threshold
   - 2. Execute the load
3. If can_eliminate flag is not set
   - Reset can_eliminate flag in SLD entry
4. Insert load PC into RMT
5. Insert load PC into AMT
6. Set can_eliminate flag
7. Register write
8. Reset can_eliminate flag in SLD entry
9. Physical address of a store is generated
10. Snoop request arrives at the core

(b) Key operations in Constable

**SAFARI**
Architecting SLD

Average number of SLD updates per cycle

- Client
- Enterprise
- FSPEC17
- ISPEC17
- Server
Effect of Wrong-Path Update

Change in performance

Client  Enterprise  FSPEC17  ISPEC17  Server

SAFARI
An Example of Constable’s Operation

<table>
<thead>
<tr>
<th>PC</th>
<th>conf</th>
<th>can_elim</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCx</td>
<td>30</td>
<td>0</td>
</tr>
</tbody>
</table>

LD₁ from PCₓ gets decoded

Look up SLD

can_eliminate is not set but conf (30) matches the threshold (30)

Mark LD₁ as likely-stable and execute
Ensuring Coherence

- Constable relies on **snoop requests** to observe modifications to a memory address by other cores.

```
add rax, 0x10
mov r8, [rbp+0x8]  
sub rax, r8
cmp rsi, rax
pop rbx
add rbp, 0xd8
```

```
add rax, 0x10
mov r8, [rbp+0x8]  
sub rax, r8
cmp rsi, rax
jle 0x40230e
```

**Register Monitor**

- **rbp**
- **PC_x**

**Address Monitor**

- **0x4200e0**
- **PC_x**

**Elimination Table**

- **PC_x**
- **0x2ae**
- Flag

*Elimination flag not set. Gets executed*
Ensuring Coherence

• Constable relies on **snoop requests** to observe modifications to a memory address by other cores

• Evicting a cacheline from core-private cache **resets the core-valid bit** in directory
  - What if a **clean eviction**?
    • Unnecessary elimination opportunity loss

• Constable **pins the CV-bit of an eliminated load’s cacheline**
  - Even if the cacheline gets evicted from core-private cache, snoop request gets delivered
# Constable’s Storage Overhead

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
</table>
| SLD       | # entries: 512 (32 sets × 16 ways)  
Entry size: tag (24b) + addr (32b) + val (64b) + confidence level (5b) + can_eliminate flag (1b) | 7.9 KB |
| RMT       | 16 load PCs for each stack registers (RSP and RBP)  
8 load PCs for each remaining 14 architectural registers in x86-64 | 0.4 KB |
| AMT       | # entries: 256 (32 sets × 8 ways)  
Entry size: physical address tag (32b) + # hashed load PCs (4 × 24b) | 4.0 KB |
| Total     |             | 12.4 KB |
# Area and Power Overhead of Constable’s Structures

<table>
<thead>
<tr>
<th>Component</th>
<th>Read access energy (pJ)</th>
<th>Write access energy (pJ)</th>
<th>Leakage power (mW)</th>
<th>Area (mm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLD (7.9KB, 3R/2W ports)</td>
<td>10.76</td>
<td>16.70</td>
<td>1.02</td>
<td>0.211</td>
</tr>
<tr>
<td>RMT (0.4KB, 2R/6W ports)</td>
<td>0.15</td>
<td>0.20</td>
<td>0.31</td>
<td>0.004</td>
</tr>
<tr>
<td>AMT (4.0KB, 1R/1W ports)</td>
<td>1.58</td>
<td>4.22</td>
<td>0.74</td>
<td>0.017</td>
</tr>
</tbody>
</table>
## Simulated System Parameters

<table>
<thead>
<tr>
<th>Basic</th>
<th>x86-64 core clocked at 3.2 GHz with 2-way SMT support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch &amp; Decode</td>
<td>8-wide fetch, TAGE/ITTAGE branch predictors [156], 20-cycle mis-prediction penalty, 32KB 8-way L1-I cache, 4K-entry 8-way micro-op cache, 6-wide decode, 144-entry IDQ, loop-stream detector [41]</td>
</tr>
<tr>
<td>Rename</td>
<td>6-wide, 288 integer, 220 512-bit and 320 256-bit physical registers, Memory Renaming [177], zero elimination [68], move elimination [64,68], constant folding [64,68], branch folding [60]</td>
</tr>
<tr>
<td>Allocate</td>
<td>512-entry ROB, 240-entry LB, 112-entry SB, 248-entry RS</td>
</tr>
<tr>
<td>Issue &amp; Retire</td>
<td>6-wide issue to 12 execution ports; 5, 3, 2, and 2 ports for ALU, load, store-address, and store-data execution. Port 0, 1, and 5 are used for vector instructions. Aggressive out-of-order load scheduling with memory dependence prediction [51,120], 6-wide retire</td>
</tr>
<tr>
<td>Caches</td>
<td>L1-D: 48KB, 12-way, 5-cycle latency, LRU, PC-based stride prefetcher [69]; L2: 2MB, 16-way, 12-cycle round-trip latency, LRU, stride + streamer [47] + SPP [101]; LLC: 3MB, 12-way, 50-cycle data round trip latency [15,36], dead-block-aware replacement policy [100], streamer, MESIF [118] protocol</td>
</tr>
<tr>
<td>Memory</td>
<td>4 channels, 2 ranks/channel, 8 banks/rank, 2KB row-buffer/rank, 64b bus/channel, DDR4, tCAS=22ns, tRCD=22ns, tRP=22ns, tRAS=56ns</td>
</tr>
</tbody>
</table>
| Optional | • EVES [155]: exact design of the CVP-1 32KB storage track winner  
• ELAR [34]: with additional adder and RSP register in decode stage  
• RFP [164]: 2K-entry prefetch table, 64-entry page address table, 128-entry RFP-inflight table. Total size: **12.5KB**  
• Constable (this work). Total size: **12.4KB** |
## Evaluated Workloads

<table>
<thead>
<tr>
<th>Suite</th>
<th>#Workloads</th>
<th>#Traces</th>
<th>Example Workloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>16</td>
<td>22</td>
<td>DaCapo [3], SYSmark [20], TabletMark [21], JetStream2 [12]</td>
</tr>
<tr>
<td>Enterprise</td>
<td>9</td>
<td>14</td>
<td>SPECjEnterprise [19], SPECjbb [18], LAMMPS [13]</td>
</tr>
<tr>
<td>FSPEC17</td>
<td>13</td>
<td>29</td>
<td>All from SPECrate FP 2017 [17]</td>
</tr>
<tr>
<td>ISPEC17</td>
<td>10</td>
<td>11</td>
<td>All from SPECrate Integer 2017 [17]</td>
</tr>
<tr>
<td>Server</td>
<td>10</td>
<td>14</td>
<td>Hadoop [1], Linpack [9], Snort [16], BigBench [2]</td>
</tr>
</tbody>
</table>
Workload-Wise Performance

Conclusions:
- EVES provides higher performance than Constable.
- EVES+Constable provides the highest performance across different workloads.

Key Observations:
- Workload 527.cam4_r shows a notable improvement with Constable.
- Workload 549.fotonik3d_r and hadoop_kmeans exhibit mixed performance across different configurations.
- Workload 538.imagick_r shows a significant increase in speedup with EVES+Constable.

Further Analysis:
- jetstream2-richards_wasm shows a significant boost with EVES+Constable compared to Constable alone.
- SYSMark-chrome highlights the effectiveness of EVES+Constable in high-performance scenarios.
- jetstream2-richards demonstrates the superior speedup of EVES+Constable over Constable.

Overall, the results indicate that EVES and Constable work well in isolation but EVES+Constable offers the best performance across a wide range of workloads.
Load Category-Wise Performance

- PC-relative loads
- Stack-relative loads
- Register-relative loads
- All loads

Geomean speedup over the baseline:
- Client: 1.011
- Enterprise: 1.026
- FSPEC17: 1.018
- ISPEC17: 1.051
- Server: 1.051
- GEOMEAN: 1.051
Performance Comparison with Prior Works

<table>
<thead>
<tr>
<th></th>
<th>ELAR</th>
<th>RFP</th>
<th>Constable</th>
<th>ELAR+Constable</th>
<th>RFP+Constable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>1.007</td>
<td>1.045</td>
<td>1.051</td>
<td>1.054</td>
<td>1.081</td>
</tr>
<tr>
<td>Enterprise</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSPEC17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISPEC17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEOMEAN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Geomean speedup over the baseline
Elimination Coverage

- EVES
- Constable
- EVES+Constable
- EVES+Ideal Constable

Fraction of all loads

- Client
- Enterprise
- FSPEC17
- ISPEC17
- Server
- AVG

Percentage:
- Client: 27.3%
- Enterprise: 35.5%
- FSPEC17: 41.6%

SAFFARI
Coverage of Global-Stable Loads

- Global-stable and eliminated
- Global-stable but not eliminated
- Not global-stable but eliminated

Fraction of stable load in each load category

- PC-rel
- Stack-rel
- Reg-rel
- All loads

Client
Enterprise
FSPEC17
ISPEC17
Server
AVG
Reduction in Dynamic Power
Performance Sensitivity to Load Execution Width Scaling

- Baseline system
- Constable

Geomean speedup over baseline processor configuration

Load execution width

1 width savings

+5% perf
Performance Sensitivity to Pipeline Depth Scaling

![Graph showing performance sensitivity to pipeline depth scaling. The x-axis represents the pipeline depth scaling factor, while the y-axis shows the geometric speedup over the baseline processor configuration. Two lines are plotted: one for the baseline system and another for Constable. As the pipeline depth scaling factor increases, the geometric speedup also increases.]
Eliminated Loads that Violates Memory Ordering

Only \textbf{0.09\%} of all eliminated loads violate memory ordering
Eliminated loads that violate memory ordering

Increase in number of allocated instructions only by 0.3%
## Executive Summary

1. A significant fraction of loads always fetch **the same data from same address throughout the entire workload**

2. These loads cause significant **resource dependence** even when using value prediction and memory renaming

3. Mitigating their resource dependence has **significant performance headroom**

### Key Findings

**Constable**

**Key Idea:** To **mitigate both load data and load resource dependence** by safely eliminating the entire execution of a load

**Key Result:** **Simultaneously improves performance and power efficiency** of a strong baseline OoO processor