AtomTracker: A Comprehensive Approach to Atomic Region Inference and Violation Detection

-Abdullah Muzahid, Norimasa Otsuki, Josep Torrellas
University of Illinois at Urbana-Champaign

Debugging Multithreaded Programs

"Debugging a multithreaded program has a lot in common with medieval torture methods"

-- Random quote found via Google search

Concurrency Bugs

- Multicore era => more parallel programs
  => more concurrency bugs

Concurrency Bugs

- Multicore era => more parallel programs
  => more concurrency bugs
- Atomicity violation bug
  - A type of concurrency bug
  - Reason: too short critical sections
  - Result: accesses from different threads interleave incorrectly
  - Very frequent, gets relatively less attention

Atomicity Violation Bug Example

class Point { int x, y; };

```
Thread1        Thread2
lock(l);
p.x = …
…
p.y = …
unlock(l);
               lock(l);
               p.x = …
   …           …
               p.y = …
               unlock(l);
```

Atomicity Violation Bug Example

class Point { int x, y; };

Thread1	Thread2
lock(l);	
p.x = …	
…	
p.y = …	
unlock(l);	
	lock(l);
	p.x = …
…	…
	p.y = …
	unlock(l);

Thread1	Thread2
lock(l);	
p.x = …	…
unlock(l);	
	lock(l);
	p.x = …
	unlock(l);
…	…
	lock(l);
	p.y = …
	unlock(l);
lock(l);	
p.y = …	…
unlock(l);	

Atomicity Violation Bug Example

class Point { int x, y; };

```
Thread1          Thread2
lock(l);
p.x = …
…
p.y = …
unlock(l);
                 lock(l);
                 p.x = …
   …             …
                 p.y = …
                 unlock(l);
```

```
Thread1          Thread2
lock(l);
p.x = …          …
unlock(l);
                 lock(l);
                 p.x = …
                 unlock(l);
   …             …
                 lock(l);
                 p.y = …
                 unlock(l);
lock(l);
p.y = …          …
unlock(l);
```

VIOLATION

Atomicity Violation Bug Example

class Point { int x, y; };

```
Thread1        Thread2
lock(l);
p.x = …
…
p.y = …
unlock(l);
               lock(l);
               p.x = …
  …            …
               p.y = …
               unlock(l);
```

```
Thread1        Thread2
lock(l);
p.x = …        …
unlock(l);
               lo…;
               … = …
               nlock(l);
  …            …
               lock(l);
               p.y = …
               unlock(l);
lock(l);
p.y = …        …
unlock(l);
```

VIOLATION

- Data race freedom does not imply atomicity bug freedom

State of the Art in Atomicity Violation Detection

- Many proposals
- We are interested in those that require no user annotations
- They are constrained in the types of Atomic Regions (AR):
  - Number of variables
  - Number of instructions
  - Type of code construction (e.g., a function)

State of the Art in Atomicity Violation Detection

- Many proposals
- We are interested in those that require no user annotations
- They are constrained in the types of Atomic Regions (AR):
  - Number of variables
  - Number of instructions
  - Type of code construction (e.g., a function)
- Example: AVIO [Lu06]

Access(x)

Access(x)

Access(x)

Outline

- Motivation
- Contributions
- Main Idea
- Results
- Conclusions

Outline

- Motivation
- Contributions
- Main Idea
- Results
- Conclusions

Contributions

Contributions

- Novel algorithm to infer arbitrary atomic regions (AR)
  - Needs no annotation at all

Contributions

- Novel algorithm to infer arbitrary atomic regions (AR)
  - Needs no annotation at all
- Novel algorithm to detect atomicity violations at runtime
  - A software implementation
  - A hardware implementation with negligible execution overhead

Contributions

- Novel algorithm to infer arbitrary atomic regions (AR)
  - Needs no annotation at all
- Novel algorithm to detect atomicity violations at runtime
  - A software implementation
  - A hardware implementation with negligible execution overhead
- First proposal that works with any AR
  - Any number of variables
  - Any number of instructions
  - Not dependent on code construct

Contributions

- Novel algorithm to infer arbitrary atomic regions (AR)
  - Needs no annotation at all
- Novel algorithm to detect atomicity violations at runtime
  - A software implementation
  - A hardware implementation with negligible execution overhead
- First proposal that works with any AR
  - Any number of variables
  - Any number of instructions
  - Not dependent on code construct
- Detects 8 atomicity violation bugs from real code

Outline

- Motivation
- Contributions
- Main Idea
- Results
- Conclusions

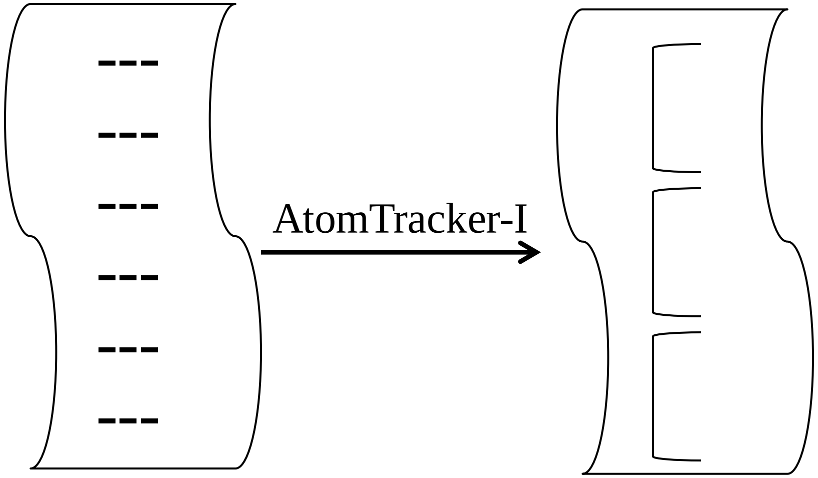
Proposal: AtomTracker

Proposal: AtomTracker

- It has two parts:

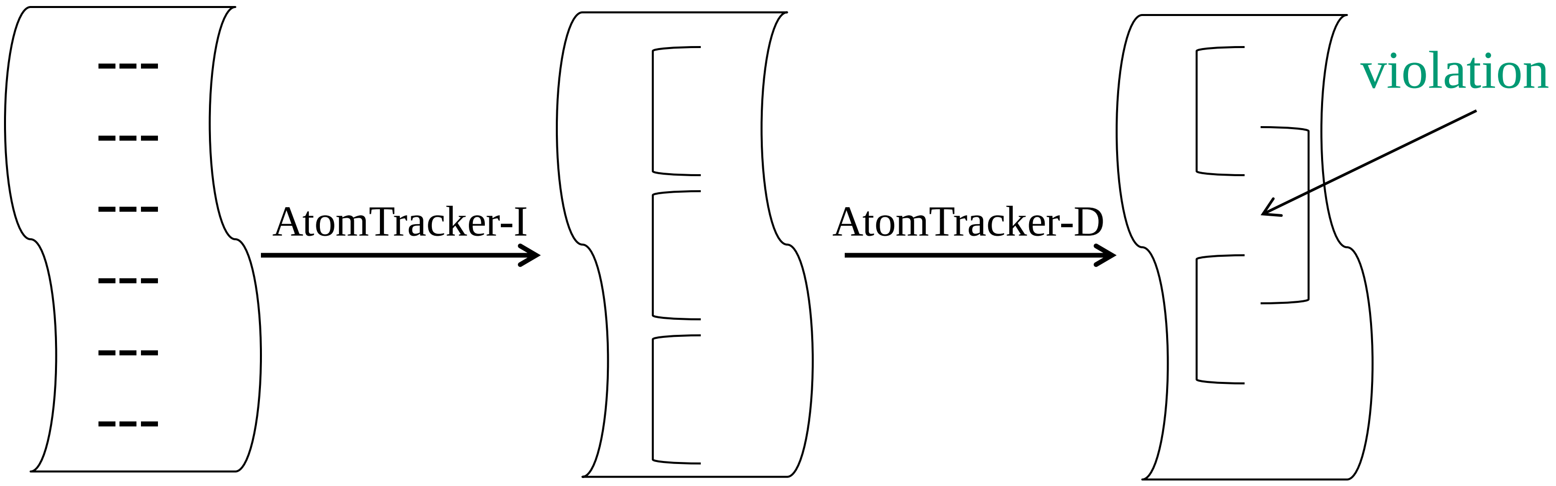
Proposal: AtomTracker

- It has two parts:
  - AtomTracker-I: Automatically infers generic ARs



Proposal: AtomTracker

- It has two parts:
  - AtomTracker-I: Automatically infers generic ARs
  - AtomTracker-D: Automatically detects violations of them at runtime



AtomTracker-I

AtomTracker-I

- Infers ARs without programmer’s annotations

AtomTracker-I

- Infers ARs without programmer's annotations
- Input:
  - Traces of multiple correct runs
  - Each trace is a total order of memory accesses during one execution

AtomTracker-I

- Infers ARs without programmer's annotations
- Input:
  - Traces of multiple correct runs
  - Each trace is a total order of memory accesses during one execution
- Approach: greedily try to find the largest possible ARs

Example of How AtomTracker-I Works

T1	T2
Rd X	
	Wr Y
Rd Y	
	Wr X
Rd X	
	Rd X
Wr X	
	Wr Y
Wr Y	
	Rd Y
Wr Y	

Example of How AtomTracker-I Works

T1	T2
Rd X	
	Wr Y
Rd Y	
	Wr X
Rd X	
	Rd X
Wr X	
	Wr Y
Wr Y	
	Rd Y
Wr Y	

Example of How AtomTracker-I Works

T1	T2
	Wr Y
Rd X, Rd Y	
	Wr X
Rd X	
	Rd X
Wr X	
	Wr Y
Wr Y	
	Rd Y
Wr Y	

Example of How AtomTracker-I Works

T1	T2
	Wr Y
Rd X, Rd Y	
	Wr X
Rd X	
	Rd X
Wr X	
	Wr Y
Wr Y	
	Rd Y
Wr Y	

Example of How AtomTracker-I Works

T1	T2
	Wr Y
Rd X, Rd Y	
	Wr X
Rd X	
	Rd X
Wr X	
	Wr Y
Wr Y	
	Rd Y
Wr Y	

Example of How AtomTracker-I Works

T1	T2
	Wr Y
Rd X, Rd Y	
	Wr X
Rd X	
	Rd X
Wr X	
	Wr Y
Wr Y	
	Rd Y
Wr Y	

Example of How AtomTracker-I Works

T1	T2
	Wr Y
Rd X, Rd Y	
	Wr X
	Rd X
Rd X, Wr X	
	Wr Y
Wr Y	
	Rd Y
Wr Y	

Example of How AtomTracker-I Works

T1	T2
	Wr Y
Rd X, Rd Y	
	Wr X
	Rd X
Rd X, Wr X	
	Wr Y
Wr Y	
	Rd Y
Wr Y	

Example of How AtomTracker-I Works

T1	T2
	Wr Y
Rd X, Rd Y	
	Wr X
	Rd X
	Wr Y
Rd X, Wr X, Wr Y	
	Rd Y
Wr Y	

Example of How AtomTracker-I Works

T1	T2
	Wr Y
Rd X, Rd Y	
	Wr X
	Rd X
	Wr Y
Rd X, Wr X, Wr Y	
	Rd Y
Wr Y	

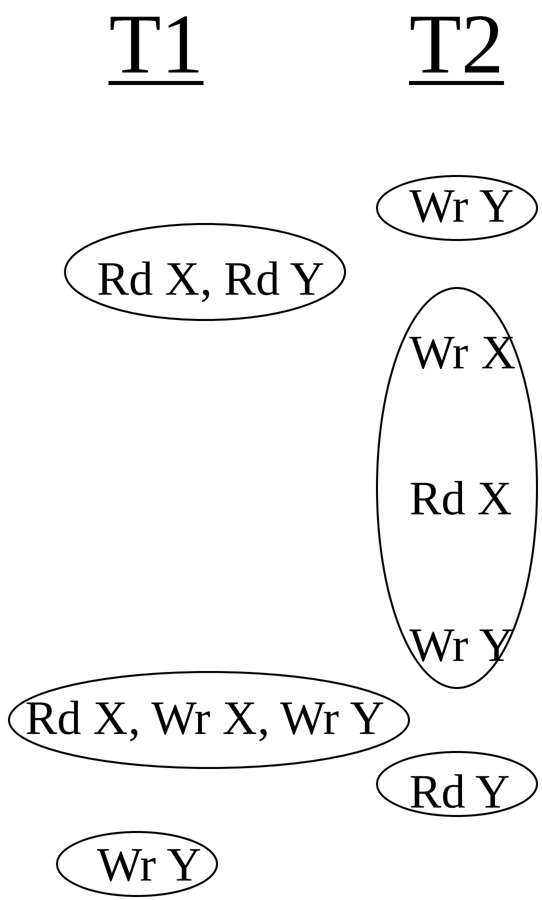
Example of How AtomTracker-I Works

T1	T2
	Wr Y
Rd X, Rd Y	
	Wr X
	Rd X
	Wr Y
Rd X, Wr X, Wr Y	
	Rd Y
Wr Y	

Example of How AtomTracker-I Works

T1	T2
	Wr Y
Rd X, Rd Y	
	Wr X
	Rd X
	Wr Y
Rd X, Wr X, Wr Y	
	Rd Y
Wr Y	

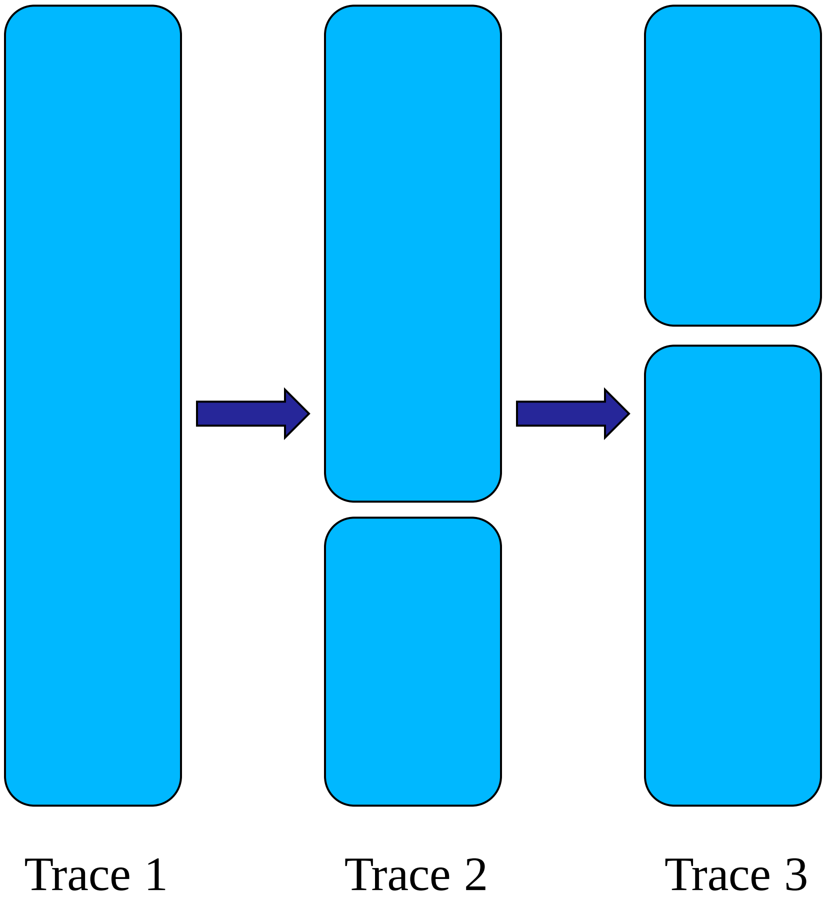
Example of How AtomTracker-I Works



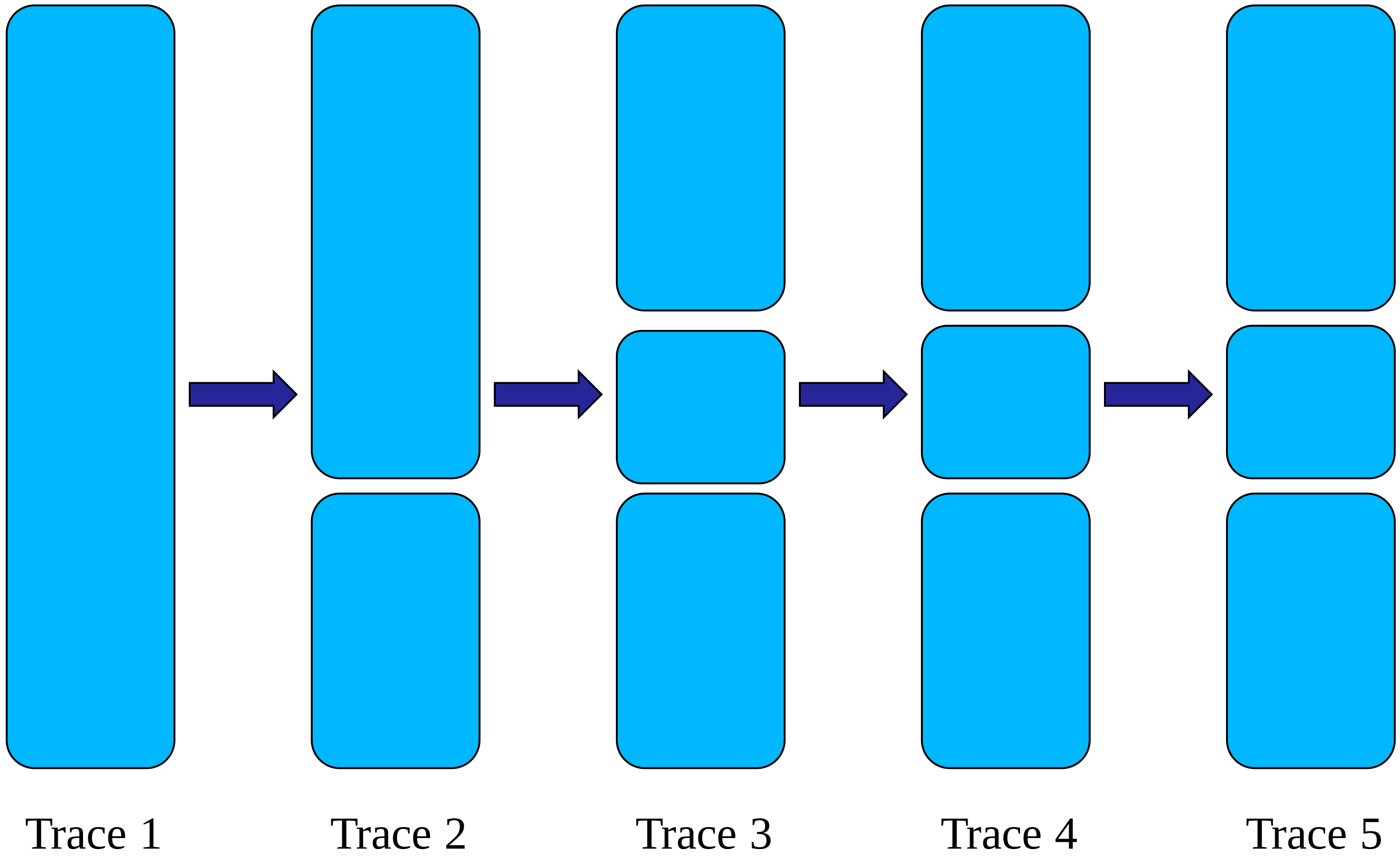
Convergence of AtomTracker-I

- Given a trace, find out the largest possible ARs
- As we see more and more traces, the larger ARs will get divided into multiple smaller ARs
- Eventually, we will find ARs close to the actual ones

Illustrative Example of Convergence



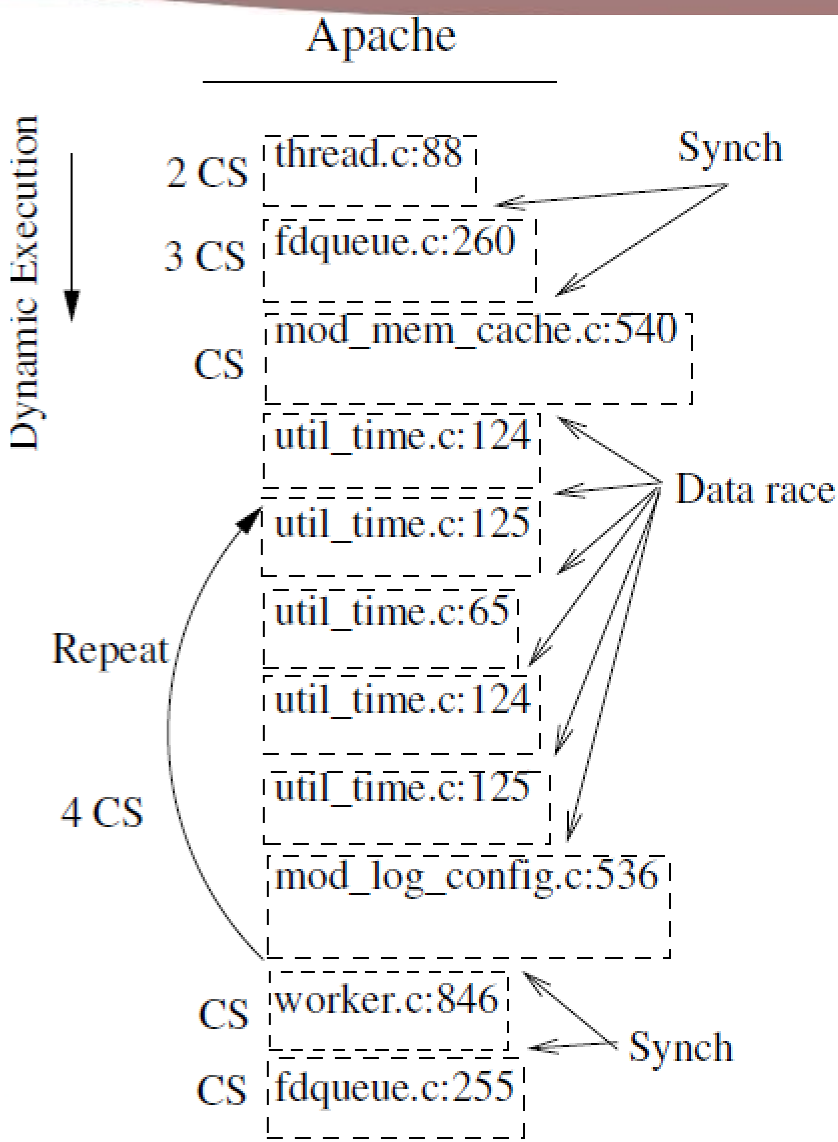
Illustrative Example of Convergence



Design Decisions of AtomTracker-I

- Ignore synchronization accesses
  - Sync accesses might be incorrect
- Treat critical sections as indivisible group of instructions during the merging process
  - Typically critical sections are supposed to be atomic
- Finish AR at loop iteration boundaries
  - AR usually do not stride loop iteration

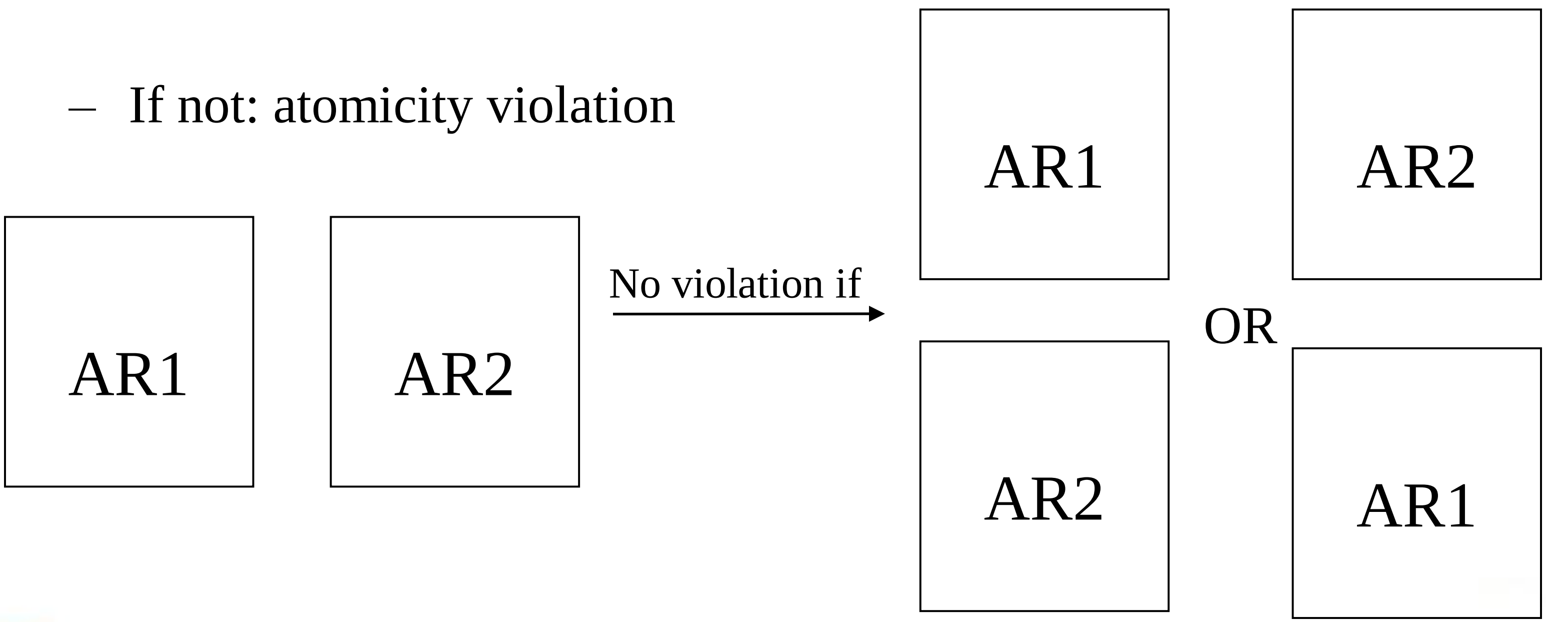
Result of Applying AtomTracker-I



- AR boundaries coincide with
  - Synchronizations
  - Data races
- Some AR contain multiple critical sections (CS)

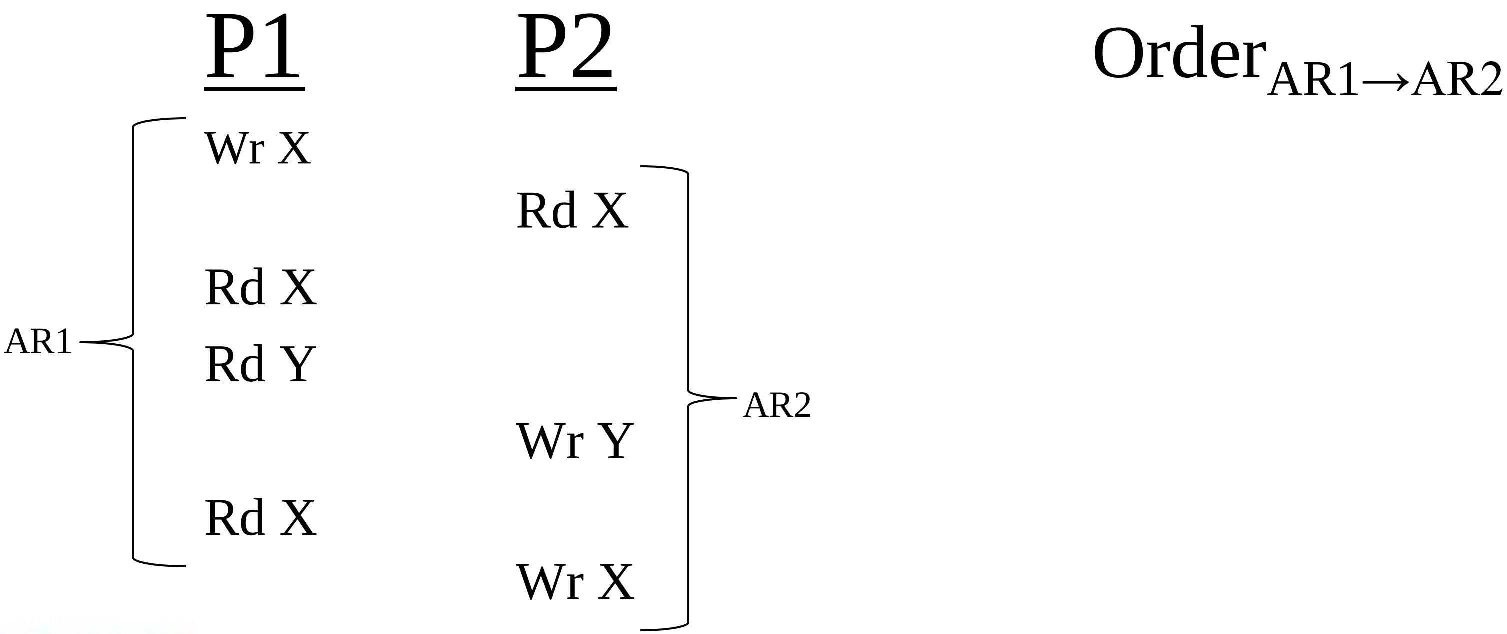
AtomTracker-D

- Takes a program with annotations in the binary
  - AR_enter, AR_exit
- Detects violations of these AR at runtime
- Idea: As two ARs execute concurrently, AtomTracker-D checks if they can be made to appear to execute in sequence
  - If not: atomicity violation



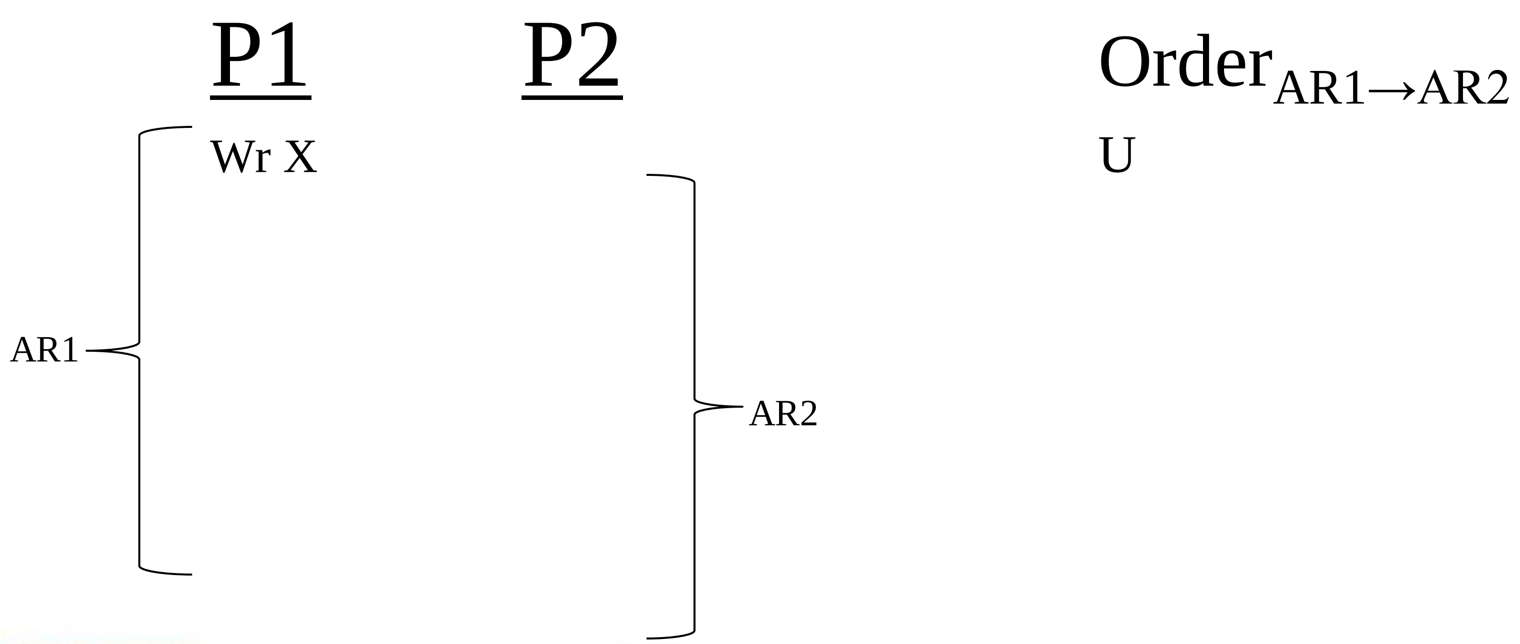
How AtomTracker-D Works

- Considers each access of the two concurrent ARs in order and checks for conflicts
- Each processor keeps a local flag
  - Tells the order of own AR relative to other AR
  - Values: {Before (B), After (A), Unordered (U)}



How AtomTracker-D Works

- Considers each access of the two concurrent ARs in order and checks for conflicts
- Each processor keeps a local flag
  - Tells the order of own AR relative to other AR
  - Values: {Before (B), After (A), Unordered (U)}



How AtomTracker-D Works

- Considers each access of the two concurrent ARs in order and checks for conflicts
- Each processor keeps a local flag
  - Tells the order of own AR relative to other AR
  - Values: {Before (B), After (A), Unordered (U)}

P1	P2	$\text{Order}_{AR1 \rightarrow AR2}$
Wr X		U
	Rd X	U & B = B

AR1

AR2

How AtomTracker-D Works

- Considers each access of the two concurrent ARs in order and checks for conflicts
- Each processor keeps a local flag
  - Tells the order of own AR relative to other AR
  - Values: {Before (B), After (A), Unordered (U)}

<u>P1</u>	<u>P2</u>	$\text{Order}_{AR1 \rightarrow AR2}$
Wr X		U
	Rd X	U & B = B
Rd X		B & U = B

AR1

AR2

How AtomTracker-D Works

- Considers each access of the two concurrent ARs in order and checks for conflicts
- Each processor keeps a local flag
  - Tells the order of own AR relative to other AR
  - Values: {Before (B), After (A), Unordered (U)}

P1	P2	$Order_{AR1 \rightarrow AR2}$
Wr X		U
	Rd X	U & B = B
Rd X		B & U = B
Rd Y		B & U = B

AR1

AR2

How AtomTracker-D Works

- Considers each access of the two concurrent ARs in order and checks for conflicts
- Each processor keeps a local flag
  - Tells the order of own AR relative to other AR
  - Values: {Before (B), After (A), Unordered (U)}

P1 (AR1)	P2 (AR2)	$\text{Order}_{AR1 \rightarrow AR2}$
Wr X		U
	Rd X	U & B = B
Rd X		B & U = B
Rd Y		B & U = B
	Wr Y	B & B = B

How AtomTracker-D Works

- Considers each access of the two concurrent ARs in order and checks for conflicts
- Each processor keeps a local flag
  - Tells the order of own AR relative to other AR
  - Values: {Before (B), After (A), Unordered (U)}

P1	P2	Order$_{AR1 \rightarrow AR2}$
Wr X		U
	Rd X	U & B = B
Rd X		B & U = B
Rd Y		B & U = B
	Wr Y	B & B = B
Rd X		B & U = B => No Violation

AR1

AR2

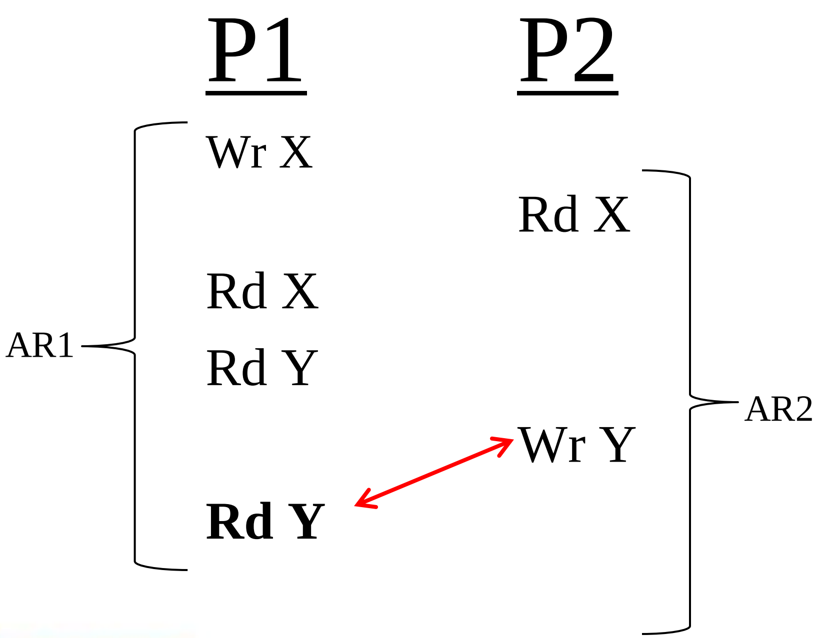
How AtomTracker-D Works

- Considers each access of the two concurrent ARs in order and checks for conflicts
- Each processor keeps a local flag
  - Tells the order of own AR relative to other AR
  - Values: {Before (B), After (A), Unordered (U)}

	P1	P2		$Order_{AR1 \rightarrow AR2}$
AR1	Wr X		AR2	U
		Rd X		U & B = B
	Rd X			B & U = B
	Rd Y			B & U = B
		Wr Y		B & B = B
	Rd Y			

How AtomTracker-D Works

- Considers each access of the two concurrent ARs in order and checks for conflicts
- Each processor keeps a local flag
  - Tells the order of own AR relative to other AR
  - Values: {Before (B), After (A), Unordered (U)}



$Order_{AR1 \rightarrow AR2}$

U

U & B = B

B & U = B

B & U = B

B & B = B

B **& A ==>** Violation

AtomTracker-D Implementation

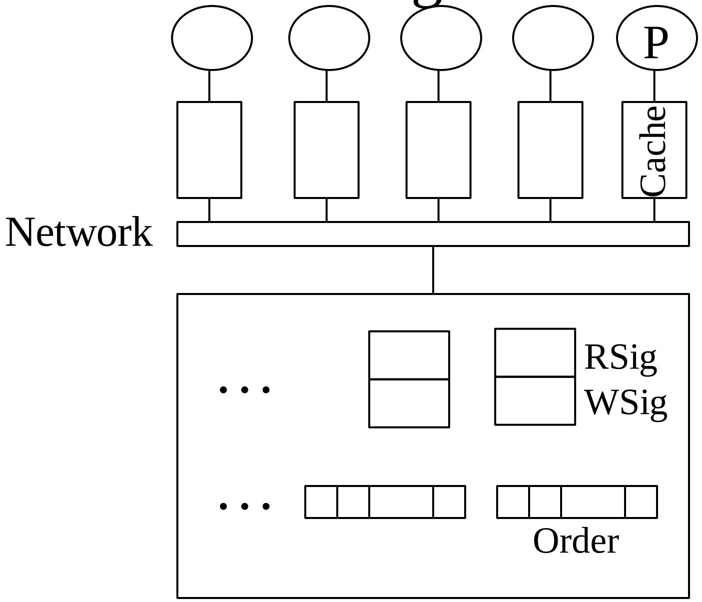
- Software:
  - Use PIN
  - For each access, check a software data structure
- Hardware:
  - Leverage the cache coherence protocol messages
  - Negligible execution overhead

AtomTracker-D Hardware Implementation

- Key insight: AtomTracker-D
  - Does not need to see all accesses
  - Only needs to see those that can change Order flag
- What are these accesses?
  - Those that introduce WAR, WAW, RAW deps
    - First one of these induces cache coh msg
- Problem: first access in an AR may be invisible to coherence protocol
  - First read to line in S state or first R/W to line in D state
  - Solution: R and W FirstAccess bits in cache tags

Bus Based Implementation

- Hardware module (AVM) on bus
- Signatures to summarize accesses with little state
  - Detect ordering conflicts of ARs
- Each memory access updates the Order flags in all the processors using signatures
- Suffers from false sharing and false positives



Atomicity Violation Detection Module (AVM)

Outline

- Motivation
- Contributions
- Main Idea
- Results
- Conclusions

Evaluation

- 8 real atomicity bugs from
  - Apache,
  - MySql and
  - Mozilla suite.

Evaluation

- Simulated AtomTracker-D hardware with Simics

Multicore Core	4 cores at 4 GHz 4 issue out-of-order
L1 cache (private)	32 KB, 4 way, 2 cycle lat
L2 cache (private)	512 KB, 8 way, 12 cycle lat
Cache line	64B
Memory	80 cycle round trip lat
Network	Bus
Bus bandwidth	128B/cycle
Coherence protocol	MESI
Signatures	2 Kbit

Bug Description

Bug #	Version	# Variables involved
Apache 1	2.0.48	Single
Apache 2	2.0.46	Single
Mozilla 1	0.8	Multiple
Mozilla 2	0.8	Multiple
Mozilla 3	0.9	Multiple
MySQL 1	4.0.12	Single
MySQL 2	3.23.56	Multiple
MySQL 3	4.0.16	Multiple

Bug Detection

Bug #	AVIO[Lu06]	Pset[Yu09]	MUVI[Lu07]	AtomTracker
Apache 1	Y	Y	N	Y
Apache 2	Y	Y	N	Y
Mozilla 1	N	N	Y	Y
Mozilla 2	N	N	Y	Y
Mozilla 3	N	N	Y	Y
MySQL 1	Y	Y	N	Y
MySQL 2	N	N	N	Y
MySQL 3	N	N	Y	Y

Bug Detection

Bug #	AVIO[Lu06]	Pset[Yu09]	MUVI[Lu07]	AtomTracker
Apache 1	Y	Y	N	Y
Apache 2	Y	Y	N	Y
Mozilla 1	N	N	Y	Y
Mozilla 2	N	N	Y	Y
Mozilla 3	N	N	Y	Y
MySQL 1	Y	Y	N	Y
MySQL 2	N	N	N	Y
MySQL 3	N	N	Y	Y

Bug Detection

Bug #	AVIO[Lu06]	Pset[Yu09]	MUVI[Lu07]	AtomTracker
Apache 1	Y	Y	N	Y
Apache 2	Y	Y	N	Y
Mozilla 1	N	N	Y	Y
Mozilla 2	N	N	Y	Y
Mozilla 3	N	N	Y	Y
MySQL 1	Y	Y	N	Y
MySQL 2	N	N	N	Y
MySQL 3	N	N	Y	Y

Bug Detection

Bug #	AVIO[Lu06]	Pset[Yu09]	MUVI[Lu07]	AtomTracker
Apache 1	Y	Y	N	Y
Apache 2	Y	Y	N	Y
Mozilla 1	N	N	Y	Y
Mozilla 2	N	N	Y	Y
Mozilla 3	N	N	Y	Y
MySQL 1	Y	Y	N	Y
MySQL 2	N	N	N	Y
MySQL 3	N	N	Y	Y

Bug Detection

Bug #	AVIO[Lu06]	Pset[Yu09]	MUVI[Lu07]	AtomTracker
Apache 1	Y	Y	N	Y
Apache 2	Y	Y	N	Y
Mozilla 1	N	N	Y	Y
Mozilla 2	N	N	Y	Y
Mozilla 3	N	N	Y	Y
MySQL 1	Y	Y	N	Y
MySQL 2	N	N	N	Y
MySQL 3	N	N	Y	Y

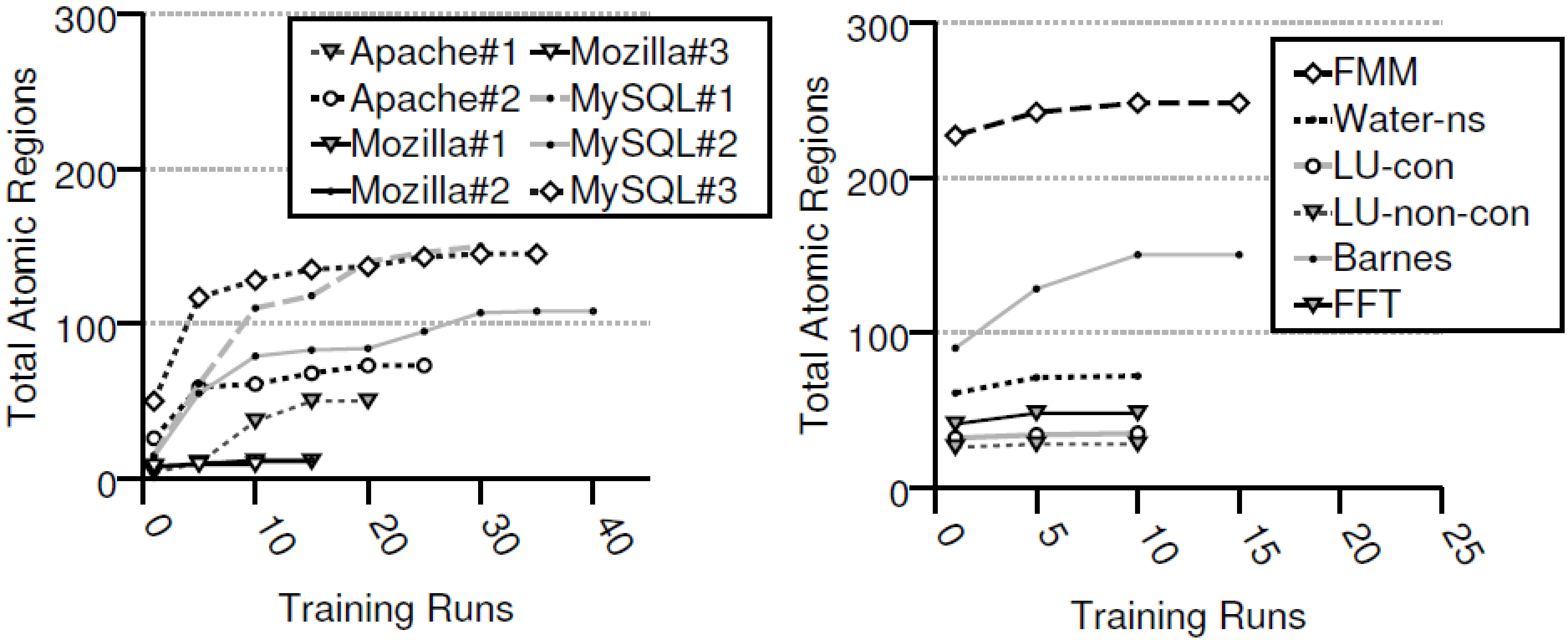
- Detects both single and multiple variable bugs

Bug Detection

Bug #	AVIO[Lu06]	Pset[Yu09]	MUVI[Lu07]	AtomTracker
Apache 1	Y	Y	N	Y
Apache 2	Y	Y	N	Y
Mozilla 1	N	N	Y	Y
Mozilla 2	N	N	Y	Y
Mozilla 3	N	N	Y	Y
MySQL 1	Y	Y	N	Y
MySQL 2	**N**	**N**	**N**	**Y**
MySQL 3	N	N	Y	Y

- Detects both single and multiple variable bugs
- Detects all bugs, even those not detected by others

AtomTracker-I Training



- It takes 10 - 40 training runs

AtomTracker-D Overhead

App	Hardware Impl Overhead	
	Execution Time Increase (%)	Traffic Increase(%)
Apache 1	0.1	1.3
Apache 2	0.1	1.0
Mozilla 1	0.1	5.5
Mozilla 2	0.5	6.3
Mozilla 3	0.1	2.7
MySQL 1	0.1	4.2
MySQL 2	0.1	1.5
MySQL 3	0.3	3.8
AVG	0.2	3.3

- Negligible execution time and traffic overhead
- Suitable for production runs

AtomTracker-D Overhead

App	Software Impl Overhead	
	Pin Only (X)	Total (X)
Apache 1	8.7	80.4
Apache 2	6.9	74.1
Mozilla 1	2.9	14.6
Mozilla 2	1.3	2.1
Mozilla 3	1.5	1.9
MySQL 1	6.2	15.9
MySQL 2	7.5	13.9
MySQL 3	1.8	2.8
AVG	4.6	25.7

- Reasonable for testing environment
- Software implementation is improvable

False Positives (FP)

Application	Software Impl	Hardware impl	
	No false sharing & aliasing	Only false sharing, no aliasing	Both false sharing & aliasing
Apache 1	1.8	2.0	2.0
Apache 2	2.6	10.4	14.4
Mozilla 1	0.4	1.8	1.8
Mozilla 2	0.0	3.0	6.0
Mozilla 3	0.0	0.0	0.0
MySQL 1	0.6	12.2	15.0
MySQL 2	0.8	7.6	9.6
MySQL 3	0.2	17.2	18.0
AVG	0.8	6.8	8.4

Outline

- Motivation
- Contributions
- Main Idea
- Results
- Conclusions

Conclusions

- AtomTracker – A novel technique for atomicity violation detection
  - AtomTracker-I automatically infers ARs
  - AtomTracker-D automatically detects violations at run-time
    - Suitable to implement in hardware using signatures
- First proposal to handle arbitrary AR
- Detected 8 atomicity violations in real world code
- Hardware implementation induces negligible exec overhead → production runs

AtomTracker: A Comprehensive Approach to Atomic Region Inference and Violation Detection

-Abdullah Muzahid, Norimasa Otsuki, Josep Torrellas

University of Illinois at Urbana-Champaign

i-acoma group

UPCRC Illinois Universal Parallel Computing Research Center

Atomic Regions

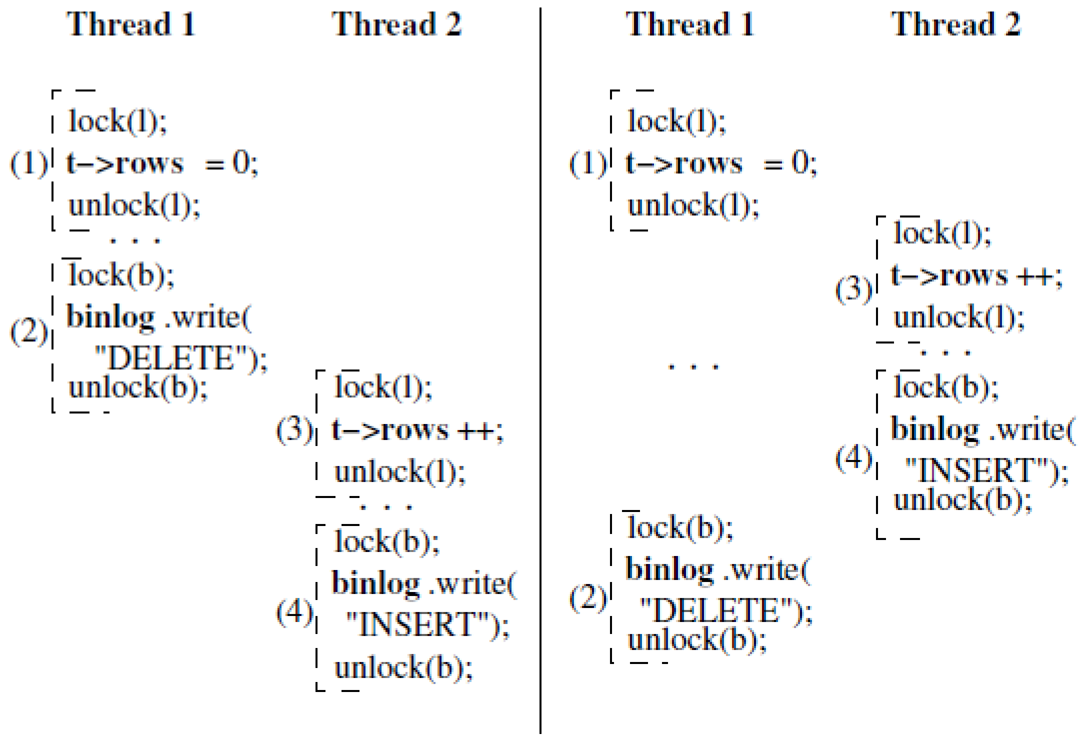
Applications	# LOC	# Shared Var
Apache 1	44.3	62.4
Apache 2	76.4	125.6
Mozilla 1	62.2	82.9
Mozilla 2	83.8	197.1
Mozilla 3	46.4	102.1
MySQL 1	43.9	230.4
MySQL 2	44.3	82.4
MySQL 3	17.0	31.8
AVG	52.3	114.3
SP2 kernels	4.6	215.5
SP2 apps	4.7	28.2

Effectiveness of Prepasses

Microbenchmarks	ATI (%)	ATI-CS (%)	ATI-LP (%)	ATI-LP – CS (%)
LinkedList (17)	100	58.8	94.1	52.9
ProdCons (4)	100	100	75	75
FFT (14)	100	78.6	100	78.6
AVG	100	79.1	89.7	68.8

- Both passes are essential

MySQL 2 Bug



DELETE & INSERT recorded in correct order | DELETE & INSERT recorded in wrong order

Problem With Nesting

thread 1	thread 2	thread 1	thread 2
lock(l);		lock(l);	
...= y;		...= y;	
lock(m);		lock(m);	
x = ...;	. . .	x = ...;	. . .
unlock(m);		unlock(m);	
	lock(m);		lock(m);
	x = ...;		x = ...;
lock(m);	unlock(m);	lock(m);	unlock(m);
x = ...;	. . .	x = ...;	. . .
unlock(m);		unlock(m);	
y = ...;		y = ...;	
unlock(l);		unlock(l);	