Latr: Lazy Translation Coherence

Mohan Kumar*, Steffen Maass*, Sanidhya Kashyap, Ján Veselý‡, Zi Yan‡, Taesoo Kim, Abhishek Bhattacharjee‡, Tushar Krishna

Georgia Institute of Technology ‡Rutgers University

* Co-First Authors

March 28, 2018
Supermicro Debuts 8-Socket Server for Intel Xeon Processors

By Sue Smith / NewsFactor Network

Supermicro just announced the latest addition to its line of SuperServer systems, designed for data centers and the SuperServer 7089P-TR4T is a server for Intel Xeon scalable architecture.
Large NUMA machines

Terabytes of memory
Large NUMA machines

Terabytes of memory

Microsecond latency

Microsecond-scale I/O means tension between performance and productivity that will need new latency-mitigating ideas, including in hardware.

BY LUIZ BARROSO, MIKE MARTY, DAVID PATTERSON, AND PARTHASARATHY RANGANATHAN
Problem of Microsecond Latency in System Services

TLB Coherence is Contributor in Important Subset

Large NUMA machines

Terabytes of memory

Microsecond latency
Impact of TLB coherence on applications

- Multi-core MapReduce application
  - Prior research: **10x increase in shootdown time** with increasing core counts
- Web servers (e.g., Apache)
  - Prior research and our findings: ≈**35% of time spent in TLB shootdown**
- Die-stacked Memory
  - Swapping between on-chip and off-chip memory
- Disaggregated Memory
  - Swapping between local and remote memory
Impact of TLB coherence on applications

- Multi-core MapReduce application
  - Prior research: \textbf{10x increase in shootdown time} with increasing core counts
- Web servers (e.g., Apache)
  - Prior research and our findings: $\approx 35\%$ of time spent in TLB shootdown
- Die-stacked Memory
  - Swapping between on-chip and off-chip memory
- Disaggregated Memory
  - Swapping between local and remote memory

$\Rightarrow$ Can we mitigate this costly TLB shootdown?
Table of contents

1 TLB Shootdown Background

2 LATR: Asynchronous TLB Shootdowns

3 Evaluation

4 Conclusion
1 TLB Shootdown Background

2 LATR: Asynchronous TLB Shootdowns

3 Evaluation

4 Conclusion
Translation lookaside buffer: Introduction

- Cache for virtual → physical mapping, per-core structures
- Accessed on every load/store
- Unlike data caches (L3, etc.), coherence managed by OS
- TLB coherence significantly impacts application performance
TLB coherence: Background

- **Hardware-based Approaches**
  - Providing cache coherence to TLBs
  - ISA-level instruction support (ARM)
  - Microcode-based approaches

- **Software-based Approaches**
  - Current commodity OS design: Use Inter-Processor Interrupts (IPI)
  - Optimization: Reduce number of shootdowns, better tracking
  - Multikernel design: Use Message-Passing
TLB coherence: Background

- **Hardware-based Approaches**
  - Providing cache coherence to TLBs
    \[ \Rightarrow \text{More Hardware Complexity} \]

- **Software-based Approaches**
  \[ \Rightarrow \text{TLB shootdowns still significant} \]
  - Optimization: Reduce number of shootdowns, better tracking
  - Multikernel design: Use Message-Passing
munmap() on core 1, application running on cores 1, 2, and 5:
**TLB shootdown internals in Linux**

- `munmap()` on core 1, application running on cores 1, 2, and 5:

```
<table>
<thead>
<tr>
<th>Core1</th>
<th>Core2</th>
<th>Core3</th>
<th>Core4</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>TLB</td>
<td>TLB</td>
<td>TLB</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Core5</th>
<th>Core6</th>
<th>Core7</th>
<th>Core8</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>TLB</td>
<td>TLB</td>
<td>TLB</td>
</tr>
</tbody>
</table>
```

Timeline:

```
<table>
<thead>
<tr>
<th>App1</th>
<th>App2</th>
<th>Idle</th>
<th>Idle</th>
<th>App5</th>
<th>Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>OS</td>
<td></td>
<td></td>
<td>OS</td>
<td></td>
</tr>
</tbody>
</table>
```

#### Application

**Application**

**Operating System**

**TLB**

**O.S.**

**OS**
TLB shootdown internals in Linux

- Context switch on core 1, local TLB shootdown:

<table>
<thead>
<tr>
<th>Core1</th>
<th>Core2</th>
<th>Core3</th>
<th>Core4</th>
<th>Core5</th>
<th>Core6</th>
<th>Core7</th>
<th>Core8</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>TLB</td>
<td>TLB</td>
<td>TLB</td>
<td>TLB</td>
<td>TLB</td>
<td>TLB</td>
<td>TLB</td>
</tr>
</tbody>
</table>

Timeline:

1. `munmap()`
2. Local Shootdown

Application

<table>
<thead>
<tr>
<th>App1</th>
<th>App2</th>
<th>Idle</th>
<th>Idle</th>
<th>App5</th>
<th>Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>OS</td>
<td>...</td>
<td></td>
<td>OS</td>
<td></td>
</tr>
</tbody>
</table>

Operating System
TLB shootdown internals in Linux

- Notify cores 2 and 5 via IPI, application blocked on core 1:

Timeline:
1. munmap()
2. Local Shootdown
3. Send IPIs

Spin-wait

2.2µs
**TLB shootdown internals in Linux**

- Execute context switch and TLB shootdown on cores 2 and 5:

**Timeline:**
- 1. `munmap()`
- 2. Local Shootdown
- 3. Send IPIs
- 4. Remote Shootdown

**Duration:** 2.2\(\mu\)s
Cores 2 and 5 respond ACK via shared memory:

Timeline:

Timeline: ① munmap() ② Local Shootdown ③ Send IPIs ④ Remote Shootdown ⑤ IPI ACK

2.2µs
Control is returned on all cores, TLB shootdown completed:

Timeline:

1. `munmap()`
2. Local Shootdown
3. Send IPIs
4. Remote Shootdown
5. IPI ACK
6. `munmap()` complete

Savings potential for asynchronous approach with LATR
Observation

- **Synchronous TLB shootdown is expensive:**
  - Up to 6 $\mu$s delay with two sockets

- **Processing IPIs is expensive:**
  - Interrupt handler on remote core
  - Long wait time on initiating core

- **IPI send-and-wait delay:**
  - Unicast delivery of the IPIs (one at a time)
TLB shootdown: A necessary evil

- Cost of a simple memory unmap operation (**munmap()**):
  - 1 page on 16 cores with 2 sockets: **up to 8 µs**
  - ≈ 70% from TLB shootdown alone

- More expensive with more sockets:

![Graph showing latency for munmap() operation with varying number of cores and sockets]
Cost of a simple memory unmap operation (`munmap()`):
- 1 page on 16 cores with 2 sockets: **up to 8 µs**
- ≈ 70% from TLB shootdown alone

More expensive with more sockets:
TLB shootdown: A necessary evil

- Cost of a simple memory unmap operation (munmap()):
  - 1 page on 16 cores with 2 sockets: **up to 8 µs**
  - \( \approx 70\% \) from TLB shootdown alone

- More expensive with more sockets:
Table of contents

1 TLB Shootdown Background

2 LATR: Asynchronous TLB Shootdowns

3 Evaluation

4 Conclusion
In this talk: **LATR**

- **LATR**: Lazy Translation Coherence
- **Perform asynchronous TLB shootdown**
  - Remove remote shootdown from the critical path
  - Take advantage of change in ABI without affecting applications’ correctness
- **Use shared memory instead of IPI**
  - Eliminate send-and-wait delay of IPIs
- **Scope**: 
  - *free* operations (in this talk)
  - *migration* operations (see our paper)
In this talk: Latr

- **Latr**: Lazy Translation Coherence
- **Perform asynchronous TLB shootdown**
  - Remove remote shootdown from the critical path
  - Take advantage of change in ABI without affecting applications’ correctness
- **Use shared memory instead of IPI**
  - Eliminate send-and-wait delay of IPIs
- **Scope:**
  - *free* operations (in this talk)
  - *migration* operations (see our paper)

⇒ But: How to perform asynchronous shootdown?
**LATR States**

- Store virtual addresses to be flushed
- Remote cores shootdown local TLB during
  - OS context switch
  - OS scheduler tick (**upper bound**: 1ms in Linux)

![Diagram showing LATR States and Cache Coherency](image)
**LATR: Example**

- `munmap()` initiated on core 1:

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>App₁</td>
</tr>
<tr>
<td>OS</td>
</tr>
</tbody>
</table>

  **Operating System**

  | Core₁ | Core₂ | Core₃ | Core₄ |
  | LATR States | LATR States | LATR States | LATR States |
  |        |        |        |        |

  | Core₅ | Core₆ | Core₇ | Core₈ |
  | LATR States | LATR States | LATR States | LATR States |
  |        |        |        |        |

  **Timeline:**

  1

  (Diagram of application and operating system states across different cores and timeline indicated by numbered points.)
**LATR: Example**

- **munmap()** initiated on core 1:

  ![Diagram](image-url)

  - Timeline:
  - Application
  - Operating System
  - LATR States

  ![Timeline](image-url)
Set up **LATR** state (for cores 2 and 5), local shootdown:

Timeline:

1. **munmap()**
2. **Local Shootdown**
3. **Create LATR State**

---

**Core1, LATR State 1:**

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>mm</th>
<th>flags</th>
<th>Core list</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x0F</td>
<td>0x1234</td>
<td>0x1</td>
<td>{2, 5}</td>
<td>True</td>
</tr>
</tbody>
</table>
**LATR: Example**

- Return control on core 1. Time taken: $2.3\mu s$, 70% reduction:

  
  ![Diagram](image)

  Core1, LATR State1:

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>mm</th>
<th>flags</th>
<th>Core list</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x0F</td>
<td>0x1234</td>
<td>0x1</td>
<td>{2, 5}</td>
<td>True</td>
</tr>
</tbody>
</table>

  Timeline:

  1. **munmap()**
  2. Local Shootdown
  3. Create LATR State
  4. **munmap() complete**

  ![Timeline](image)
Scheduler tick on core 2, local shootdown, reset state:


- Core list active: 0x1234 {5}

- Core1, LATR State1:

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>mm</th>
<th>flags</th>
<th>Core list</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x0F</td>
<td>0x1234</td>
<td>0x1</td>
<td>{5}</td>
<td>True</td>
</tr>
</tbody>
</table>
Scheduler tick on core 5, local shootdown, reset state:

<table>
<thead>
<tr>
<th>App_1</th>
<th>App_2</th>
<th>Idle</th>
<th>Idle</th>
<th>App_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>OS</td>
<td>...</td>
<td></td>
<td>OS</td>
</tr>
</tbody>
</table>

Core_1, LATR State_1:

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>mm</th>
<th>flags</th>
<th>Core list</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x0F</td>
<td>0x1234</td>
<td>0x1</td>
<td>{}</td>
<td>False</td>
</tr>
</tbody>
</table>

1. **munmap()**
2. Local Shootdown
3. Create LATR State
4. **munmap() complete**
5. Shootdown Core_2
6. Shootdown Core_5
**LATR: Example**

- Shootdown complete, LATR entry can be reused:

<table>
<thead>
<tr>
<th>App1</th>
<th>App2</th>
<th>Idle</th>
<th>Idle</th>
<th>App5</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>OS</td>
<td>⋮</td>
<td></td>
<td>OS</td>
</tr>
</tbody>
</table>

- Core1, LATR State1:

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>mm</th>
<th>flags</th>
<th>Core list</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x0F</td>
<td>0x1234</td>
<td>0x1</td>
<td>{}</td>
<td>False</td>
</tr>
</tbody>
</table>
Same physical memory or virtual memory is reused
  Leads to memory corruption
⇒ Avoid same physical/virtual page reuse
  Upper bound for TLB shootdown with LATR is 1ms
  OS physical/virtual memory reclamation delayed by two scheduler ticks (2ms)
  Memory overhead is bounded by 21 MB
Lazy TLB shootdown: Incorrect accesses

- Memory accesses before \texttt{LATR} shootdown:
  - Consequence of incorrect application: Use After Free
  - Before \texttt{LATR} shootdown, access (reads and writes) allowed
  - Exists in the current OS implementation
  - After \texttt{LATR} shootdown, access results in segmentation fault
Scope of **LATR**

- ABI change for *free* operations
- Support for operations limited to few, frequently used operations:

<table>
<thead>
<tr>
<th>Classification</th>
<th>Operations</th>
<th>Lazy operation possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free</td>
<td><code>munmap()</code>: unmap address range</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td><code>madvise()</code>: free memory range</td>
<td>✓</td>
</tr>
<tr>
<td>Migration</td>
<td>AutoNUMA page migration (⇒ See paper)</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Page swap: swap page to disk</td>
<td>✓</td>
</tr>
<tr>
<td>Permission</td>
<td><code>mprotect()</code>: change page permission</td>
<td>-</td>
</tr>
<tr>
<td>Ownership</td>
<td>CoW: Copy on Write</td>
<td>-</td>
</tr>
<tr>
<td>Remap</td>
<td><code>mremap()</code>: change physical address</td>
<td>-</td>
</tr>
</tbody>
</table>
Table of contents

1 TLB Shootdown Background

2 LATR: Asynchronous TLB Shootdowns

3 Evaluation

4 Conclusion
Evaluation: Questions

- **LATR** prototype developed for Linux 4.10
- Evaluation questions
  - What are **LATR**’s benefits with microbenchmarks?
  - What are **LATR**’s benefits with real-world applications exhibiting many TLB shootdowns?
  - What is the cost for **LATR**?
Microbenchmark on eight sockets

- Linux and LATR calling `munmap()` with one page on 120 cores:

  ![Graph showing the cost of `munmap()` and TLB Shootdown latency](image)

  ✓ Up to 66.7% reduction for `munmap()`
Serving files with Apache

- Linux, ABIS [ATC17], and LATR on 2 sockets:

\[ \Rightarrow \text{Up to 59.9}\%\ \text{more} \ \frac{\text{requests}}{\text{second}} \ \text{than Linux, 37.9}\%\ \text{higher than ABIS.} \]
Cost of **Latr**

- Memory overhead is bounded by 21 MB
- Performance overheads for applications with few TLB shootdowns:

$$\Rightarrow \text{Latr} \text{ shows small performance overheads of up to 1.7\% due to added operations during scheduling.}$$
Future work

Further applications of LATR in:
- Disaggregated data centers
- Heterogeneous memory
- Applicability to PCID/ASID-based approaches
- Impact on new features such as KPTI, ...?
The synchronous TLB shootdown is expensive

We propose a software-based asynchronous shootdown mechanism

Significant improvement in application performance with LATR

- 70% reduction for munmap(), for 16-core and 120-core machines
- Improves Apache’s throughput by 60%

Asynchronous mechanism applicable to other services:

- AutoNUMA (see our paper)
The **synchronous** TLB shootdown is expensive

- We propose a software-based **asynchronous** shootdown mechanism
- Significant improvement in application performance with **LATR**
  - 70% reduction for `munmap()`, for 16-core and 120-core machines
  - Improves Apache’s throughput by 60%
- Asynchronous mechanism applicable to other services:
  - AutoNUMA (see our paper)

**Thanks!**