

Runtime Validation of Memory Ordering Using Constraint Graph Checking

Kaiyu Chen[†], Sharad Malik[†] and Priyadarsan Patra[‡]

[†] Dept. of Electrical Engineering, Princeton University

[‡] Validation Research Lab, Intel Corporation

International Symposium on High-
Performance Computer Architecture

February 20th, 2008

Memory Ordering Problem

- Multiprocessor execution result is dependent on the relative order of concurrent memory operations
 - Program example

```
Processor-1  
  
...  
(1.1) A = 1;  
(1.2) if (B == 0)  
  {  
    // critical section  
    ...  
  }
```

```
Processor-2  
  
...  
(2.1) B = 1;  
(2.2) if (A == 0)  
  {  
    // critical section  
    ...  
  }
```

Memory Ordering Problem

- Multiprocessor execution result is dependent on the relative order of concurrent memory operations
 - Program example

Processor-1

```
...  
(1.1) A = 1;  
(1.2) if (B == 0)  
  {  
    // critical section  
  }  
...
```

Processor-2

```
...  
(2.1) B = 1;  
(2.2) if (A == 0)
```

Possible results of read (A, B):

Legal:

(1.1)→(1.2)→(2.1)→(2.2): (1, 0)

Memory Ordering Problem

- Multiprocessor execution result is dependent on the relative order of concurrent memory operations
 - Program example

Processor-1

```
...  
(1.1) A = 1;  
(1.2) if (B == 0)  
{  
    // critical section  
...
```

Processor-2

```
...  
(2.1) B = 1;  
(2.2) if (A == 0)  
{  
    // critical section  
...
```

Possible results of read (A, B):

Illegal:

(1.2) → (2.1) → (2.2) → (1.1): (0, 0)

Memory Ordering Problem

- Multiprocessor execution result is dependent on the relative order of concurrent memory operations
 - Program example

```
Processor-1
...
(1.1) A = 1;
(1.2) if (B == 0)
{
    // critical section
}
```

```
Processor-2
...
(2.1) B = 1;
(2.2) if (A == 0)
{
    // critical section
}
```

```
Possible results of read (A, B):

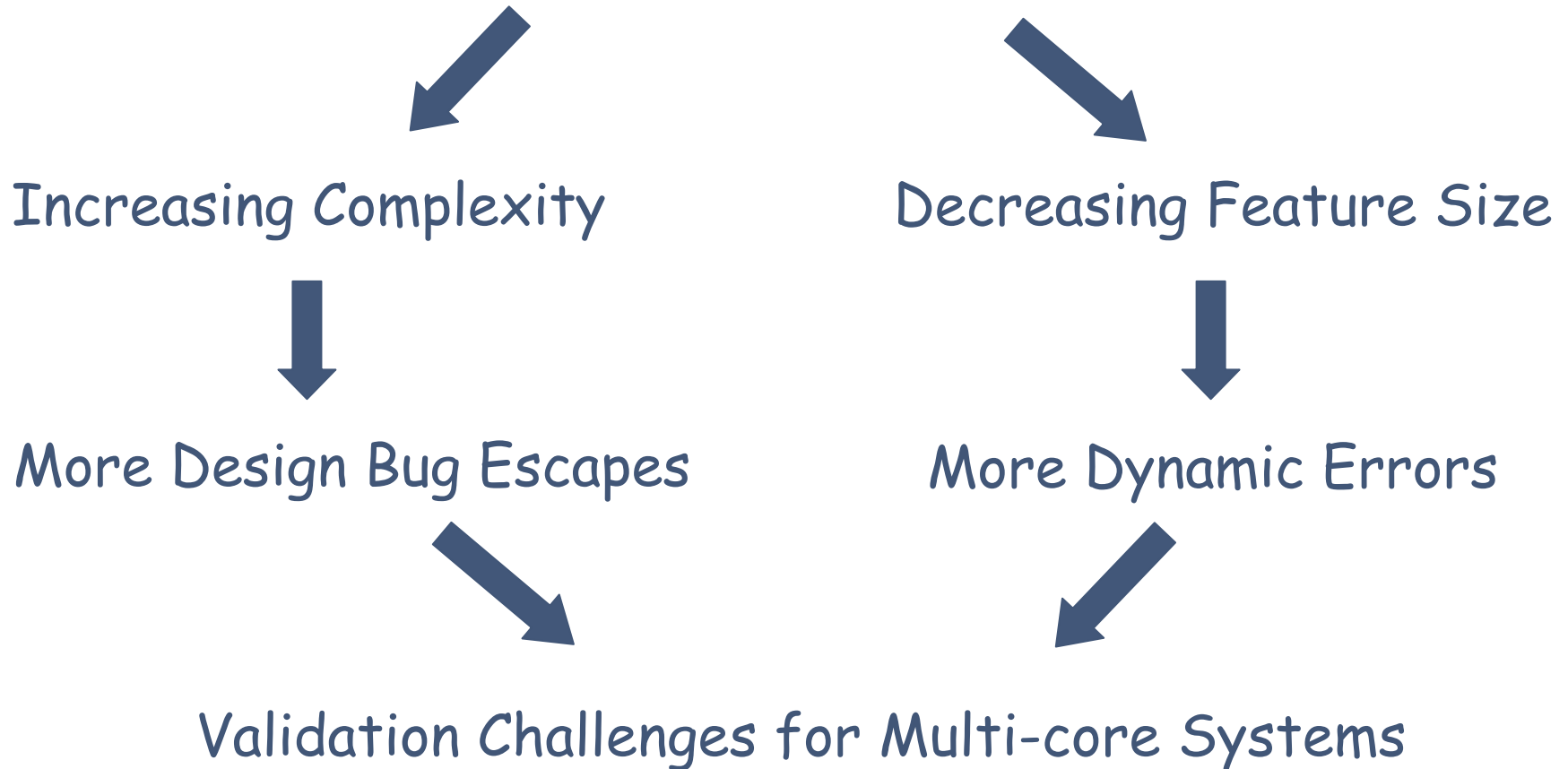
Legal:
(1.1)→(1.2)→(2.1)→(2.2): (1, 0)

Illegal:
(1.2)→(2.1)→(2.2)→(1.1): (0, 0)
```

- Single processor optimizations may break global consistency
- Causes of memory ordering errors
 - Design bugs/dynamic errors in numerous components
 - Processor pipeline, cache/memory controller, interconnection network ...

Motivation

Technology Scaling



Presentation Outline

- Motivation
- Runtime Validation Methodology
 - Validation Model
 - Basic Solution
 - Design Optimization
- Evaluation
- Conclusions

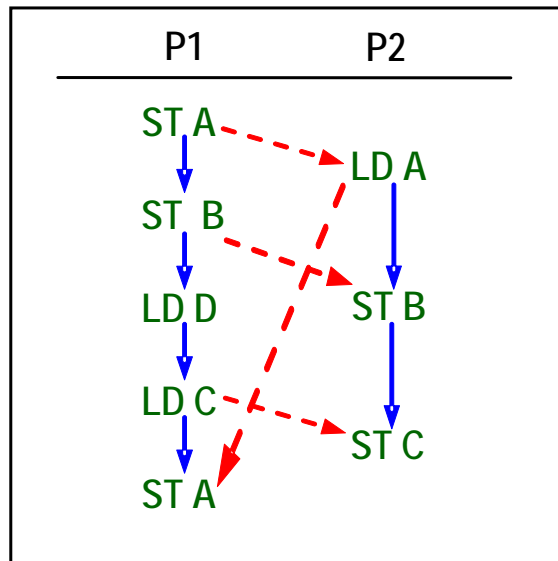
Memory Ordering Models

- Specification of correct shared-memory system behavior
 - Defines the legal ordering of the memory operations
 - The contract between the system designer and the programmer
- Common Memory Models [S. V. Adve *et al.*, IEEE Computer 1996]
 - Sequential Consistency (SC)
 - Requires a global sequential order among all memory operations
 - The program order must be maintained for each processor
 - Examples: SGI MIPS, HP PA-RISC
 - Total Store Ordering (TSO) and its variants
 - Relax the store to load order
 - Examples: Sun SPARC, Intel X86
 - Weak Ordering (WO)
 - Relax all memory orders but provide memory barrier instructions
 - Examples: DEC Alpha, IBM PowerPC, Intel Itanium

Constraint Graph Model

[D. Shasha *et al.*, TOPLAS'88] [H. W. Cain *et al.*, PACT'03]

- Definition: A directed graph that models memory ordering constraints
 - **Vertices**: dynamic memory instruction instances
 - **Edges**:



Constraint Graph under
Sequential Consistency

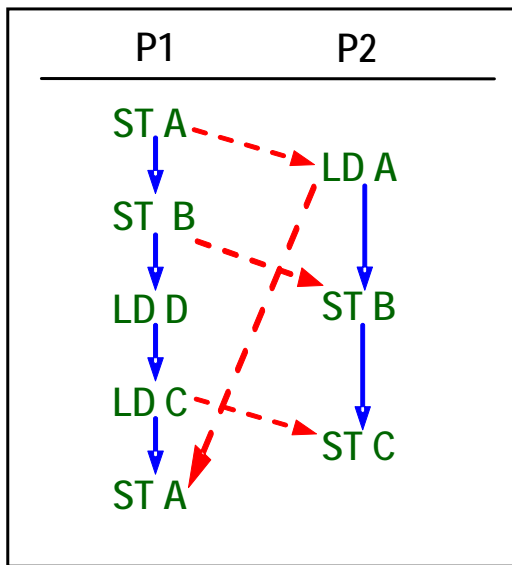
- **Dependence edges**

- Between conflicting instructions on the same memory location
- Capture the relative ordering at runtime

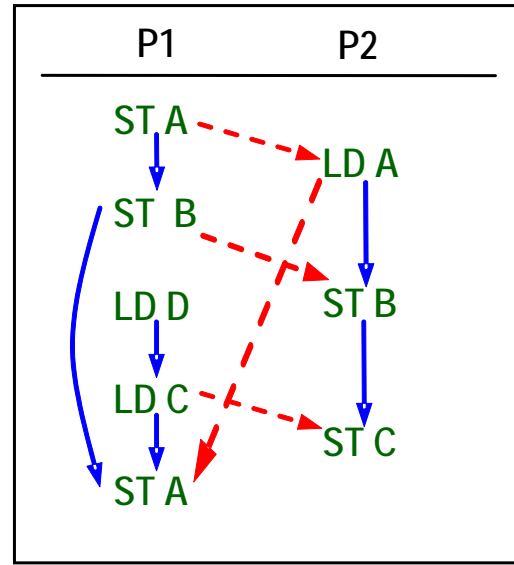
Constraint Graph Model

[D. Shasha *et al.*, TOPLAS'88] [H. W. Cain *et al.*, PACT'03]

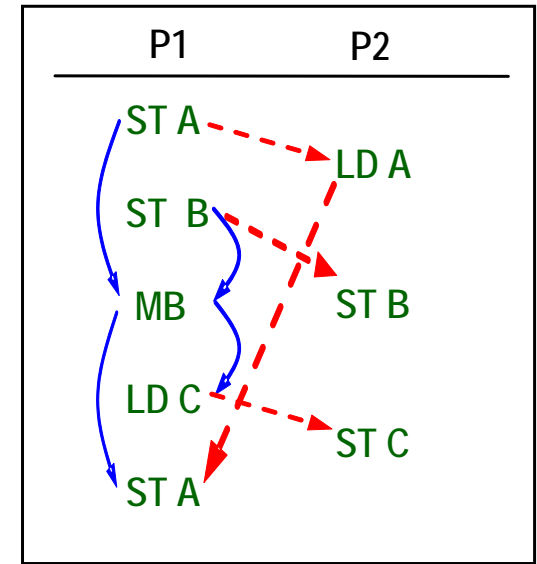
- Definition: A directed graph that models memory ordering constraints
 - **Vertices**: dynamic memory instruction instances
 - **Edges**:
 - **Consistency edges**
 - **Dependence edges**



Sequential Consistency



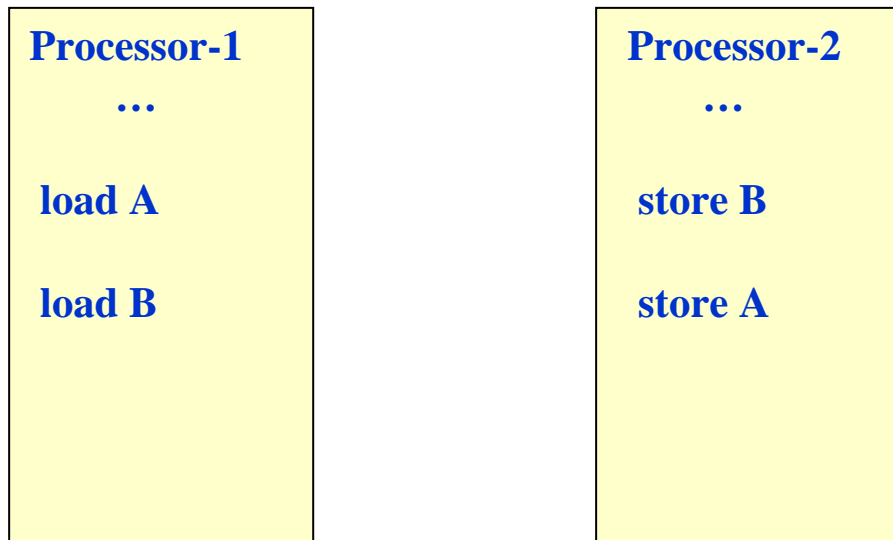
Total Store Ordering



Weak Ordering

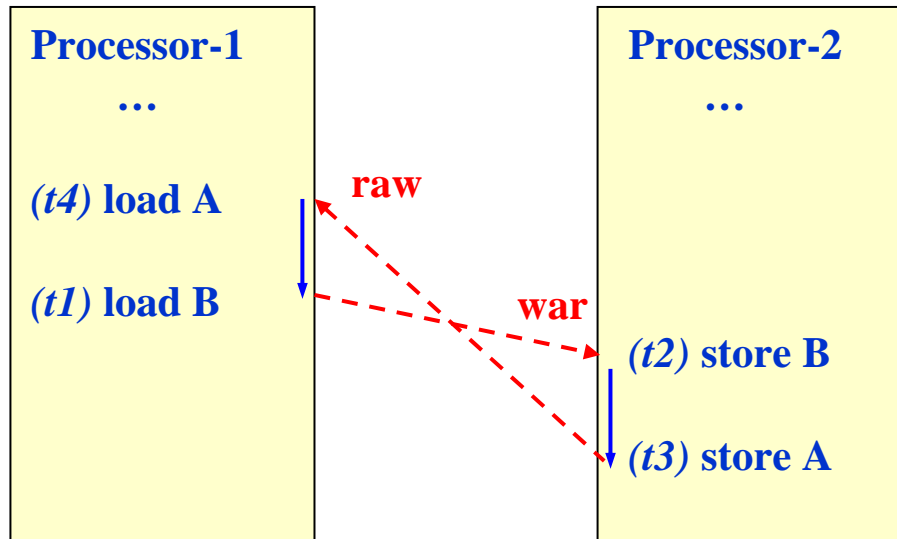
Constraint Graph Property

- **Correctness property:** A cycle in the graph indicates a memory ordering violation
 - Program example (assuming SC model):
 - Possible cycle created by reordering of memory operations



Constraint Graph Cycle Example

- Runtime execution sequence
 - Initially P1 has local copies of A and B
 - (t1) "load A" is stalled on address calculation,
P1 performs "load B" and reads the local copy
 - (t2) P2 performs "store B" and overwrites the copy in P1
 - (t3) P2 performs "store A" and overwrites the copy in P1
 - (t4) P1 performs "load A" and reads the new value modified by P2



Offline Verification Approach

- Previous work based on graph checking
 - Successfully used in industry
 - Alpha's Weak Ordering (WO) model [S. A. Taylor *et al.*, ICCAD'01]
 - Sun's Total Store Ordering (TSO) model [S. Hangal *et al.*, ISCA'04]

Offline Verification Approach

- Previous work based on graph checking
 - Successfully used in industry
 - Alpha's Weak Ordering (WO) model [S. A. Taylor *et al.*, ICCAD'01]
 - Sun's Total Store Ordering (TSO) model [S. Hangal *et al.*, ISCA'04]
 - Basic steps
 - Run short test programs that exercise the target memory system
 - Build the constraint graph for the execution trace.
 - Perform offline analysis on the graph to verify the acyclic property

Offline Verification Approach

- Previous work based on graph checking
 - Successfully used in industry
 - Alpha's Weak Ordering (WO) model [S. A. Taylor *et al.*, ICCAD'01]
 - Sun's Total Store Ordering (TSO) model [S. Hangal *et al.*, ISCA'04]
 - Basic steps
 - Run short test programs that exercise the target memory system
 - Build the constraint graph for the execution trace.
 - Perform offline analysis on the graph to verify the acyclic property
 - Limitations and inefficiency
 - Limited coverage of corner cases
 - Not scalable to large programs
 - Ineffective for soft errors and aging induced faults
 - Rely on data coloring to establish mapping from load to store
 - Expensive recursive operation to infer other necessary edges

Proposed Online Verification Approach

- Monitor and validate correct system behavior at runtime
 - Hardware checking of the constraint graph on-the-fly
 - End-to-end error detection
 - Effective against escaped design bugs and dynamic errors

Proposed Online Verification Approach

- Monitor and validate correct system behavior at runtime
 - Hardware checking of the constraint graph on-the-fly
 - End-to-end error detection
 - Effective against escaped design bugs and dynamic errors
- **Basic Steps**
 - Local observation of graph edges
 - Global checking of the constraint graph
 - Global recovery if a cycle is detected

Design Assumptions

- Uniprocessor correctness
 - Correct intra-processor computation
 - Ensured by existing design verification techniques or runtime validation techniques (e.g., DIVA) [T. M. Austin, MICRO'99]

Design Assumptions

- Uniprocessor correctness
 - Correct intra-processor computation
 - Ensured by existing design verification techniques or runtime validation techniques (e.g., DIVA) [T. M. Austin, MICRO'99]
- Cache coherence
 - General system implementation with hardware cache coherence support
 - Well-addressed in existing verification work
 - Simulation and formal verification based solutions [D. J. Sorin *et al.*, DSN'03]
 - Recent dynamic verification techniques [A. Meixner *et al.*, HPCA'07]

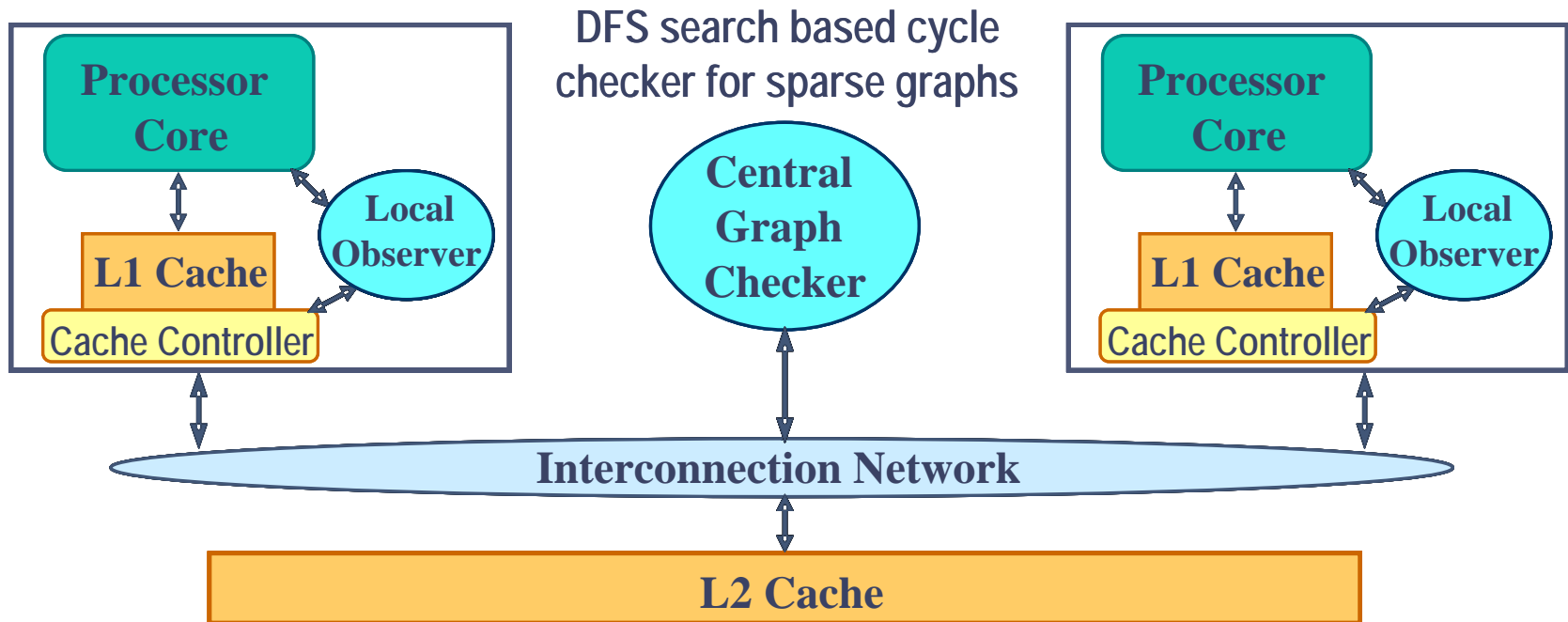
Design Assumptions

- Uniprocessor correctness
 - Correct intra-processor computation
 - Ensured by existing design verification techniques or runtime validation techniques (e.g., DIVA)
- Cache coherence
 - General system implementation with invalidation-based hardware cache coherence support
 - Well-addressed in existing verification work
 - Simulation and formal verification based solutions
 - Recent dynamic verification techniques
- Storage protection
 - Use techniques such as ECC

Design Assumptions

- Uniprocessor correctness
 - Correct intra-processor computation
 - Ensured by existing design verification techniques or runtime validation techniques (e.g., DIVA)
- Cache coherence
 - General system implementation with invalidation-based hardware cache coherence support
 - Well-addressed in existing verification work
 - Simulation and formal verification based solutions
 - Recent dynamic verification techniques
- Storage protection
 - Use techniques such as ECC
- Verification message robustness
 - Exploit end-to-end protection or request time out schemes

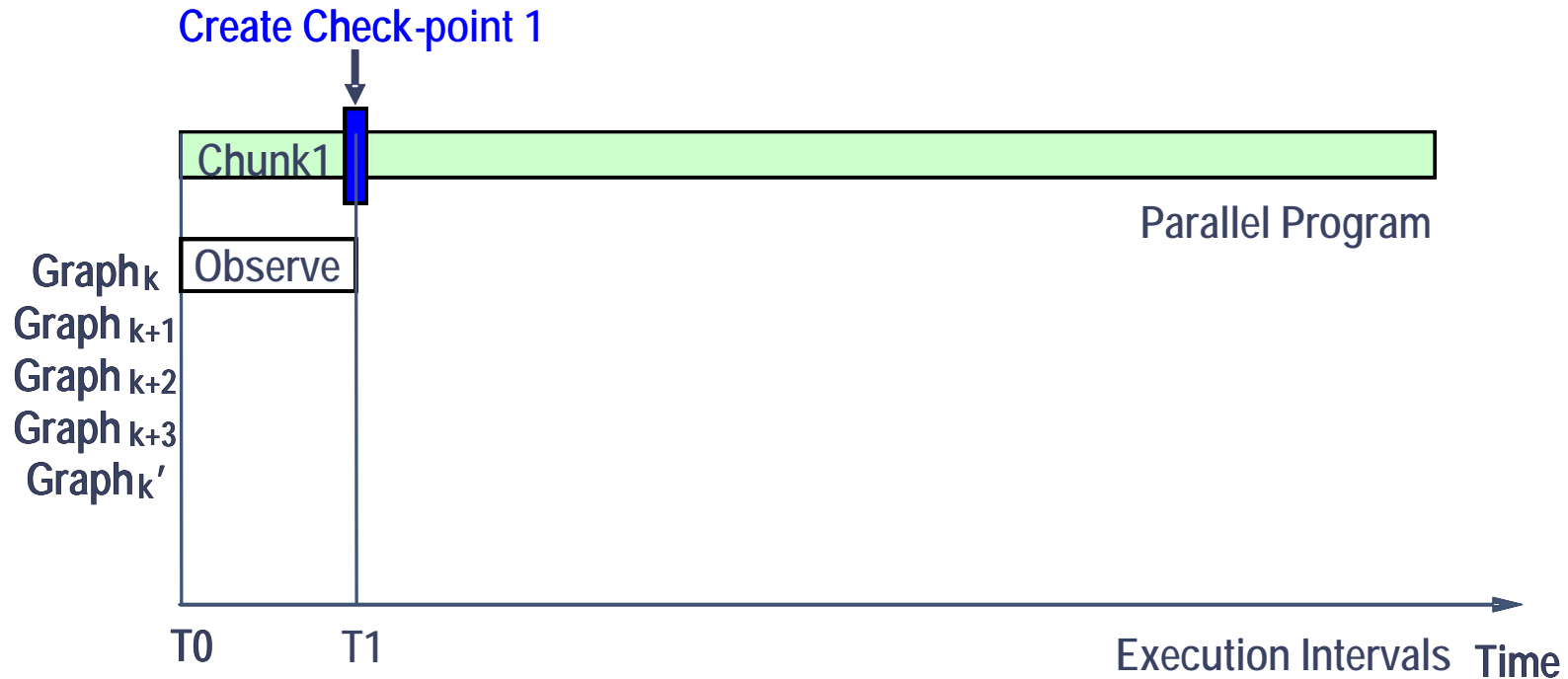
Runtime Validation Architecture Overview



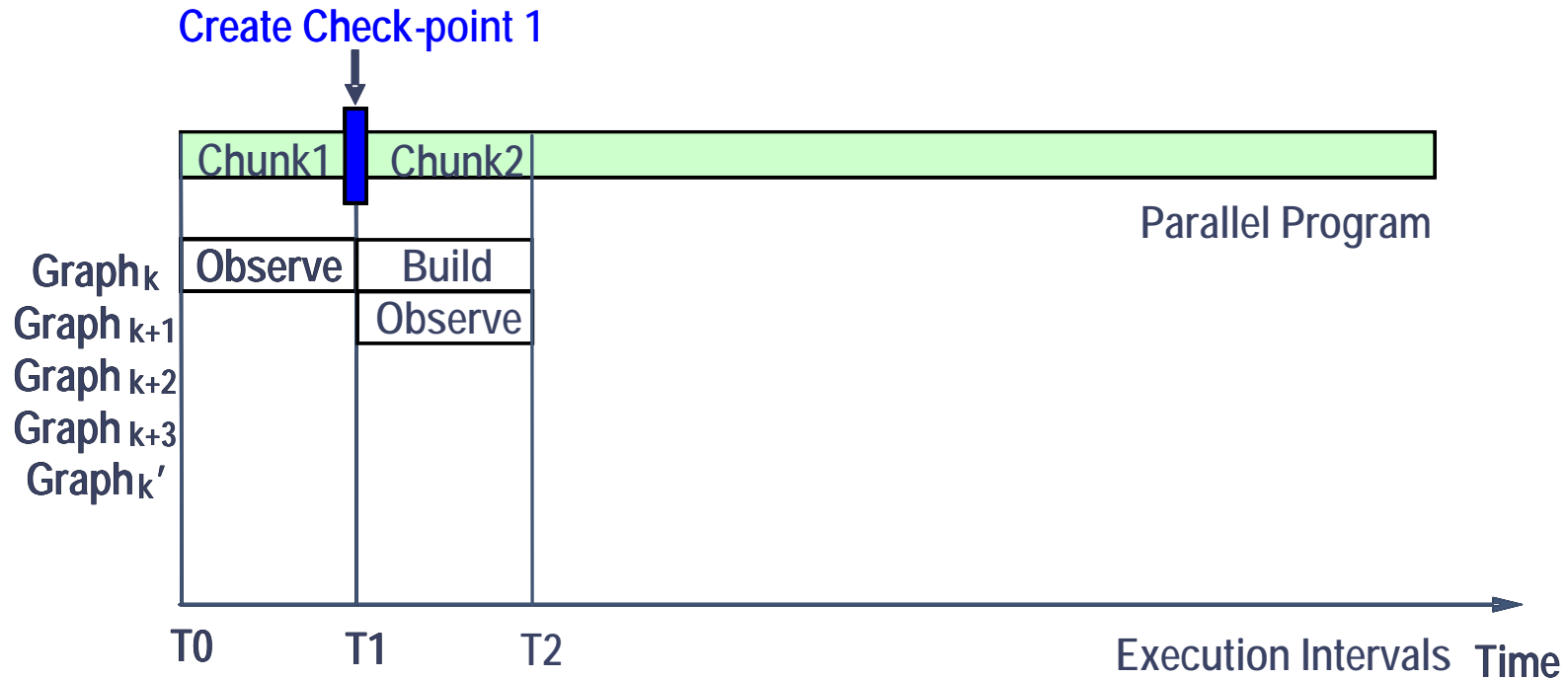
- **Local observer:**

- Local instruction ordering
- Build the global constraint graph
- Local access history
- Check for the acyclic property
- Locally observed inter-processor edges

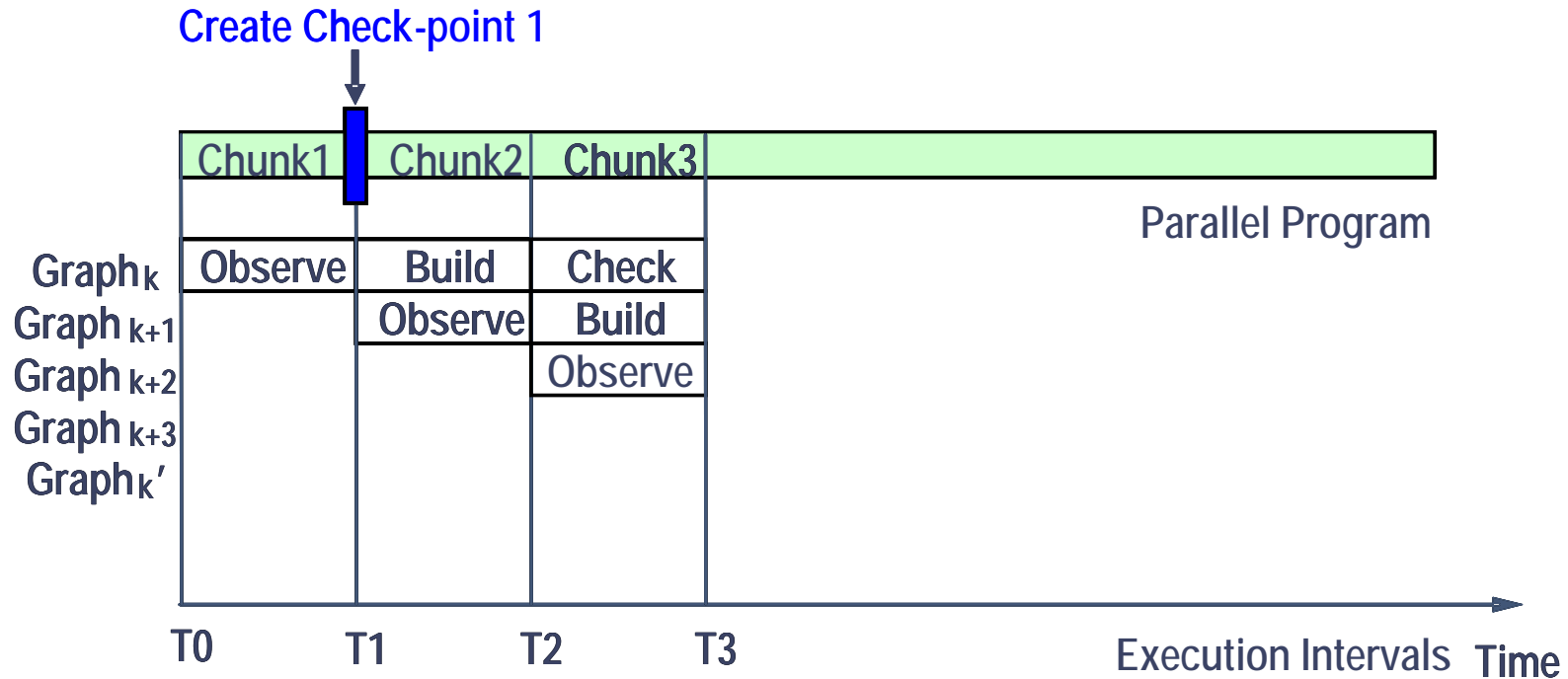
Runtime Validation Timing Diagram



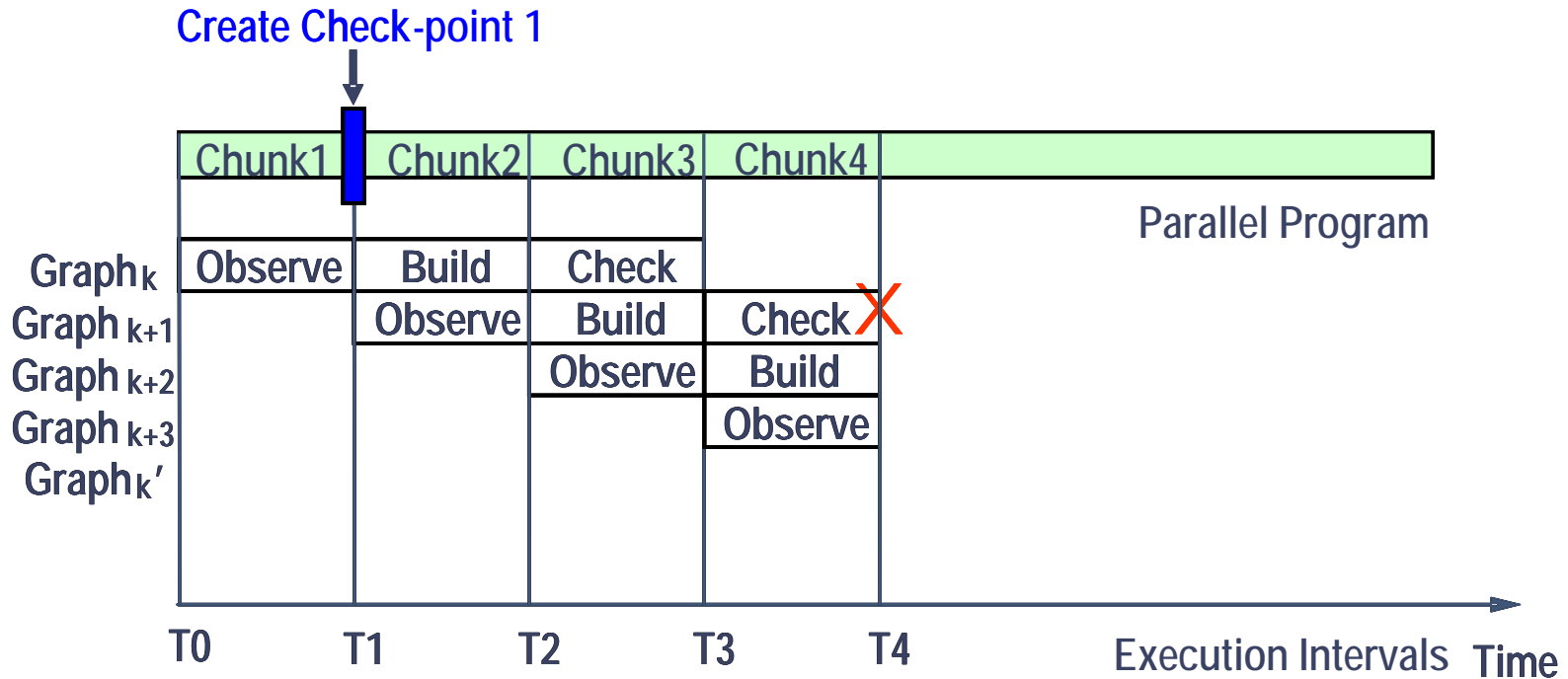
Runtime Validation Timing Diagram



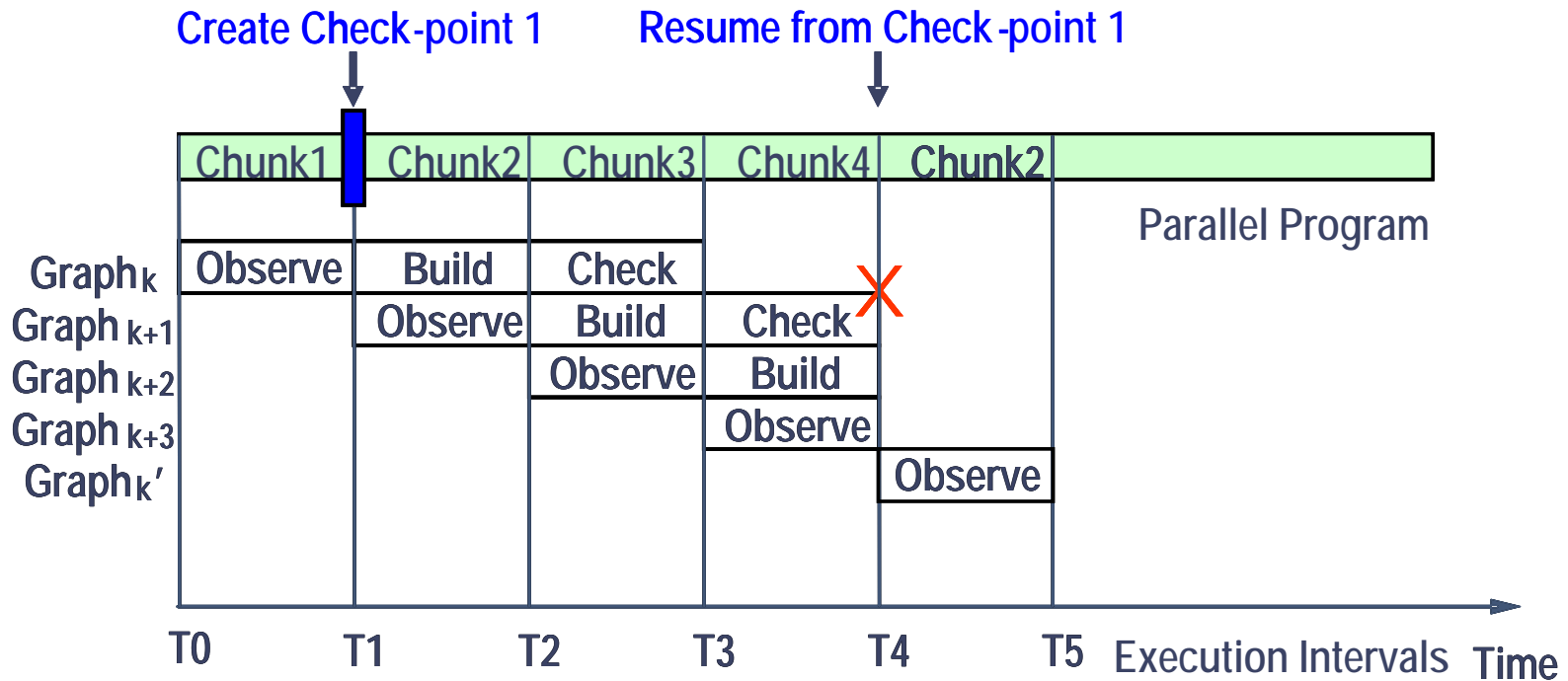
Runtime Validation Timing Diagram



Runtime Validation Timing Diagram



Runtime Validation Timing Diagram



Construction of Consistency Edges

- Constructed between memory instructions when dispatched in program order
 - Assign a monotonically increasing id to each dynamic instruction when it is dispatched
 - Tailored tracking scheme for different memory ordering rules
 - SC: edge for each adjacent pair of memory instructions
 - TSO: need to track the last load id and the last store id
 - Weak ordering: need to track last memory barrier id

P1	Inst_id
ST A	1
ST B	2
LDB	3
LDC	4
ST A	5

Sequential Consistency

P1	Inst_id
ST A	1
ST B	2
LDD	3
LDC	4
ST A	5

Total Store Ordering

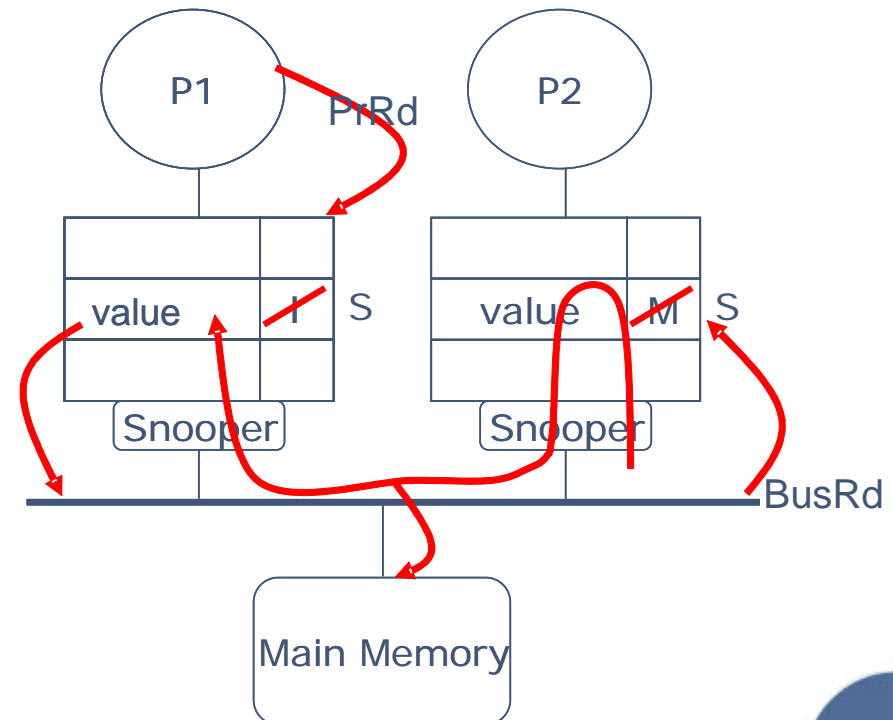
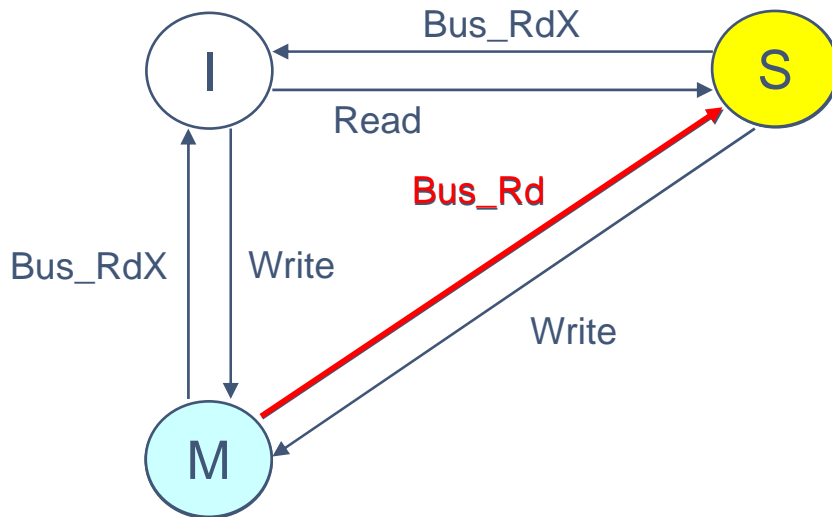
P1	Inst_id
MB	1
ST A	2
ST B	3
MB	4
LDC	5

Weak Ordering

Construction of Inter-processor Dependence Edges

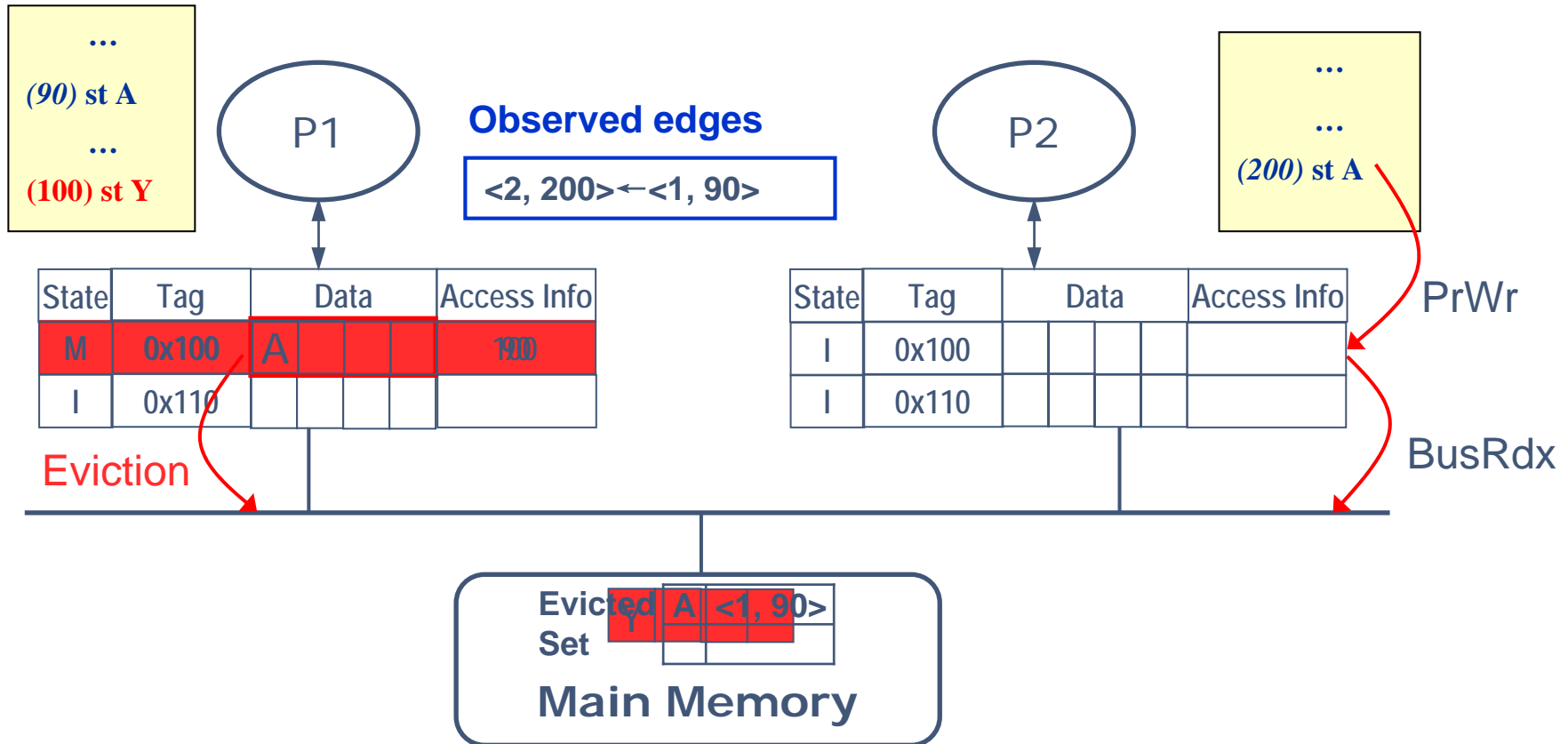
- Constructed between different processors when inter-processor communication is involved
 - Piggyback on cache coherence events to construct an edge

- RAW**



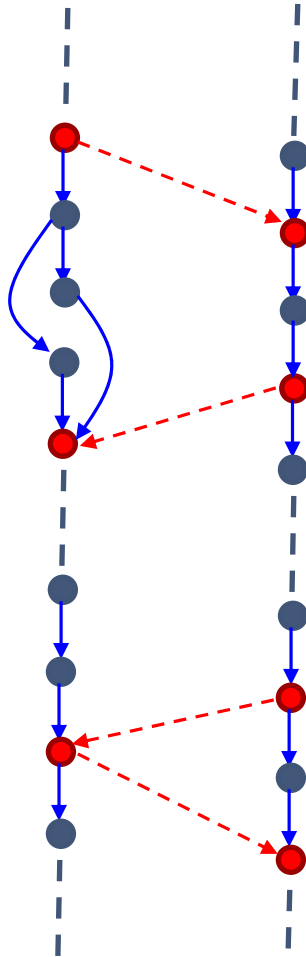
Further Design Issues

- Cache eviction:



- Add a small buffer to keep the evicted access records
- Employ effective filtering techniques to reduce the buffer size

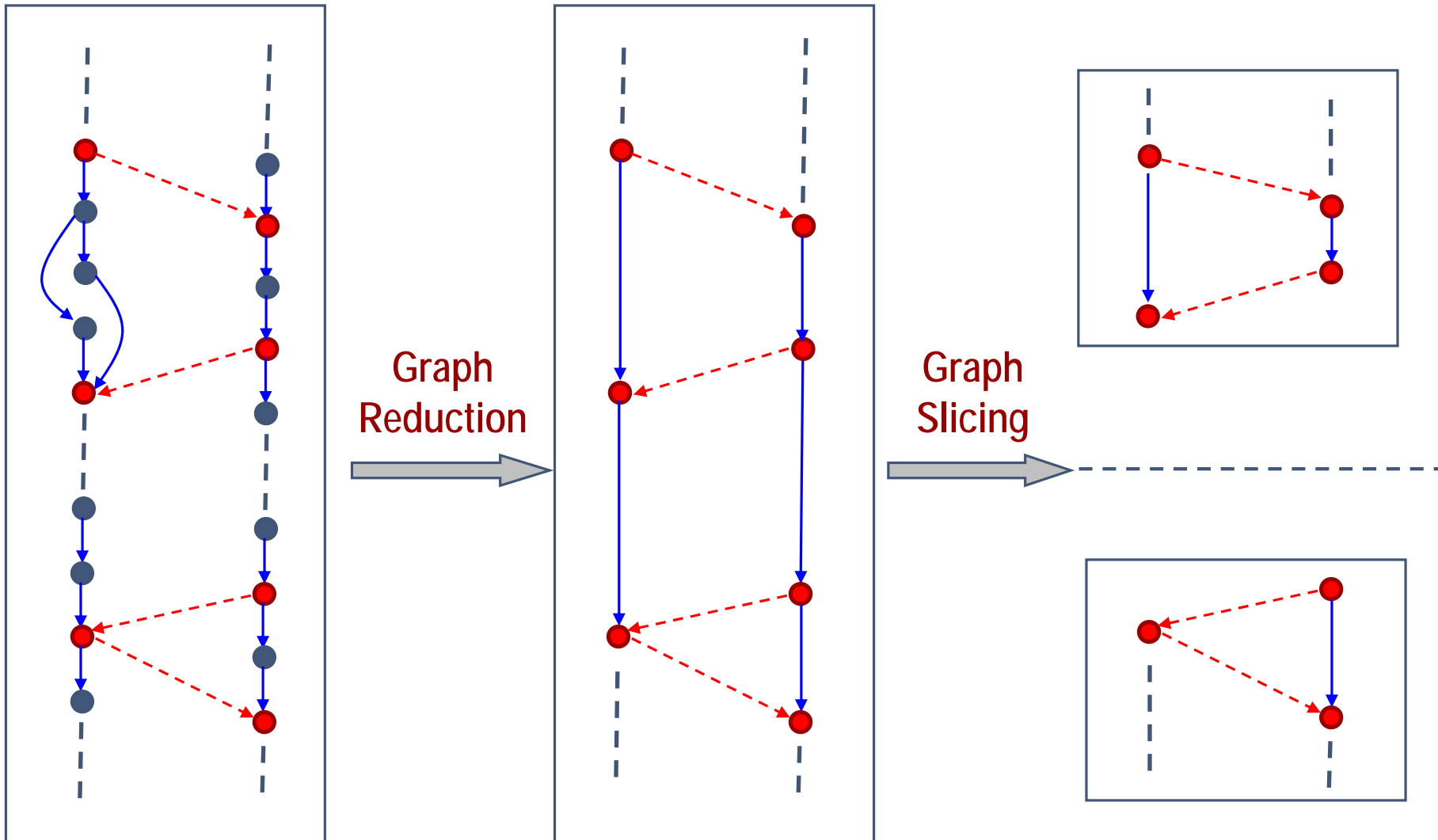
Practical Design Challenges



A naively built constraint graph that includes all executed memory instructions

- **Billions of vertices**
- **Unbounded graph size**

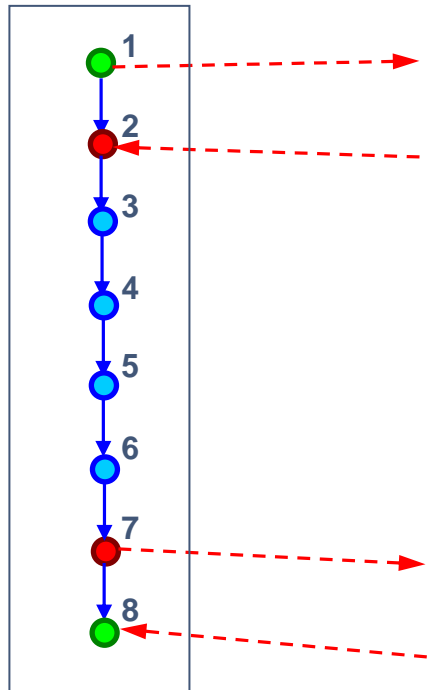
Key Enabling Techniques



Graph Reduction

- Use an equivalent but significantly reduced graph for detecting the existence of cycles
 - Characterization of SC constraint graph

P1



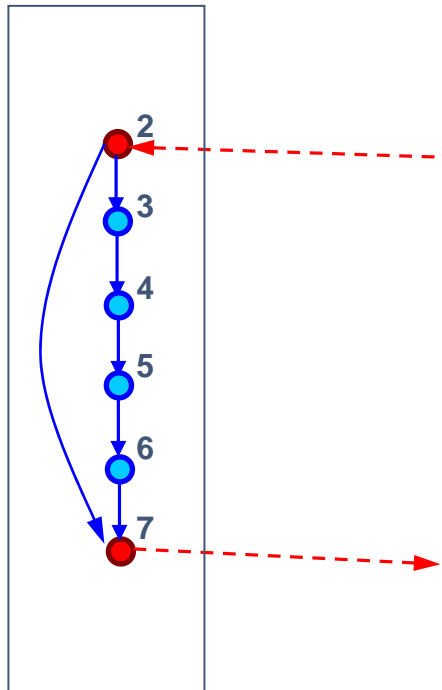
Observations

- Green vertices cannot be in a cycle
 - Intra-processor edges follow program order
 - A cycle must involve an incoming edge at the top and an outgoing edge at the bottom

Graph Reduction

- Use an equivalent but significantly reduced graph for detecting the existence of cycles
 - Characterization of SC constraint graph

P1



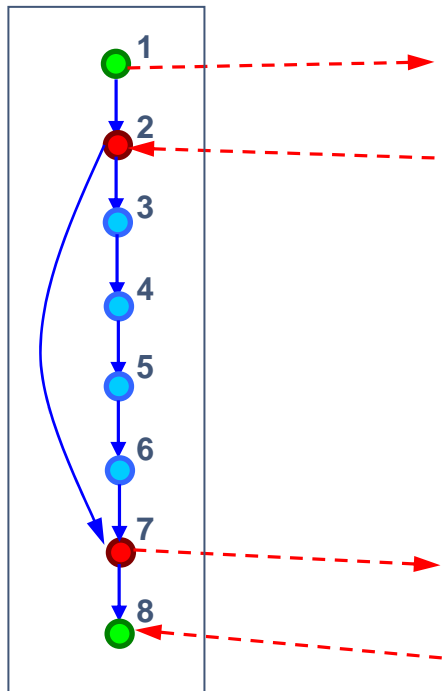
Observations

- Blue vertices can be compressed
 - No inter-processor edges
 - For the purpose of cycle detection, only need the transitive closure edge between instruction 2 and 7

Graph Reduction

- Use an equivalent but significantly reduced graph for detecting the existence of cycles
 - Characterization of SC constraint graph

P1



Reduction method

- Only track vertices with inter-processor dependence edges
- Infer the transitive closure of the intra-processor edges based on the graph pattern of different memory models

- Similar reduction for TSO and WO.

Graph Slicing

- Enable incremental validation of short execution intervals
 - Use a loosely-synchronized physical clock as the logical time base
 - 3-phase checking protocol
 - Program example:

Processor-1

...

(9) load C**(10) store D****(11) store B****(12) load A****Processor-2**

...

(10) store A**(11) load B****(12) Store E****(13) load D**

Graph Slicing

- Program example
 - Runtime execution order

Processor-1

...

(t1) (9) load C

(10) store D

(11) store B

(12) load A

Processor-2

...

(10) store A

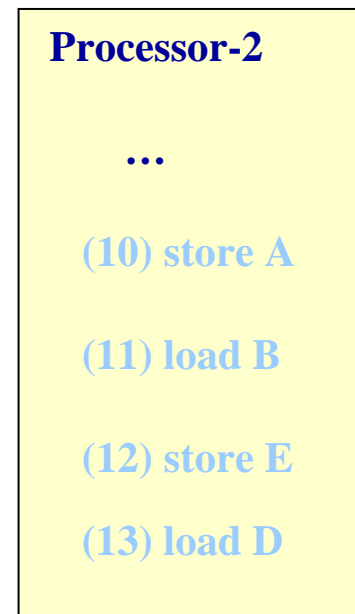
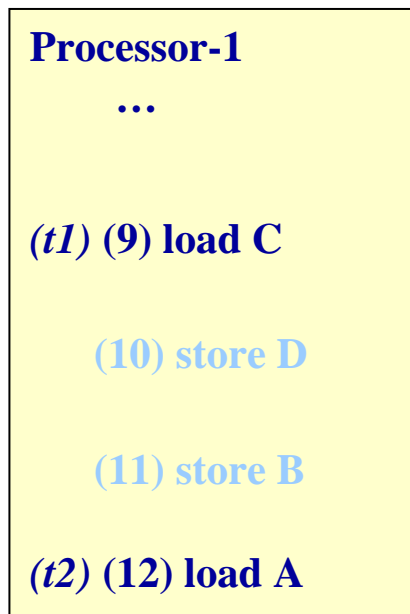
(11) load B

(12) store E

(13) load D

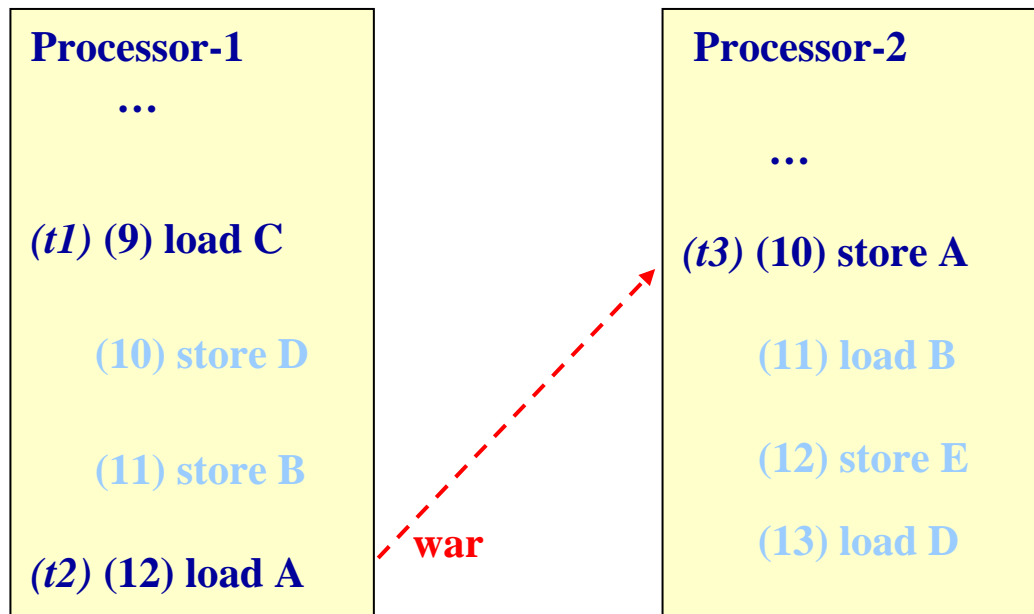
Graph Slicing

- Program example
 - Runtime execution order



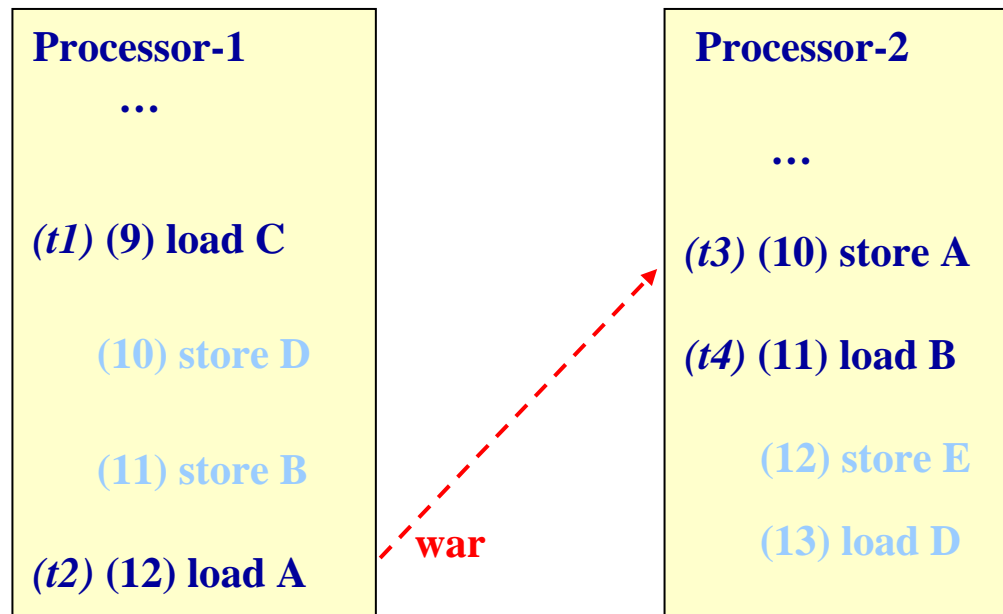
Graph Slicing

- Program example
 - Runtime execution order



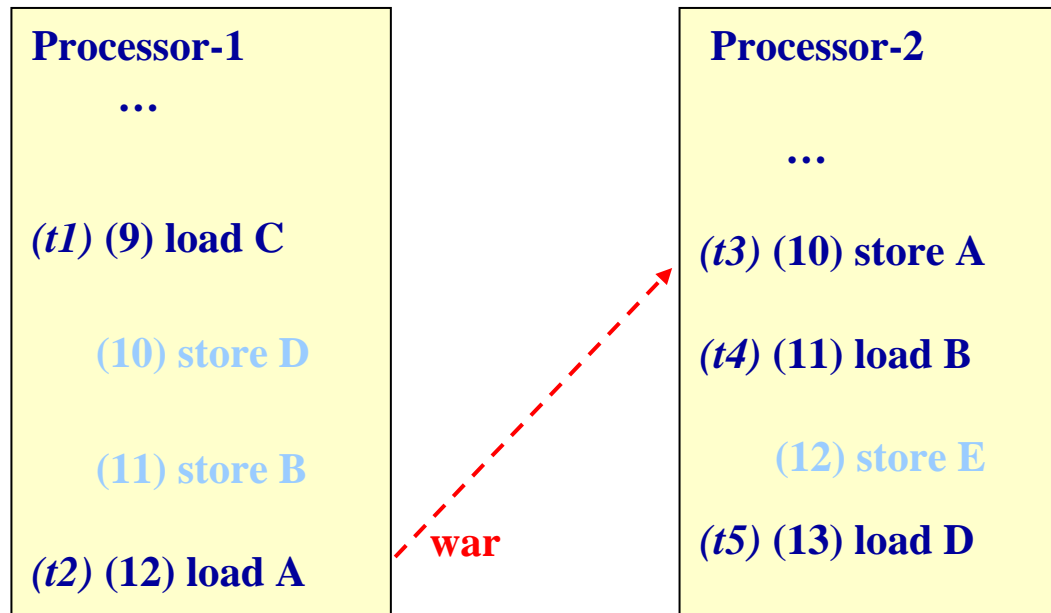
Graph Slicing

- Program example
 - Runtime execution order



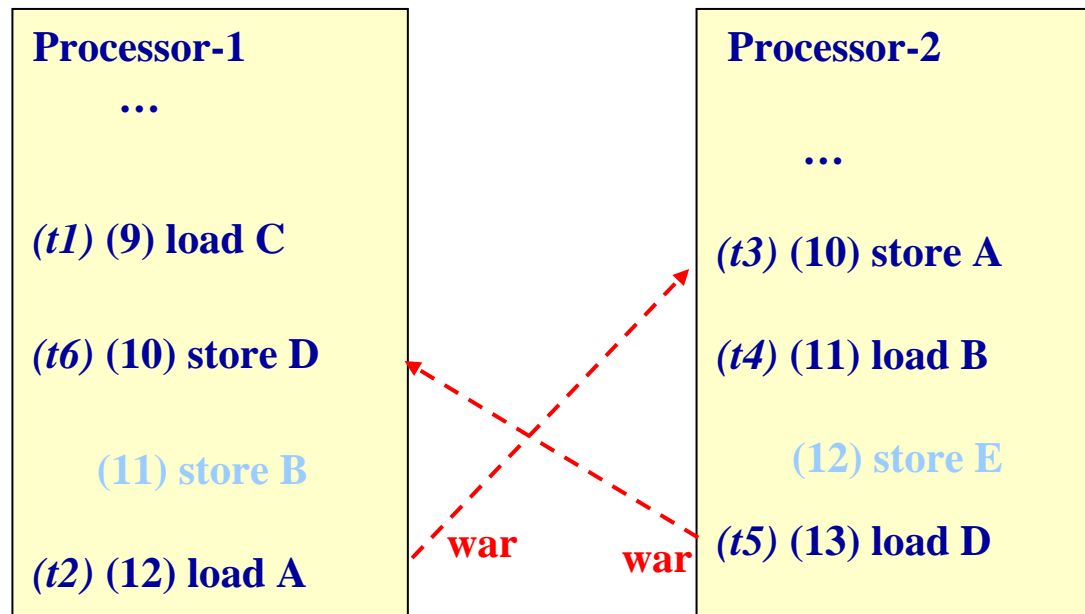
Graph Slicing

- Program example
 - Runtime execution order



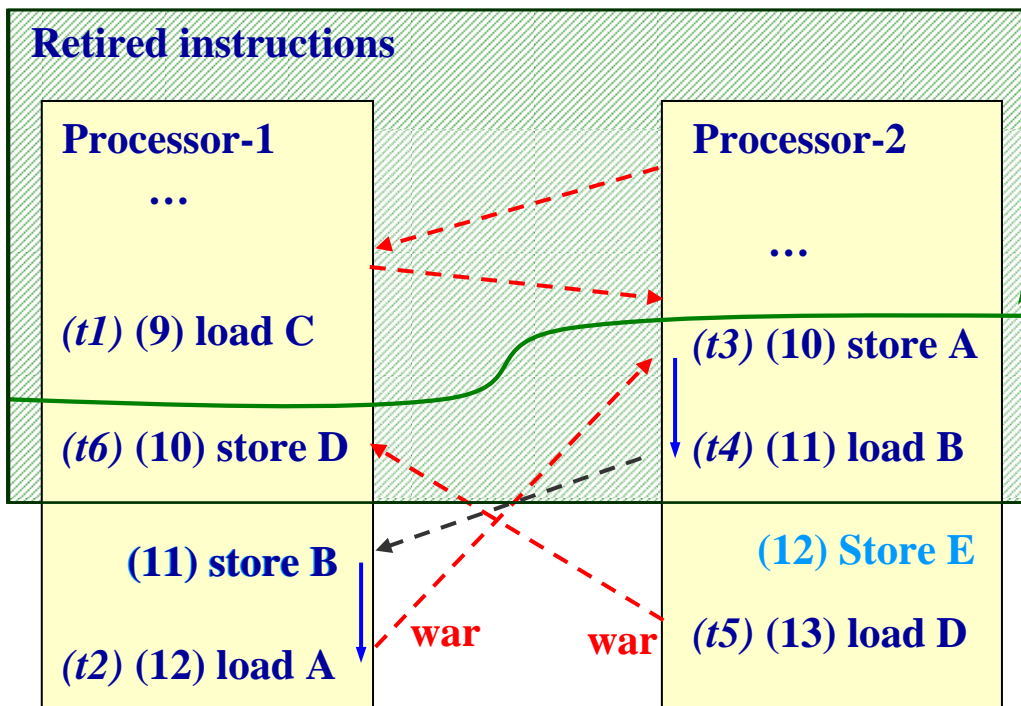
Graph Slicing

- Program example
 - Runtime execution order



Graph Slicing

- Program example (assuming SC model)
 - Execution status at the end of a validation interval T



Forward Causality Frontier

- **Definition:**

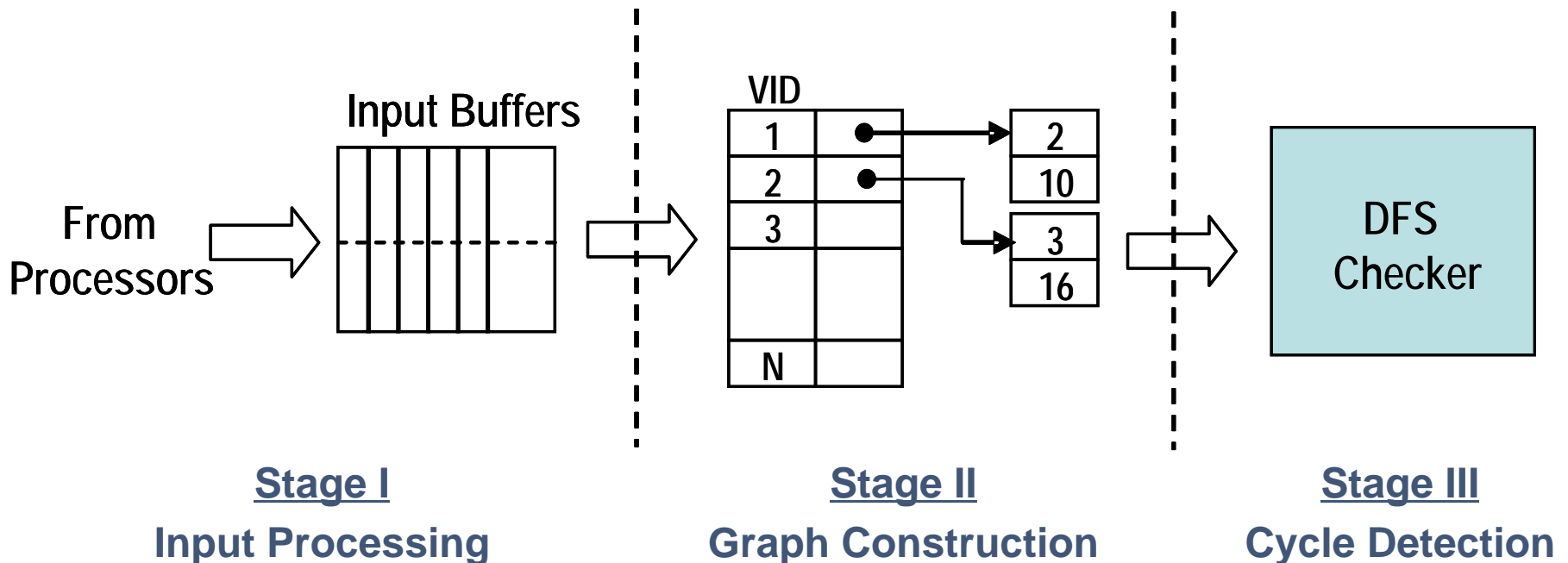
- Only forward edges cross the boundary
- Represents “happens-before” relation
- No cycle involves instructions both before and after the boundary
- E.g., Instruction (2, 10) must be excluded

- **Construction:**

- A fixed-point iterative scheme for computing the forward causality frontier

Graph Checker Design

- Exploit hardware accelerated graph processor
 - Experiments show that the constraint graphs are fairly sparse
 - Pipelined design of a dedicated hardware graph checking engine based on DFS search



Presentation Outline

- Motivation
- Runtime Validation Methodology
 - Validation Model
 - Basic Solution
 - Design Optimization
- Evaluation
- Conclusions

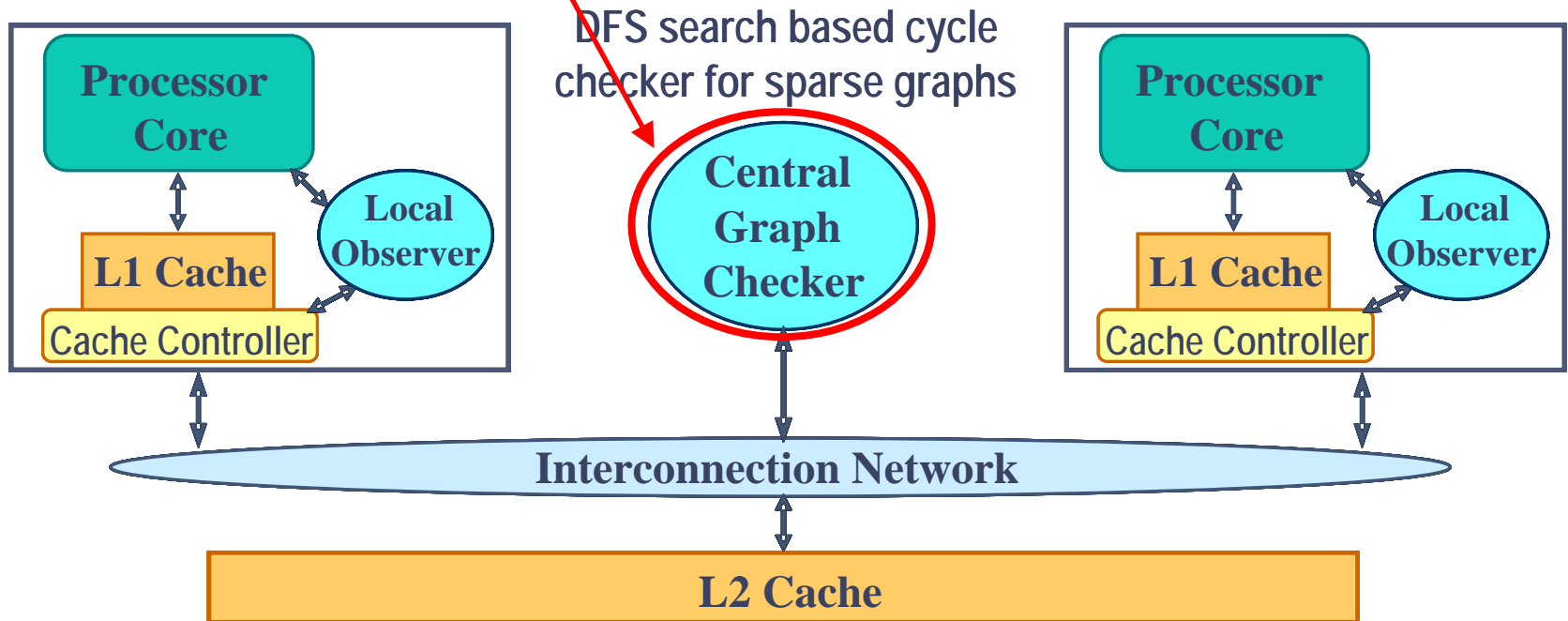
Experiment Setup

- Simulation infrastructure
 - Built on Multi-facet GEMS multiprocessor simulator
 - Dual-core CMP system configuration
 - Selected Splash-2 parallel benchmarks
- System configuration

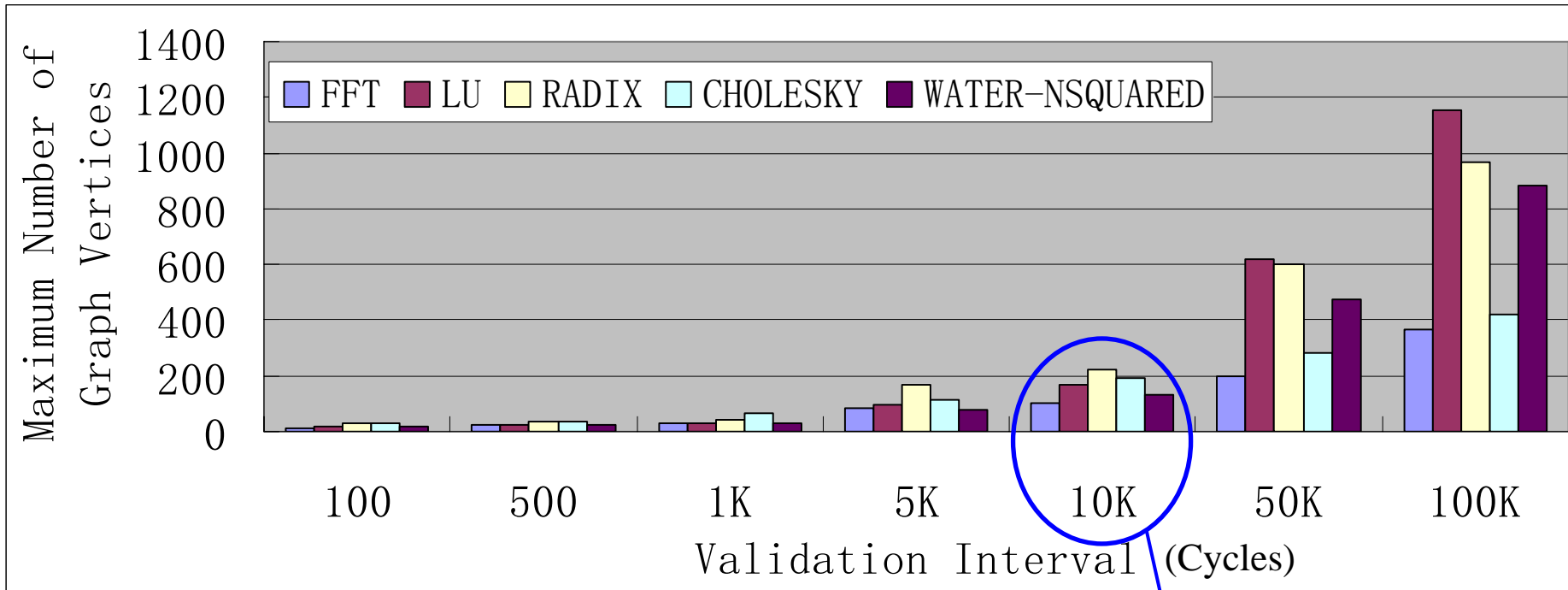
Target Architecture	Sun SPARC V9
Processor Configuration	4-way out-of-order processor core
L1 Cache (Private)	64KB, 4-way 64-byte blocks
L2 Cache (Shared)	4MB, 4-way 64-byte blocks
Memory	1G bytes
Cache Coherence Protocol	MSI_MOSI_CMP_Directory
Interconnection Network	PT_TO_PT

Experiment Evaluation

- **Graph Size**



Constraint Graph Evaluation

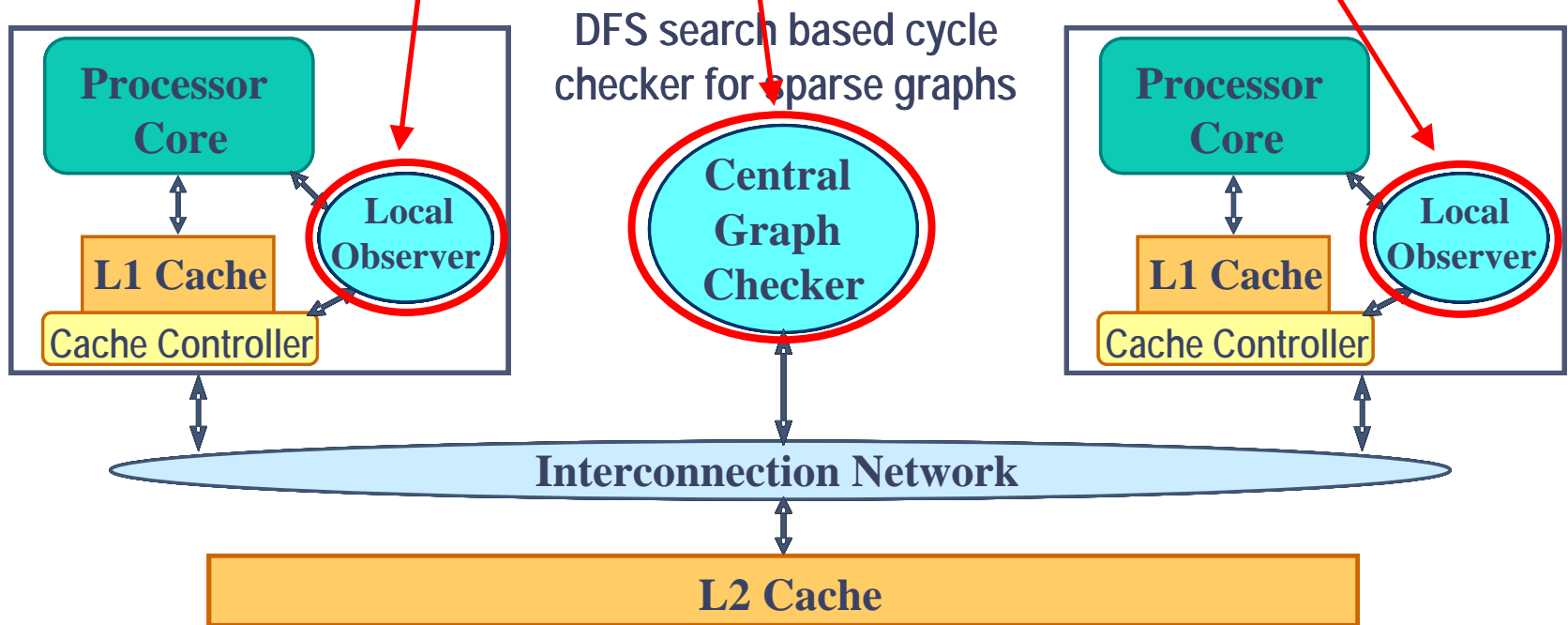


- **Small graph size**

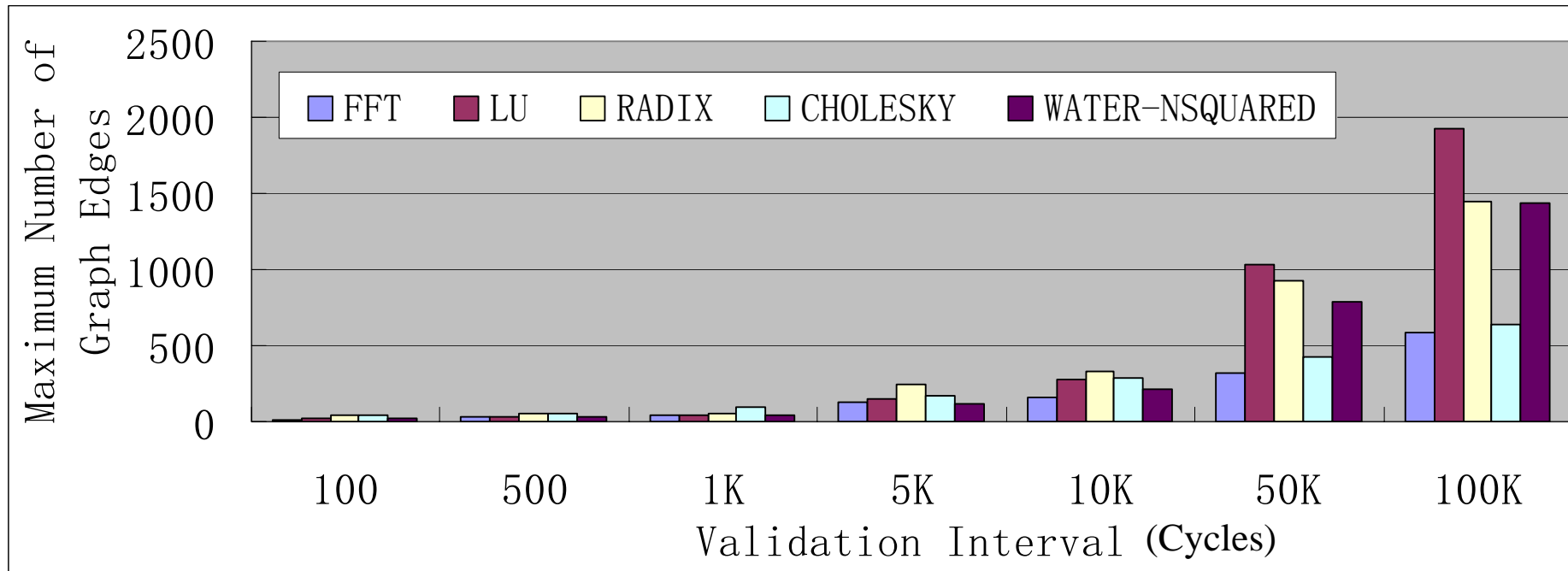
Sweet-spot in the trade-off between validation latency and hardware overhead

Experiment Evaluation

- Locally observed edge record size
- Graph checking complexity

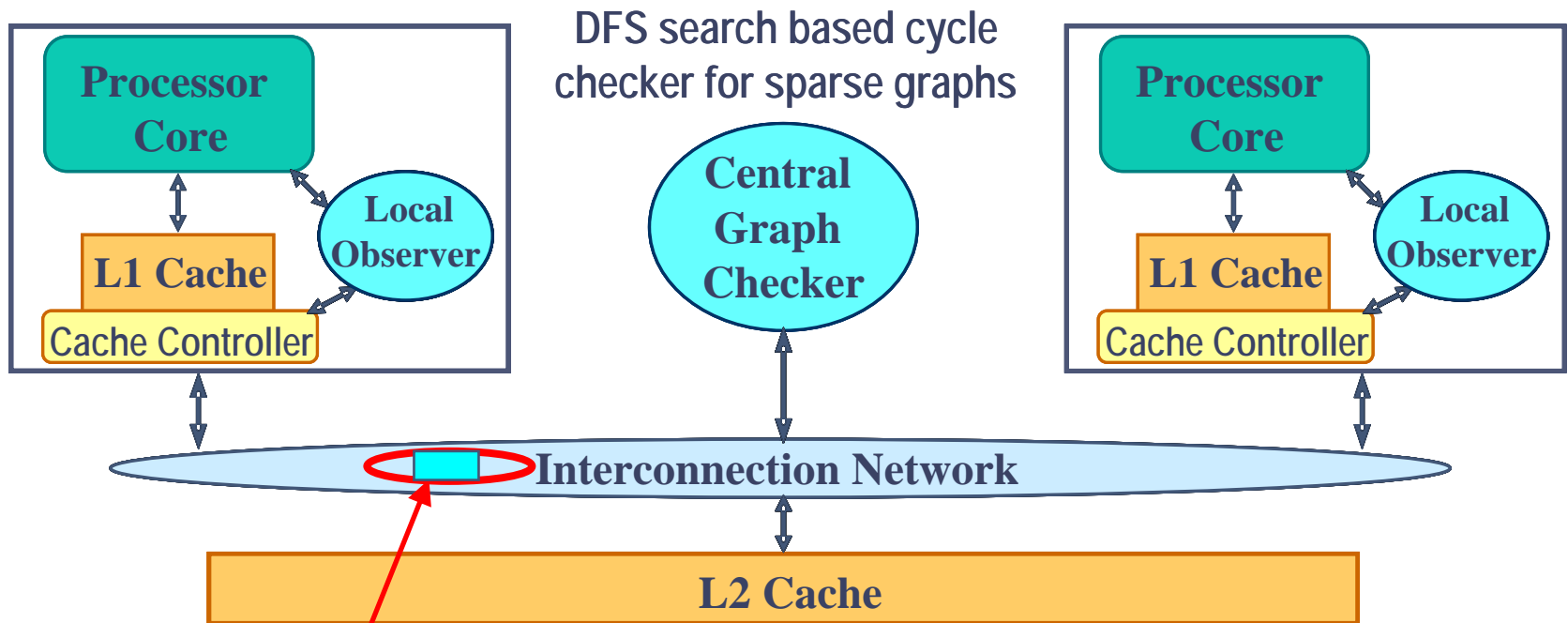


Constraint Graph Evaluation



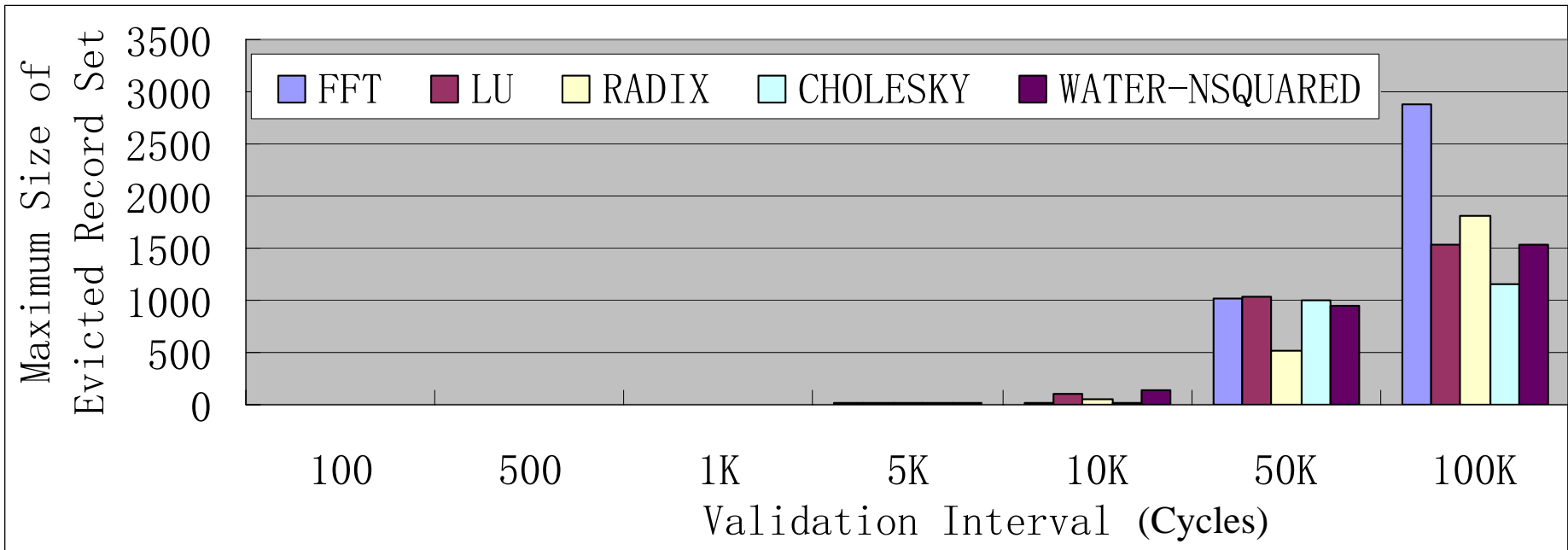
- **Sparse graph**

Experiment Evaluation



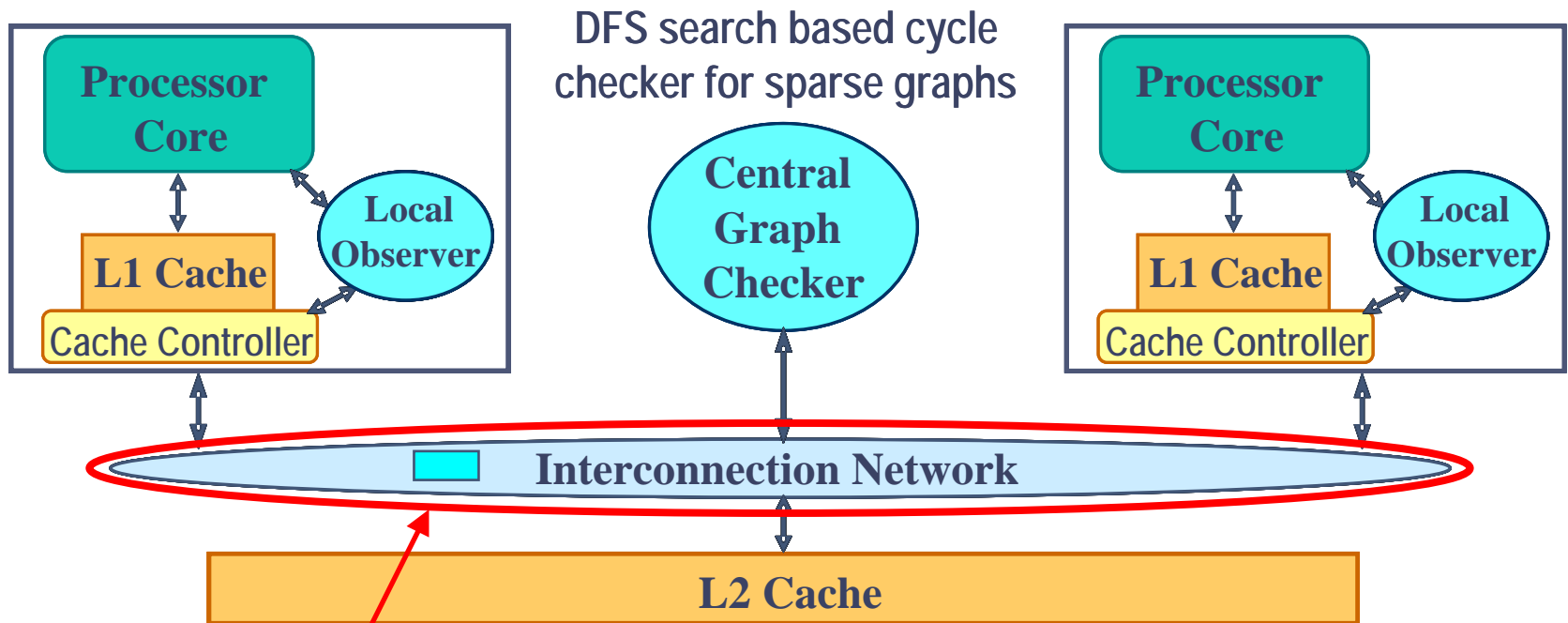
- Evicted cache record buffer size

Constraint Graph Evaluation



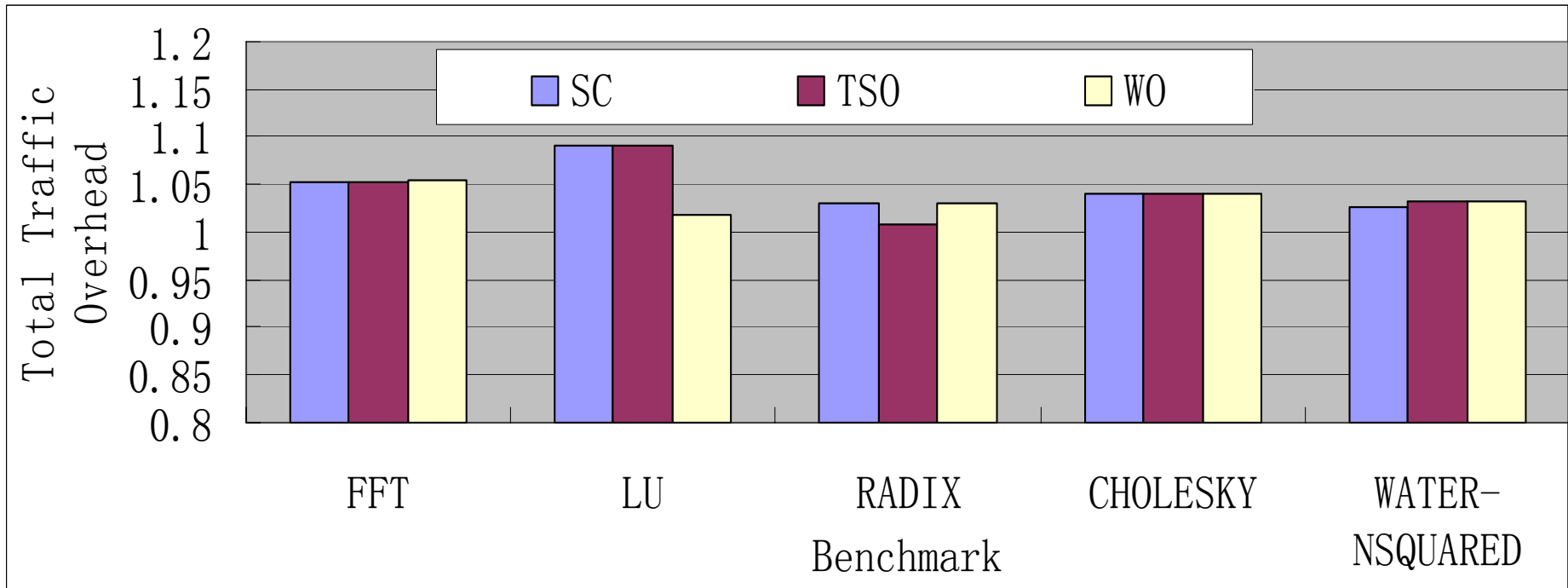
- **Modest hardware cost**

Experiment Evaluation



- **Communication Overhead**

Bandwidth Overhead Evaluation



- **Small communication overhead**

Presentation Outline

- Motivation
- Runtime Validation Methodology
 - Validation Model
 - Basic Solution
 - Design Optimization
- Evaluation
- **Conclusions**

Conclusions

- A runtime approach for validating multiprocessor memory ordering models
 - Effective end-to-end correctness checking against general errors
 - Efficient online causality tracking with low performance penalty
 - Key enabling techniques for reducing validation overhead
- Other potential applications
 - Support for debugging
 - Assist in dynamic reconfiguration
 - Synergy with performance optimization techniques
- Future work
 - Further design evaluation of the proposed schemes
 - Explore runtime validation for other aspects of parallel architectures