Hiding the Java Memory Model with Compilers

Jaejin Lee

Department of Computer Science & Engineering

Michigan State University

jlee@cse.msu.edu
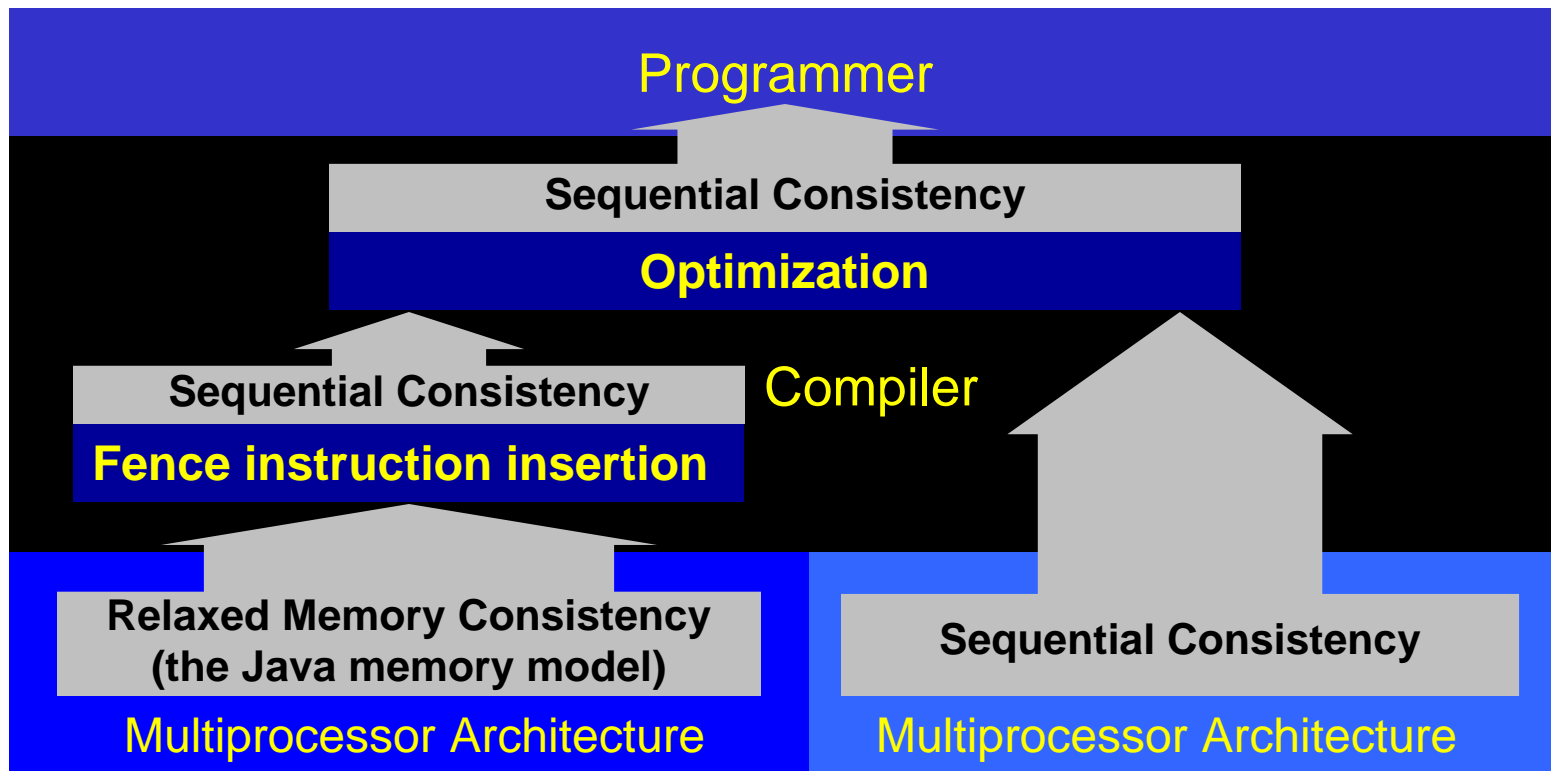
http://www.cse.msu.edu/~jlee

1

Motivation

- The Java memory model is widely known as difficult to understand.
    - It follows a relaxed memory consistency model.
- Data races and synchronization make it impossible to apply classical compiler optimization and analysis techniques directly to Java concurrent programs.
    - It follows a shared memory programming model.
- The synergic effect of the non-intuitive Java memory model and the non-deterministic behavior prevents the programmer from writing correct and efficient Java concurrent programs.
- What if the burden of considering the underlying memory model is shifted to a compiler?

The Compiler

Programmer

Sequential Consistency

Optimization

Sequential Consistency

Compiler

Fence instruction insertion

Relaxed Memory Consistency
(the Java memory model)

Sequential Consistency

Multiprocessor Architecture

Multiprocessor Architecture

- The compiler presents to programmers a sequentially consistent view of the underlying architecture.
- The compiler makes it possible to apply classical compiler optimization techniques correctly to parallel programs that are not handled by conventional compilers.

3

An Example of Incorrect Execution in Java

Initially, x = 0, y = 0

Thread 1	Thread 2
❶ X = x	❸ Y = y
❷ y = 1	❹ x = 1

Incorrect outcome: X = 1, Y = 1

[Pugh, Java Grande'99]

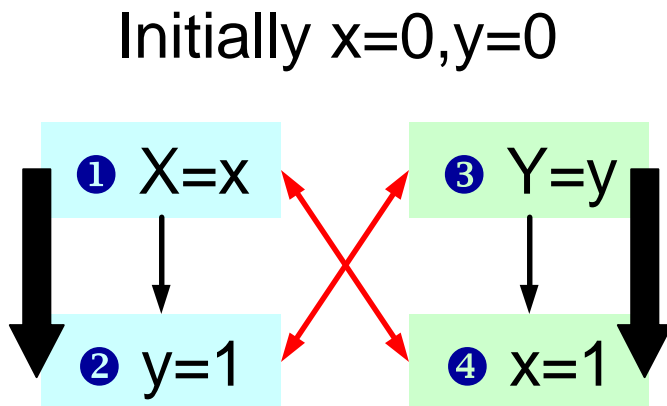|	
❷ y = 1	X: 0, Y: 0
❸ Y = y	X: 0, Y: 1
❹ x = 1	X: 0, Y: 1
❶ X = x	X: 1, Y: 1

- With prescient stores.
- If X is equal to 1 then Y should be 0 (reasoning based on sequential consistency).
- A counter-intuitive result can occur, and this violates sequential consistency. It appears that instructions (i.e., ❶ and ❷) are reordered.

4

Correctness Criterion (Sequential Consistency)

- A multiprocessor system is sequentially consistent if the result of any execution is the same *as if the operations of all the processors were executed in some sequential order*, and *the operations of each individual processor appear in this sequence in the order specified by its program*. [Lamport, IEEE TOC, 1979]

Initially, x = 0, y = 0

Thread 1	Thread 2
❶ X = x	❸ Y = y
❷ y = 1	❹ x = 1

Execution	X Y
❶X=x❷y=1❸Y=y❹x=1	0 1
❸Y=y❹x=1❶X=x❷y=1	1 0
❸Y=y❶X=x❹x=1❷y=1	0 0
…	…
❶X=x❷y=1❹x=1❸Y=y	0 1
❷y=1❸Y=y❹x=1❶X=x	1 1

Unordered, but sequentially consistent

Not sequentially consistent

(Operations are atomic)
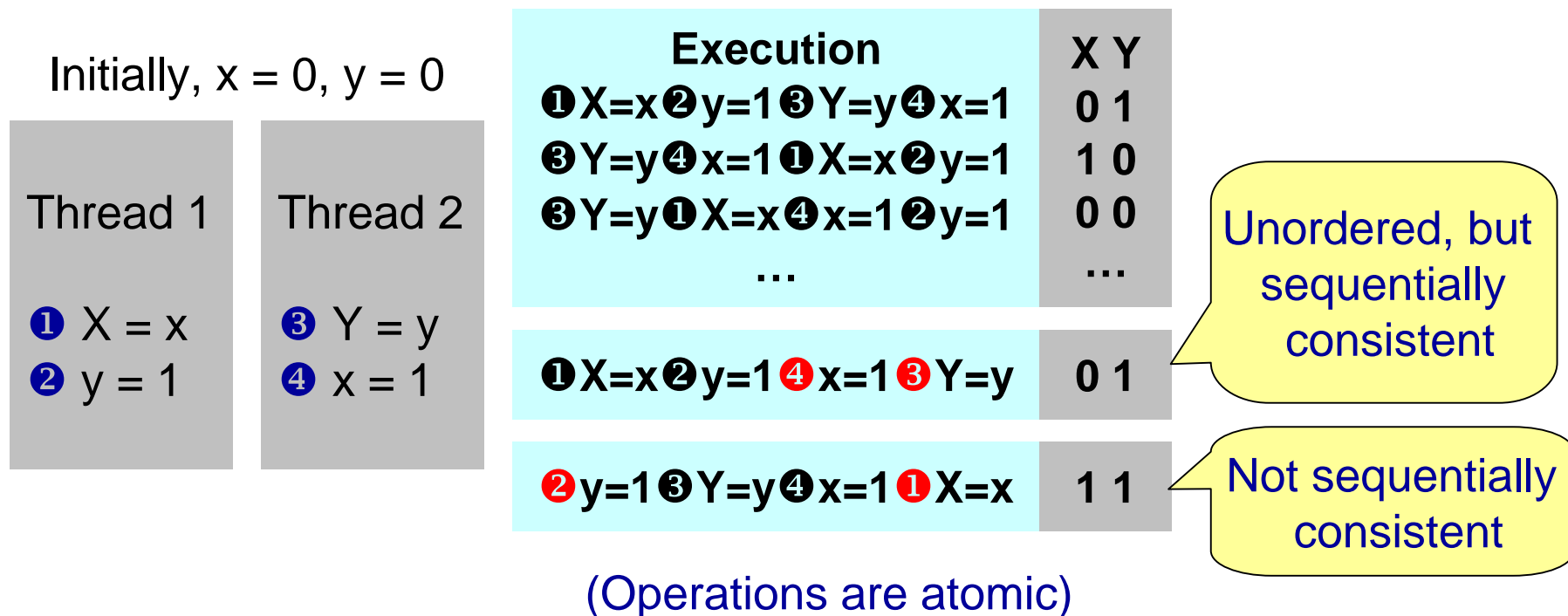
Delay Set Analysis [Shasha and Snir, TOPLAS, 1988]

Initially x=0,y=0

❶ X=x   ❸ Y=y

❷ y=1   ❹ x=1

- Finds conflict relation (conflict edges) using an escape analysis.
- Finds critical cycles (minimal mixed cycles that consist of program ordering edges and conflict edges).

- Delays (❶D❷ and ❸D❹) are the program ordering edges in the critical cycles (e.g., ❷ ❸ ❹ ❶ ❷).

- Enforcing delays is a sufficient condition to guarantee sequential consistency.

- Delays are implemented by fence instructions or the ordering constraints of the underlying machine.

6

Inserting Fence Instructions

u

w

v

control flow
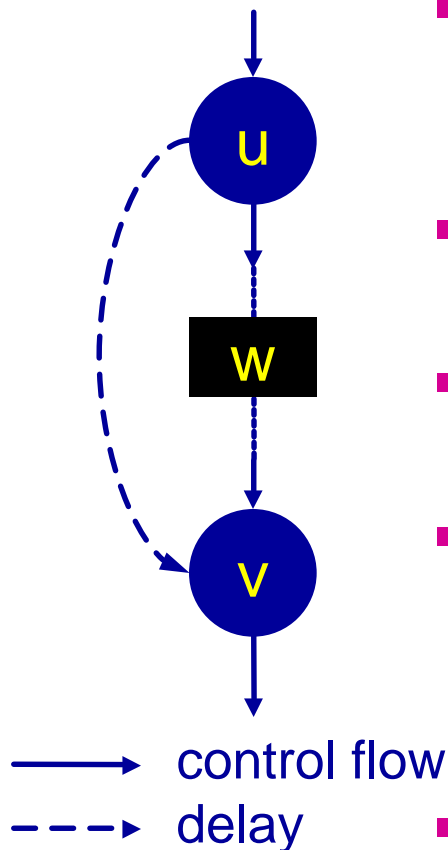
- - -> delay

- $D_m = ((D \cup D_o)^+)^{tr} - D_o$ (reduce the number of delays by using the ordering constraints of the underlying architecture)
- We insert a fence instruction for each delay $u D_m v$ in a *memory-barrier node w*.
- w always executes after u and before v whenever *u* and *v* executes.
- A node *s dominates* a node *v* with *respect to a node u* (*s* $\mathbf{dom_u}$ *v*) if every control flow path from *u* to *v* goes through *s* [Lee and Padua, PACT'00].
- Any node in $\mathbf{dom_u}[v]$ can be a memory-barrier node to enforce the delay *u$\mathbf{D}$v.*

7

Minimizing
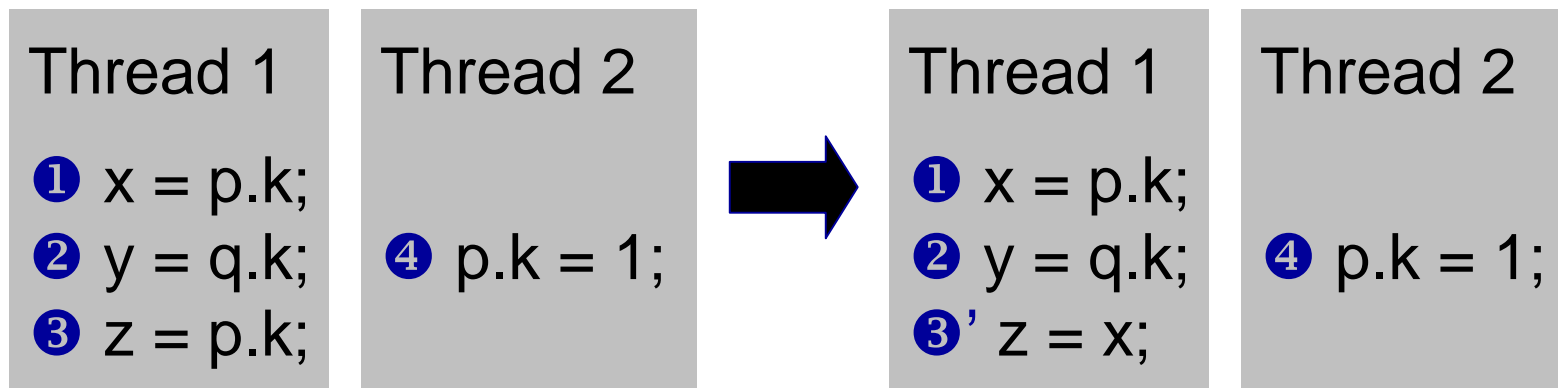# the Number of Memory-Barrier Nodes
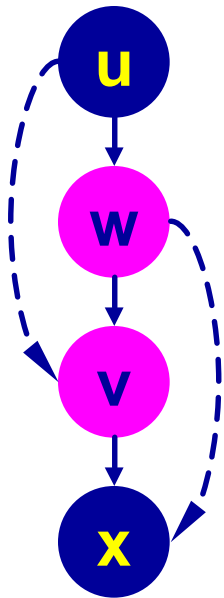
control flow

- - - → delay

- If a node $w \in \mathbf{dom}_u[v]$, then w enforces the delay $u\mathbf{D}v$.

- One memory-barrier node w (or v) is enough to enforce both delays $u\mathbf{D}v$ and $w\mathbf{D}x$.

    $(\ \mathbf{dom}_u[v] \ \boldsymbol{\zeta} \ \mathbf{dom}_w[x] = \{\ w,\ v\ \}\ )$

- Minimizing the number of memory-barrier nodes by using dominators with respect to a node is an *NP-hard* problem [Lee and Padua, PACT'00].

- Use a greedy approximation algorithm.

An Example of an Incorrect Compiler Optimization in Java
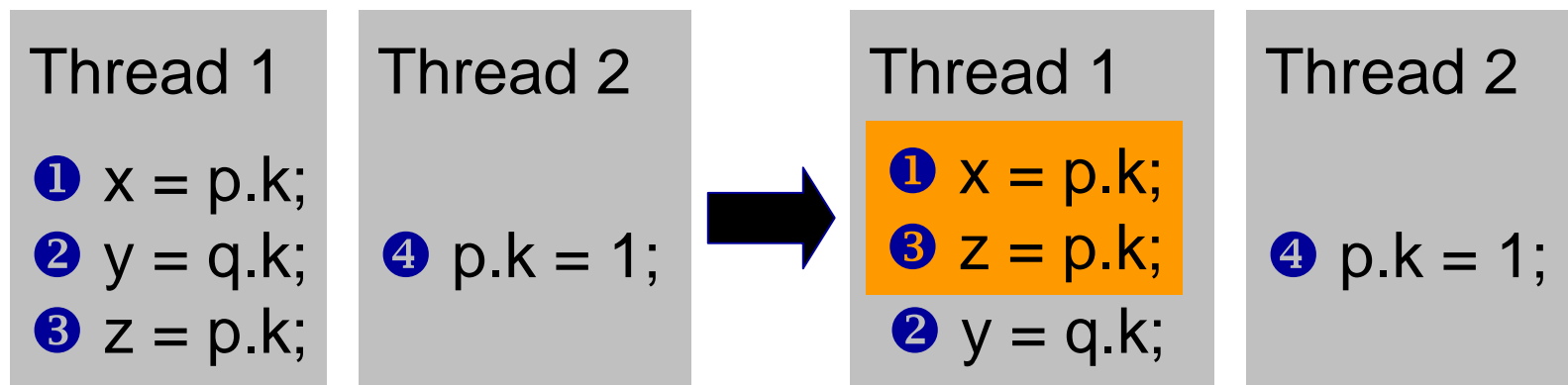
Initially, p and q are aliases and p.k = 0

Thread 1	Thread 2
❶ x = p.k;	
❷ y = q.k;	❹ p.k = 1;
❸ z = p.k;	

Incorrect outcome: x=0, y=1, z=0

Thread 1	Thread 2
❶ x = p.k;	
❷ y = q.k;	❹ p.k = 1;
❸' z = x;	

[Pugh, Java Grande'99]

- p.k is not modified in between ❶ and ❸. In classical sense, it is OK to replace p.k in ❸ with x (one form of classical common subexpression elimination).

- If z is equal to 0 then both x and y should be 0 (reasoning based on sequential consistency).
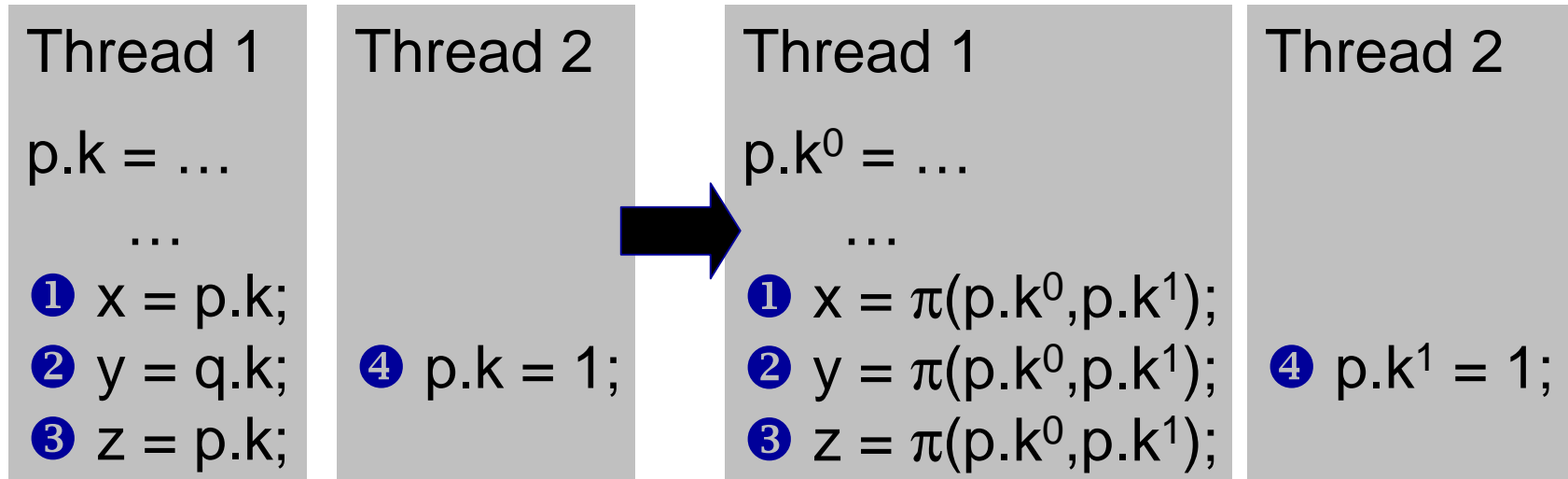
Why It Is an Incorrect Compiler Optimization?

Thread 1	Thread 2
❶ x = p.k;	
❷ y = q.k;	❹ p.k = 1;
❸ z = p.k;	

➡

Thread 1	Thread 2
❶ x = p.k;	
❸ z = p.k;	❹ p.k = 1;
❷ y = q.k;	

- The same as moving ❸ immediately after ❶ and executing ❶ and ❸ together without allowing ❹ interleaved in between them, i.e., ❷ and ❸ are reordered.

- A counter-intuitive result can occur, and this violates sequential consistency.

Another Correctness Criterion (Subset Correctness)

- A compiler transformation is correct if the set of all possible observable behaviors of a transformed program is a subset of all possible observable behavior of the original program [Lee, Padua, Midkiff, PPoPP'99].

- A sequential consistency violation implies a subset correctness violation, but not *vice versa*.

How to Avoid the Incorrect Optimization?

Thread 1	Thread 2		Thread 1	Thread 2
p.k = …			$p.k^0$ = …	
…		➡	…	
❶ x = p.k;			❶ x = $\pi(p.k^0, p.k^1)$;	
❷ y = q.k;	❹ p.k = 1;		❷ y = $\pi(p.k^0, p.k^1)$;	❹ $p.k^1$ = 1;
❸ z = p.k;			❸ z = $\pi(p.k^0, p.k^1)$;	

- Use delays as constraints for the compiler transformations.
- Use a confluence function π to summarize the interaction between threads (concurrent static single assignment (CSSA) form [Lee,Padua,Midkiff. PPoPP'99]).
- Use global value numbering to detect equivalent variables (concurrent global value numbering [Lee,Padua,Midkiff. PPoPP'99]).

Conclusions

- The compiler presents to programmers a natural and intuitive memory model (sequential consistency) irrespective of whether the underlying memory consistency model follows a sequentially consistent or a relaxed model.

- The compiler makes it possible to apply classical compiler optimization techniques correctly to parallel programs that are not handled by conventional compilers.

References

- Jaejin Lee and David Padua, "Hiding Relaxed Memory Consistency with Compilers", *IEEE International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2000.

- Jaejin Lee, "Compilation Techniques for Explicitly Parallel Programs", *Ph.D. Thesis*, University of Illinois, UIUCDCS-R-93-1814, Oct. 1999.

- Jaejin Lee, David A. Padua, and Samuel P. Midkiff, "Basic Compiler Algorithms for Parallel Programs", *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1999.

- Jaejin Lee, Samuel P. Midkiff, and David A. Padua, "A Constant Propagation Algorithm for Explicitly Parallel Programs", *International Journal of Parallel Programming*, 26(5):563-589, 1998.

- Jaejin Lee, Samuel P. Midkiff, and David A. Padua, "Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs", *The 10th International Workshop on Languages and Compilers for Parallel Computing,* Aug. 1997.