CARE

Yanyan Jiang
Tianxiao Gu
Chang Xu
Xiaoxing Ma
Jian Lu

Cache Guided Deterministic Replay for Concurrent Java Programs

What is it about?

- Concurrent programs are difficult to debug.
- Use deterministic replay
- Search-based
  - Small log, small record cost, incomplete
  - Best-effort exhaustive state space search
- Order-based
  - Record dependences among key events: R/W, (un)lock
  - Huge logs: STRIDE 30MB/s → Performance degradation
  - Easy replay

CARE

- Cache guided deterministic replay
- Key Idea: Take advantage of thread locality
  - i.e. no need to record access to same variable by same thread twice.
- Only record dependencies among different Threads
- Cache miss → other thread did access same variable
- Cache miss detected by value prediction cache

Value Prediction Cache

No cache miss

T1

x = 1;

y = 2;

read x;

T2

z = 2;

T1 VPC:	T1 Real cache:
x = 1	x = 1
y = 2	y = 2

T2 VPC:	T2 Real cache:
z = 2	z = 2

Value Prediction Cache

T1

x = 1;

y = 2;

read x;

T2

x = 2;

VPC:	Real cache:
x = 1	x = Invalid
y = 2	y = 2

VPC:	Real cache:
x = 2	x = 2

Value Prediction Cache

Cache miss detected

T1

x = 1;

y = 2;

read x;

VPC:	Real cache:
x = 1	x = 2
y = 2	y = 2

T2

x = 2;

VPC:	Real cache:
x = 2	x = 2

Value Prediction Cache

T1

x = 1;

y = 2;

read x;

T2

x = 1;

VPC:	Real cache:
x = 1	x = invalid
y = 2	y = 2

VPC:	Real cache:
x = 1	x = 1

Value Prediction Cache

Cache miss not detected

T1

x = 1;

y = 2;

read x;

T2

x = 1;

VPC:	Real cache:
x = 1	x = 1
y = 2	y = 2

VPC:	Real cache:
x = 1	x = 1

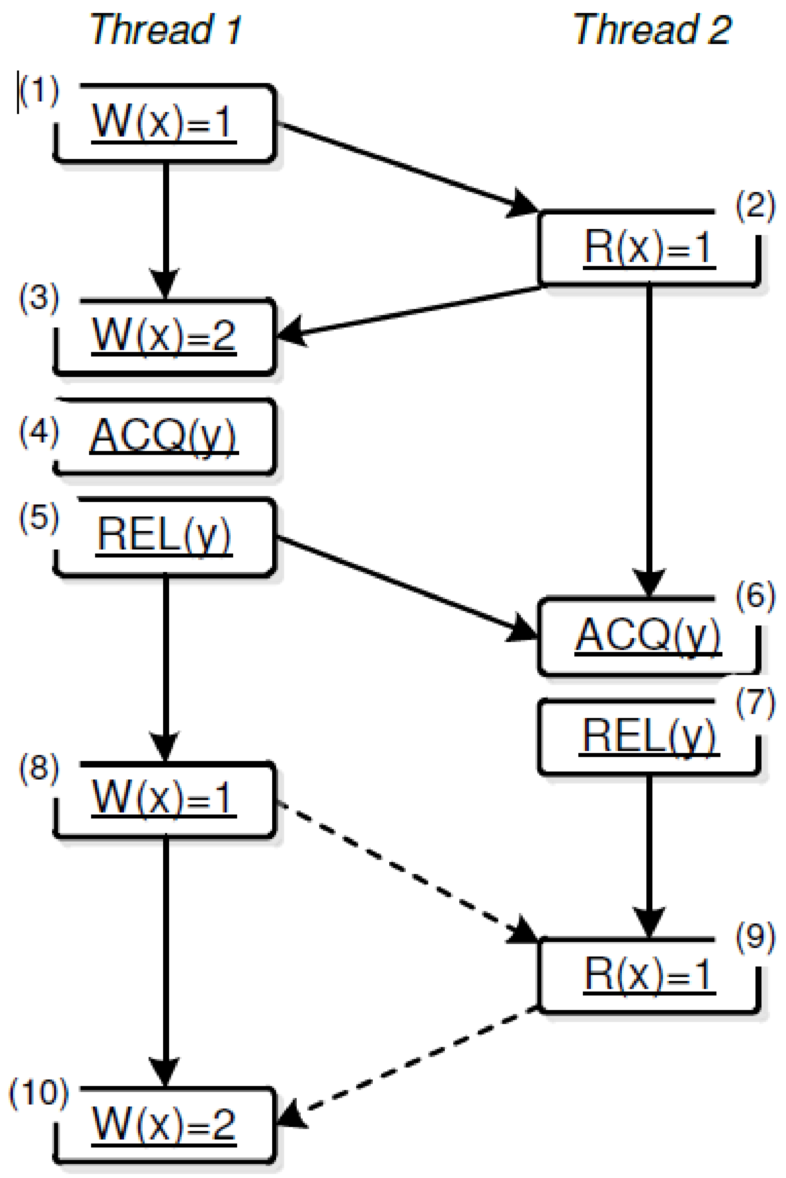


Figure 1: Illustration of missing dependences

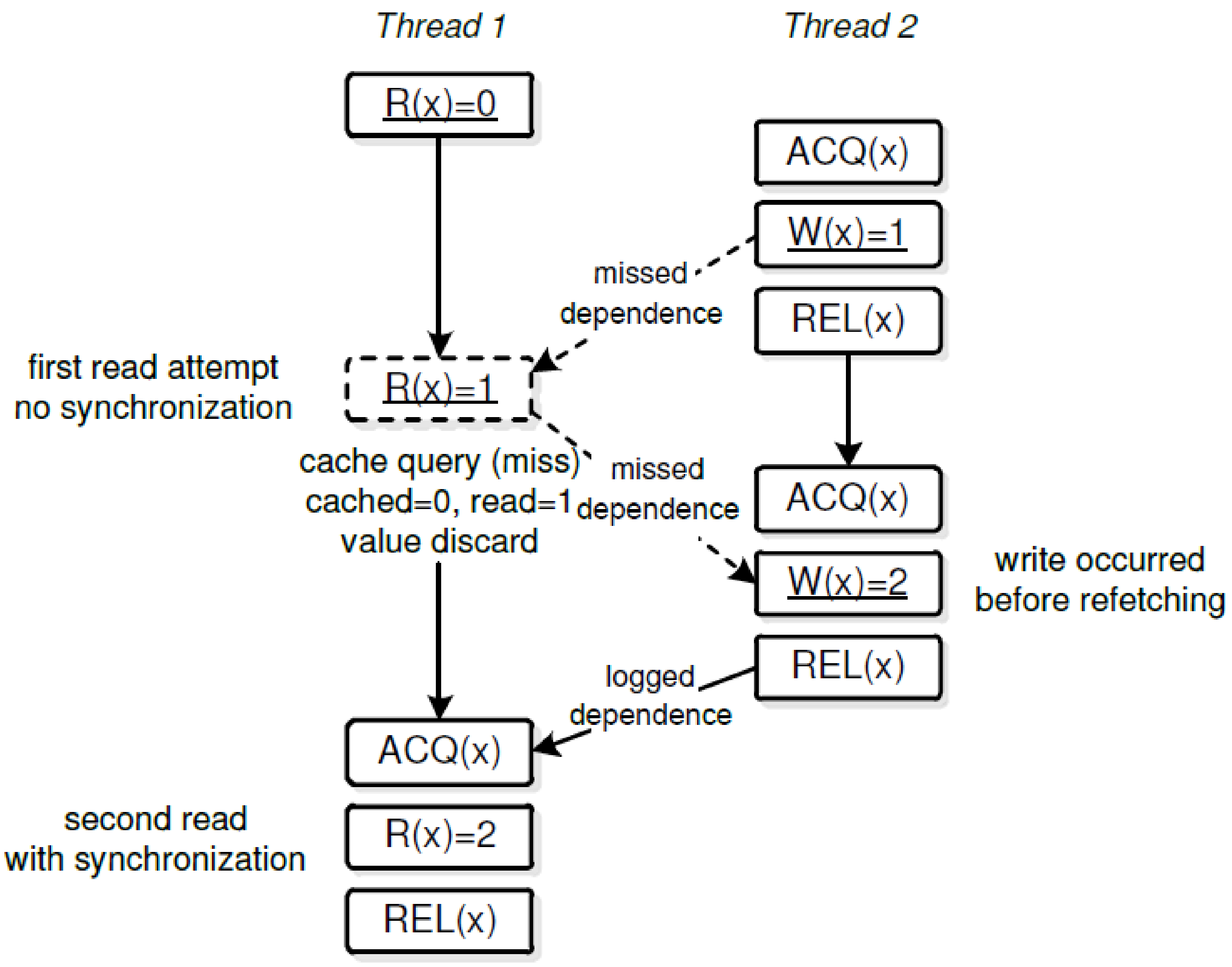


Figure 2: Illustration of refetching

Algorithm 1: read

```
d ← heap(v)
if cache(v) ≠ d then
    synchronized v
        d ← heap(v)
        H ← H ∪ (last(v),r)
        G ← G ∪ {r}
        last(v) ← r
    cache(v) ← d
```

G: set of read actions with cache miss

H: inter thread dependences

r = <tid,read,v,uniqueId>

Algorithm 2: write

synchronized v
    heap(v) $\leftarrow$ d
    if last(v).t $\neq$ t **then**
        H $\leftarrow$ H $\cup$ (last(v),w)
        last(v) $\leftarrow$ w
cache(v) $\leftarrow$ d

G: set of read actions with cache miss

H: inter thread dependences

w = <t, write, v, uniqueId>

Algorithm 3: lock

acquire(v)

if last(v).t ≠ t **then**

   H ← H ∪ (last(v),l)

   last(v) ← w

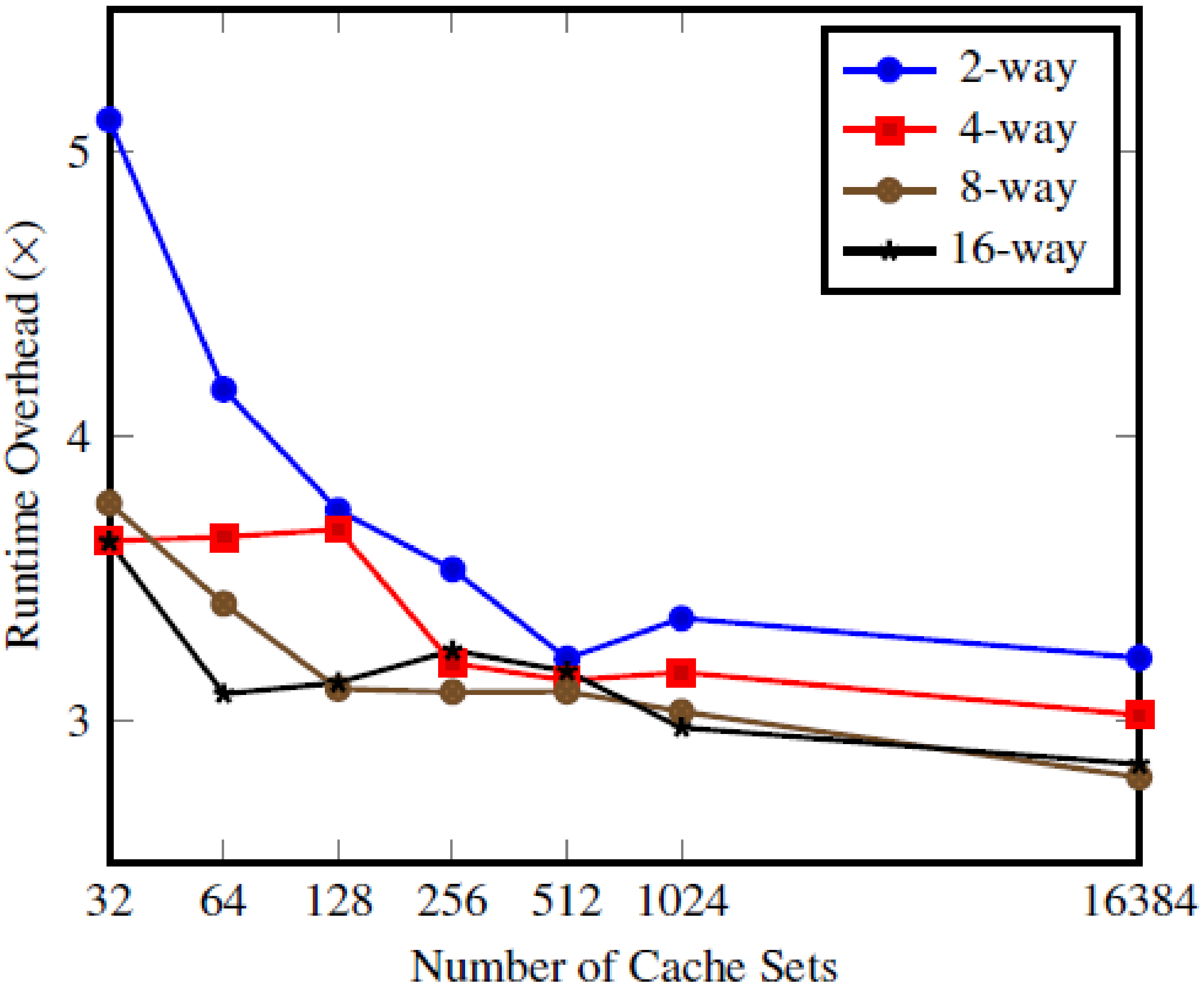
G: set of read actions with cache miss

H: inter thread dependences

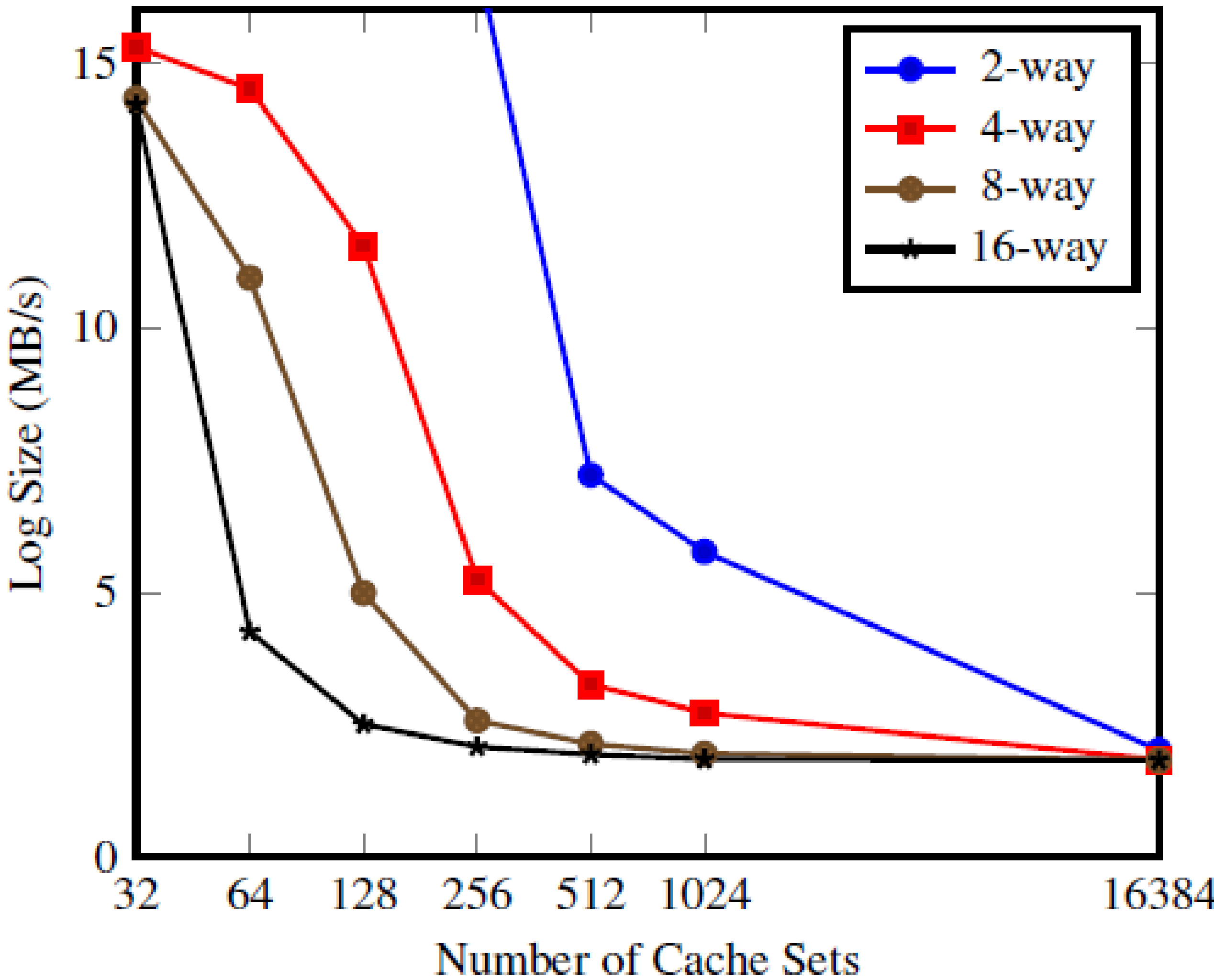
w = <t, acquire, v, uniqueId>

Cache Organization

- Big cache
  - Less unnecessary cache misses → smaller log
  - Disables the garbage collection mechanism → drains memory
- Small cache
  - Unnecessary cache misses
- Optimal cache
  - Efficient updates and queries
  - Moderate memory consumption
  - High cache hit rate



(a) *Study of runtime overhead against cache organization*



(b) *Study of log size against cache organization*

Heuristics to still get SC replay

- Try to schedule read actions first
  - If desired value is inconsistent with the one in the heap suspend thread
  - Immediately after desired value is written resume
- Add sequence number to groups of variables
  - Sequence numbers define dependences between variables

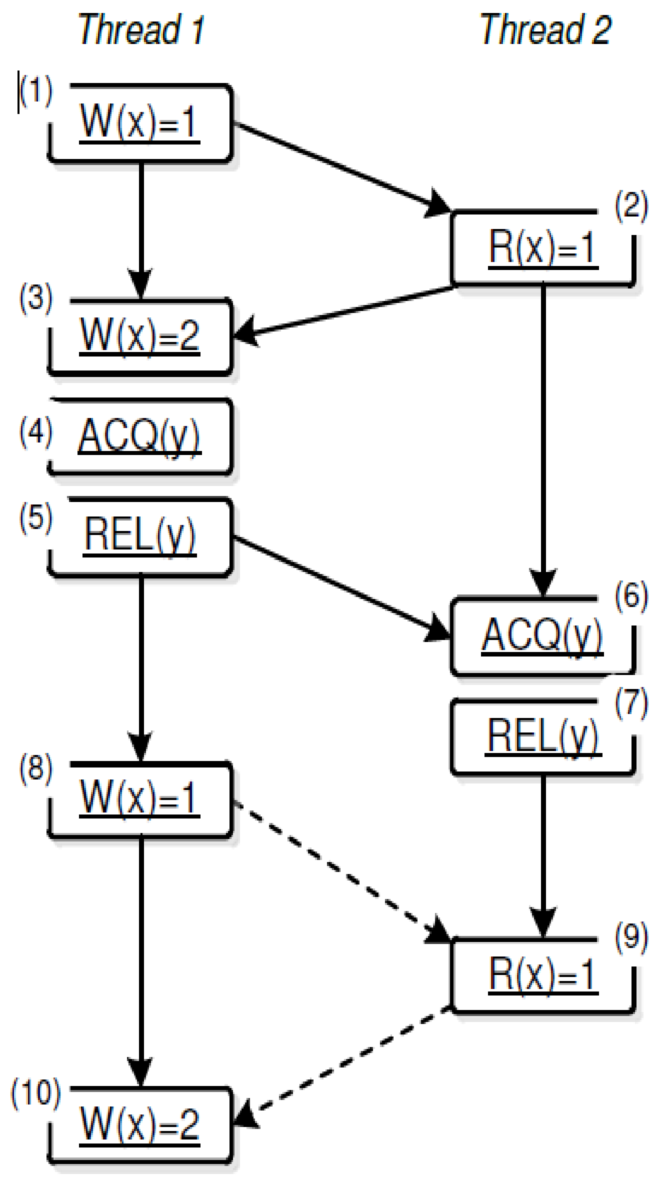


Figure 1: Illustration of missing dependences

Performance

Table 1: Comparison of CARE and LEAP under benchmark programs

Benchmark	CARE				LEAP [18]	
	Overhead (×)	*Log Size (/s)*	*Unordered (#)*	*Resolved (?)*	*Overhead (×)*	*Log Size (/s)*
Avrora	1.52	2.18MB	23K	Y	9.48	24.3MB
Batik	1.49	1.51KB	0	Y	3.77	2.32KB
H2	18.5	24.2MB	0	Y	62.8	27.4MB
Lusearch	3.41	6.53MB	0	Y	9.01	46.0MB
Sunflow	64.9	886MB	0	Y	389	6029MB
Tomcat	4.76	7.80MB	15	Y	11.9	23.5MB
Xalan	7.18	13.6MB	0	Y	12.2	143MB
Tsp	2.79	1.84MB	0	Y	111	570MB
Moldyn	11.9	24.1MB	0	Y	50.5	303MB

Performance

Table 2: Comparison of CARE and Stride with normalized values

Benchmark	CARE		Stride [37]	
	Overhead	*Log Size*	*Overhead*	*Log Size*
Avrora	16.0%	8.97%	54.0%	36.4%
Batik	39.5%	65.1%	50.0%	34.9%
H2	29.5%	88.3%	29.8%	23.9%
Lusearch	37.9%	14.2%	34.7%	30.0%
Sunflow	16.7%	14.7%	38.5%	9.17%
Tomcat	40.0%	33.2%	64.3%	34.6%
Xalan	59.2%	9.52%	19.0%	23.1%
Tsp	2.51%	0.32%	9.36%	7.18%
Moldyn	23.8%	7.94%	1.32%	0.71%

Conclusion

- CARE records only inter-thread dependencies
- Takes use of cache: cache miss = dependency
- Good performance: small log, low overhead