Toward Extreme-Scale Manycore Architectures

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

USC
October 2016
Accelerated Progress in Transistor Integration

- Large multicores for data centers and cloud
- 3D stacked chips

Intel Xeon Phi 7290F (Oct 2016)
72 cores, 288 contexts, 260W

Intel 3D Xpoint memory

Micron’s Hybrid Memory Cube
Research is Pushing Ever Farther Ahead

- More integration → 1,000 cores/chip

Runnemede prototype [HPCA-13]

- Research on stacking multiple processor and memory dies
Meanwhile: Power Wall… and Performance Wall

• University of Illinois Blue Waters Supercomputer

  Performance: 11 PF
  Power: 6-11 MW (idle to loaded)
  1MW = $1M per year electricity

• Technology improvements in speed and power slowing down

  Computer architecture innovations become strategic
What We Need

- Very high energy efficiency
- Faster communication and synchronization
- Ease of programming

All at the same time
Today’s Discussion

• Focus: Reducing the cost of basic primitives for parallelism

• Flavor of other challenges: energy, programmability
Making synchronization inexpensive
Making Synchronization Inexpensive

- Make memory fences free
  
  \[
  \begin{align*}
  &\text{wr } x \\
  &\text{fence} \\
  &\text{rd } z \\
  &\text{wr } y \\
  &\text{fence} \\
  &\text{rd } x \\
  &\text{wr } z \\
  &\text{fence} \\
  &\text{rd } y
  \end{align*}
  \]

- Breaking serialization in lock-free synchronization

  Compare\&Swap(CAS) \[\xrightarrow{\text{CAS}}\] x \[\xrightarrow{\text{CAS}}\] \[\xrightarrow{\text{CAS}}\] \[\xrightarrow{\text{CAS}}\]

- Scalable concurrent priority queues

  QueueHead \[\xrightarrow{\text{node}}\] node \[\xrightarrow{\text{node}}\] node \[\xrightarrow{\text{node}}\] node
Making Synchronization Inexpensive

- Make memory fences free (WeeFence)

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

- Breaking serialization in lock-free synchronization
- Scalable concurrent priority queues
Fence: a Primitive for Parallelism

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

Read x
Write y
Fence
Read z

Until these are finished
- reads retired
- writes retired + drained from write buffer

Cannot be observed by another processor
Use of Fences (I)

Enforce the correct order between accesses

- Programmers insert fences in codes with fine-grain sharing:
  - Work-stealing algorithm in Cilk

Worker dequeues from tail and checks head

\[
\text{Tail} = \ldots \quad \text{fence} \quad \ldots = \text{Head}
\]

Thief takes from head and checks tail

\[
\text{Head} = \ldots \quad \text{fence} \quad \ldots = \text{Tail}
\]
Use of Fences (II)

• **Compilers** insert fences in C++:
  – Programmer uses *intentional data race* for performance → declares variable as **atomic**
  – Compiler inserts fence after the access, does not reorder
  – Hardware does not reorder across fence
If We Remove Fences: **Incorrect Execution**

With fences: \( t_1 = 1, t_0 = 1 \) or both\( = 1 \)

A0 B0 A0
A1 B1 B0

Without fences: \( t_0 = t_1 = 0 \)

B0 A0 A1
B1 A1 B1

**Unintuitive bug:**
Sequential Consistency(SC) Violation

SC: execution appears as if accesses from multiple threads were interleaved in a uniprocessor
Fence Overhead

- Naïve implementation: stall all memory operations following the fence
  - The processor quickly stalls
Modern Implementations: Perform Speculation

- Reads following fences can load data speculatively
  - If no processor observes it, no problem
  - If coherence transaction received, rd is squashed and retried
- Still: speculative reads cannot retire until the WB is drained

Expensive:
- Fence in Xeon desktop stalls for 20—200 cycles.

In a large MP?
What if Fences Were Free?

• Programmers could write faster fine-grained concurrent algorithms

• C++/Java programmers would not have to worry about data races
  – Declare all shared variables as atomic
  – Compiler puts many fences, hardware still runs fast
  – Guaranteed Sequential Consistency (SC)
Proposal: **WeeFence** (or **W Fence**)

- **Goal:** Eliminate any stall in the pipeline
- **Post-fence read** [retires before](#) the pre-fence writes have drained
  - “Skip” the fence

Substantial gains when write misses pile-up before the fence

---

**Write**

**Fence**

**Read**

Spec execution

<table>
<thead>
<tr>
<th>Spec execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>r f W₂</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write Fence Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reorder Buffer (ROB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>W₂</td>
</tr>
<tr>
<td>W₁</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>W₁</td>
</tr>
<tr>
<td>W₂</td>
</tr>
</tbody>
</table>

**Substantial gains when write misses pile-up before the fence**
But… Not Stalling Can Cause Incorrect Execution

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>PA</th>
<th>PB</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0: x = 1</td>
<td>B0: y = 1</td>
</tr>
<tr>
<td>A1: t0 = y</td>
<td>B1: t1 = x</td>
</tr>
</tbody>
</table>

What we want: not stall but avoid these SC violations

Solution: Allow the reorder, check for this case, and stall the read (B1)
But… Not Stalling Can Cause Incorrect Execution

\[ x = y = 0 \]

\begin{align*}
\text{PA} & : x = 1 \\
\text{A0: } x &= 1 \\
\text{A1: } t_0 &= y \\
\text{WeeFence} & \\
\text{PB} & : y = 1 \\
\text{B0: } y &= 1 \\
\text{WeeFence} & \\
\text{B1: } t_1 &= x \\
\end{align*}

Write Fence Read

Time

What we want: not stall but avoid these SC violations

Solution: Allow the reorder, check for this case, and stall the read (B1)

Conventional fences always conservatively stall \( \leftrightarrow \) Not WeeFence
WeeFence: The Idea

- At a fence: record the thread’s incomplete writes in a HW structure
- Allow post-fence reads to execute before pre-fence writes complete
- Check post-fence reads (rd x) against HW structure to find conflicts with other threads’ incomplete writes.
  - Conflict? Stall read
  - Else: Retire

Prevent “rd x” retiring early if:
- There is a concurrent fence
- Accesses vars in opposite order
How WFence Works

PS: Pending Set

BS: Bypass Set

(1) PS

Table

(2)

execute

(3)


How WFence Works

PS: Pending Set → wr x

BS: Bypass Set → rd y

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

PA

PB

Wfence1
Wfence2

wr x
wr x
wr y
wr y

dr y
rd x
rd x

PS: Pending Set
BS: Bypass Set
How WFence Works (II)

PS: Pending Set \( \rightarrow \) wr \( x \)  
PFence1

BS: Bypass Set \( \rightarrow \) rd \( y \)

PA

PB

No fence present in x86

(1) PS

(2) Table

(3) execute

(4) BS y
How WFence Works (II)

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

PA

PS

WS

PA

(3)
execute

(4)
BS

y

stall

(1)
Wfence1

(2)
Table

(5)
coherence

PB

wr y

wr x

No fence present in x86

Table

x
Summary: How WFence Works

PS: Pending Set
BS: Bypass Set

PS: Pending Set
BS: Bypass Set

Table

execute & retire
WFence

- Cycles are rare: Wfence typically executes without stalling the processor.
- Works with cycles with any number of processors.

```
PA     PB     PC
wr x   wr y   wr z
Wfence Wfence Wfence
rd z   rd x   rd y
```

- No compiler support needed: Unmodified off-the-shelf executable.
Improving WeeFence

• The Global State is expensive to maintain with many threads

• Can we eliminate the Global State (Pending Set)
Eliminating the Global State

Deadlock....
Insight: no deadlock if one processor stalls at the fence and generates no BS

[ASPLOS-15]
Asymmetric Fences: Strong Fence + Weak Fences

- Given a conflict cycle with N processors:
  - 1 strong fence (no BS) = conventional fence
  - N-1 weak fences that allow reordering = WeeFences without PS

![Diagram of Asymmetric Fences](image)
Where to Put Strong Fence?

\[
\begin{align*}
\text{PA} & \quad \text{PB} \\
\text{tmp->field } &= 10; & \text{if (obj) } & \{
\text{fence1; } & \quad \text{fence2; } \\
\text{obj } &= \text{tmp; } & \text{a } &= \text{obj->field; }
\end{align*}
\]

Put strong fence in fence1, why?

It only executes once, at initialization

- Work stealing algorithm in Cilk:
  - Weak fences $\rightarrow$ workers
  - Strong fences $\rightarrow$ thieves

- Software transaction memory:
  - Weak fences $\rightarrow$ reads
  - Strong fences $\rightarrow$ writes
Results

Kernels with fences: WFence eliminates >90% of the fence stall time

Full apps: WFence reduces the overhead of fences-everywhere (hence guaranteeing SC) from 40% to 2%
Making Synchronization Inexpensive

- Make memory fences free
- **Breaking serialization** in lock-free synchronization (CASPAR)

![Diagram showing Compare&Swap (CAS) and multiple CAS operations]

- Scalable concurrent priority queues

[ASPLOS-16]
Bottleneck: Many Processors Synch on Same Var

- Operating systems, databases, language runtimes, mem allocators
- **Lock-free synchronization**: Manipulates data using atomic instructions instead of locks

```c
if (mem[addr] == old) {
    mem[addr]=new
}
```

[ASPLOS-16]
Simple Example Lock-Free Synchronization

Compare\&Swap(addr,old,new) \rightarrow if (mem[addr] == old) {
mem[addr]=new
}

Everyone adds 1:
while (true) {
old = x
new = old +1
if (CAS(mem, old, new))
return
}
Example: Pushing Nodes into Stack

new = malloc();
while (true) {
    old = top
    new->next = old
    if (CAS(&top, old, new))
        return
}

PA

node

top

PB
Example: Pushing Nodes into Stack

```c
new = malloc();
while (true) {
    old = top
    new->next = old
    if (CAS(&top, old, new))
        return
}
```
new = malloc();
while (true) {
    old = top
    new->next = old
    if (CAS(&top, old, new))
        return
}

Example: Pushing Nodes into Stack

new

PA

old_A = top

PB

old_B = top
Example: Pushing Nodes into Stack

```
new = malloc();
while (true) {
    old = top
    new->next = old
    if (CAS(&top, old, new))
        return
}
```
Example: Pushing Nodes into Stack

```
new = malloc();
while (true) {
    old = top
    new->next = old
    if (CAS(&top, old, new))
        return
}
```

Josep Torrellas
Toward Extreme Scale...
Problem: Serialization

new = malloc();
while (true) {
    old = top
    new→next = old
    if (CAS(&top, old, new))
        return
}

Our Goal:
All processors perform a successful CAS at the same time, in parallel
CASPAR: Main Idea

Two steps:

• Queue the “ld old” requests in HW in the directory
  – Provides efficient serialization: only one proc attempts the CAS at a time (others remain idle)
  – Similar to past work

• Break serialization: Two new ideas:
  – Eager forwarding
  – Parallel validation
Queue “ld old” Requests in HW in Directory

```c
new = malloc();
while (true) {
    old = top
    new->next = old
    if (CAS(&top, old, new))
        return
}
```

Similar to past work....
Basic Hardware: Queue of Requests in Directory

Diagram showing a queue of requests in a directory with cache lines and pointers to PA, PB, PC, and PD.
new = malloc();
while (true) {
    old = top
    new->next = old
    if (CAS(&top, old, new))
        return
}

```
Directory
```

```
PA  PB  PC  PD
```

```
CAS
```

```
l_{PB}  l_{PC}  l_{PD}
```

Basic Hardware: Queue of Requests in Directory

Directory

PA  PB  PC  PD

Directory

$ld_{PB}$  $ld_{PC}$  $ld_{PD}$

Josep Torrellas
Toward Extreme Scale
Basic Hardware: Queue of Requests in Directory

new = malloc();
while (true) {
    old = top
    new\rightarrow next = old
    if (CAS(&top, old, new))
        return
}

Directory

PA  PB  PC  PD

CAS

\rightarrow

\rightarrow

ld_{PC}  ld_{PD}
Basic Hardware: Queue of Requests in Directory

Directory

PA  PB  PC  PD

ld_{PC}  ld_{PD}
new = malloc();
while (true) {
    old = top
    new→next = old
    if (CAS(&top, old, new))
        return
}

Directory

Completely serial execution 😞
Breaking Serialization (1): Eager Forwarding

new = malloc();
while (true) {
    old = top
    new = next = old
    if (CAS(&top, old, new))
        return
}

Observation: In a proc,
“new” does not depend on “old”
“new” is ready well before CAS

PA
new

PB
new

PC
new

Id old
CAS

Id old
CAS

Waste

Waste

Time
Breaking Serialization (1): Eager Forwarding

```
new = malloc();
while (true) {
    old = top
    new→next = old
    if (CAS(&top, old, new))
        return
}
```

Predecessors: Eagerly forward “new”
Successors:
* Use “new” to satisfy “ld old”
* Perform a successful CAS,
* Continue speculatively like TM, no stall
Breaking Serialization (1): Eager Forwarding

new = malloc();
while (true) {
    old = top
    new = next = old
    if (CAS(&top, old, new))
        return
}

* All CAS succeeded in parallel
* All wasted time is eliminated
* Execution continues; does not stop
* Need a validation step to compare forwarded and real value
Breaking Serialization (1): Eager Forwarding

new = malloc();
while (true) {
    old = top
    new->next = old
    if (CAS(&top, old, new))
        return
}

Directory

PA  PB  PC  PD

line

new = malloc();
while (true) {
    old = top
    new->next = old
    if (CAS(&top, old, new))
        return
}
new = malloc();
while (true) {
    old = top
    new\rightarrow next = old
    if (CAS(&top, old, new))
        return
}

All procs decode CAS, find that “new” has been produced, and forward it to the directory in parallel

Directory

PA PB PC PD

IdPA IdPB IdPC IdPD
new = malloc();
while (true) {
    old = top
    new->next = old
    if (CAS(&top, old, new))
        return
}

```c
while (true) {
    old = top
    new->next = old
    if (CAS(&top, old, new))
        return
}
```

```
new = malloc();
while (true) {
    old = top
    new->next = old
    if (CAS(&top, old, new))
        return
}
```
Breaking Serialization (1): Eager Forwarding

new = malloc();
while (true) {
  old = top
  new→next = old
  if (CAS(&top, old, new))
    return
}

Proc_i uses new_{i-1} as the response to its “ld old” and proceeds speculatively in parallel
new = malloc();
while (true) {
    old = top
    new¬next = old
    if (CAS(&top, old, new))
        return
}

```
while (true) {
    old = top
    new¬next = old
    if (CAS(&top, old, new))
        return
}
```

```
new = malloc();
while (true) {
    old = top
    new¬next = old
    if (CAS(&top, old, new))
        return
}
```
Breaking Serialization (1): Eager Forwarding

Directory

Cache line

PA   PB   PC   PD

\[ \text{id}_{PA} \quad \text{id}_{PB} \quad \text{id}_{PC} \quad \text{id}_{PD} \]

new\_A \quad new\_B \quad new\_C \quad new\_D
Breaking Serialization (1): Eager Forwarding

Validate:
* Compare the final value of the line to new\textsubscript{A} forwarded earlier on
* Commit speculative execution

Directory

\[
\begin{array}{cccc}
\text{PA} & \text{PB} & \text{PC} & \text{PD} \\
\text{ld}_{PB} & \text{ld}_{PC} & \text{ld}_{PD} \\
\text{new}_B & \text{new}_C & \text{new}_D \\
\end{array}
\]
Breaking Serialization (1): Eager Forwarding

Directory

PA  PB  PC  PD

Validate

\[\text{ld}_{PB} \quad \text{ld}_{PC} \quad \text{ld}_{PD}\]

new\textsubscript{B} \quad new\textsubscript{C} \quad new\textsubscript{D}\]
Breaking Serialization (1): Eager Forwarding
Breaking Serialization (1): Eager Forwarding

PA  PB  PC  PD

Directory

\[ \text{id}_{PC} \quad \text{id}_{PD} \]

new\(_C\)  new\(_D\)

Validate
Breaking Serialization (1): Eager Forwarding

Parallel CAS execution 😊 Still serial validation 😞
new = malloc();
while (true) {
    old = top
    new->next = old
    if (CAS(&top, old, new))
        return
}

Limitation of Eager Forwarding

Long speculation increases the chances of squashing the threads
Breaking Serialization (2): Parallel Validation

new = malloc();
while (true) {
    old = top
    new→next = old
    if (CAS(&top, old, new))
        return
}

Idea: Validate in the directory without ever sending line to cores
How: Use new_i stored in directory
Breaking Serialization (2): Parallel Validation

new = malloc();
while (true) {
    old = top
    new→next = old
    if (CAS(&top, old, new))
        return
}

PA
new_A

PB
new_B

PC
new_C

CAS

Id old

Id old

Id old

i_1 Speculative execution

i_2 Speculative execution

i_3

DIRECTORY
Parallel validation

Speculative execution reduced to a minimum
Execution does not stop
Breaking Serialization (2): Parallel Validation

PA  PB  PC  PD
CAS  CAS  CAS  CAS  Parallel CAS

Directory

\[ ld_{PA} \quad ld_{PB} \quad ld_{PC} \quad ld_{PD} \]
\[ new_A \quad new_B \quad new_C \quad new_D \]
Breaking Serialization (2): Parallel Validation

Cache line

Directory

PA  PB  PC  PD

ld_{PA}  ld_{PB}  ld_{PC}  ld_{PD}
new_A  new_B  new_C  new_D
Breaking Serialization (2): Parallel Validation

Parallel CAS execution 😊 Parallel validation 😊
Summary

• Full parallel synchronization
  – Parallel successful CAS execution
  – Parallel validation

• Large speedups for 64-core runs:
  – Throughput of kernels increases by 80% avg
  – Execution time of application sections reduces by 60% avg
Making Synchronization Inexpensive

- WeeFence: Make memory fences free
- CASPAR: Breaking the serialization in lock-free synchronization
- Scalable concurrent priority queues
Today’s Discussion

• Focus: Reducing the cost of basic primitives for parallelism

• Flavor of other challenges: energy, programmability
QuickRec: A Prototype of Record and Replay (RnR)

- HW + OS record all non-deterministic events, so that a parallel program can be replayed deterministically
- Finds non-deterministic software bugs and security intrusions
- Built FPGA platform with a Pentium multicore
ScalCore: a Core for Voltage Scalability [HPCA-16]

- Decouple the $V_{dd}$ of logic and storage structures in the pipeline
- Reconfigure pipeline to fuse the faster storage-intensive stages
Control-Theoretic Energy/Performance Controllers

• Design control-theoretic controllers that track multiple outputs while actuating on multiple inputs

• Attains most efficient use of resources to deliver highest performance
Conclusion

• Lots of room to innovate in computer architecture at this time
• Many exciting interdisciplinary venues of research:
  – Performance, energy-efficiency & programmability
Toward Extreme-Scale Manycore Architectures

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

USC
October 2016