EECS 583 – Class 17
Research Topic 1
Decoupled Software Pipelining

University of Michigan

November 9, 2011

Announcements + Reading Material

- 2nd paper review due today
  - » Should have submitted to andrew.eecs.umich.edu:/y/submit
- Next Monday – Midterm exam in class
- Today's class reading
  - » "Automatic Thread Extraction with Decoupled Software Pipelining," G. Ottoni, R. Rangan, A. Stoler, and D. I. August, *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, Nov. 2005.
- Next class reading (Wednes Nov 16)
  - » "Spice: Speculative Parallel Iteration Chunk Execution," E. Raman, N. Vachharajani, R. Rangan, and D. I. August, *Proc 2008 Intl. Symposium on Code Generation and Optimization*, April 2008.

Midterm Exam

- When: Monday, Nov 14, 2011, 10:40-12:30
- Where
  - 1005 EECS
    - Uniquenames starting with A-H go here
  - 3150 Dow (our classroom)
    - Uniquenames starting with I-Z go here
- What to expect
  - Open book/notes, no laptops
  - Apply techniques we discussed in class on examples
  - Reason about solving compiler problems – why things are done
  - A couple of thinking problems
  - No LLVM code
  - Reasonably long but you should finish
- Last 2 years exams are posted on the course website
  - Note – Past exams may not accurately predict future exams!!

Midterm Exam

- Office hours between now and Monday if you have questions
  - Daya: Thurs and Fri 3-5pm
  - Scott: Wednes 4:30-5:30, Fri 4:30-5:30
- Studying
  - Yes, you should study even though its open notes
    - Lots of material that you have likely forgotten
    - Refresh your memories
    - No memorization required, but you need to be familiar with the material to finish the exam
  - Go through lecture notes, especially the examples!
  - If you are confused on a topic, go through the reading
  - If still confused, come talk to me or Daya
  - Go through the practice exams as the final step

Exam Topics

- Control flow analysis
  - Control flow graphs, Dom/pdom, Loop detection
  - Trace selection, superblocks
- Predicated execution
  - Control dependence analysis, if-conversion, hyperblocks
  - Can ignore control height reduction
- Dataflow analysis
  - Liveness, reaching defs, DU/UD chains, available defs/exprs
  - Static single assignment
- Optimizations
  - Classical: Dead code elim, constant/copy prop, CSE, LICM, induction variable strength reduction
  - ILP optimizations - unrolling, renaming, tree height reduction, induction/accumulator expansion
  - Speculative optimization – like HW 1

Exam Topics - Continued

- Acyclic scheduling
  - Dependence graphs, Estart/Lstart/Slack, list scheduling
  - Code motion across branches, speculation, exceptions
- Software pipelining
  - DSA form, ResMII, RecMII, modulo scheduling
  - Make sure you can modulo schedule a loop!
  - Execution control with LC, ESC
- Register allocation
  - Live ranges, graph coloring
- Research topics
  - Can ignore these

Last Class

- Scientific codes – Successful parallelization
  - KAP, SUIF, Parascope, gcc w/ Graphite
  - Affine array dependence analysis
  - DOALL parallelization
- C programs
  - Not dominated by array accesses – classic parallelization fails
  - Speculative parallelization – Hydra, Stampede, Speculative multithreading
    - Profiling to identify statistical DOALL loops
    - But not all loops DOALL, outer loops typically not!!
- This class – Parallelizing loops with dependences

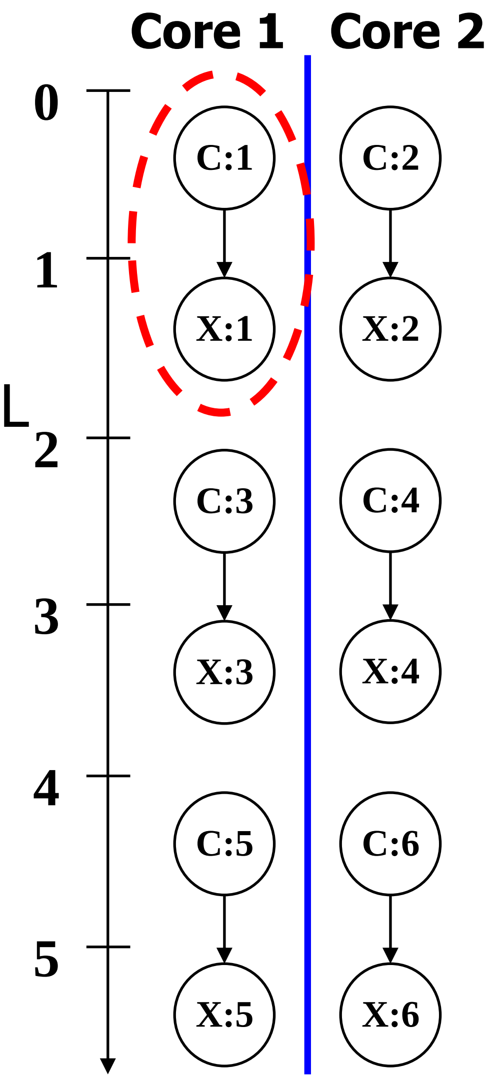
What About Non-Scientific Codes???

Scientific Codes (FORTRAN-like)

```
for(i=1; i<=N; i++) // C
  a[i] = a[i] + 1;  // X
```

Independent Multithreading (IMT)

Example: DOALL parallelization

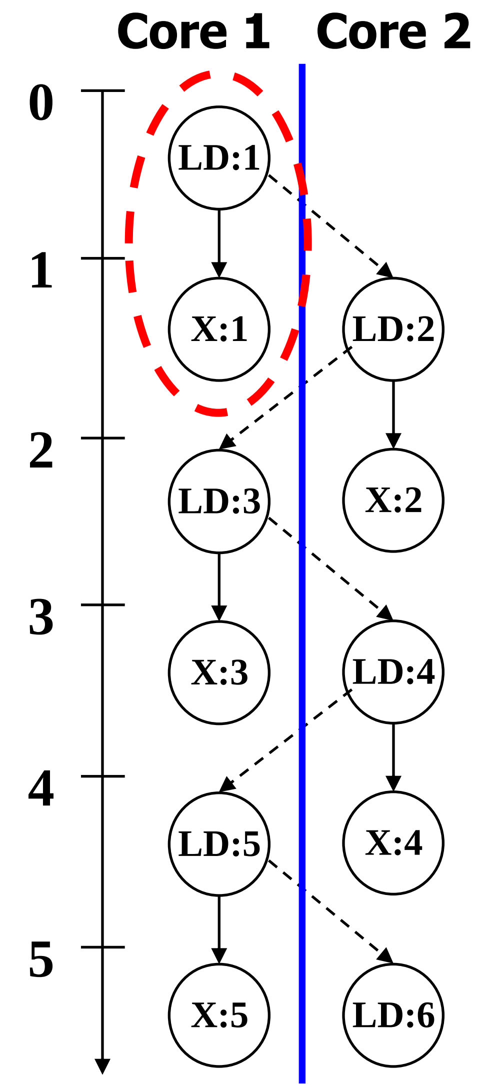


General-purpose Codes (legacy C/C++)

```
while(ptr = ptr->next)     // LD
  ptr->val = ptr->val + 1; // X
```

Cyclic Multithreading (CMT)

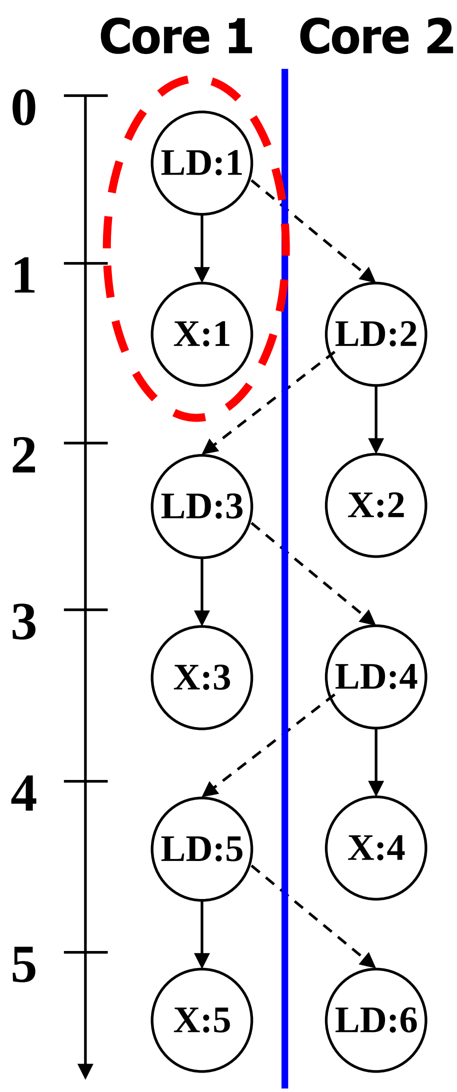
Example: DOACROSS [Cytron, ICPP 86]



Alternative Parallelization Approaches

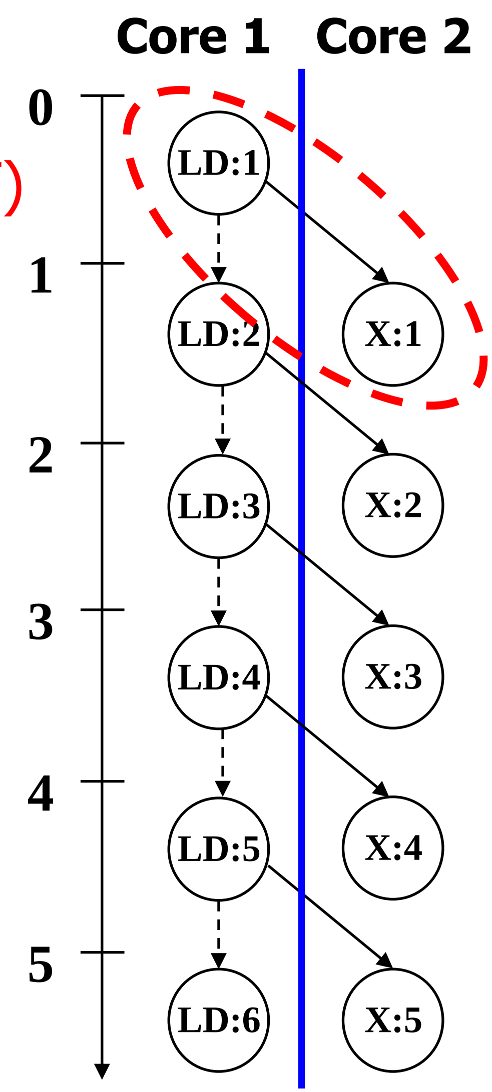
```
while(ptr = ptr->next)      // LD
  ptr->val = ptr->val + 1; // X
```

Cyclic Multithreading (CMT)

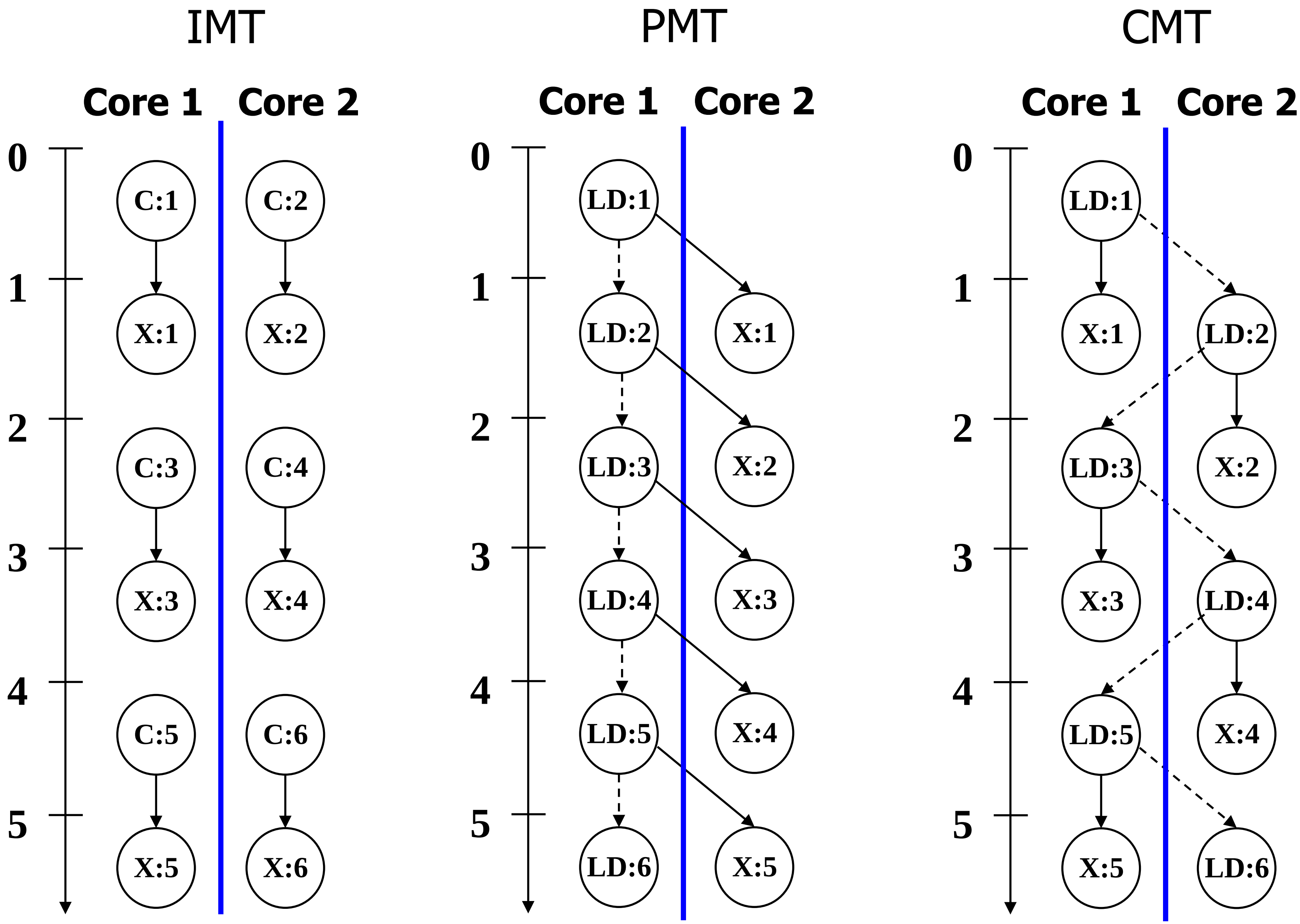


Pipelined Multithreading (PMT)

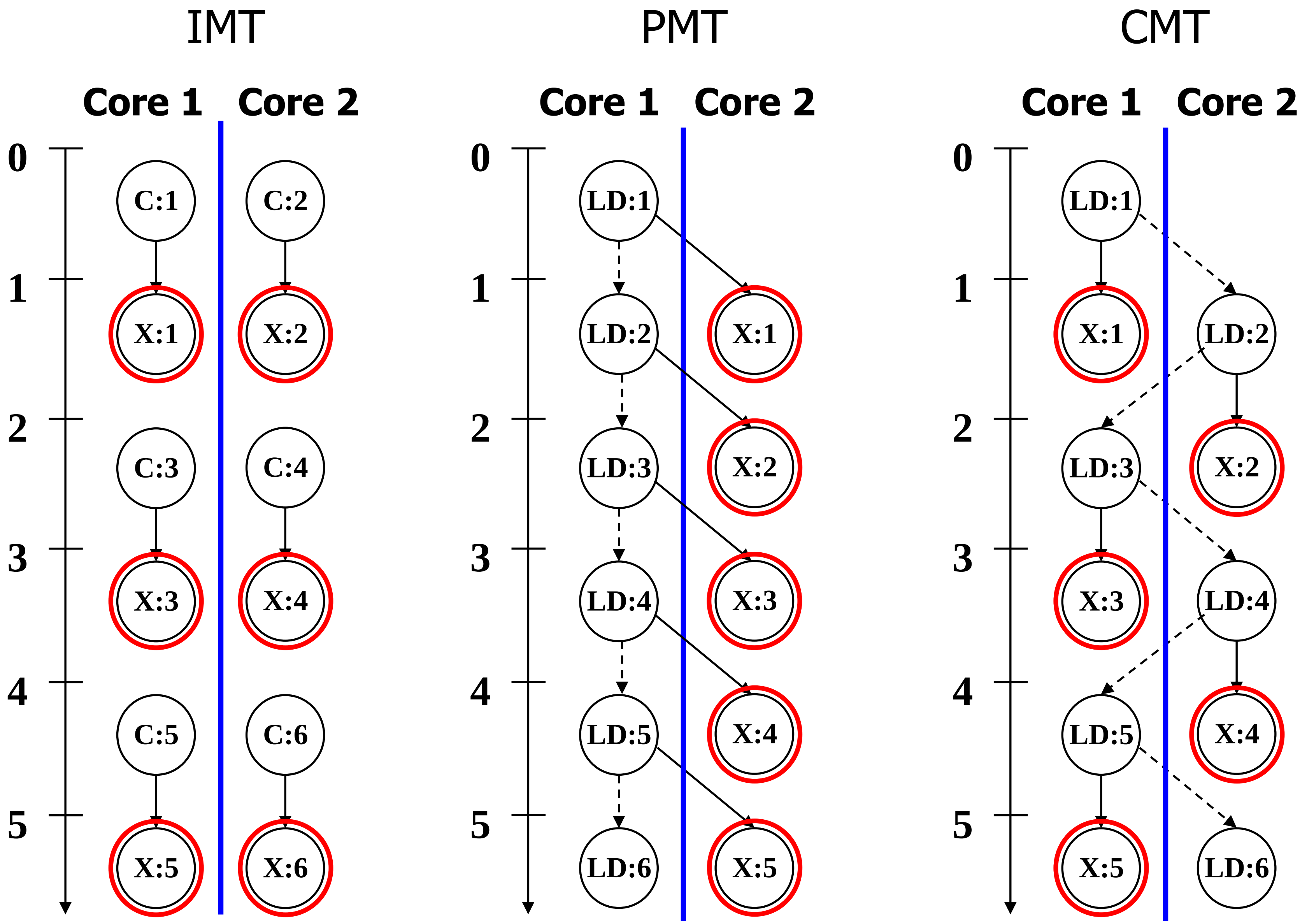
Example: DSWP **[PACT 2004]**



Comparison: IMT, PMT, CMT

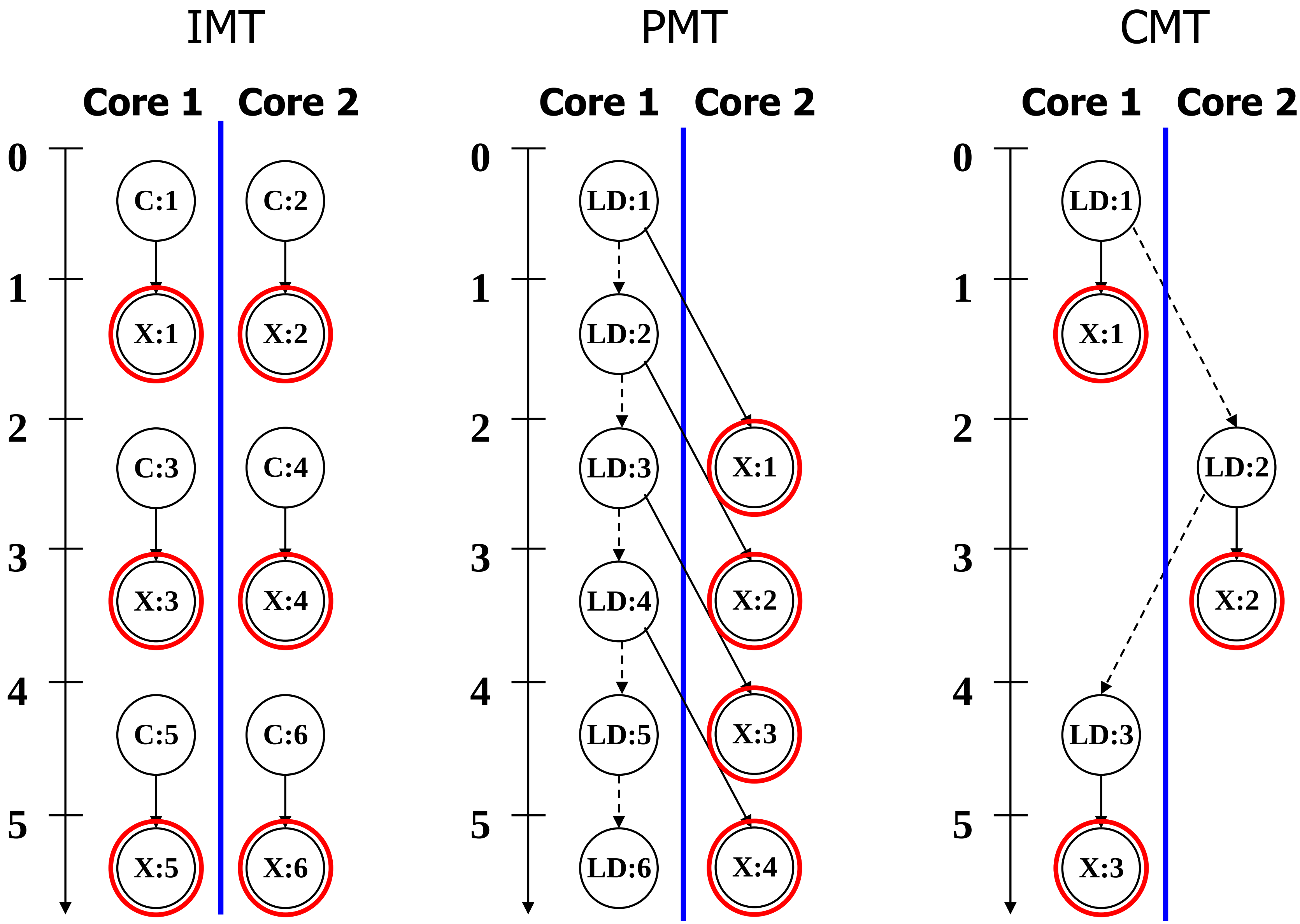


Comparison: IMT, PMT, CMT



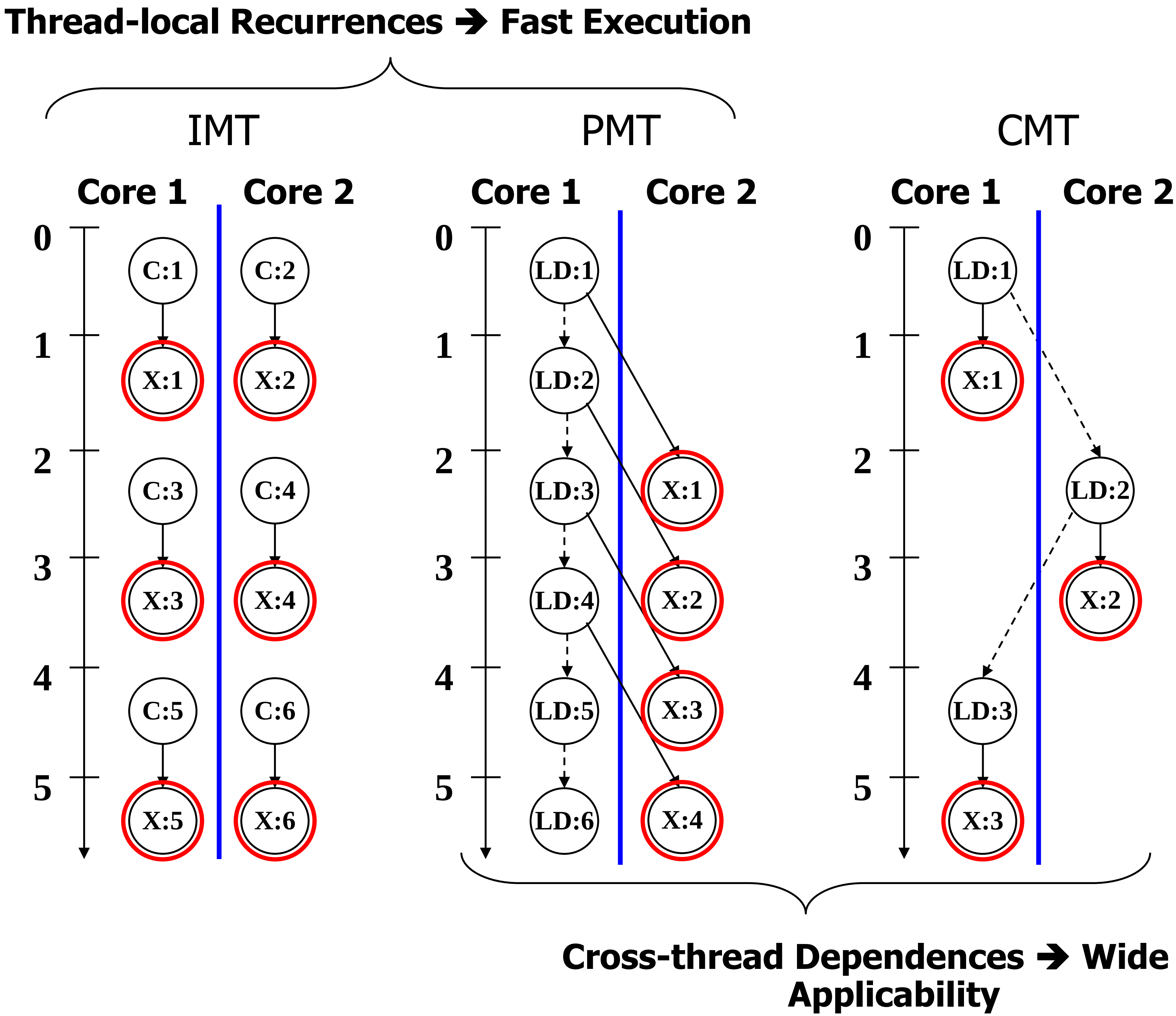
lat(comm) = 1: 1 iter/cycle 1 iter/cycle 1 iter/cycle

Comparison: IMT, PMT, CMT

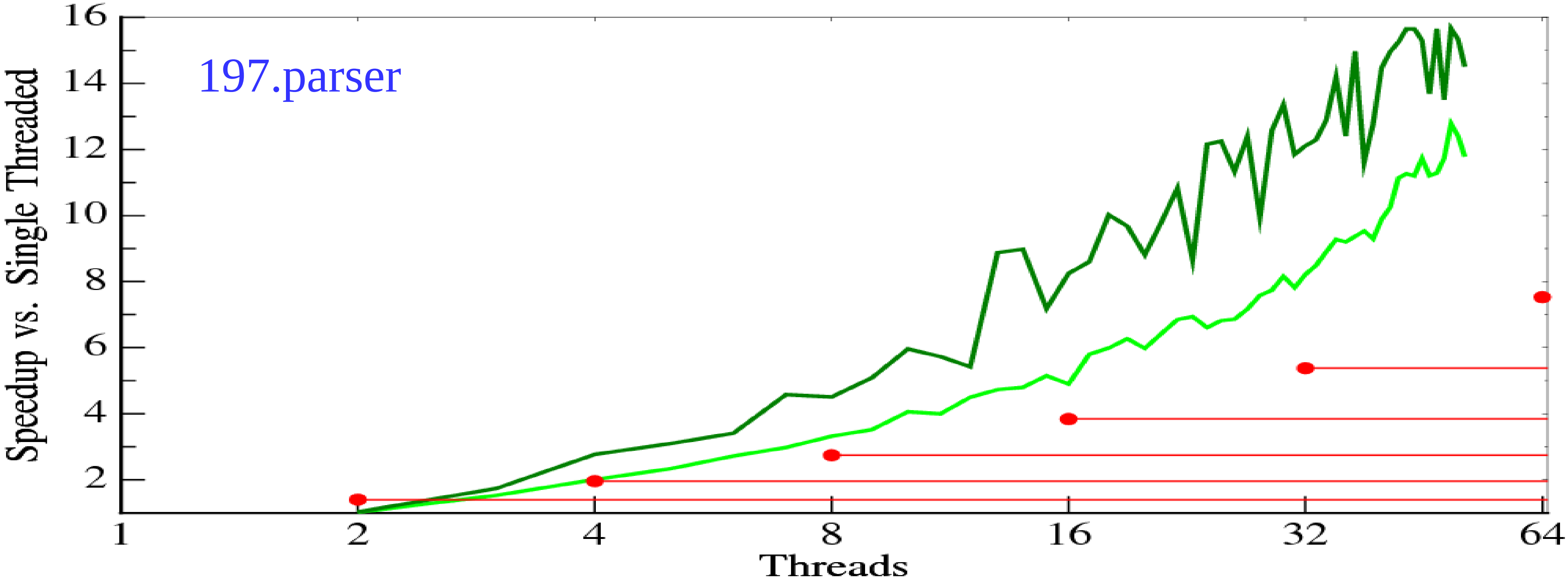


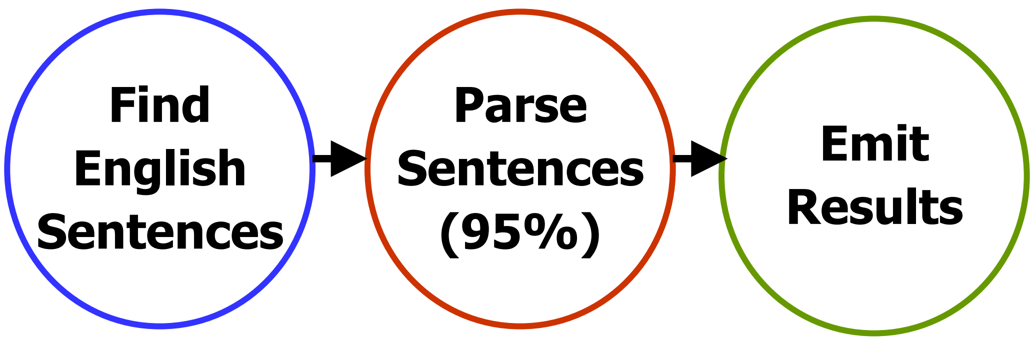
lat(comm) = 1:	1 iter/cycle	1 iter/cycle	1 iter/cycle
lat(comm) = 2:	1 iter/cycle	1 iter/cycle	0.5 iter/cycle

Comparison: IMT, PMT, CMT

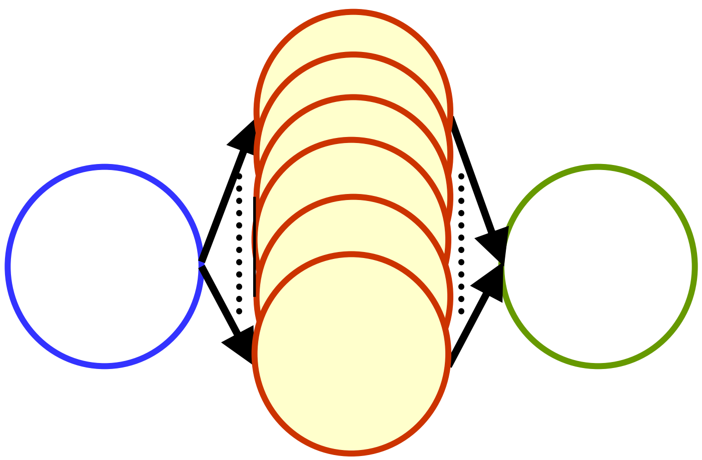


Our Objective: Automatic Extraction of Pipeline Parallelism using DSWP





Decoupled Software Pipelining



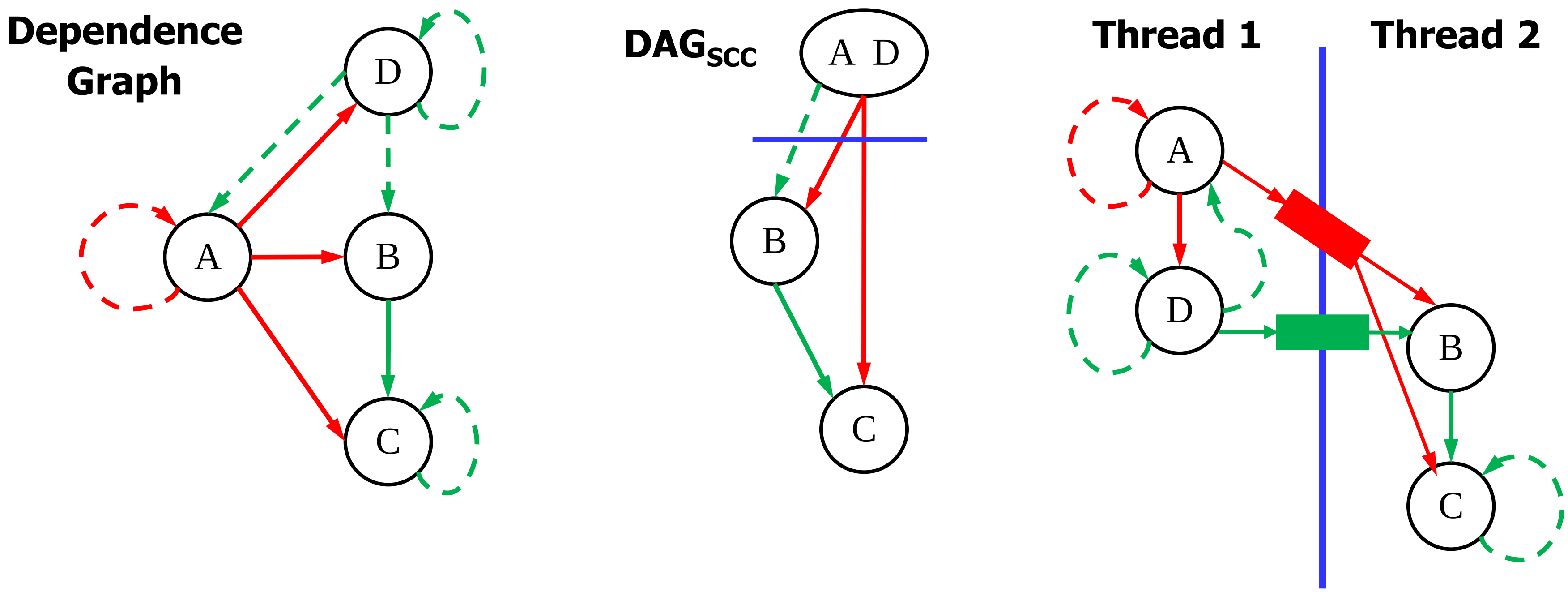
PS-DSWP (Spec DOALL Middle Stage)

Decoupled Software Pipelining

Decoupled Software Pipelining (DSWP)

[MICRO 2005]

```
A:  while(node)
B:    ncost = doit(node);
C:    cost += ncost;
D:    node = node->next;
```



register

control

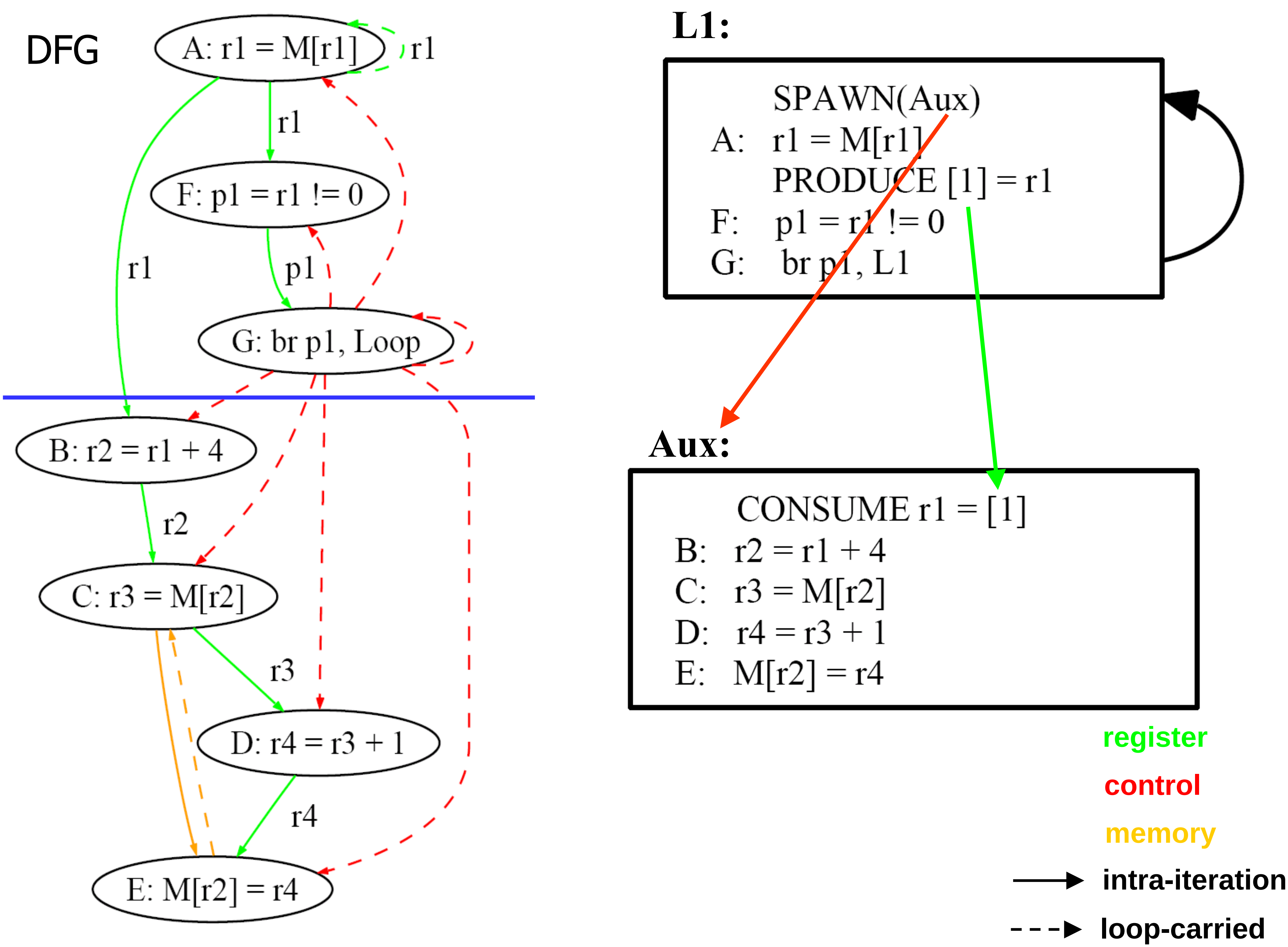
intra-iteration

loop-carried

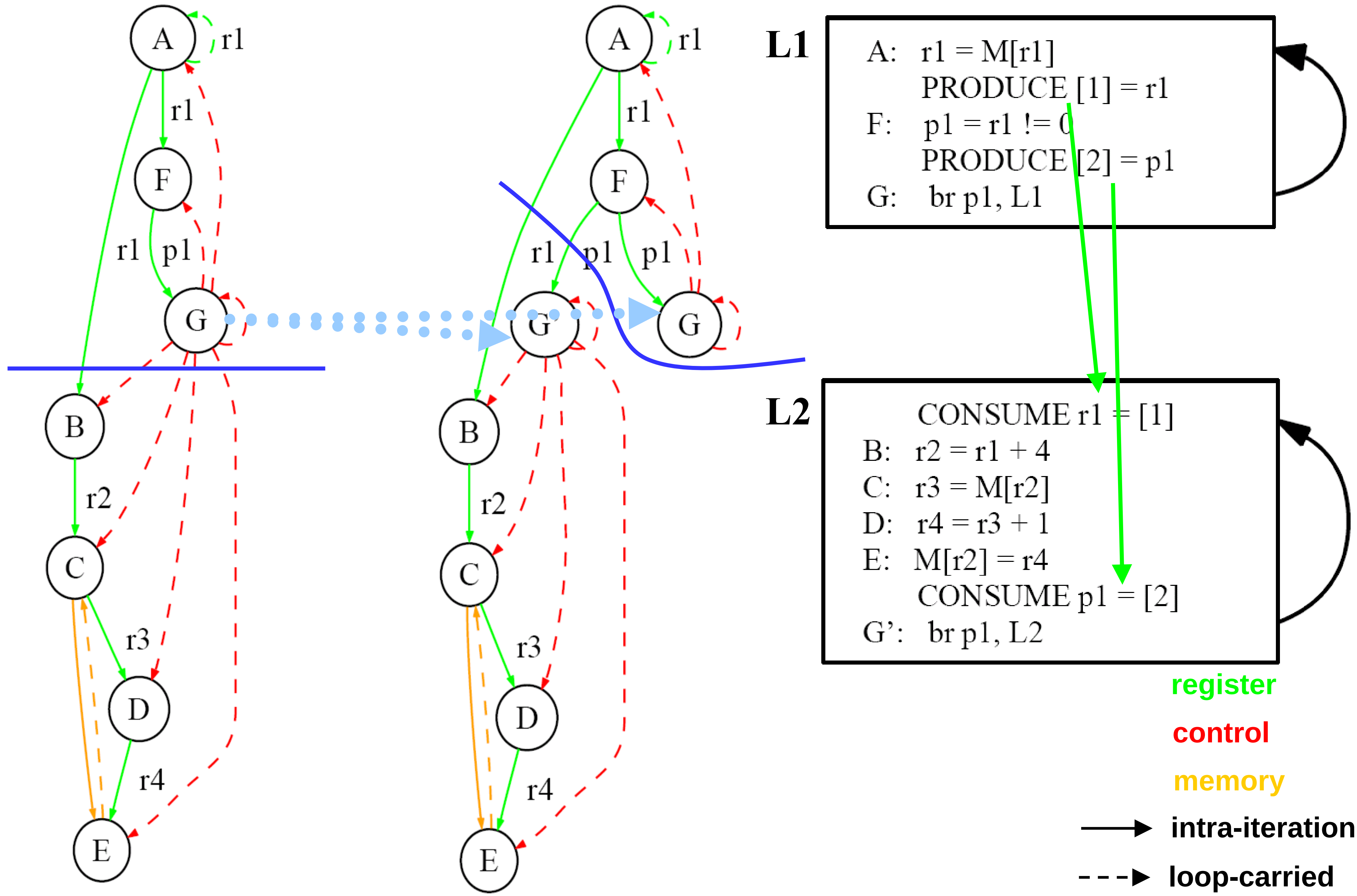
communication queue

Inter-thread communication latency is a one-time cost

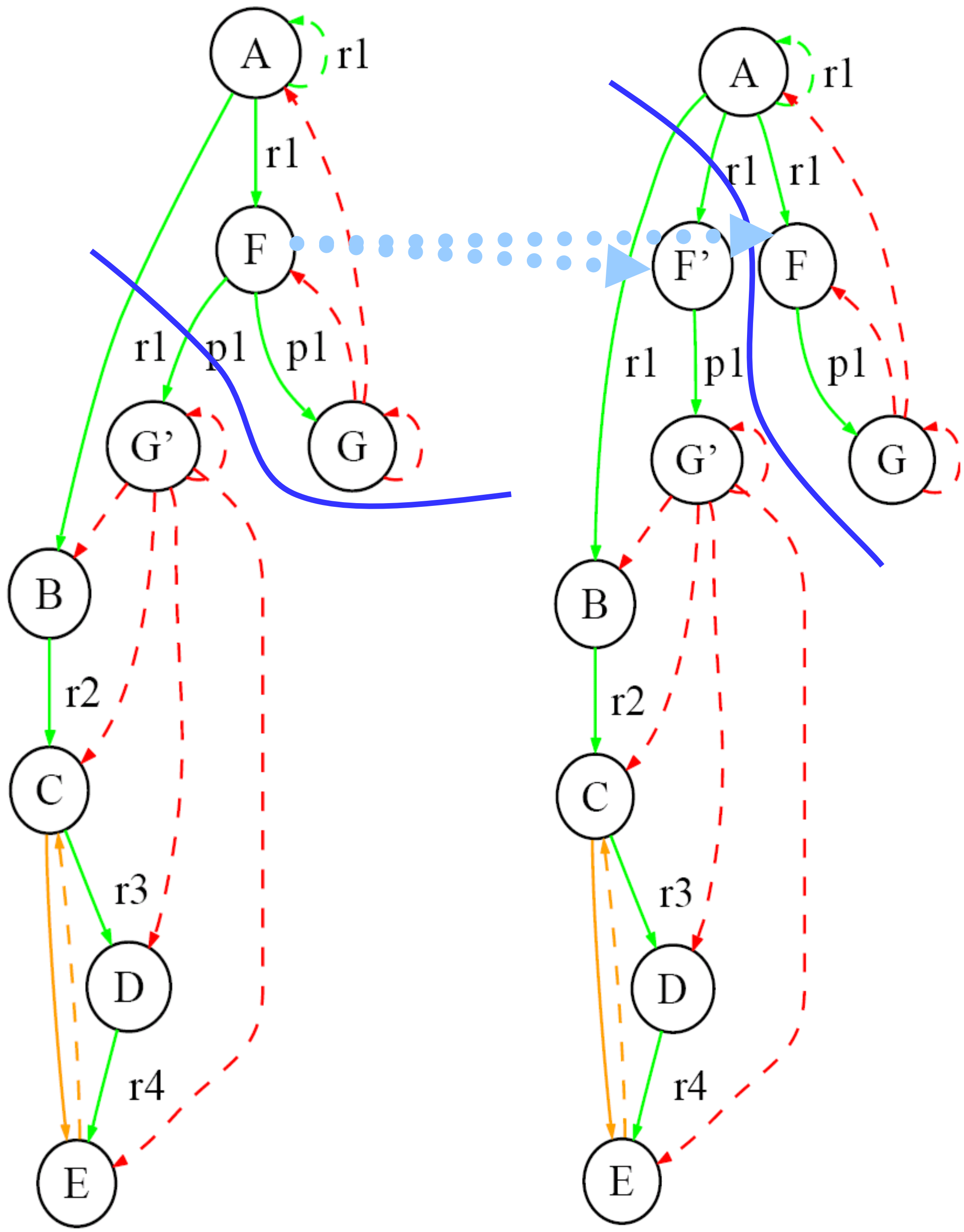
Implementing DSWP



Optimization: Node Splitting
To Eliminate Cross Thread Control



Optimization: Node Splitting To Reduce Communication



L1

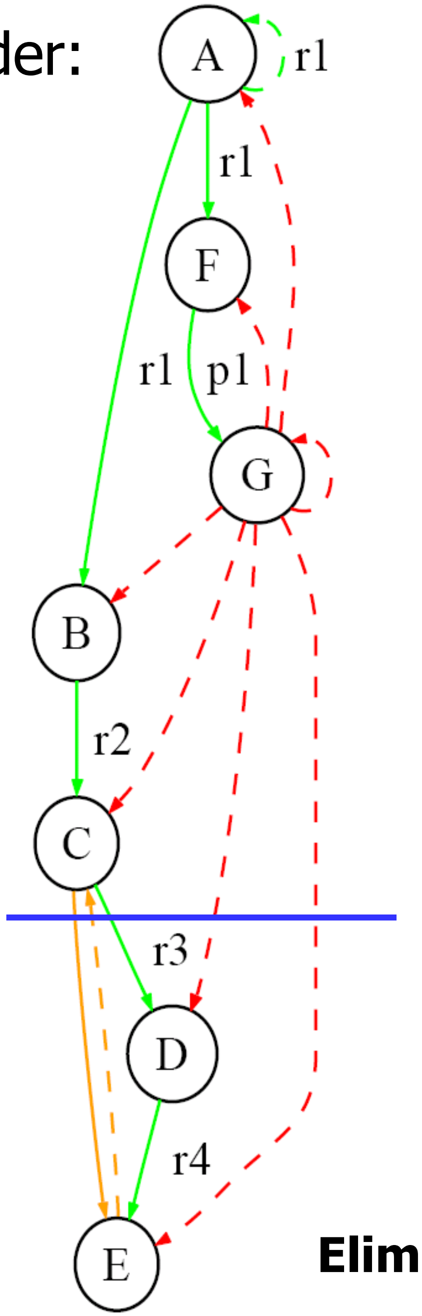
```
A:   r1 = M[r1]
     PRODUCE [1] = r1
F:   p1 = r1 != 0
G:   br p1, L1
```

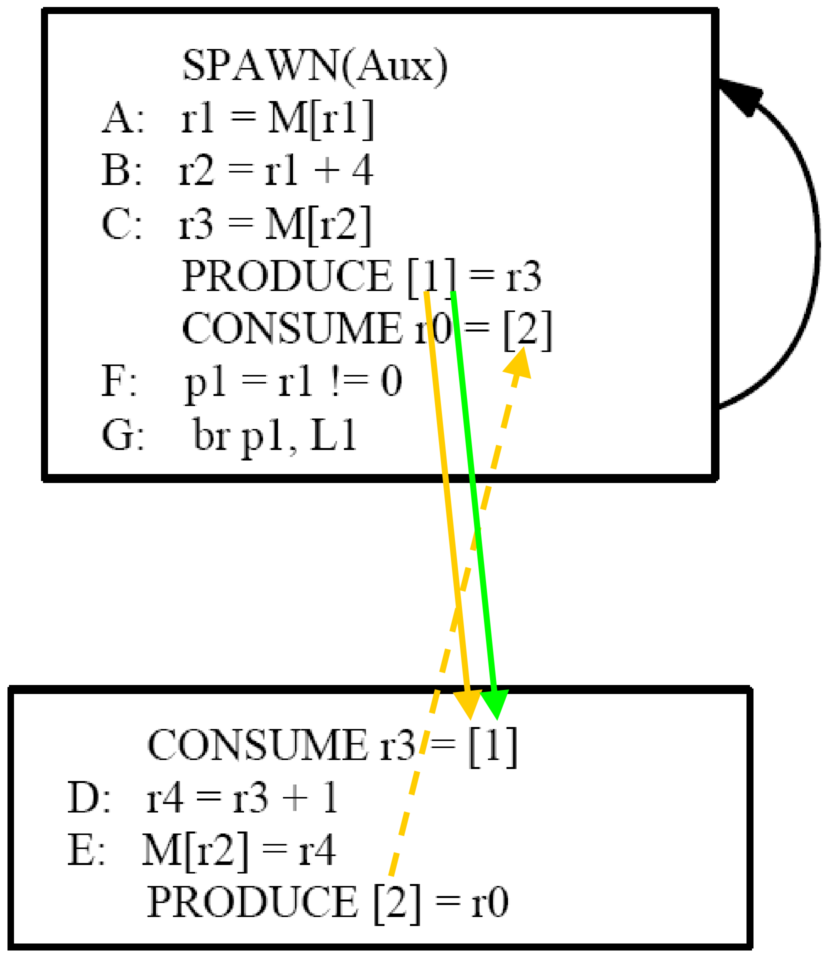
L2

```
     CONSUME r1 = [1]
B:   r2 = r1 + 4
C:   r3 = M[r2]
D:   r4 = r3 + 1
E:   M[r2] = r4
F':  p1 = r1 != 0
G':  br p1, L2
```

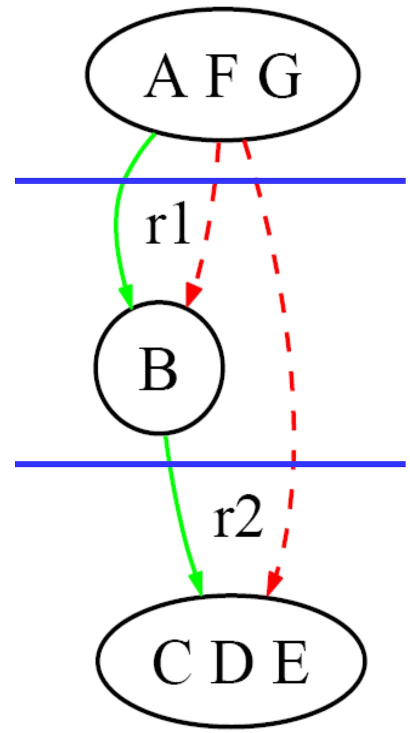
Constraint: Strongly Connected Components

Consider:





Solution: DAG_{SCC}



Eliminates pipelined/decoupled property

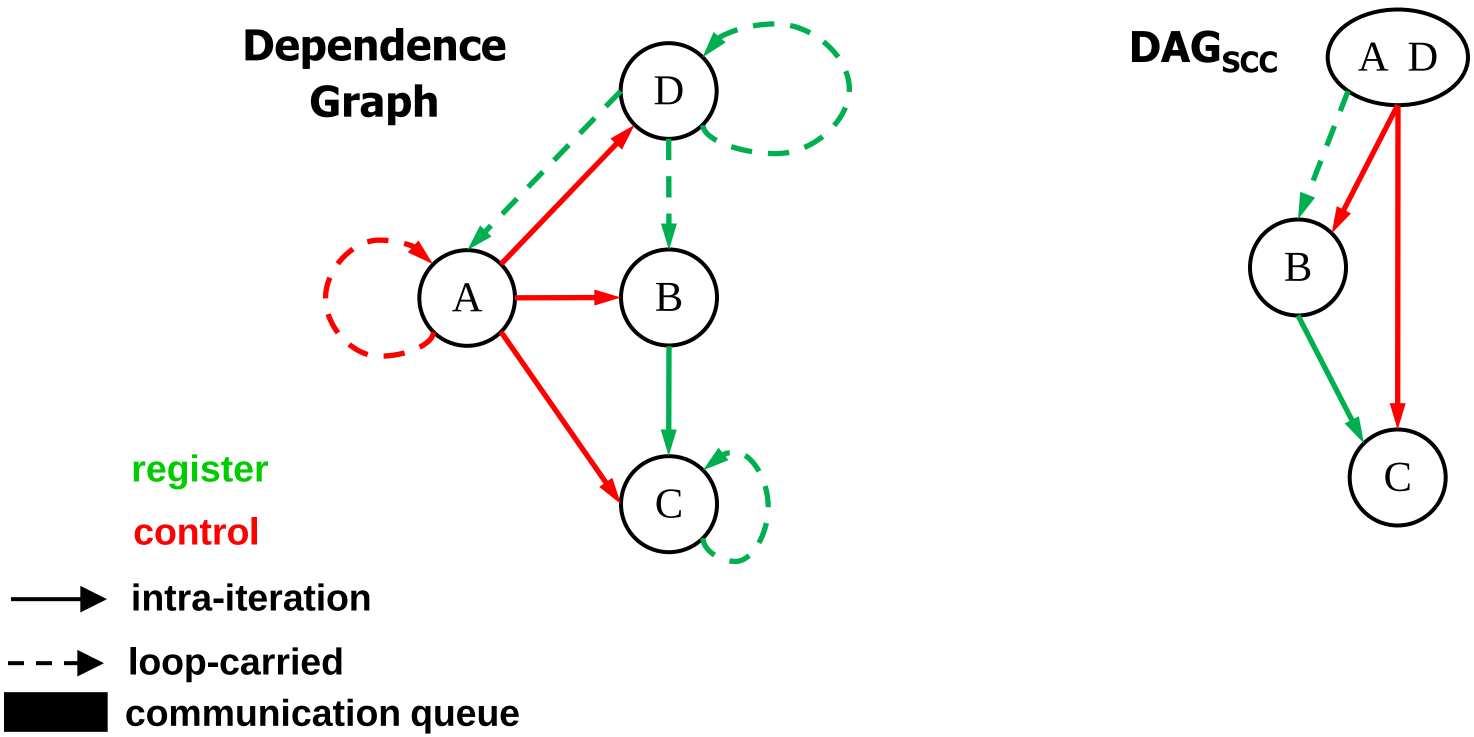
register
control
memory
intra-iteration
loop-carried

2 Extensions to the Basic Transformation

- Speculation
  - Break statistically unlikely dependences
  - Form better-balanced pipelines
- Parallel Stages
  - Execute multiple copies of certain "large" stages
  - Stages that contain inner loops perfect candidates

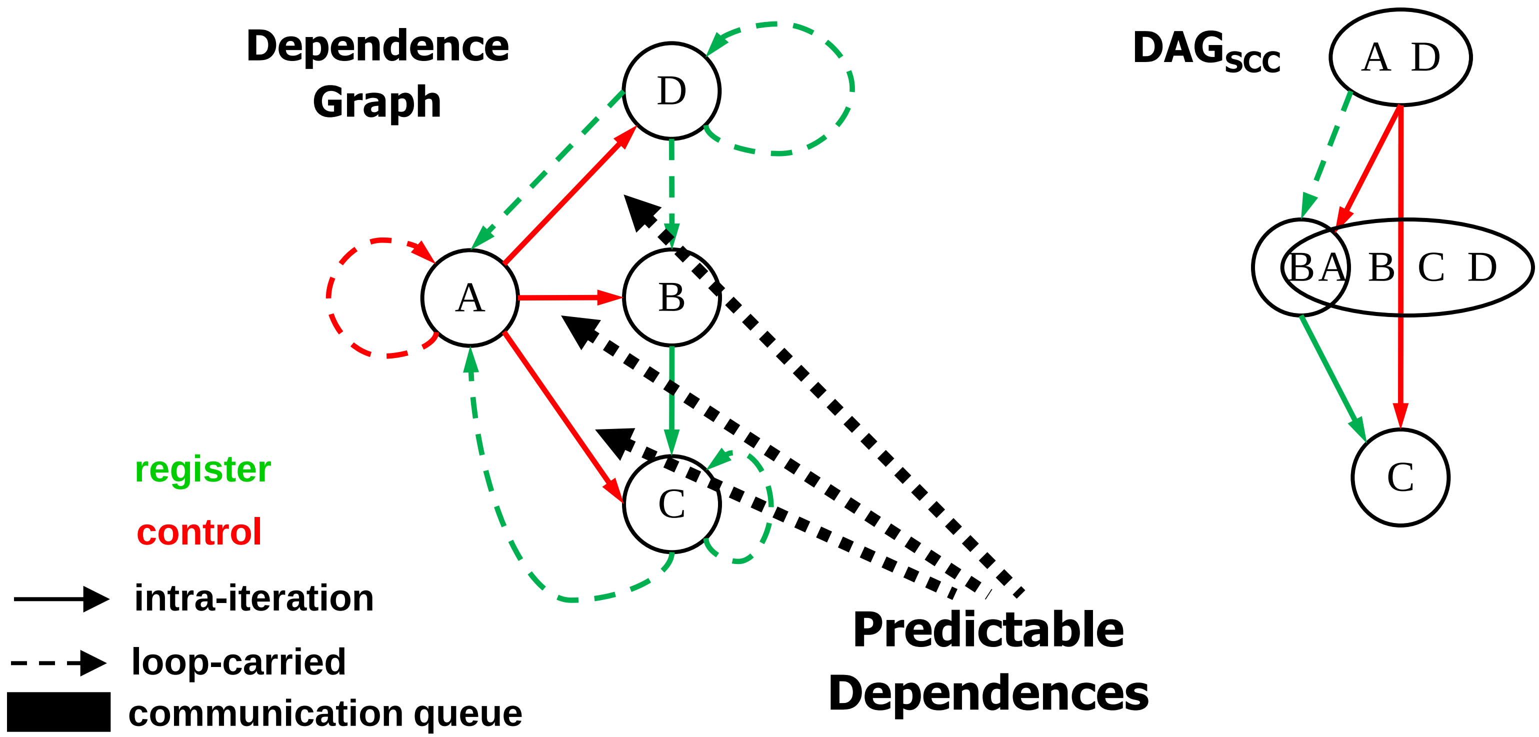
Why Speculation?

```
A: while(node)
B:    ncost = doit(node);
C:    cost += ncost;
D:    node = node->next;
```



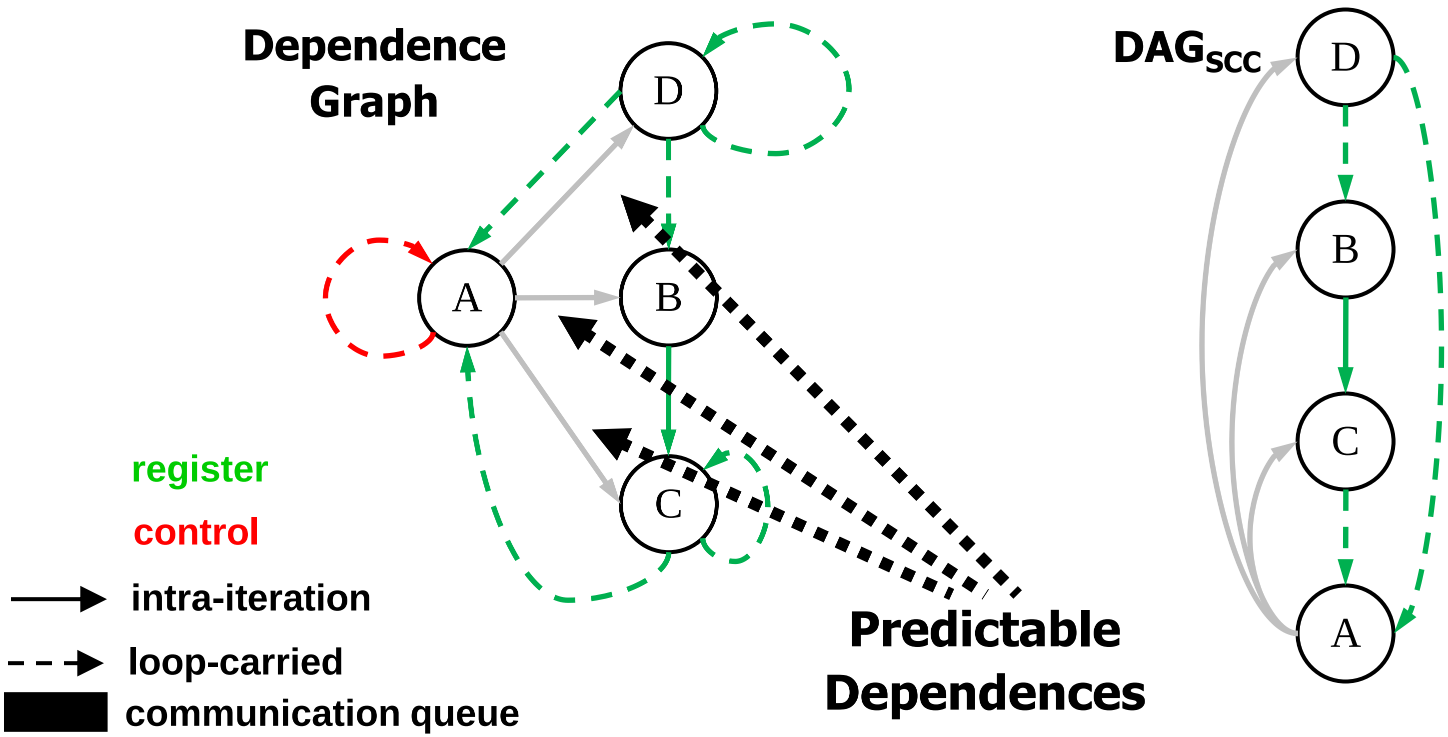
Why Speculation?

```
A: while(cost < T && node)
B:    ncost = doit(node);
C:    cost += ncost;
D:    node = node->next;
```

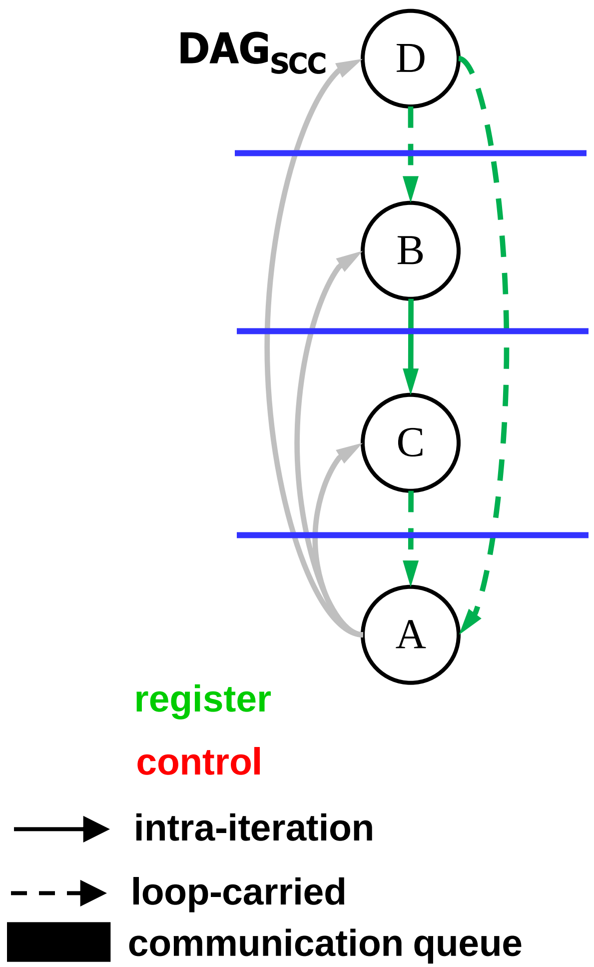


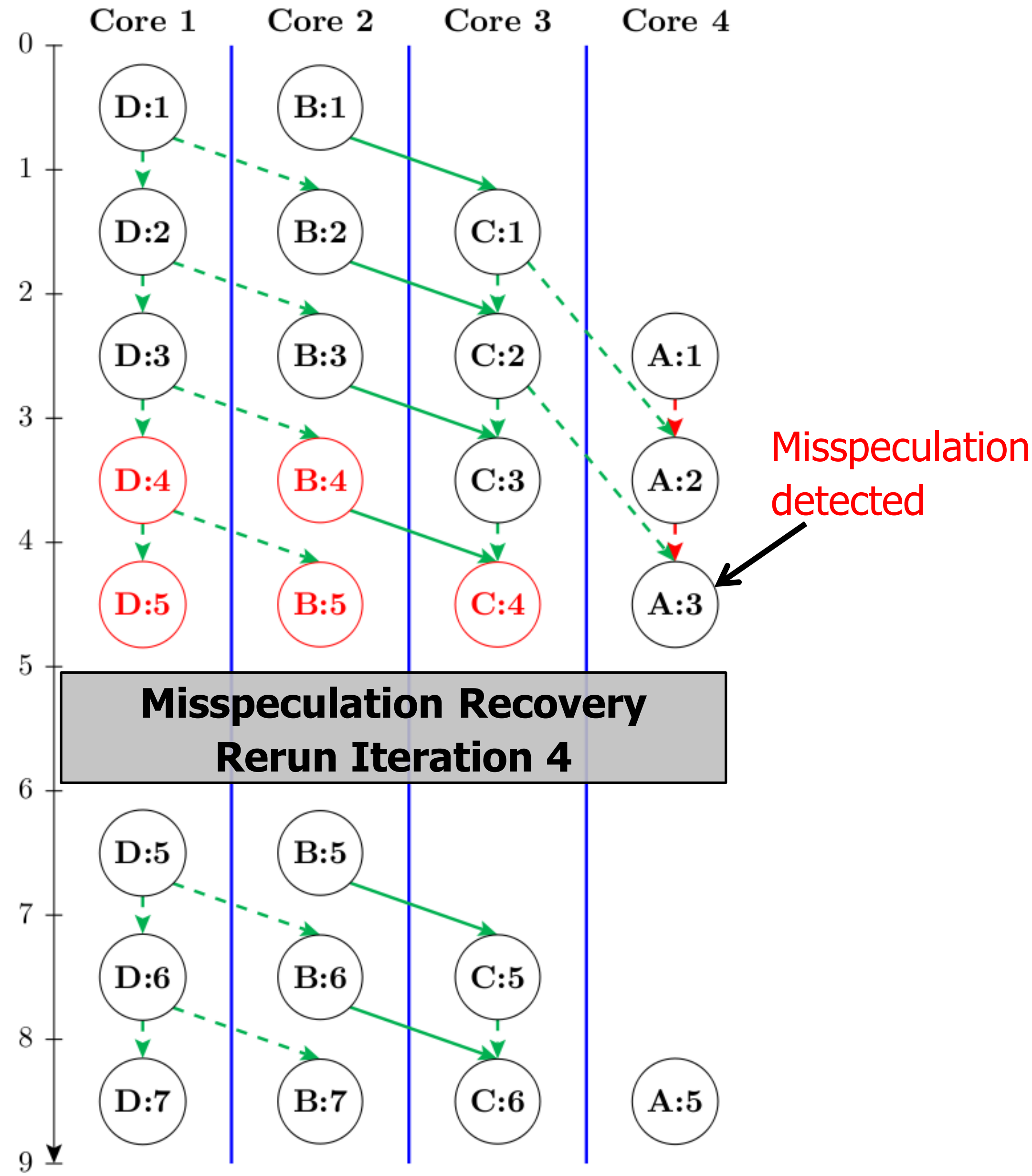
Why Speculation?

```
A: while(cost < T && node)
B:    ncost = doit(node);
C:    cost += ncost;
D:    node = node->next;
```

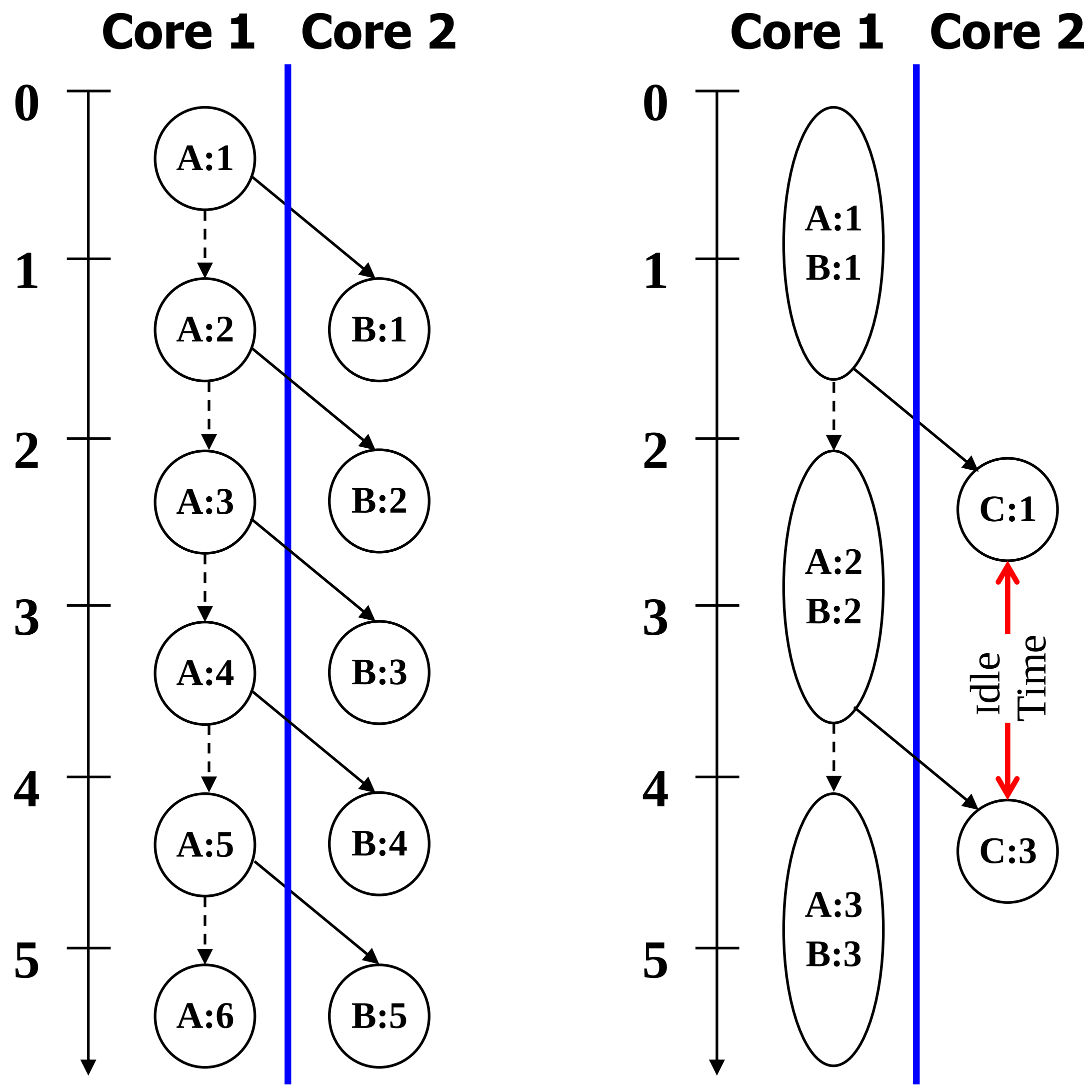


Execution Paradigm





Understanding PMT Performance



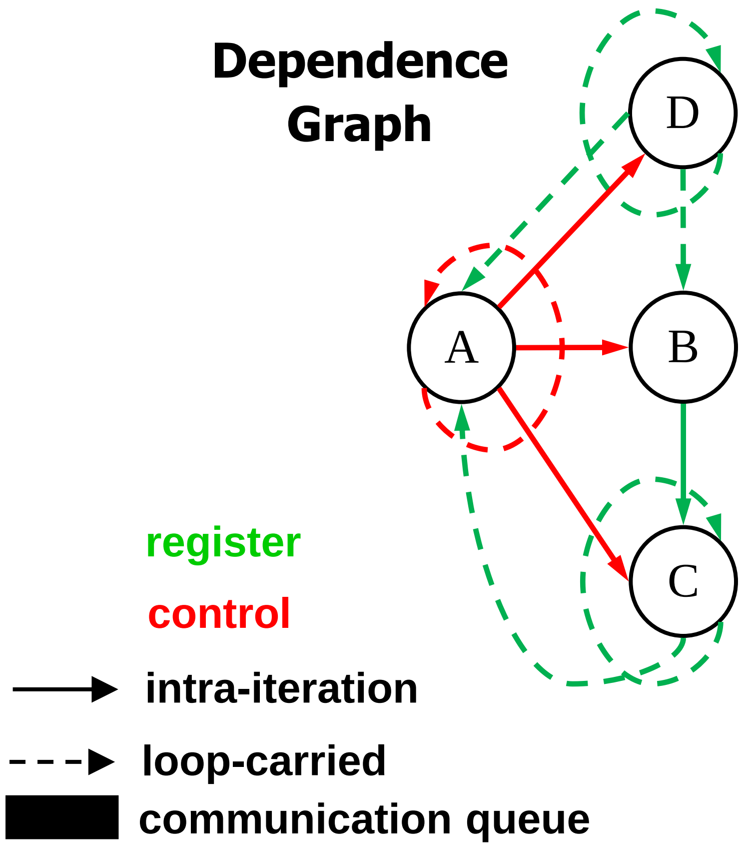
$$T \propto \max(t_i)$$

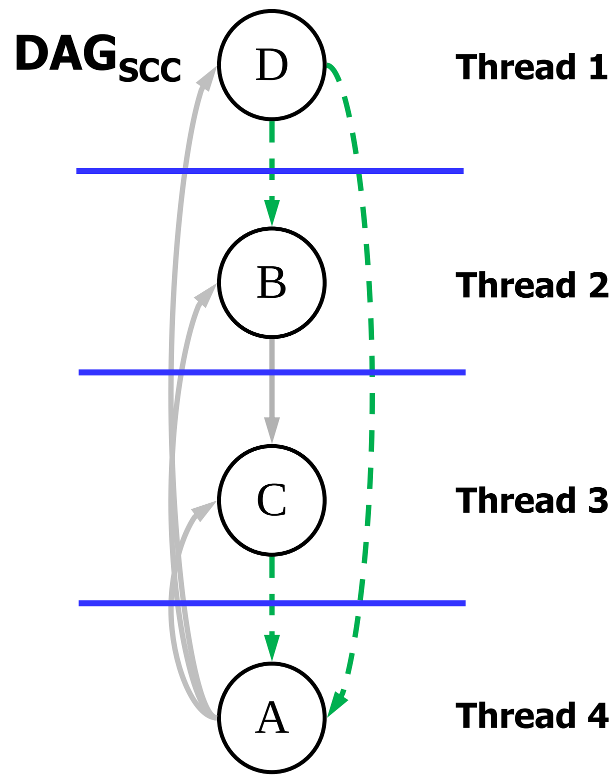
1. **Rate t_i is at least as large as the longest dependence recurrence.**
2. **NP-hard to find longest recurrence.**
3. **Large loops make problem difficult in practice.**

Slowest thread:	1 cycle/iter	2 cycle/iter
Iteration Rate:	1 iter/cycle	0.5 iter/cycle

Selecting Dependences To Speculate

```
A: while(cost < T && node)
B:    ncost = doit(node);
C:    cost += ncost;
D:    node = node->next;
```





Detecting Misspeculation

Thread 1

```
A^1: while(consume(4))
D :    node = node->next
       produce({0,1},node);
```

Thread 2

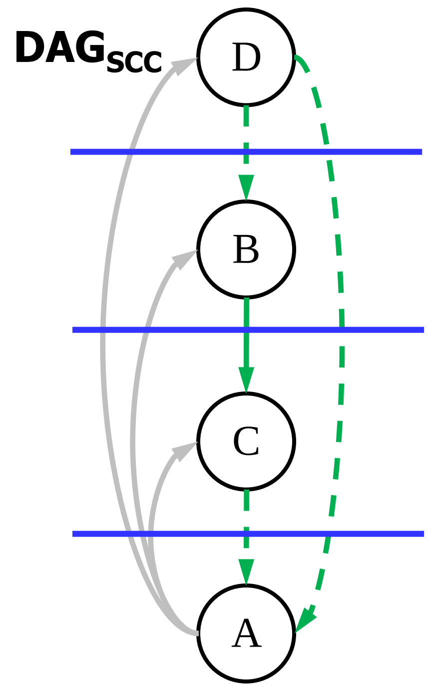
```
A^2: while(consume(5))
B :    ncost = doit(node);
       produce(2,ncost);
D^2:   node = consume(0);
```

Thread 3

```
A^3: while(consume(6))
B^3:   ncost = consume(2);
C :    cost += ncost;
       produce(3,cost);
```

Thread 4

```
A : while(cost < T && node)
B^4:   cost = consume(3);
C^4:   node = consume(1);
       produce({4,5,6},cost < T
                       && node);
```



Detecting Misspeculation

Thread 1

```
A^1: while(TRUE)
D  :    node = node->next
        produce({0,1},node);
```

Thread 2

```
A^2: while(TRUE)
B  :    ncost = doit(node);
        produce(2,ncost);
D^2:    node = consume(0);
```

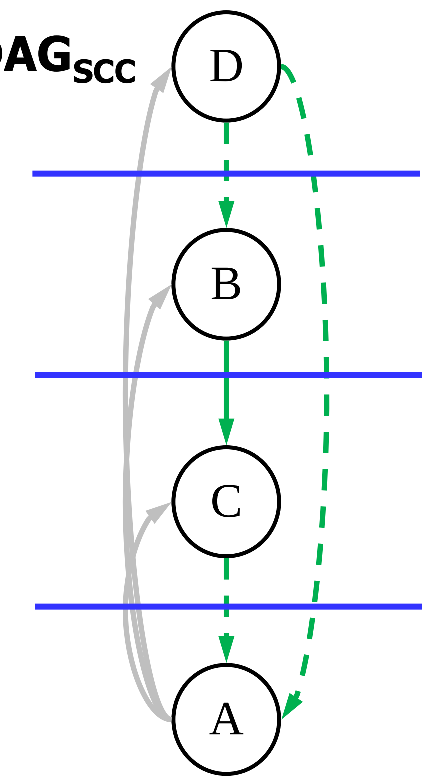
Thread 3

```
A^3: while(TRUE)
B^3:    ncost = consume(2);
C  :    cost += ncost;
        produce(3,cost);
```

Thread 4

```
A  : while(cost < T && node)
B^4:    cost = consume(3);
C^4:    node = consume(1);
        produce({4,5,6},cost < T
                          && node);
```

DAG$_{SCC}$



Detecting Misspeculation

Thread 1

```
A^1: while(TRUE)
D  :   node = node->next
       produce({0,1},node);
```

Thread 2

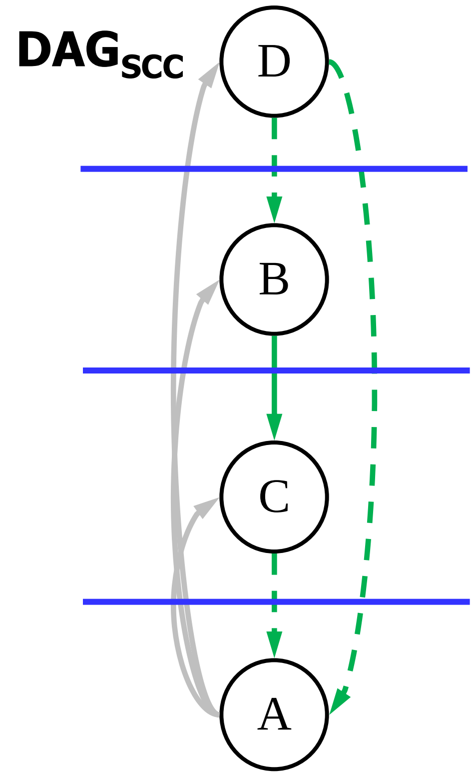
```
A^2: while(TRUE)
B  :   ncost = doit(node);
       produce(2,ncost);
D^2:   node = consume(0);
```

Thread 3

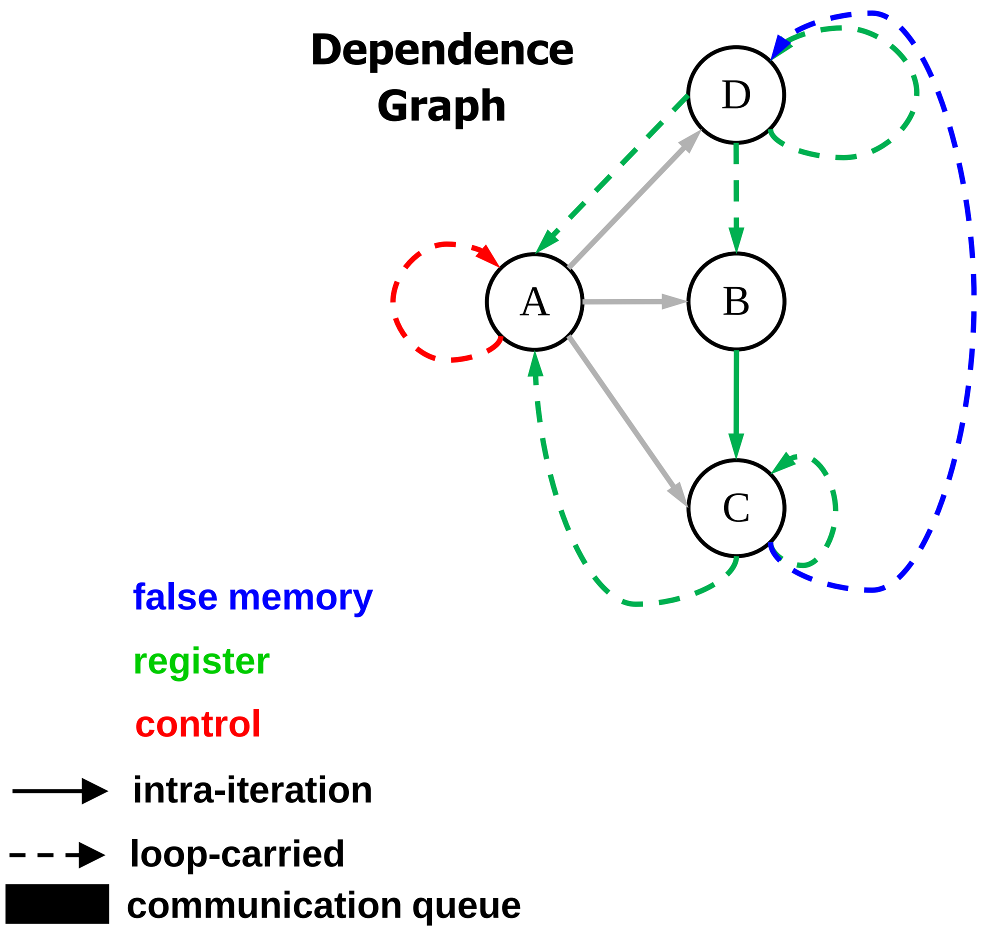
```
A^3: while(TRUE)
B^3:   ncost = consume(2);
C  :   cost += ncost;
       produce(3,cost);
```

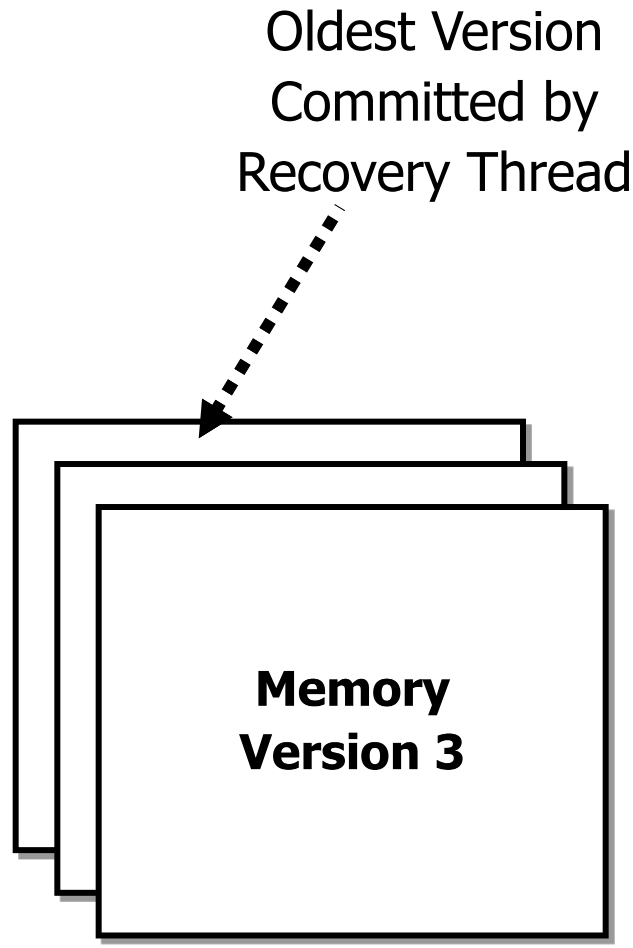
Thread 4

```
A  : while(cost < T && node)
B^4:   cost = consume(3);
C^4:   node = consume(1);
       if(!(cost < T && node))
         FLAG_MISSPEC();
```

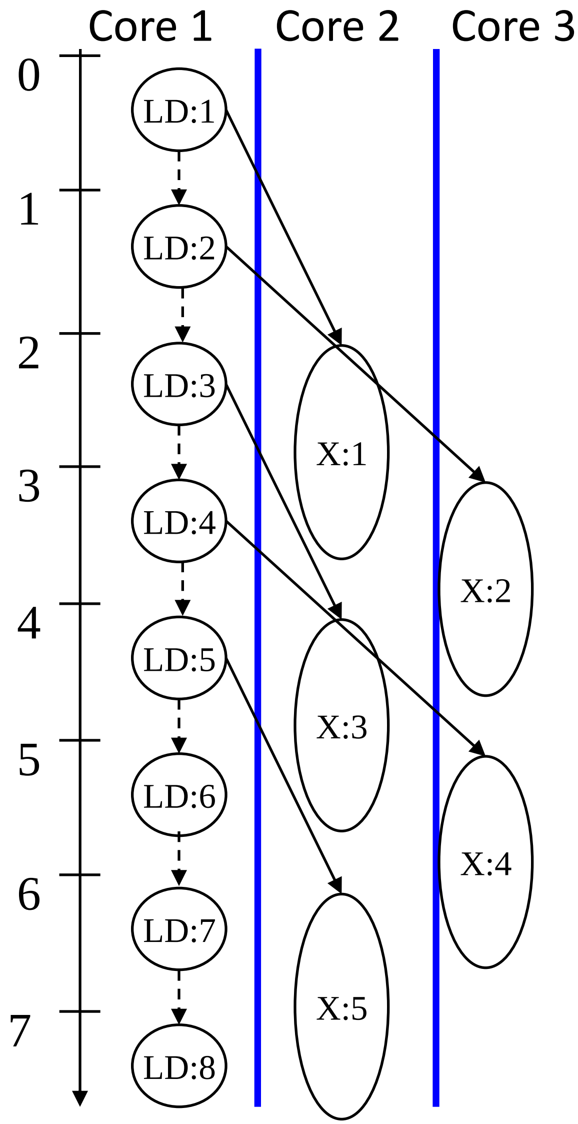


Breaking False Memory Dependences





Adding Parallel Stages to DSWP




```
while(ptr = ptr->next)      // LD
  ptr->val = ptr->val + 1; // X
```

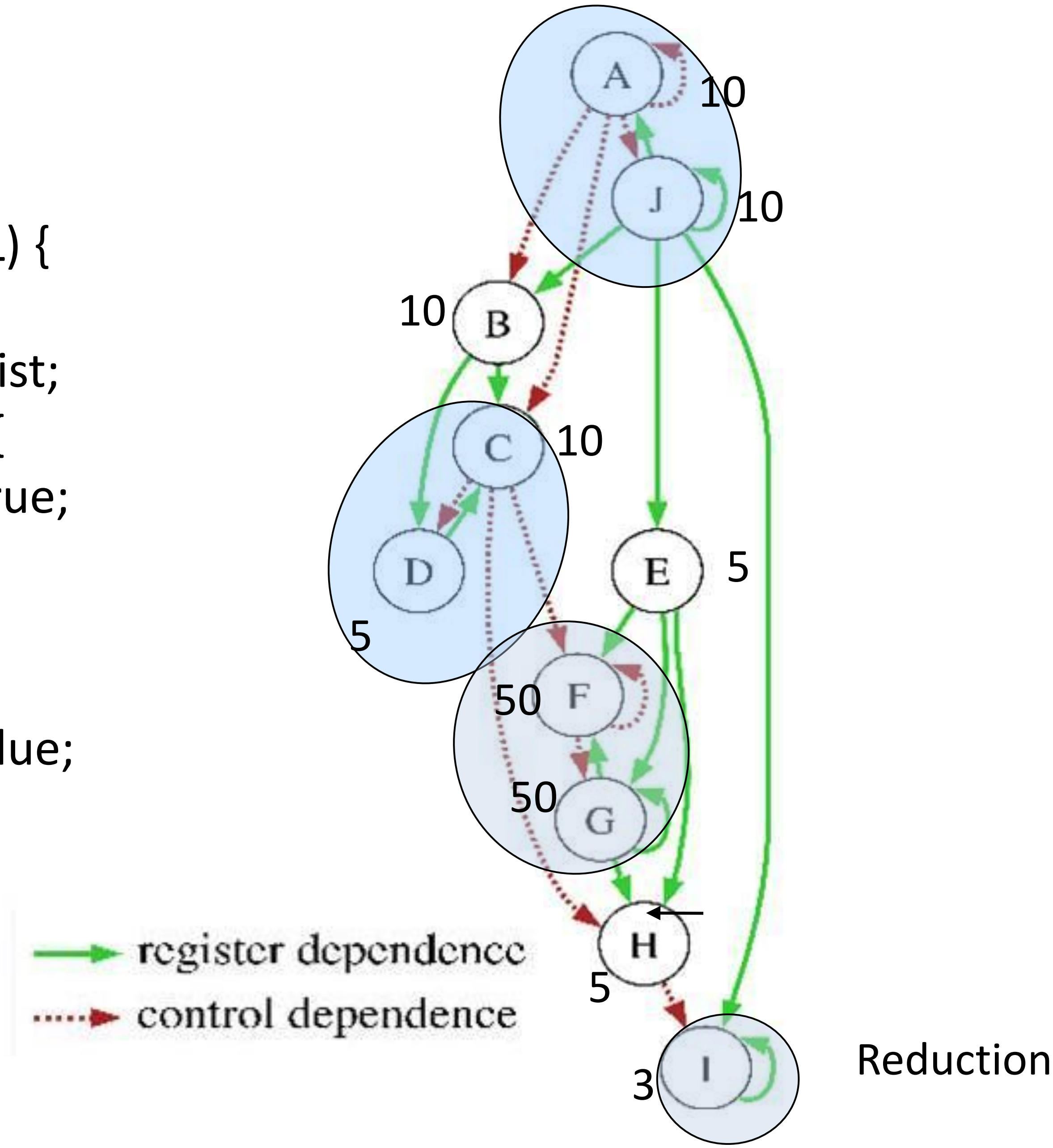
LD = 1 cycle
X = 2 cycles

Comm. Latency = 2 cycles

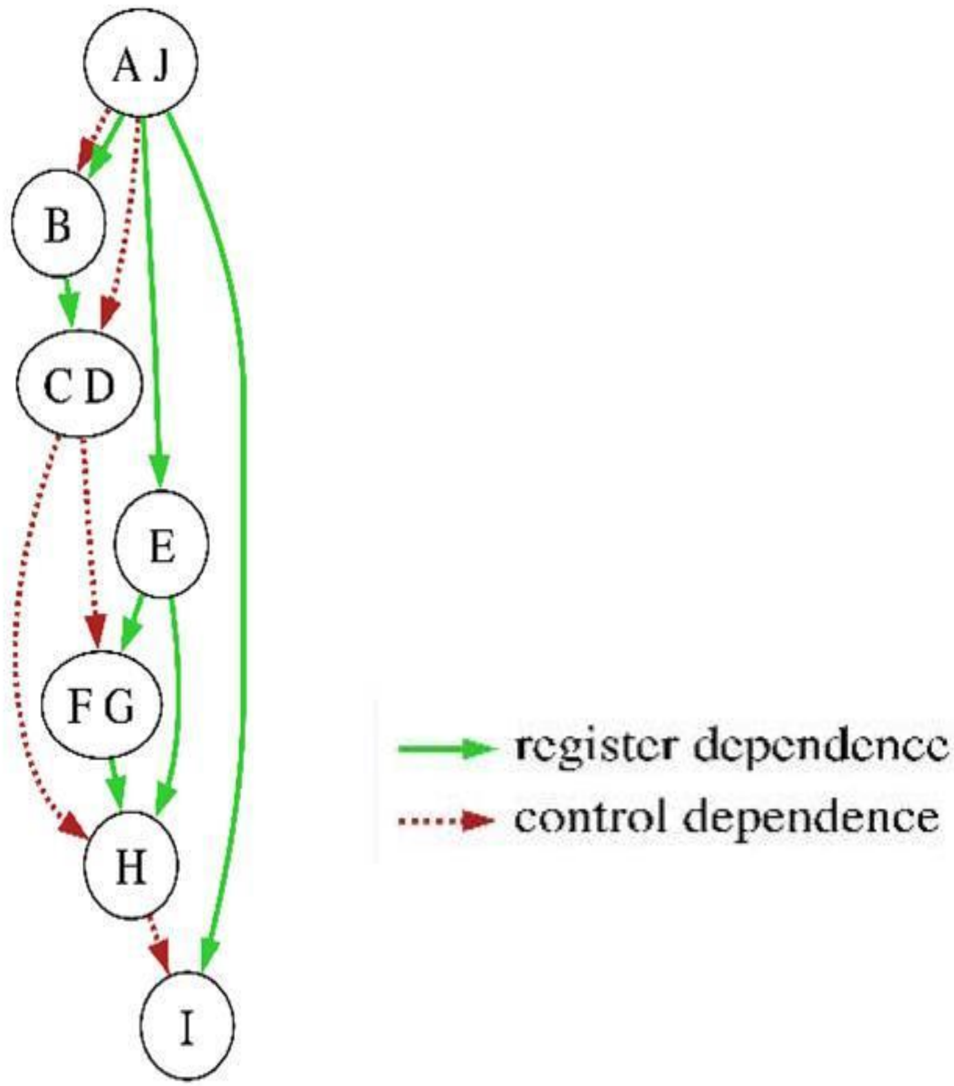
Throughput
DSWP: 1/2 iteration/cycle
DOACROSS: 1/2 iteration/cycle
PS-DSWP: 1 iteration/cycle

Thread Partitioning

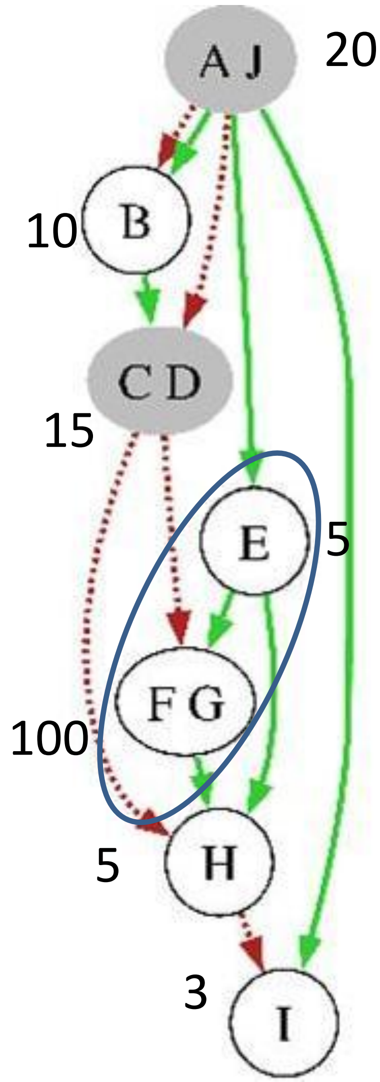
```
       p = list;
      sum = 0;
A:   while (p != NULL) {
B:      id = p->id;
E:        q = p->inner_list;
C:      if (!visited[id]) {
D:         visited[id] = true;
F:        while (foo(q))
G:           q = q->next;
H:         if (q != NULL)
I:           sum += p->value;
       }
J:     p = p->next;
      }
```



Thread Partitioning: DAG_{SCC}

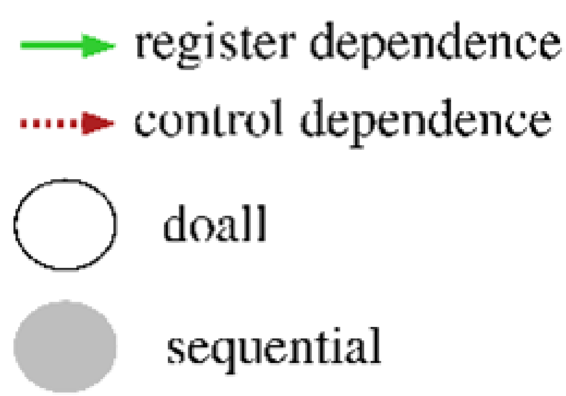


Thread Partitioning

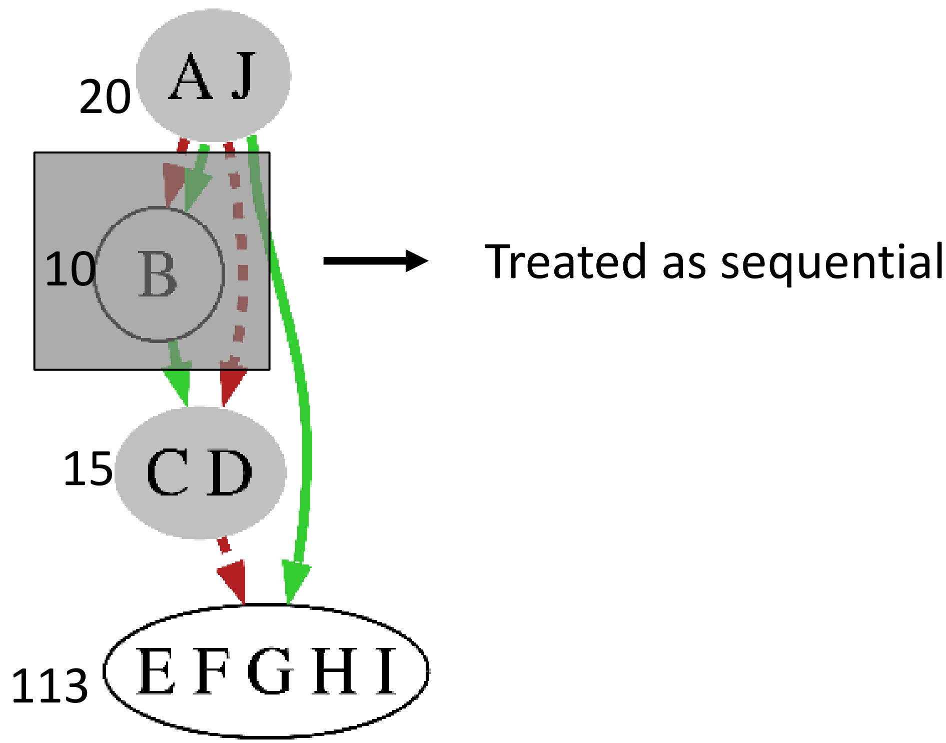


Merging Invariants

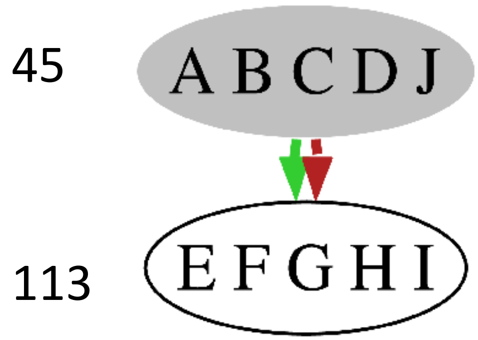
- No cycles
- No loop-carried dependence inside a doall node



Thread Partitioning



Thread Partitioning



- Modified MTCG[Ottoni, MICRO'05] to generate code from partition

Discussion Point 1 – Speculation

- How do you decide what dependences to speculate?
  - Look solely at profile data?
  - What about code structure?
- How do you manage speculation in a pipeline?
  - Traditional definition of a transaction is broken
  - Transaction execution spread out across multiple cores

Discussion Point 2 – Pipeline Structure

- When is a pipeline a good/bad choice for parallelization?

- Is pipelining good or bad for cache performance?
  - Is DOALL better/worse for cache?

- Can a pipeline be adjusted when the number of available cores increases/decreases?