Asymmetric Memory Fences: Optimizing Performance & Programmability

Yuelu Duan, Nima Honarmand, Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

ASPLOS, March 2015
Fence: a (Slow) Primitive for Parallelism

- Instruction inserted by programmers or compilers
- Prevents the compiler and HW from reordering memory accesses

\[
\begin{align*}
\text{rd } x \quad &\quad \text{Until these are finished} \\
\text{wr } y \quad &\quad \cdot \text{ reads retired} \\
\text{fence} \quad &\quad \cdot \text{ writes retired + drained from write buffer} \\
\text{rd } z \quad &\quad \text{Cannot be observed by another processor}
\end{align*}
\]

Expensive: cost of a fence in Xeon-based desktop is 20—200 cycles

There are HW proposals to eliminate fence stall: WeeFence [ISCA13]… … but they need complex HW
Contributions

• **Asymmetric Fence Groups**: eliminate fence stall with simple HW
  – Weak Fence (wF): aggressive access reordering
  – Strong Fence (sF): conventional
• Best fence performance-cost tradeoff yet
• Taxonomy of Asymmetric Fence Groups under TSO
Past Work: *WeeFence*

- Aggressive access reorder to eliminate pipeline stall
- Post-fence reads *retire* before the pre-fence writes have drained
  - “Skip” the fence

Substantial gains when write misses pile-up before the fence
But… Reordering Can Cause Incorrect Execution

\[ x = y = 0 \]

With fences: \( t0=1 \) or \( t1=1 \) or both=1

\[
\begin{align*}
\text{PA} & \quad \text{PB} \\
A0: x &= 1 \\
A1: t0 &= y \\
B0: y &= 1 \\
B1: t1 &= x
\end{align*}
\]

Sequential Consistency (SC) Violation

\[
\begin{align*}
\text{PA} & \quad \text{PB} \\
\text{wr } x & \quad \text{wr } y \\
\text{fence} & \quad \text{fence} \\
\text{rd } y & \quad \text{rd } x
\end{align*}
\]

Solution: WeeFence stall rds/wrs if reordering may cause a cycle
How WeeFence Works

PS: Pending Set → wr x
BS: Bypass Set → rd y

Fence group

PA

PB

wr y
rd x
How WeeFence Works

WeeFence works in shared memory and requires no hardware support. It uses a global reorder table (GRT) that tracks the global order of memory accesses. When a processor (PA) writes to a shared memory location, the WeeFence system checks if the location is in the pending set (PS) or bypass set (BS). If it is in the PS, the write is delayed until it is propagated to the GRT. If it is in the BS, the write is bypassed. The GRT is updated to reflect the global order of memory accesses, which allows the system to prevent data races and ensure memory consistency.
WeeFence Hardware is **Suboptimal**

- Requires GRT **global hardware** and additional messages
- GRT is **hard to distribute** like the directory
  - Creates coherence protocol races
  - If **PS** maps to multiple directory modules → turns into conventional fence

**Our goal: High performance and simpler hardware**
Weak Fence (wF)

- Reordering capabilities of WeeFence + no global state

If all the fences in a Fence Group are wF → Deadlock
Insight: No Deadlock If At Least One Conventional Fence

- **Strong Fence (sF):** Conventional fence
- If the group has at least one sF → no deadlock or SC violation
Asymmetric Fence Group

- Contains one or more wF and one or more sF
- Much simpler HW than WeeFence: no global state
- Similar performance (or higher) than WeeFence:
  - Insight: In a fence group, some threads more critical than others
  - Use wF for the critical threads and sF for others

Best fence performance-cost trade-off yet
Use of Asymmetric Fences: Work Stealing

\[
\begin{align*}
\text{take()} & \quad \text{steal()}
\hline
\text{Tail} = & \quad \text{Head} = \\
\text{wF} & \quad \text{sF}
\end{align*}
\]

- Cilk, TBB and other runtime schedulers
- steal() < 5%
Use of Asymmetric Fences: Software Transactional Memory (STM)

read(M)  
\[
\begin{align*}
\text{Lock}(M) &. \text{readers} = wF \\
&= \text{Lock}(M) . \text{writer}
\end{align*}
\]

write(M)  
\[
\begin{align*}
\text{Lock}(M) &. \text{writer} = sF \\
&= \text{Lock}(M) . \text{readers}
\end{align*}
\]

- Read and Write Barriers in STM:
  - Perform the requested rd/wr
  - Update STM metadata to ensure transaction serialization
- Reads are 3.5x more frequent than writes
**Taxonomy of Asymmetric Fence Groups under TSO**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>wF</th>
<th>sF</th>
<th>sF</th>
</tr>
</thead>
<tbody>
<tr>
<td>S+</td>
<td>Fence group only contains sFs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WS+</td>
<td>Asymmetric group with at most one wF</td>
<td>wF</td>
<td>sF</td>
<td>sF</td>
</tr>
<tr>
<td>SW+</td>
<td>Any Asymmetric group</td>
<td>wF</td>
<td>sF</td>
<td>wF</td>
</tr>
<tr>
<td>W+</td>
<td>Group with only wFs</td>
<td>wF</td>
<td>wF</td>
<td>wF</td>
</tr>
</tbody>
</table>

They trade off hardware cost for performance
WS+: Asymmetric Groups with at Most one wF

- Pre-wF accesses never bounce-off from another processor’s BS
- Addresses and checks always at line level (as conventional protocol)
- False sharing may create bouncing

Write proceeds (no SC violation possible)
Other processor kept as sharer so that it sees future coherence activity
Supported with the Order bit (see paper)
W+: Groups with only wFs

- Can deadlock under unfavorable timing
  - True cycle or due to false sharing
- Insight: Under TSO only, recovery is not too costly
  - Completed accesses are only reads
  - No speculative writes
- When wF reaches ROB head $\rightarrow$ checkpoint and proceed
- If HW detects bouncing and being bounced $\rightarrow$ timeout
  - Rollback
  - Wait for write buffer to drain (no deadlock again)
Evaluation

• Simulations of 8-core multicore
• Workloads:
  – 10 Cilk apps using the THE work-stealing algorithm
  – 10 Software Transactional Memory (STM) kernels
  – 6 STAMP apps that use STM
• Goal: Speed-up execution
• Compare execution time of WS+ and W+ to:
  – S+: conventional fences
  – WeeFence
Per-Transaction Execution Time of STM Kernels

- Wee has only small fence time reductions → turns many fences into sFs
- WS+ and W+:
  - Eliminate avg. 1/2 and 2/3 of the fence stall time
  - Avg. transaction takes 24% and 35% fewer cycles
- Choice WS+ vs W+: Order bit for false sharing vs hardware timeouts
Conclusions

• Asymmetric Fences for performance and simple HW:
  – Weak Fence (wF)
  – Strong Fence (sF)
• Best fence performance-cost trade-off yet
  – Simpler HW than WeeFence: no global hardware
  – Higher perf. than WeeFence: 13% (WS+) and 21% (W+) avg.
• Taxonomy of Asymmetric fence groups under TSO
• Outlined uses
• Future Work: programmability issues
Asymmetric Memory Fences: Optimizing Performance & Programmability

Yuelu Duan, Nima Honarmand, Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

ASPLOS, March 2014
Use of Asymmetric Fences: Bakery Algorithm

Fence groups of many different sizes

E[ownpid] = …
fence
for all pid
... = E[pid]

PA
E[pid4] = …
wF
... = E[pid1]

PB
E[pid1] = …
sF
... = E[pid3]

PC
E[pid3] = …
sF
... = E[pid4]

Give priority to PA
Execution Time of Cilk Apps

- **WS+ and W+:**
  - Eliminate most of the fence stall time
  - Similar impact as Wee, which is more expensive

- Overall execution time reduced by avg. 9%
Also in the Paper

- Description of taxonomy in detail
- Hardware implementation issues
  - Line displacements, RC…
- Outline interesting programming issues
- Evaluation of STAMP benchmarks
- Scalability to 32 threads
How WeeFence Works

PS: Pending Set → wr x

BS: Bypass Set → rd y

Weefence1

execute

Wfence1

(2)

(1)

PS

x

Wfence2

rd x

Weefence2

wr x

wr y

Duan, Honarmand, Torrellas
Asymmetric Memory Fences
How WFence Works

PS: Pending Set → wr x
BS: Bypass Set → rd y

Wfence1
Wfence2

(1) PS
(2) execute
(3) PS
(4) Table
(5) local check stall

x
y

PA
PB

wr y
rd x

wr x
rd y

PA
PB

Duan, Honarmand, Torrellas
Asymmetric Memory Fences
How WFence Works (II)

PS: Pending Set  →  wr x  
Wfence1

BS: Bypass Set  →  rd y

execute

(2)

(3)

BS

y

Table

No fence present in TSO

write

write

write

write

write

write

write

write

write
How WFence Works (II)

PS: Pending Set → \text{wr x} \\
Wfence1

BS: Bypass Set → \text{rd y}

(2) execute

(3) BS → y

(1) PS

Table

squash or bounce

PA

PB

→ No fence present in TSO

\text{wr y}

\text{wr x}

\text{wr x}

\text{wr y}

\text{coherence}
Summary: How WFence Works

PS: Pending Set
BS: Bypass Set

(1) PS
(2) Table
(3) Wfence1
(4) BS
(5) execute
(6) squash or bounce
Summary: How WFence Works

**Global Reorder Table (GRT)** in shared memory (signatures)

- **PS: Pending Set**
- **BS: Bypass Set**

**Register in the processor (signature)**

**List of addresses in the cache**

1. Register in the processor
2. Execute
3. Check
4. PS Table
5. Squash or bounce
6. List of addresses in the cache