Rethinking Support for Region Conflict Exceptions

Swaranendu Biswas, Rui Zhang, Michael D. Bond, and Brandon Lucia

IPDPS 2019
C++ Program with Data Race

```cpp
X* x = NULL;
bool done = false;

Thread T1

x = new X();
done = true;

Thread T2

if (done) {
    x->func();
}
```
Catch-Fire Semantics in C++

C++ treats data races as errors

```
X* x = NULL;
bool done = false;
```

Thread T1

```
x = new X();
done = true;
```

Thread T2

```
if (done) {
x->func();
}
```
Catch-Fire Semantics in C++

ANYTHING  BAD CAN  HAPPEN!
Thread T1

```cpp
x = new X();
done = true;
```

Thread T2

```cpp
while (!done) {} 
x->func();
```
```cpp
X *x = NULL;
bool done = false;

// Thread T1
x = new X();
done = true;

// Thread T2
while (!done) {}
x->func();

// Thread T1
x = new X();
done = true;

// Thread T2
temp = done;
while (!temp) {}
```

**Infinite Loop:**

- Thread T1: `while (!done)`
- Thread T2: `while (!temp)`

The diagram illustrates the infinite loop due to the simultaneous threads that continuously check the `done` or `temp` flag, creating a deadlock.
X *x = NULL;
bool done = false;

Thread T1

x = new X();
done = true;

Thread T2

while (!done) {}
x->func();

Thread T1

done = true;
x = new X();

Thread T2

NPE

while (!done) {}
x->func();
Technical Perspective
Data Races are Evil with No Exceptions

By Sarita Adve

EXPLORING PARALLELISM HAS become the primary means to higher performance. Racy code. Java's safety requirements preclude the use of "undefined" behavior. Therefore, we must try to control the correctness of our programs by detecting the presence of data races.

The Therac-25 was not a device anyone was happy to see. It was a radiation therapy machine. In layman's terms it was a "cancer zapper"; a linear accelerator with a human as its target. Using X-rays or a beam of electrons, it was to destroy cancerous cells.

How to miscompile programs with "benign" data races

Hans-J. Boehm
HP Laboratories
Need for Stronger Semantics for Programs with Data Races

“The inability to define reasonable semantics for programs with data races is not just a theoretical shortcoming, but a fundamental hole in the foundation of our languages and systems.”

“We call upon software and hardware communities to develop languages and systems that enforce data-race-freedom, ...”
What Do We Mean by Strong Semantics?

End-to-end guarantees even for programs with data races
Outline

- Impact of Data Races on Language Models
- Strong Semantics with Region Conflict Exceptions
- Providing Region Conflict Exceptions
- ARC: Practical Architecture Support for Region Conflict Exceptions
- Comparison of ARC with Related Approaches
Strong Execution Semantics with Region Conflict Exceptions
C++ Program with Data Race

```cpp
X* x = NULL;
bool done = false;

Thread T1

x = new X();
done = true;

Thread T2

if (done) {
    x->func();
}
```
Data Race Exceptions

Thread T1

```cpp
X* x = NULL;
bool done = false;
x = new X();
done = true;
```

Thread T2

```cpp
if (done) {
x->func();
}
```

EXCEPTION
Region Conflicts

Thread T1

Thread T2

Conflict

wr x

rd x
Synchronization Free Regions (SFRs)

Thread T1
- lock m
- lock n

Thread T2
- unlock n
Region Conflicts

Thread T1

Thread T2

Report a subset of true data races that can potentially violate region serializability
Semantics with Region Conflict Exceptions

Conflict-free execution $\Rightarrow$ SFR serializability

Synchronisation-free regions (SFRs) execute atomically
Semantics with Region Conflict Exceptions

Conflict-free execution

Region conflict

true data races

SFR serializability

Potential serializability violation
Semantics with Region Conflict Exceptions

- Conflict-free execution
- Region conflict
- SFR serializability
- True data races
- Exception
Providing Region Conflict Exceptions
Providing Region Conflict Exceptions

Valor: Efficient, Software-Only Region Conflict Exceptions

Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races

Drawbacks with Conflict Exceptions

Builds on top of M(O)ESI-style cache coherence

• Introduces hardware on top existing structures
• Increases complexity

Inter-core communication at region boundaries

• Metadata in private cache lines are forwarded to other cores
• Increases on-chip interconnect bandwidth requirement

Private line evictions communicate with memory

• Relies on in-memory backup for evicted metadata
• Increases off-chip memory bandwidth requirement

ARC: Practical Architecture Support for Region Conflict Exceptions

Design Overview
Architectural Modifications
Example Executions with ARC
ARC: Practical Architecture Support for Region Conflict Exceptions

• Design Overview
Architectural Modifications
Example Executions with ARC
Baseline Architecture in ARC

- Core 1
  - Controller
  - L1
  - L2

- Core 2
  - Controller
  - L1
  - L2

- Core n
  - Controller
  - L1
  - L2

- LLC

- Main memory
Baseline Architecture in ARC

- Core 1
  - Controller
  - L1
  - L2

- Core 2
  - Controller
  - L1
  - L2

- Core n
  - Controller
  - L1
  - L2

Main memory

- LLC

no coherence protocol + directory

no core-to-core communication
Key Insights in ARC

- Self invalidation
- No M(O)ESI
- Release consistency
Release Consistency

core’s private cache waits to write back its dirty data until a synchronization release operation

```java
X = new Object();
done = true;
unlock(m);
```

```java
lock(m);
while (!done) {}}
X.compute();
```
Self-Invalidation

core invalidates private cache lines that may be out-of-date at synchronization acquire operations
ARC: Our Proposed Technique for Region Conflict Detection

Explore whether synergistic use of release consistency and self-invalidation can be competitive

Provide consistency and coherence at SFR boundaries and on private cache line evictions
Understanding how ARC Works

**ARC Core**

- **private cache line**
- **private cache line**

**Last Level Cache (LLC)**

- Line is fetched
- Check for conflicts with the LLC

- Line is evicted LLC
- Check for conflicts with the LLC
Understanding how ARC Works

ARC Core

- private cache line
- private cache line

Last Level Cache (LLC)

- Line is fetched
- Check for conflicts with the LLC

Region boundary operations
Serializability of Regions

A region appears serializable if:

- There were no conflicts
- Writes appear atomic
- Values read are consistent
A region appears serializable if:

- There were no conflicts
- Writes appear atomic
- Values read are consistent

At a region boundary, an ARC core executes:

- **Pre-commit** – Write back dirty lines to the LLC
Region Boundary Operations in ARC

A region appears serializable if:
- There were no conflicts
- Writes appear atomic
- Values read are consistent

At a region boundary, an ARC core executes:

- **Pre-commit** – Write back dirty lines to the LLC
- **Read validation** – Validate reads using version and value validation
Region Boundary Operations in ARC

A region appears serializable if:

- There were no conflicts
- Writes appear atomic
- Values read are consistent

provide coherence

At a region boundary, an ARC core executes:

- **Pre-commit** – Write back dirty lines to the LLC
- **Read validation** – Validate reads using version and value validation
- **Post-commit** – Clear per-core metadata, self-invalidate private lines
ARC: Practical Architecture Support for Region Conflict Exceptions

Design Overview

• Architectural Modifications

Example Executions with ARC
Detecting Sound and Precise Conflicts

- Byte offset i
- Read/write bits
- Per-byte metadata
- Private cache line
Modifications Introduced by ARC

- Core 1: Controller → L1 → L2
- Core 2: Controller → L1 → L2
- Core n: Controller → L1 → L2

Consistency controller and LLC connect to Main memory.
Metadata Management

- Byte offset i
- Read/write bits
- Per-byte metadata
- Incoming line
- Line to be evicted
- Evicted line + metadata
- LLC
- Line + metadata evicted to memory
Access Information Memory (AIM)

AIM is a dedicated metadata cache adjacent to the LLC

AIM lines in ARC can be large

- 100 bytes for 8 cores, 178 bytes for 16 cores, and 308 bytes for 32 cores
- Impractical to have large AIM cache structures

ARC assumes a realistic AIM design with 32K entries
ARC Architecture with AIM Cache

- Core 1
  - Controller
  - L1
  - L2

- Core 2
  - Controller
  - L1
  - L2

- Core n
  - Controller
  - L1
  - L2

- Access information buffer
- Main memory

- LLC
- consistency controller
- AIM

access metadata
ARC: Practical Architecture Support for Region Conflict Exceptions

Design Overview
Architectural Modifications

• Example Executions with ARC
Example Execution with ARC: No Conflict

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Q</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example Execution with ARC: No Conflict

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Core 1

\[ X = \ldots \]
\[ \ldots = Y \]

Last Level Cache (LLC)

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Q</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Core 2

\[ P = \ldots \]
\[ Q = \ldots \]

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Q</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Example Execution with ARC: No Conflict

Core 1

X = ...

... = Y

Core 2

P = ...

Q = ...

... = R

Last Level Cache (LLC)

- R W
X 0 0
Y 0 0
P 0 0
Q 0 0

- R W
P 0 1
Q 0 1
R 1 0

no region conflict, Core 1 writes back local updates to shared memory
Example Execution with ARC: No Conflict

Core 1

\[
X = \ldots
\]

\[
\ldots = Y
\]

Core 2

\[
P = \ldots
\]

\[
Q = \ldots
\]

\[
\ldots = R
\]

Last Level Cache (LLC)

\[
- \quad R \quad W
\]

\[
X \quad 0 \quad 0
\]

\[
Y \quad 0 \quad 0
\]

\[
- \quad R \quad W
\]

\[
P \quad 0 \quad 0
\]

\[
Q \quad 0 \quad 0
\]

\[
R \quad 0 \quad 0
\]

no region conflict, Core 2 writes back local updates to shared memory
Eager Conflict Detection with ARC: Conflict on Evicted Line
Eager Conflict Detection with ARC: Conflict on Evicted Line

Core 1

\[ X = \ldots \]
\[ \ldots = Y \]

Core 2

\[ P = \ldots \]
\[ Q = \ldots \]

Last Level Cache (LLC)

- \[ R \]
- \[ W \]

<table>
<thead>
<tr>
<th>-</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>-</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Q</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Eager Conflict Detection with ARC: Conflict on Evicted Line

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Core 1**

\[ X = \ldots \]

\[ \ldots = Y \]

**Last Level Cache (LLC)**

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Q</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Core 2**

\[ P = \ldots \]

\[ Q = \ldots \]

- **Cache line containing X gets evicted**
Eager Conflict Detection with ARC: Conflict on Evicted Line

Core 1:

\[ X = \ldots \]
\[ \ldots = Y \]

Core 2:

\[ P = \ldots \]
\[ Q = \ldots \]
\[ \ldots = X \]

Last Level Cache (LLC):

\[ - \quad R \quad W \]
\[ X \quad 0 \quad 1 \]
\[ Y \quad 0 \quad 0 \]
\[ P \quad 0 \quad 0 \]
\[ Q \quad 0 \quad 0 \]

- \quad R \quad W
- \quad P \quad 0 \quad 1
- \quad Q \quad 0 \quad 1
- \quad X \quad 1 \quad 0

**Eager Conflict**
Lazy Conflict Detection with ARC: Conflict on Private Lines

Core 1

Last Level Cache (LLC)

Core 2

- R W

- R W

X 0 0
Y 0 0
P 0 0
Q 0 0
Lazy Conflict Detection with ARC: Conflict on Private Lines

Core 1

Core 2

Last Level Cache (LLC)

\[ X = 20 \]

\[
\begin{array}{ccc}
- & R & W \\
X & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
- & R & W \\
- & X & 0 \\
Y & 0 & 0 \\
P & 0 & 0 \\
Q & 0 & 0 \\
\end{array}
\]

\[ \ldots = X (5) \]

\[
\begin{array}{ccc}
- & R & W \\
X & 1 & 0 \\
\end{array}
\]

no eager invalidation in Core 2 on write
Lazy Conflict Detection with ARC: Conflict on Private Lines

### Core 1

- **X** = 20
- ... = **Y**

<table>
<thead>
<tr>
<th>-</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Core 2

- ... = **X** (5)
- **Q** = ...

<table>
<thead>
<tr>
<th>-</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Q</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Last Level Cache (LLC)

<table>
<thead>
<tr>
<th>-</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Q</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Lazy Conflict Detection with ARC: Conflict on Private Lines

Core 1

\[ X = 20 \]
\[ \ldots = Y \]

Core 2

\[ \ldots = X (5) \]
\[ Q = \ldots \]

Last Level Cache (LLC)

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Q</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

no region conflict, Core 1 writes back local updates to shared memory
Lazy Conflict Detection with ARC: Conflict on Private Lines

Core 1

-no region conflict, Core 1 writes back local updates to shared memory

Core 2

lazy conflict on X

Last Level Cache (LLC)

fails to ensure consistency of reads

X = 20

... = Y

... = X (5)

Q = ...

-   R   W

-   R   W

-   R   W

-   R   W

-   R   W

-   R   W

-   R   W

X 0 0

X 0 0

X 0 0

X 1 0

Y 0 0

Y 0 0

Y 0 0

Q 0 1

P 0 0

Q 0 0

Q 0 1

no region conflict, Core 1 writes back local updates to shared memory

fails to ensure consistency of reads
Comparison of ARC with Related Approaches

Implementation and Evaluation
## Comparing Conflict Exceptions and ARC

<table>
<thead>
<tr>
<th>Conflict Exceptions</th>
<th>ARC</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Builds on M(O)ESI</td>
<td>• Adapts release consistency and self-invalidation schemes</td>
</tr>
<tr>
<td>• Coherence at granularity of memory accesses</td>
<td>• Coherence at region granularity</td>
</tr>
<tr>
<td>• Requires support for a Directory and point-to-point communication</td>
<td>• Requires a AIM cache, write signatures, and consistency controllers</td>
</tr>
<tr>
<td>• Detects conflicts eagerly</td>
<td>• Uses a mix of eager and lazy conflict detection</td>
</tr>
</tbody>
</table>

Implementation and Evaluation

• Simulation
  • A Pintool generates a stream of memory and synchronization events
  • Events are processed by model implementations of Conflict Exceptions (CE) and ARC
  • Use McPAT to estimate energy usage

https://github.com/PLaSSticity/ce-arc-simulator-ipdps19
Run-time Performance

Run-time normalized to CE-4

- blackscholes
- bodytrack
- canneal
- dedup
- ferret
- fluidanimate
- raytrace
- streamcluster
- swaptions
- vips
- x264
- geomean

normalized to CE-4
Run-time Performance

Run-time normalized to CE-4
Run-time Performance

- blackscholes
- bodytrack
- canneal
- dedup
- ferret
- fluidanimate
- raytrace
- streamcluster
- swaptions
- vips
- x264
- geomean

Run-time normalized to CE-4

CE-4, ARC-4, CE-8, ARC-8, CE-16, ARC-16, CE-32, ARC-32
Energy Usage

(normalized to CE-4)
Overhead of Providing Region Conflict Detection

- Current shared-memory systems provide undefined semantics for racy programs

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Overhead comparison at 32 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Run-time performance (%)</td>
</tr>
<tr>
<td>CE</td>
<td>26.7</td>
</tr>
<tr>
<td>ARC</td>
<td>12.5</td>
</tr>
</tbody>
</table>
**Key Takeaways!**

Release consistency and self-invalidation techniques can be a good fit for detecting region conflicts.

Small metadata cache provides reasonable tradeoffs between performance and complexity.

Compared to state-of-art, ARC shows promise in making region conflict detection practical.
Rethinking Support for Region Conflict Exceptions

Swararendra Biswas, Rui Zhang, Michael D. Bond, and Brandon Lucia

IPDPS 2019