Compiler Techniques For Memory Consistency Models

Students: Xing Fang (Purdue), Jaejin Lee (Seoul National University), Kyungwoo Lee (Purdue), Zehra Sura (IBM T.J. Watson Research Center), David Wong (Intel KAI Research Lab)

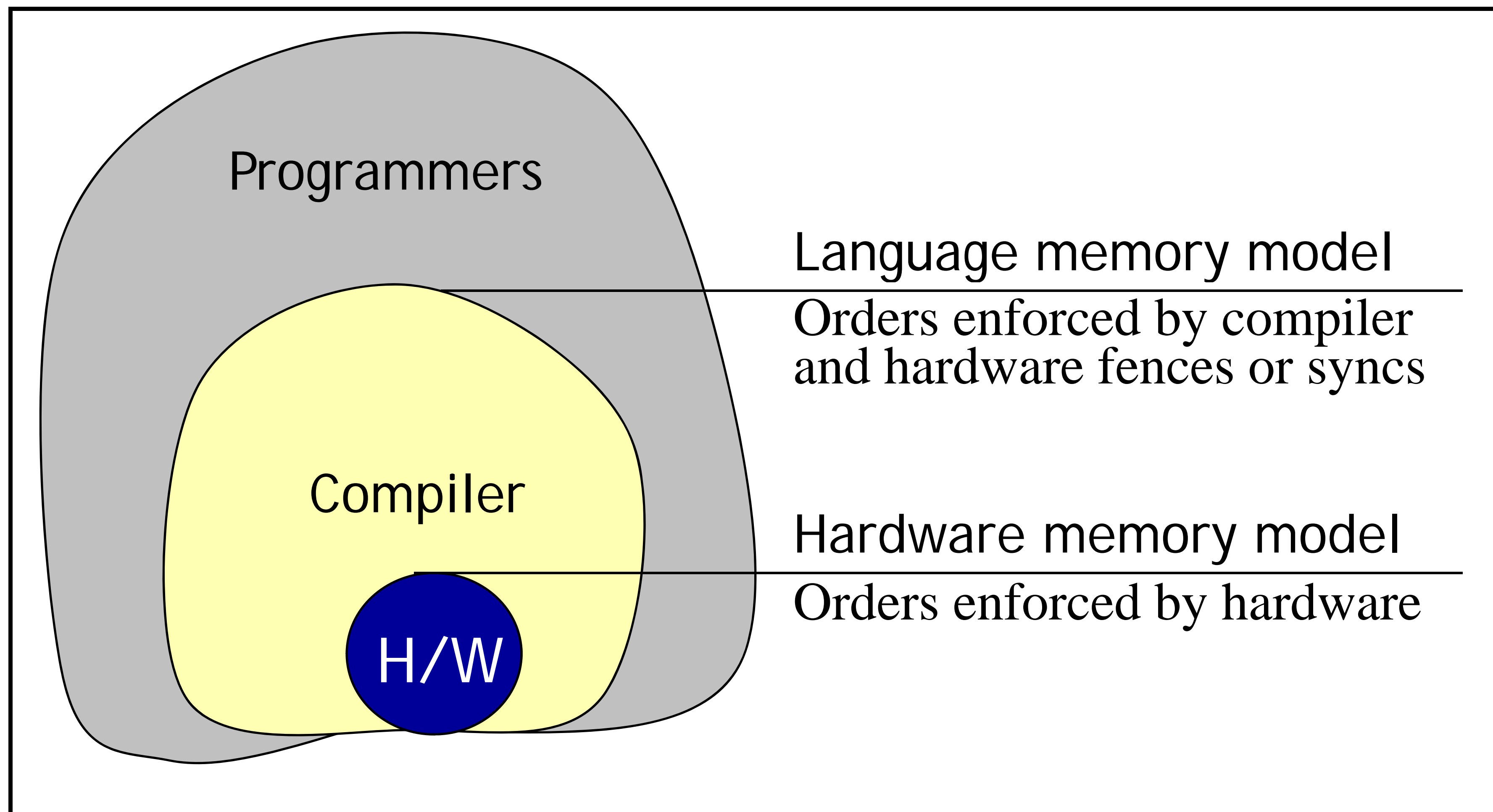
Faculty: Sam Midkiff (Purdue), David Padua (UIUC)

smidkiff@purdue.edu

Goal of this talk

- A brief overview of techniques to enable stricter consistency models to be incorporated into Cell programming models
  - *Techniques are broadly applicable*
  - *Techniques are necessitated by the programming model, not hardware*
  - *Techniques are often not necessary when input program is sequential*

Hardware and Language Models



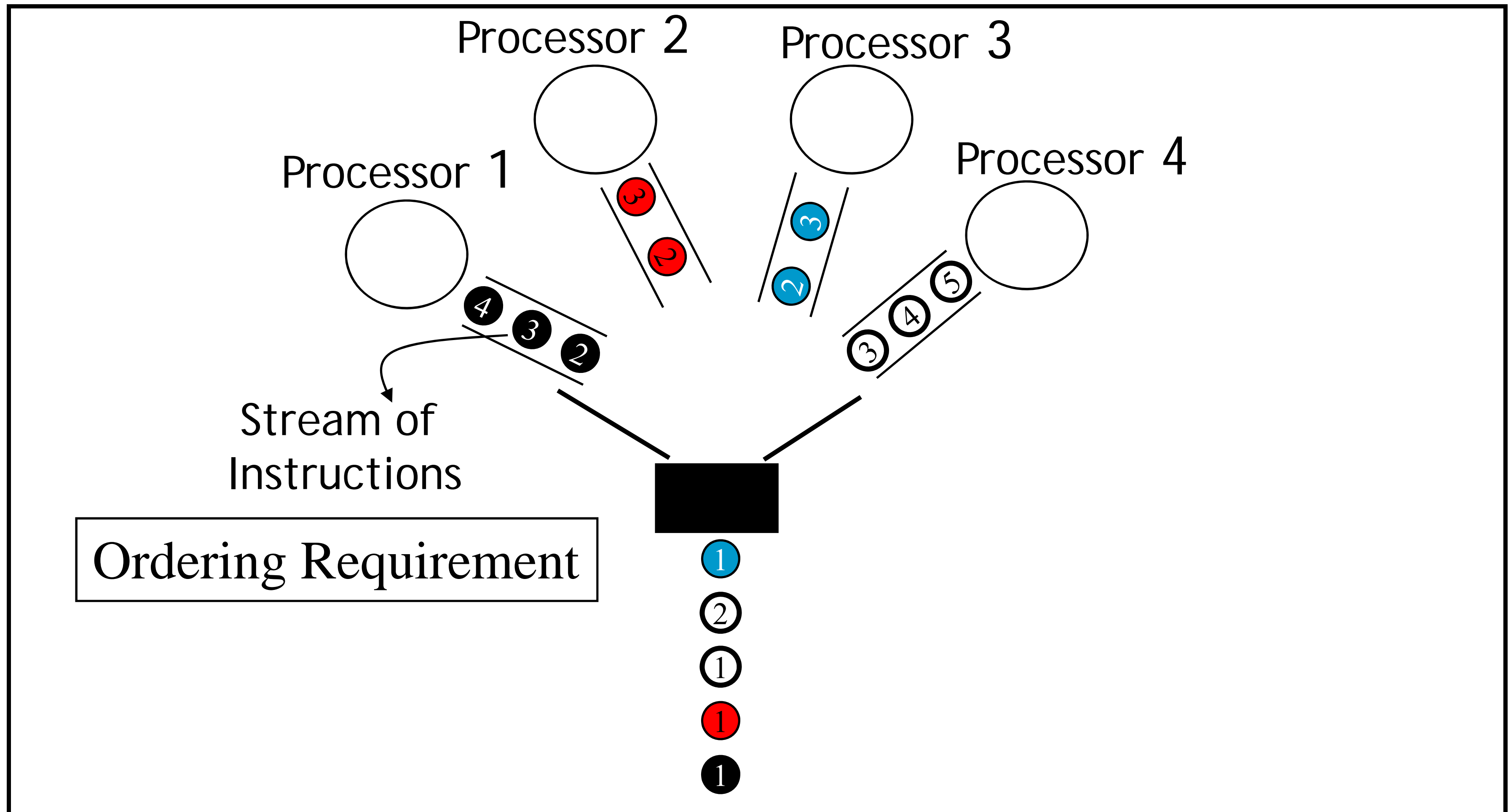
Outline

- Part I: Introduction
  - *Memory Consistency Models*
  - *Compiling for Memory Consistency*
- Part II: Compiler Analysis
  - *Delay set analysis*
  - *Synchronization analysis*
- Part III: Results and Conclusion

When do consistency issues arise?

- Issues arise whenever state is shared across different threads of execution
- Typically reads/writes to shared memory
- Synchronization, I/O, … also require attention be paid to consistency issues
  - *The typical programmer assumes a program will act as if operations occur in the order written*
  - *Not true for most consistency models*
- We will use shared memory reads/writes in our examples for simplicity. Reads/writes can be DMA operations or synchronization

Sequential Consistency (SC)



SC intuitive, but no free lunch

flag = 0; a = **null**;

Thread 0	Thread 1
a = f( );	while (flag ==0);
flag = 1;	b = a;
x[2] = 4;	for (i=0; i<n; i++) {
	x[i] = …

SC is harder to compile than sequential

- Whether a variable reference can be
  - *strength reduced to a register reference,*
  - *hoisted from a loop,*
  - *or otherwise moved*

  In part depends on how used in other threads -- *requires inter-thread analysis*

Relaxed Consistency (RC)

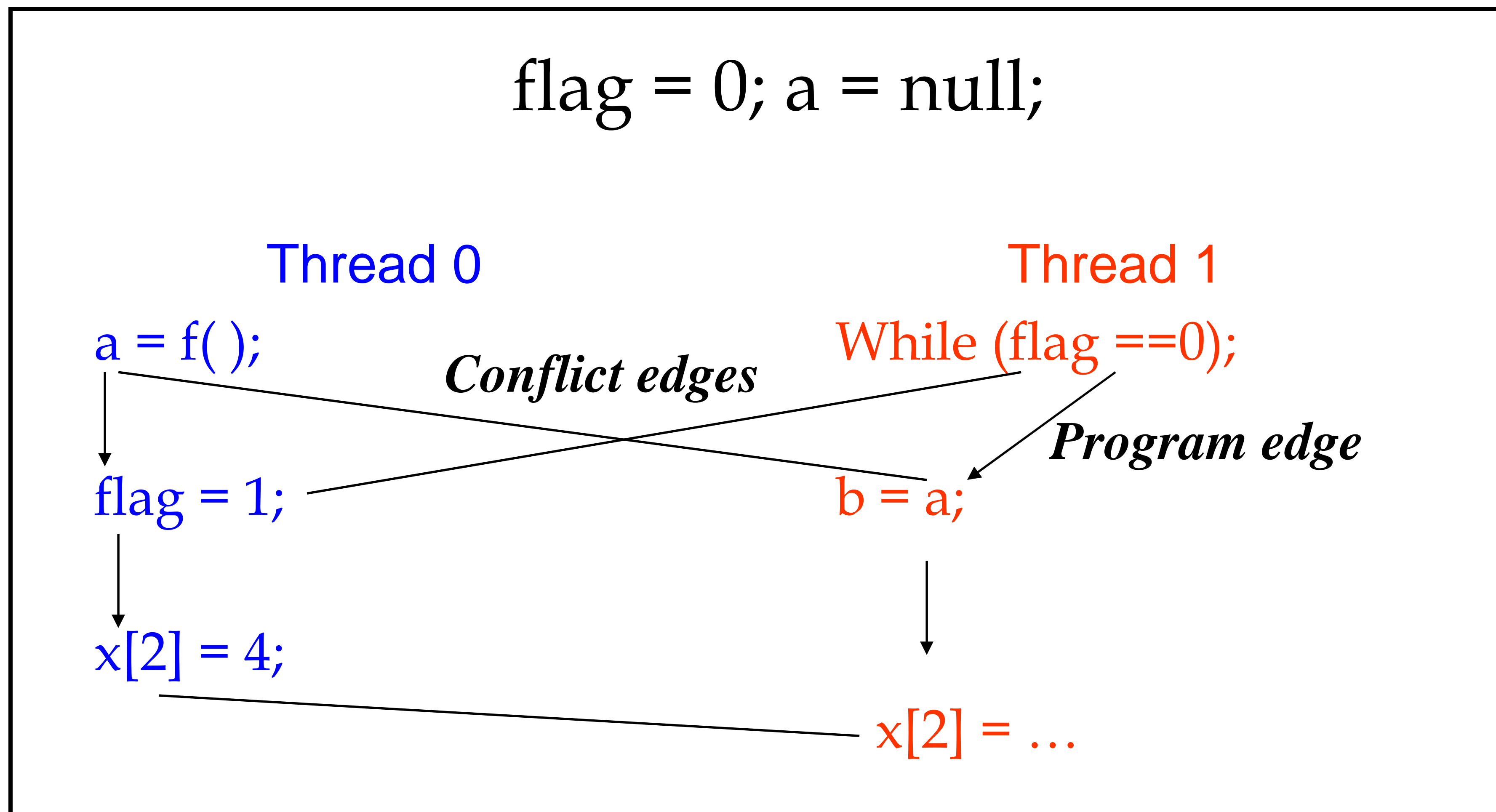
- Like sequential programs, only requires relations among variable accesses *within a thread* to be analyzed when performing optimizations (e.g. dependence/alias analysis)
- Examples:
  - *Weak consistency, Release consistency, Java memory model*
- Semantics for **well synchronized** programs same as SC

RC versus SC

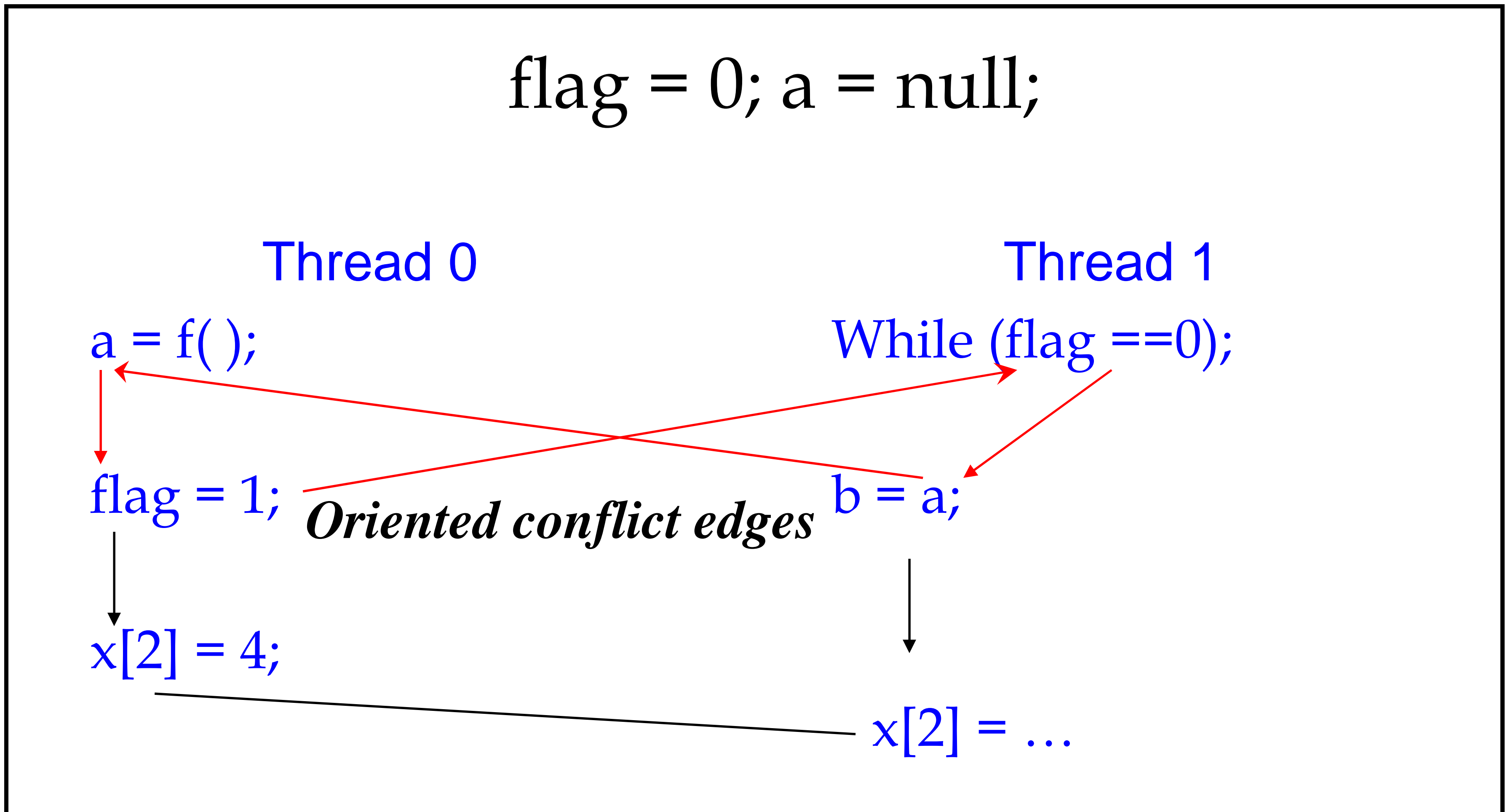
- SC better for programmability
  - *Fewer re-orderings for programmer to reason about*
  - *Compiler cannot naively reorder any memory accesses*
- RC better for performance
  - *Allow accesses to overlap or be re-ordered*
- Can we recover performance for SC?
  - *Compiler analysis to determine orders that really need to be enforced*

Mark Hill, Multiprocessors should support simple memory-consistency models, IEEE Computer, August 1998

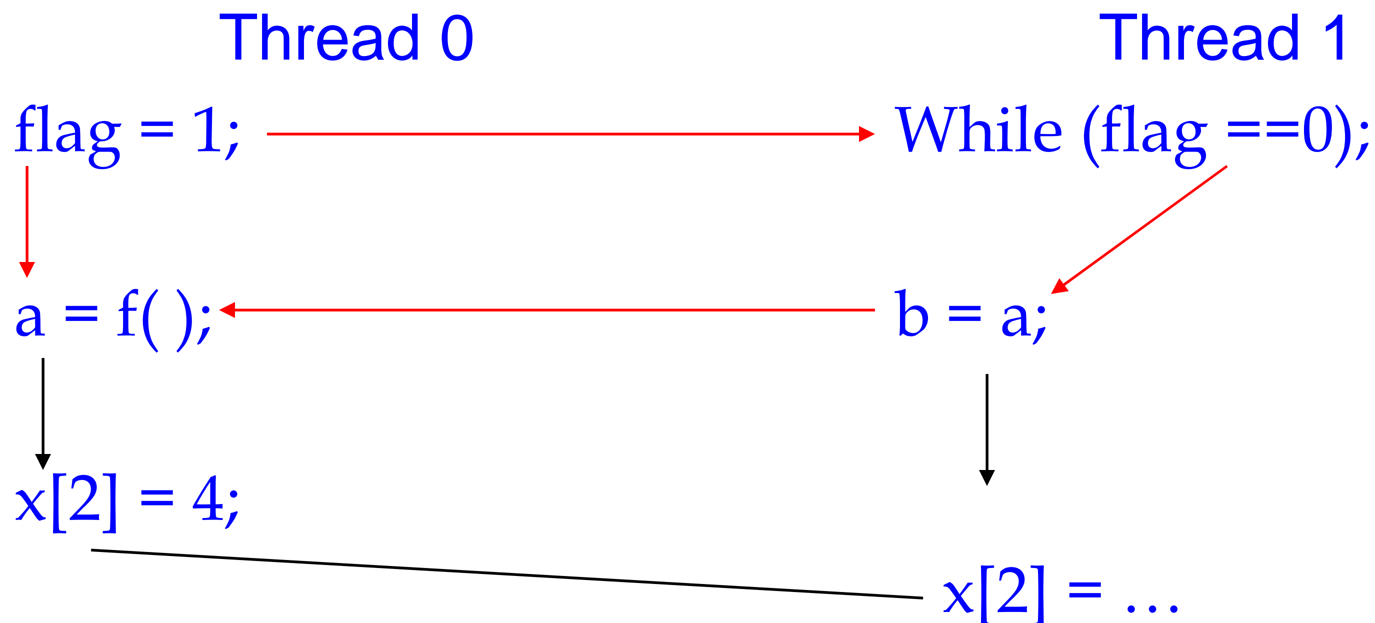
When can memory ops be moved?



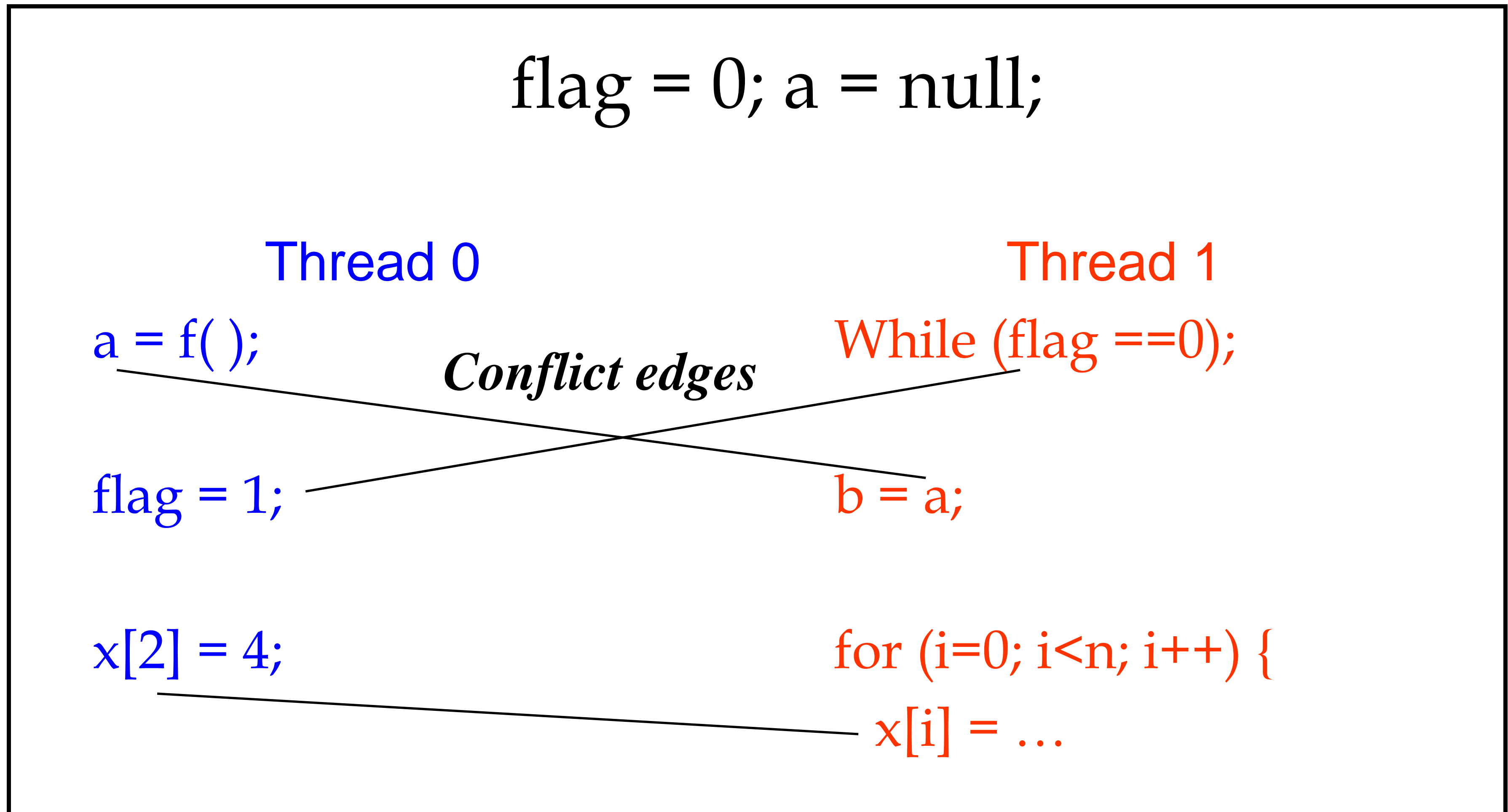
When can memory ops be moved?



How bad orientations can exist



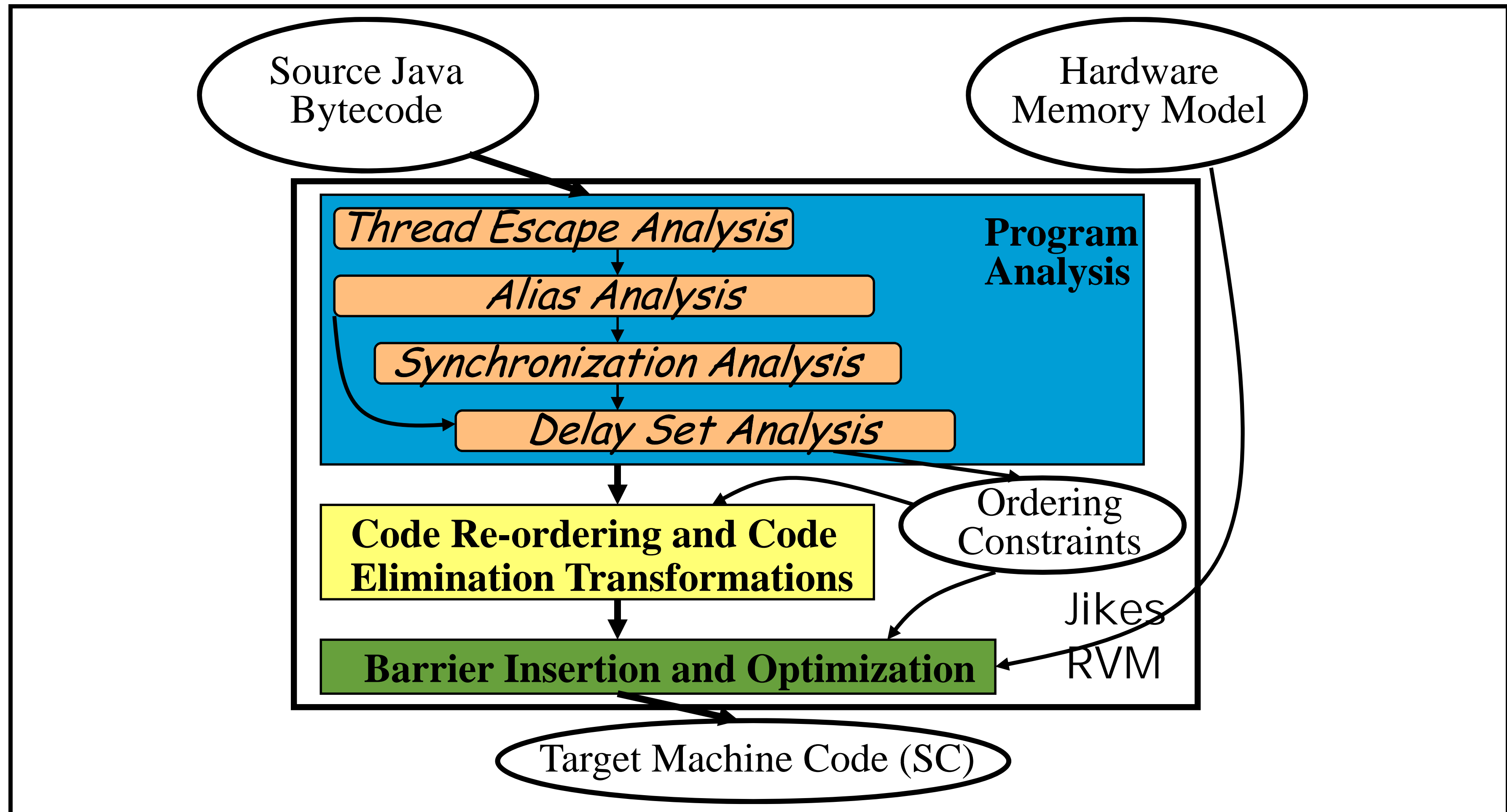
Graph for RC -- program edges only exist between dependent references



What a consistency aware compiler must do

- Program edges involved in cycles must be treated like a dependence and enforced
- Therefore, a consistency aware compiler must determine intra-thread memory operation orderings that must be enforced because of
  - *Inter-thread relationships*
  - *Traditional dependence relationships*
- Ordering of operations that cannot be violated because of inter-thread relationships are *delays*

Pensieve Compiler for SC



How is this handled in current languages?

- MPI avoids these issues by not having shared state
- OpenMP avoids these issues by requiring
  - *shared state in parallel regions to be in an atomic block*
  - *Shared state accessed via reduction, etc*
  - *Otherwise results are undefined*
- Standard Java avoids this by using a relaxed model
- C/C++/Pthreads basically undefined

These work, but …

- All of these solutions have problems
  - *Shared memory programming model sometimes useful*
  - *Undefined results make debugging hard*
  - *Most programmers think SC*

Outline

- Part I: Introduction
  - *Memory Consistency Models*
  - *Compiling for Memory Consistency*
- Part II: Compiler Analysis
  - *Escape analysis*
  - *Alias analysis (simple type based)* *[Sura, PPoPP05]*
  - *Synchronization analysis*
  - *Delay set analysis*
- Part III: Results and Conclusion

Thread Escape Analysis

- Find references to objects that may be accessed in two or more threads
  - *In Java, these are objects accessed directly, or indirectly, from static fields or thread object fields*
    - Java does not allow arguments to be passed to thread run methods
    - Rather, the "arguments" are passed to the thread constructor, and stored in a field in the constructed thread object
  - *Can be modeled as a reachability problem - an object that can be reached (directly or indirectly) by something reachable from 2 or more threads thread-escapes.*

Two phase escape analysis

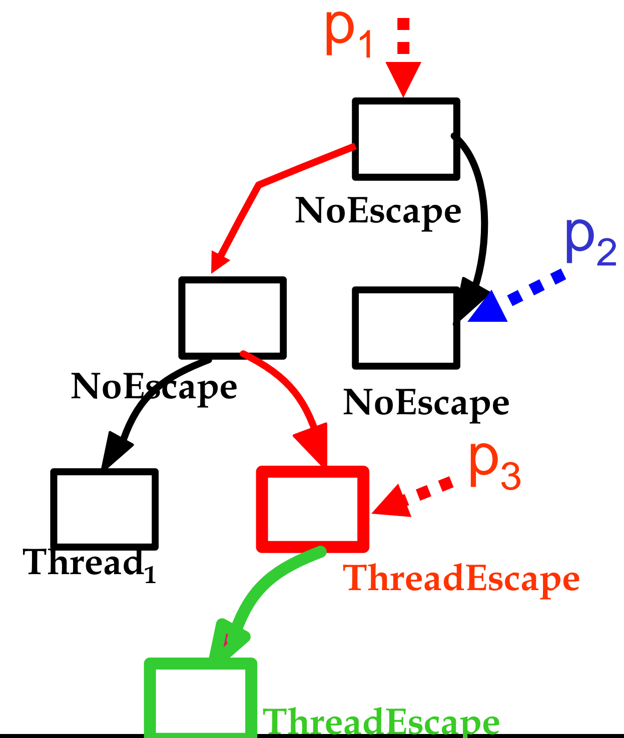
[Lee, PACT06]

- Uses a slow, off-line analysis to build a very precise *connection graph* for available classes
- Results of this analysis are converted to level summary form for the on-line phase
  - *The level summary form is used to reconcile reachability information from*
    - Classes not seen during the offline analysis,
    - Classes that have changed since being seen in the offline analysis

Online Escape Information Representation

- ***Level Summary*** : < *level, EscapeState* >
  - *A conservative, compact representation for parameter or argument at call site*
  - *Tells us where escape happens*
  - *Level summary for* p_1 *is*

    < 2, ThreadEscape>
  - *<∞, NoEscape> for* p_2
  - *<0, ThreadEscape> for* p_3



Alias analysis

- Looks for references in different threads that may access the same object
- If at least one reference writes the object, a *conflict* exists between the two references
- In the Pensieve system, a simple alias analysis that assumes references to objects of the same type are aliased

Synchronization Analysis

[Sura, PPoPP05]

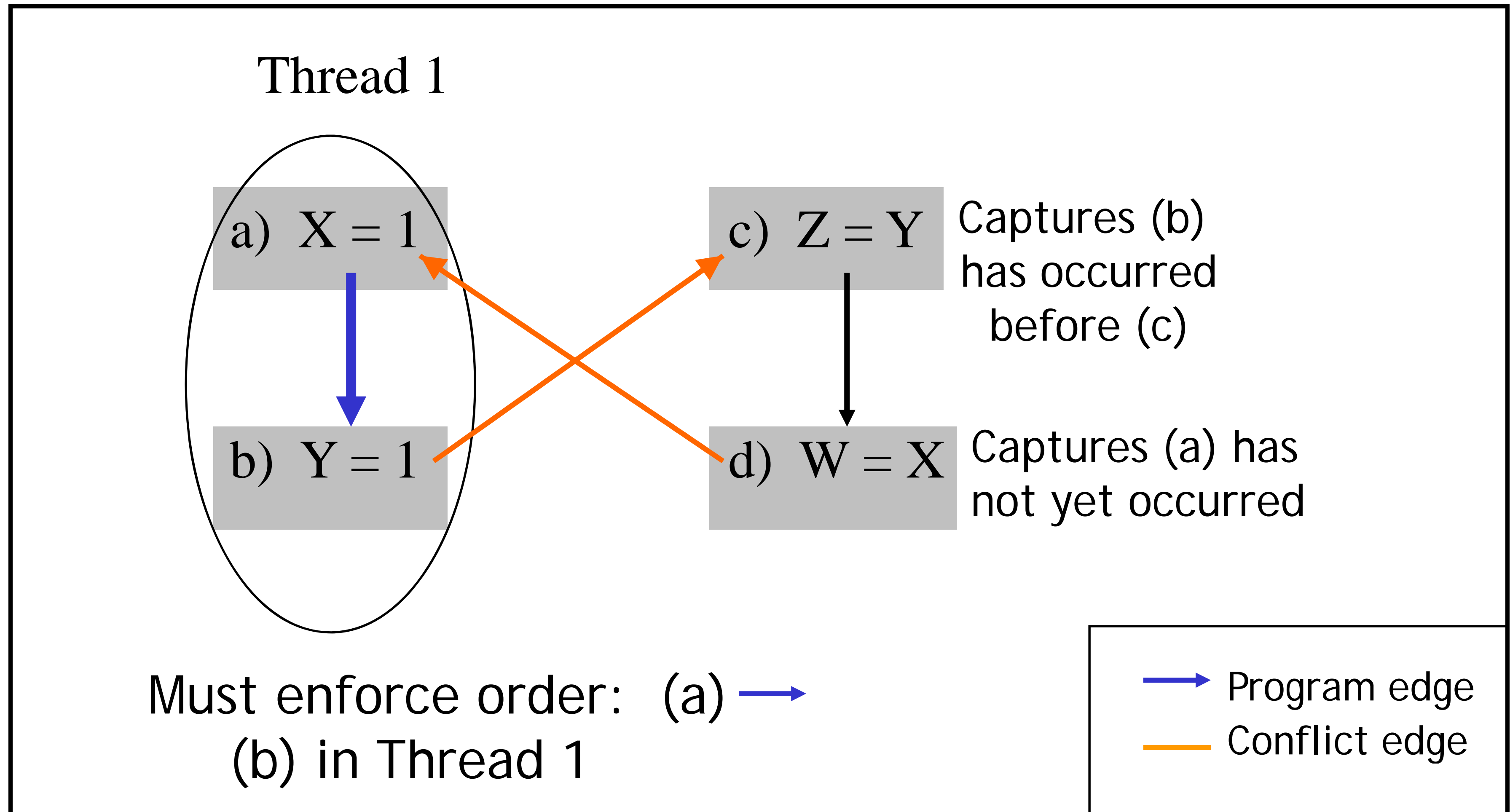
- Consider two types of synchronization:
  - *Thread structure, due to thread* start() *and thread* join() *calls*
  - *Locking, due to* synchronized *blocks*
  - *Yelick has looked at prod./consumer ordering*
- Determine access orderings across threads, code that cannot execute concurrently
- Yields more accurate graph to find delays
  - *Eliminate conflict edges*
  - *Order shared memory accesses*

Delay Set Analysis

[Sura, PPoPP05]

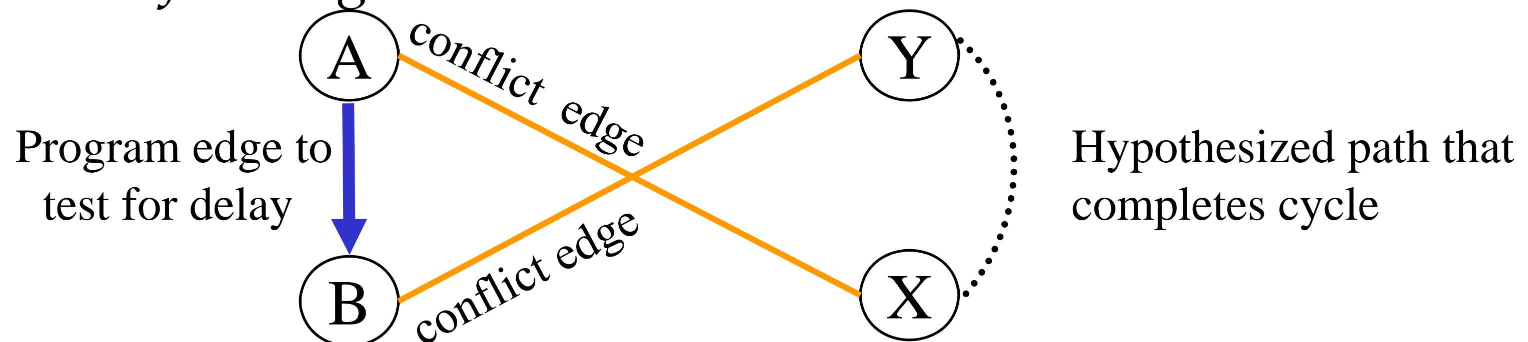
- Delay set: pairs of shared memory accesses (X,Y) in the same thread whose order must be enforced
- Shasha and Snir (TOPLAS 88) show how to find the *minimal* delay set
  - *Build graph to capture all possible access orders in program execution*
  - *Yelick showed in NP to solve this - heuristics needed*

Ordering Requirement



Simplified Delay Set Analysis

- Look for end-points of a possible cycle
- Delay from A to B if:
  - *A and B are thread escaping, and*
  - *Conflict edge between A and some access X, and*
  - *Conflict edge between B and some access Y*



Outline

- Part I: Introduction
  - *Memory Consistency Models*
  - *Compiling for Memory Consistency*
- Part II: Compiler Analysis
  - *Delay set analysis*
  - *Synchronization analysis*
- Part III: Results and Conclusions

Benchmarks

Benchmark	Source	Bytecodes	Thread Types
Hashmap	Doug Lea	24,989	1
GeneticAlgo	Stephen Hartley's code plus Doug Lea's library	30,147	6
BoundedBuf	Uses Doug Lea's library	12,050	1
Sieve	Stephen Hartley	10,811	2
DiskSched	Doug Lea	21,186	2
Montecarlo	Java Grande Forum	63,452	1
Raytracer	Java Grande Forum	33,198	1
MolDyn	Java Grande Forum	26,913	1
SPECmtrt	SPECjvm98	290,260	2

Experiments

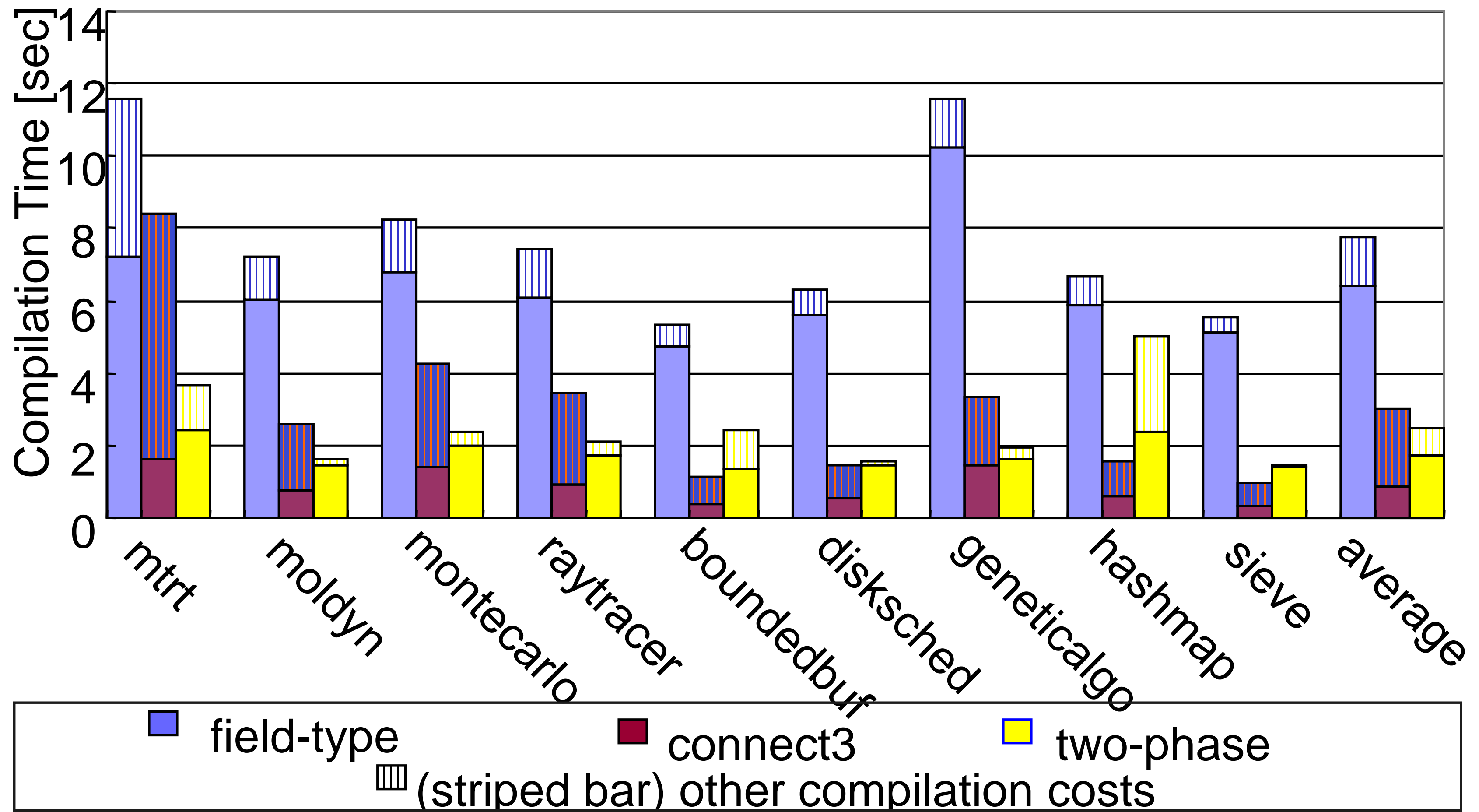
Execution time for three configurations:

1. Base*: default JikesRVM with a relaxed consistency model*
2. Escape*: sequential consistency, using:*
   - iterative, context-sensitive escape analysis
   - order enforced between each pair of escaping accesses
3. Delay*: SC using the escape analysis above, and our synchronization analysis and delay set analysis*

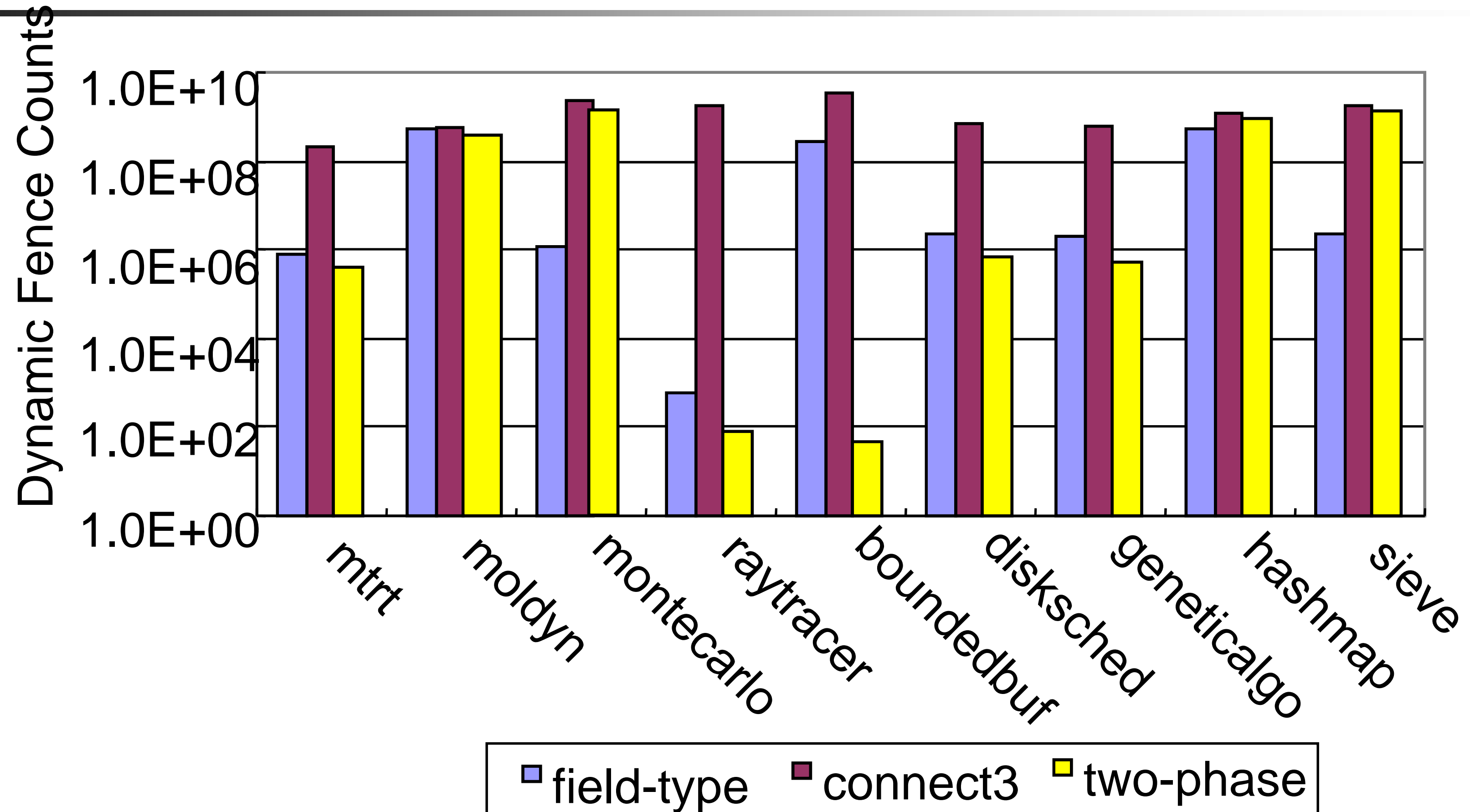
System Configuration

- Intel Pentium 4 Platform
  - *Dell PowerEdge 6600 SMP, using two 1.5GHz Xeon processors with 6GB system memory*
- PowerPC numbers available in papers

Compilation Time

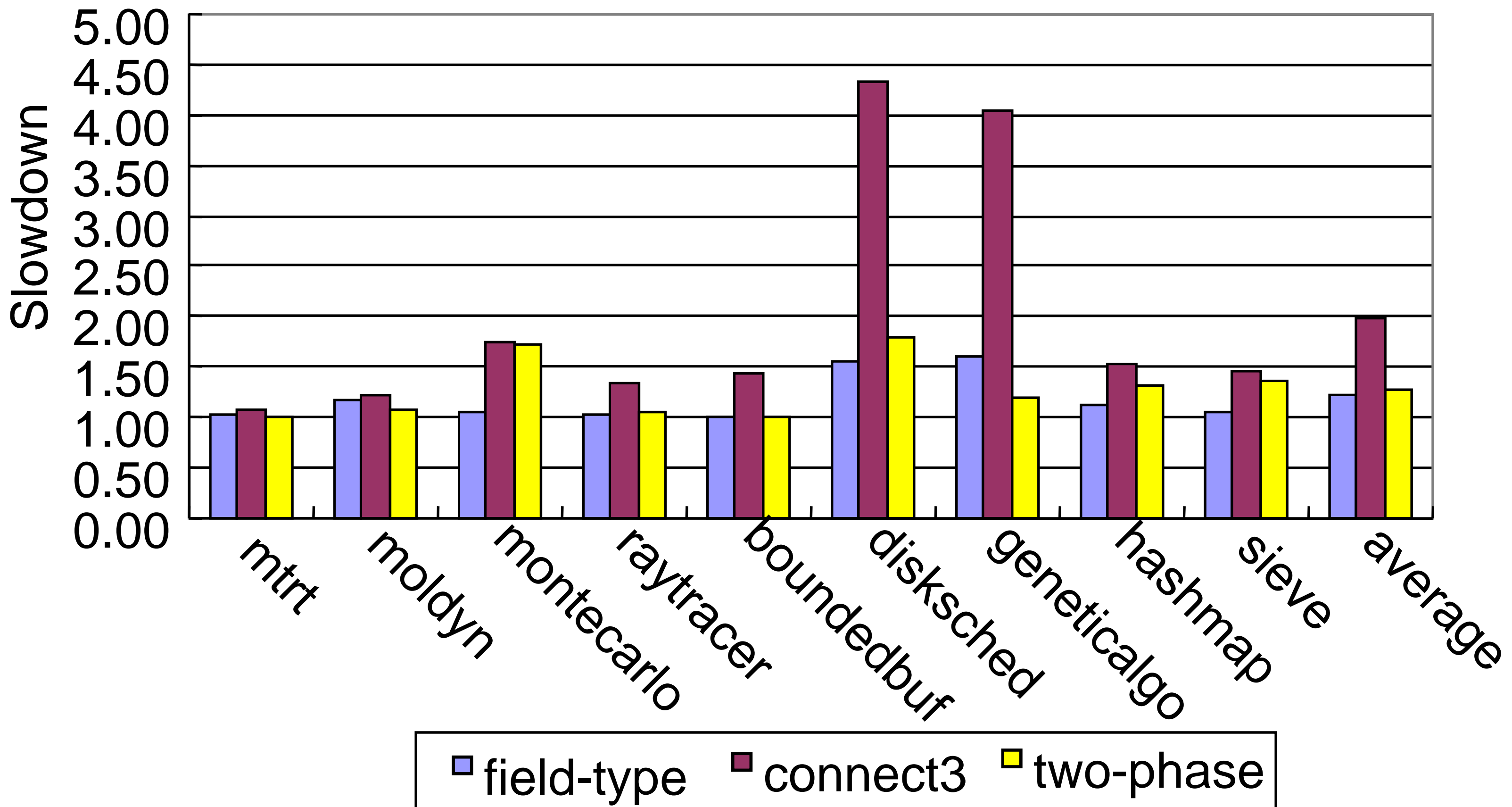


Dynamic Fence Counts

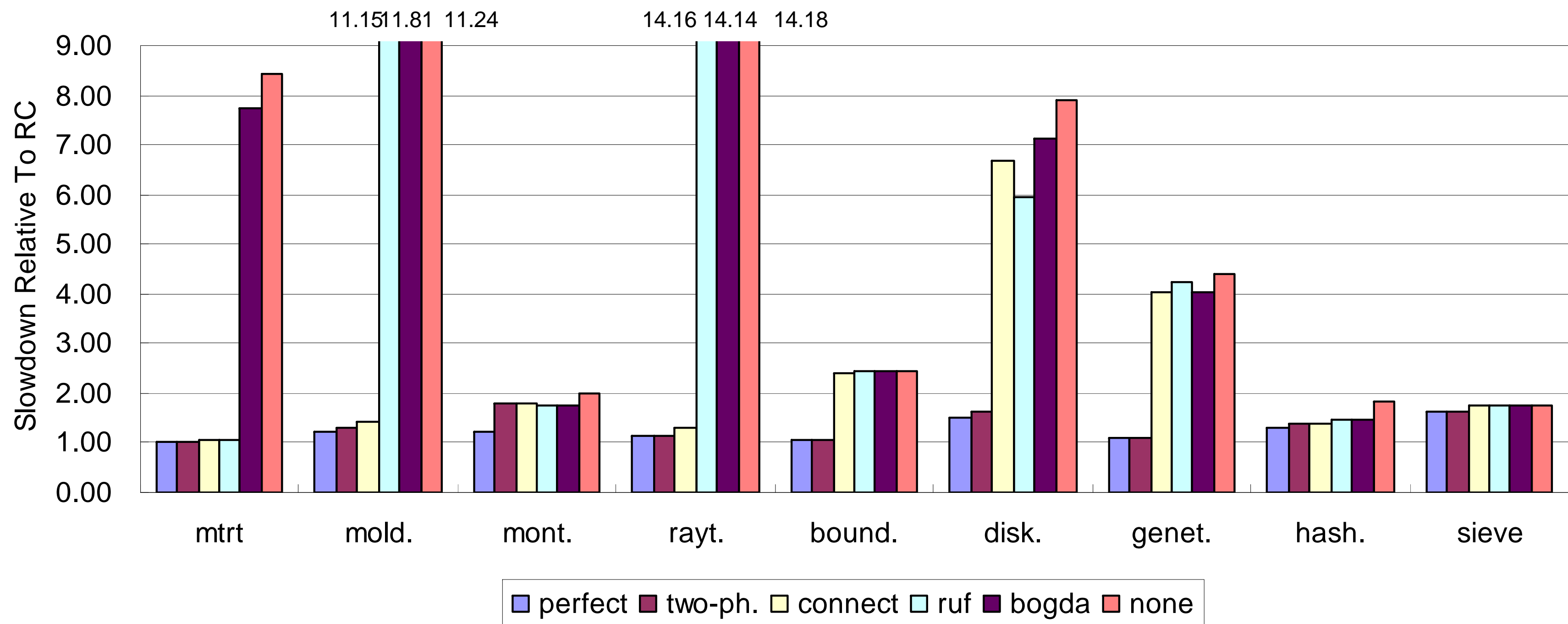


PURDUE UNIVERSITY

Slowdown Relative to RC



SC – Slowdown Relative to RC



PURDUE UNIVERSITY

Some related work

- Yelick and Krishnamurthy - Itanium and UPC - focus is on SPMD style programs
- Von Praun and Gross - optimizations on Java programs
- Sasha and Snir - original paper on delay set analysis
- Sreedhar, Gao -- lock assignment
- Early work out of DEC WRL -- no opts

Conclusions

- Techniques enable fast and effective inter-thread analysis for object-oriented programs using shared memory
- Sensitive to accuracy of escape analysis
- SC shows average slowdown of ~20% on an Intel Xeon platform (1.17 and 1.23 w/perfect, 1.26 with two phase)

Questions?