EECS 583 – Class 16
Automatic Parallelization Via Decoupled Software Pipelining

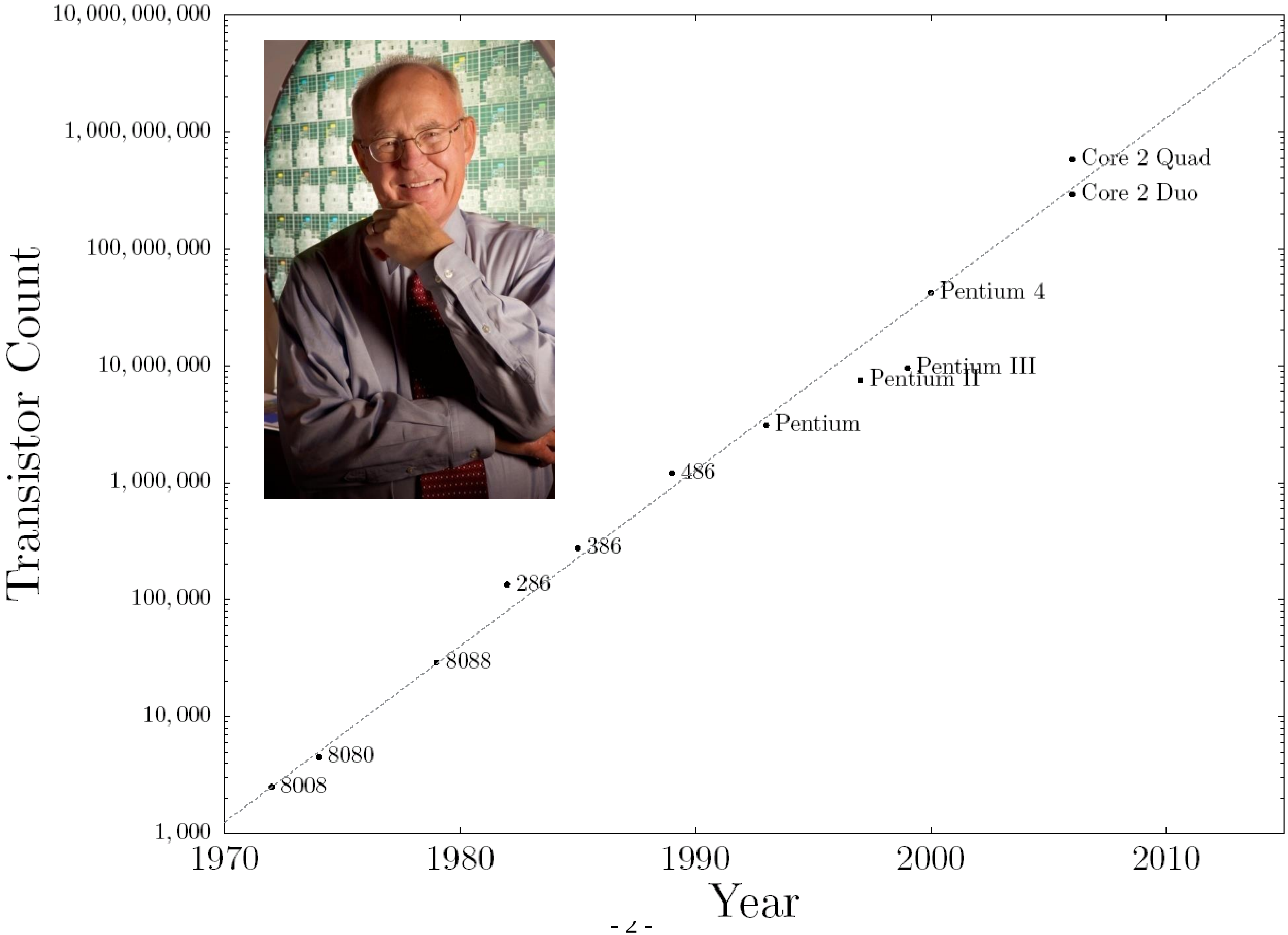
University of Michigan

November 7, 2018

Announcements + Reading Material

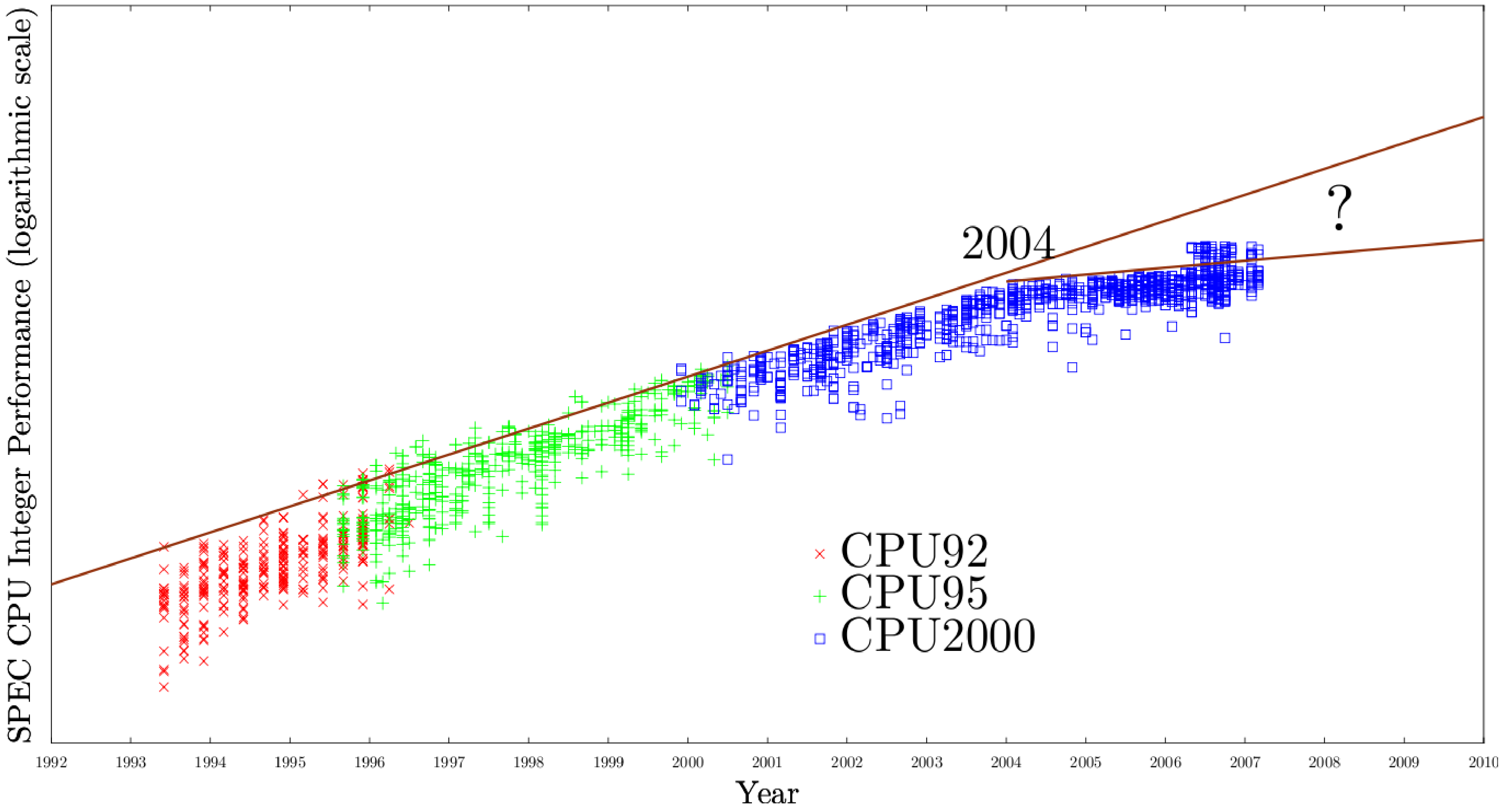
- Midterm exam – Next Wednesday in class
  - Starts at 10:40am sharp
  - Open book/notes
- No regular class next Monday
  - Group office hours in class in 2246 SRB
- Reminder – Sign up for paper presentation slot on my door if you haven't done so already
- Today's class reading
  - "Automatic Thread 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.
  - "Revisiting the Sequential Programming Model for Multi-Core," M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August, Proc 40th IEEE/ACM International Symposium on Microarchitecture, December 2007.

Moore’s Law



Source: Intel/Wikipedia

Single-Threaded Performance Not Improving



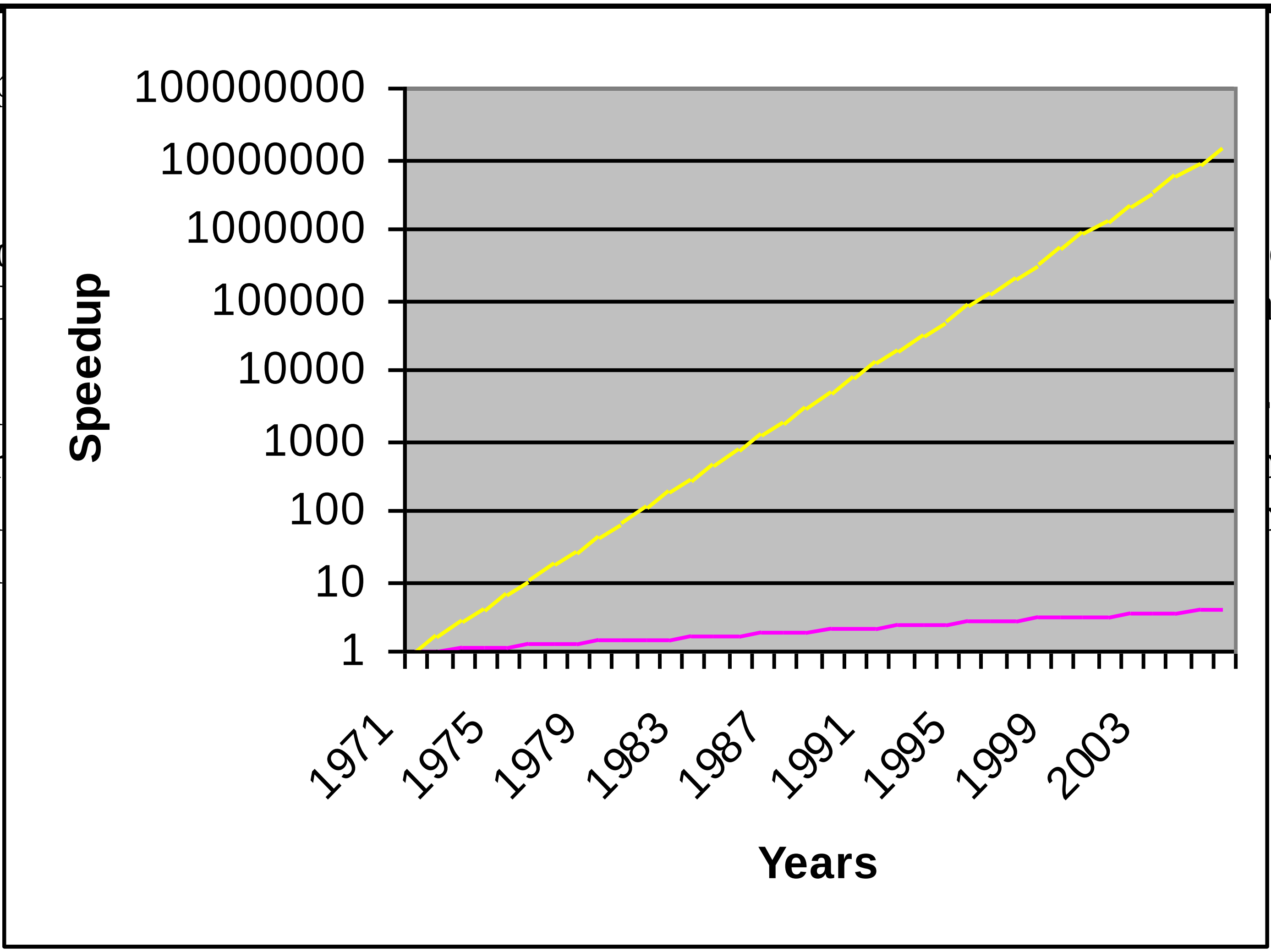
What about Parallel Programming? –or- What is Good About the Sequential Model?

- Sequential is easier
  - People think about programs sequentially
  - Simpler to write a sequential program
- Deterministic execution
  - Reproducing errors for debugging
  - Testing for correctness
- No concurrency bugs
  - Deadlock, livelock, atomicity violations
  - Locks are not composable
- Performance extraction
  - Sequential programs are portable
    - Are parallel programs? Ask GPU developers ☺
  - Performance debugging of sequential programs straight-forward

Compilers are the Answer? - Proebsting’s Law

- “Comp

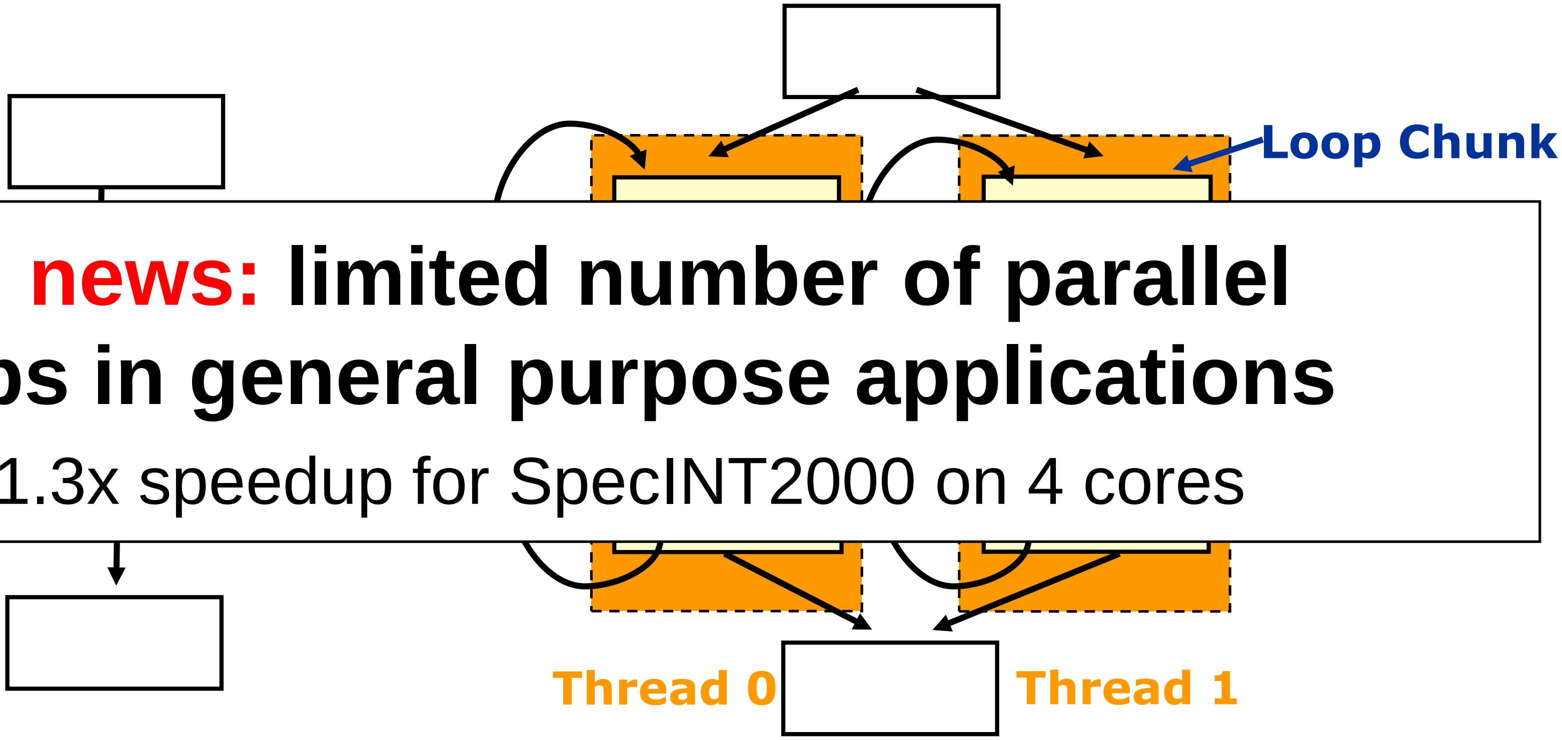
- Run y … optimizing
  compi … bled. The
  ratio o … ompiler
  optimi … atio is about
  4X for … piler
  optimi … piler
  optimi



Conclusion – Compilers not about performance!

Can We Convert Single-threaded Programs into Multi-threaded?

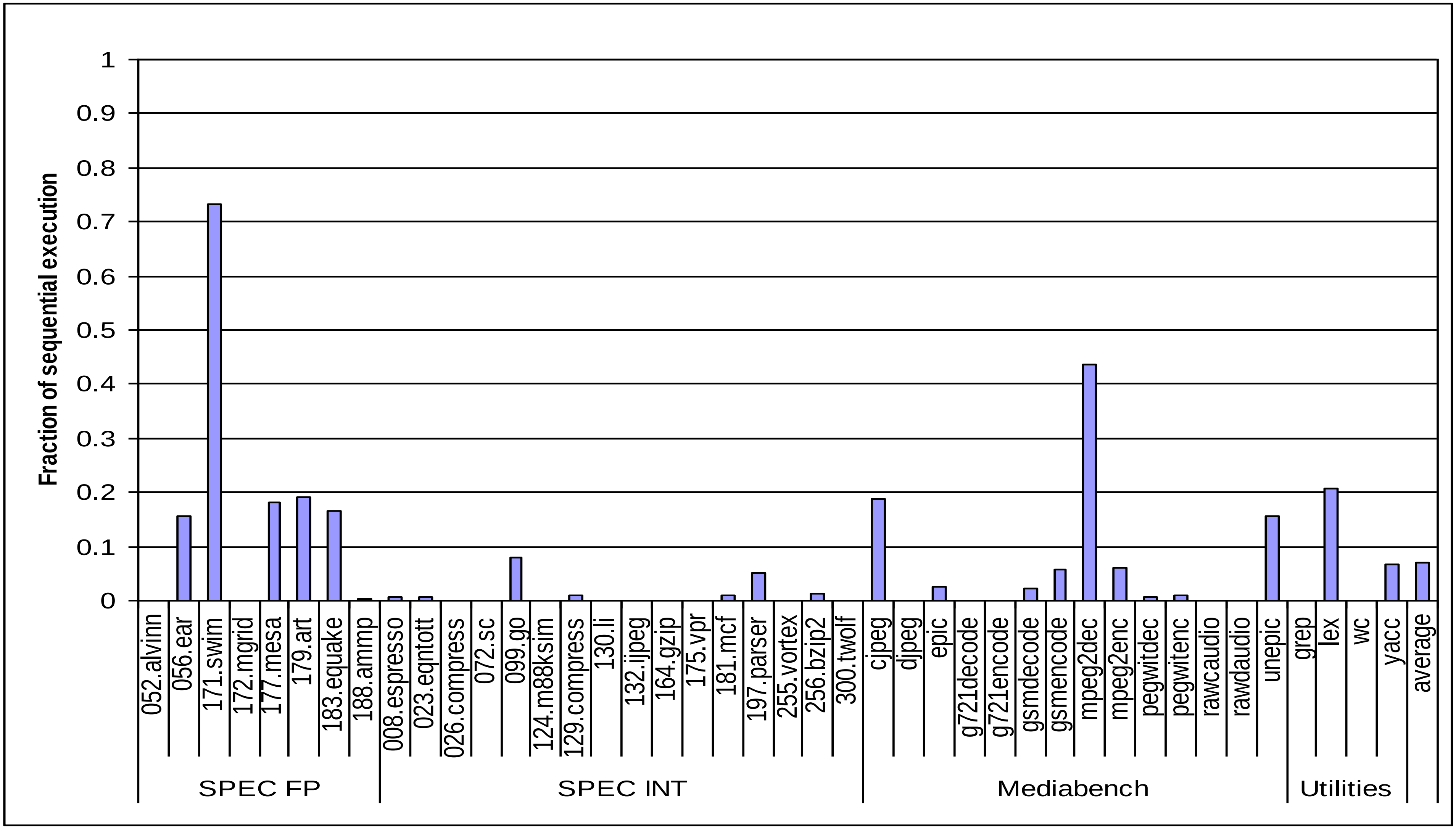
Loop Level Parallelization



Bad news: limited number of parallel loops in general purpose applications

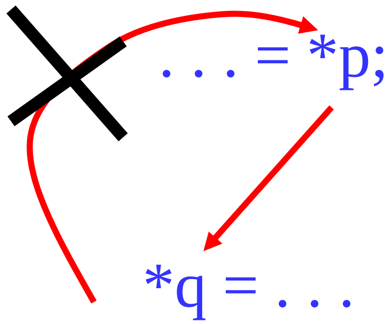
- –1.3x speedup for SpecINT2000 on 4 cores

DOALL Loop Coverage



What’s the Problem?

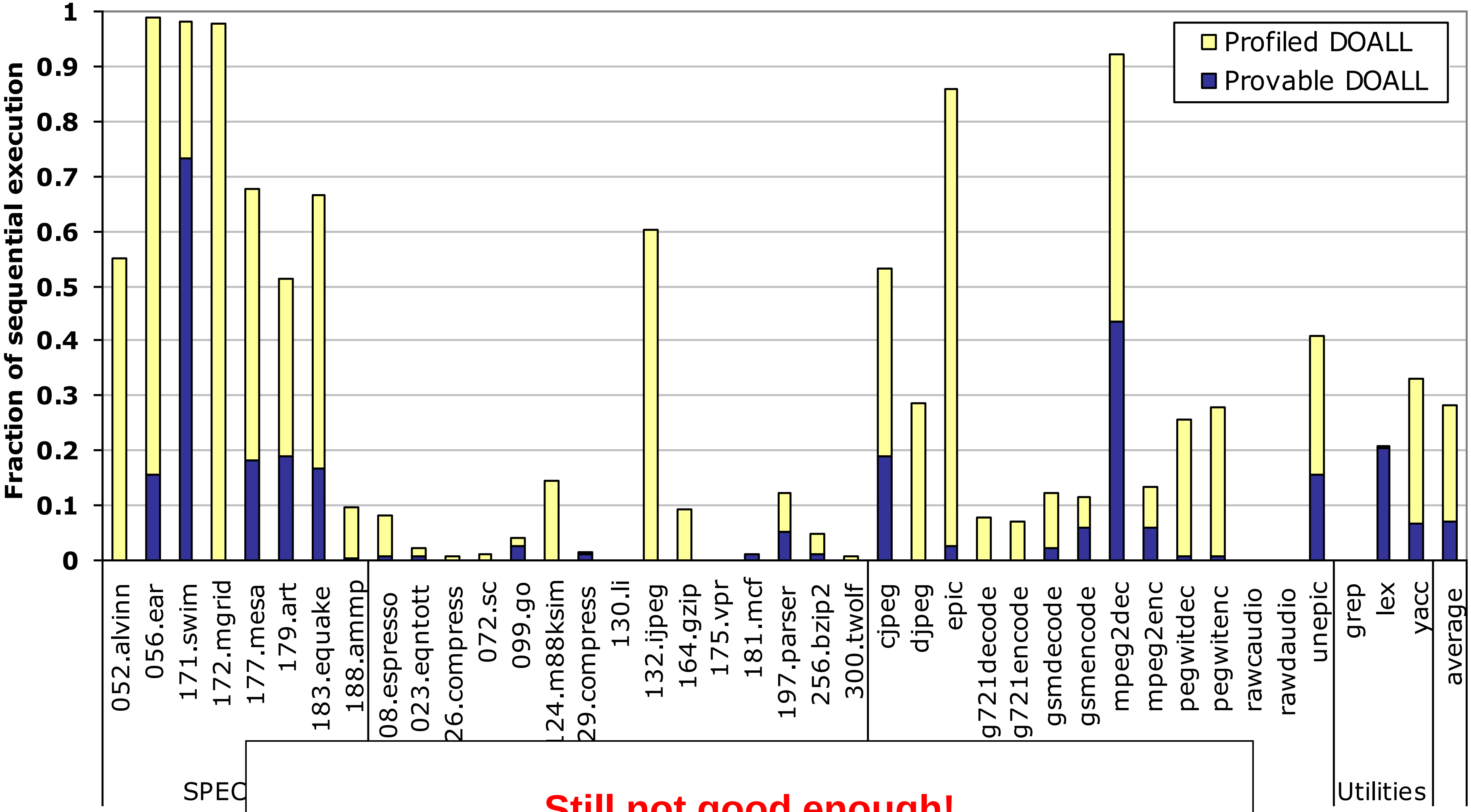
1. Memory dependence analysis



```
for (i=0; i<100; i++) {

    . . . = *p;

  *q = . . .

}
```

⇨ Memory dependence profiling and speculative parallelization

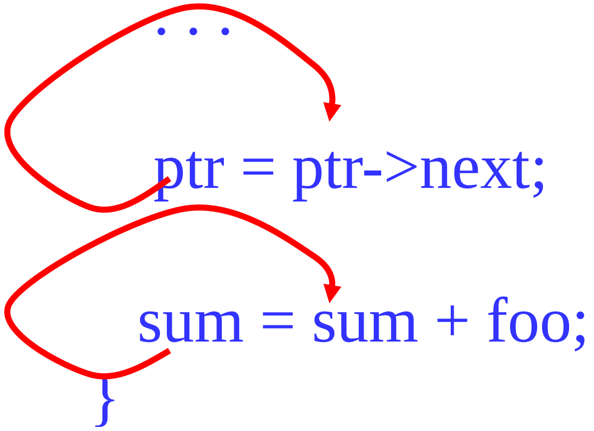
DOALL Coverage – Provable and Profiled



Still not good enough!

What’s the Next Problem?

2. Data dependences

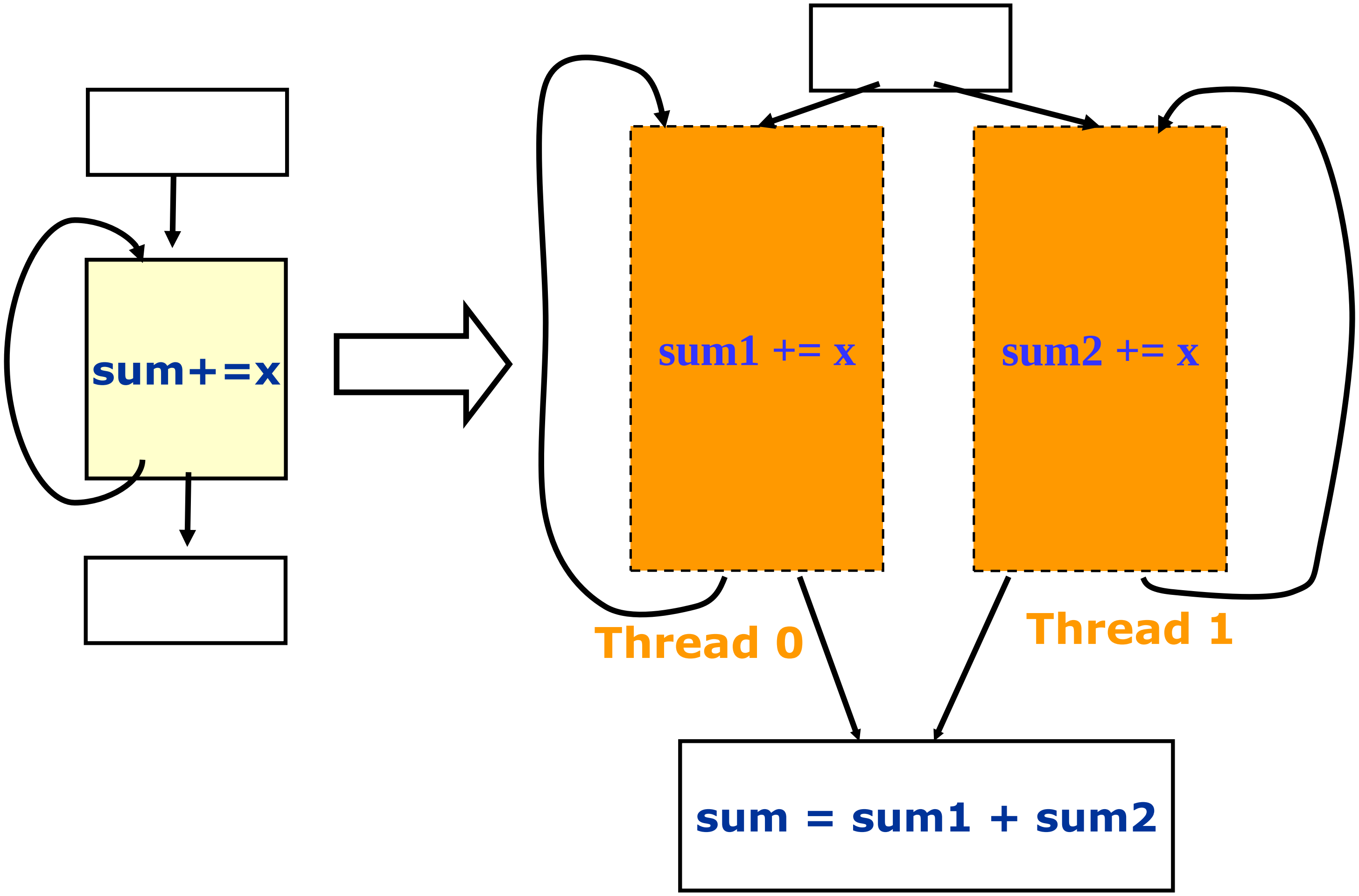


```
while (ptr != NULL) {
    . . .
    ptr = ptr->next;
    sum = sum + foo;
}
```

➡ Compiler transformations

We Know How to Break Some of These Dependences – Recall ILP Optimizations

Apply accumulator variable expansion!



Data Dependences Inhibit Parallelization

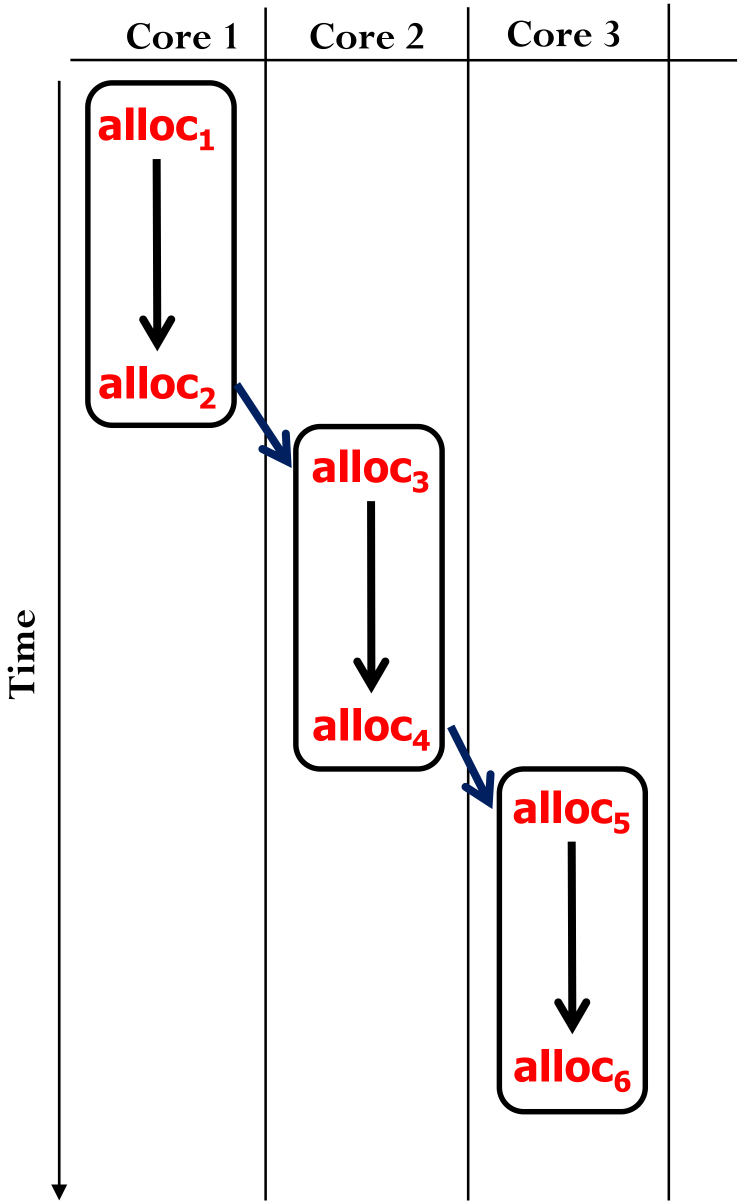
- Accumulator, induction, and min/max expansion only capture a small set of dependences
- 2 options
  - 1) Break more dependences – New transformations
  - 2) Parallelize in the presence of dependences – more than DOALL parallelization
- We will talk about both, but for now ignore this issue

What’s the Next Problem?

3. C/C++ too restrictive

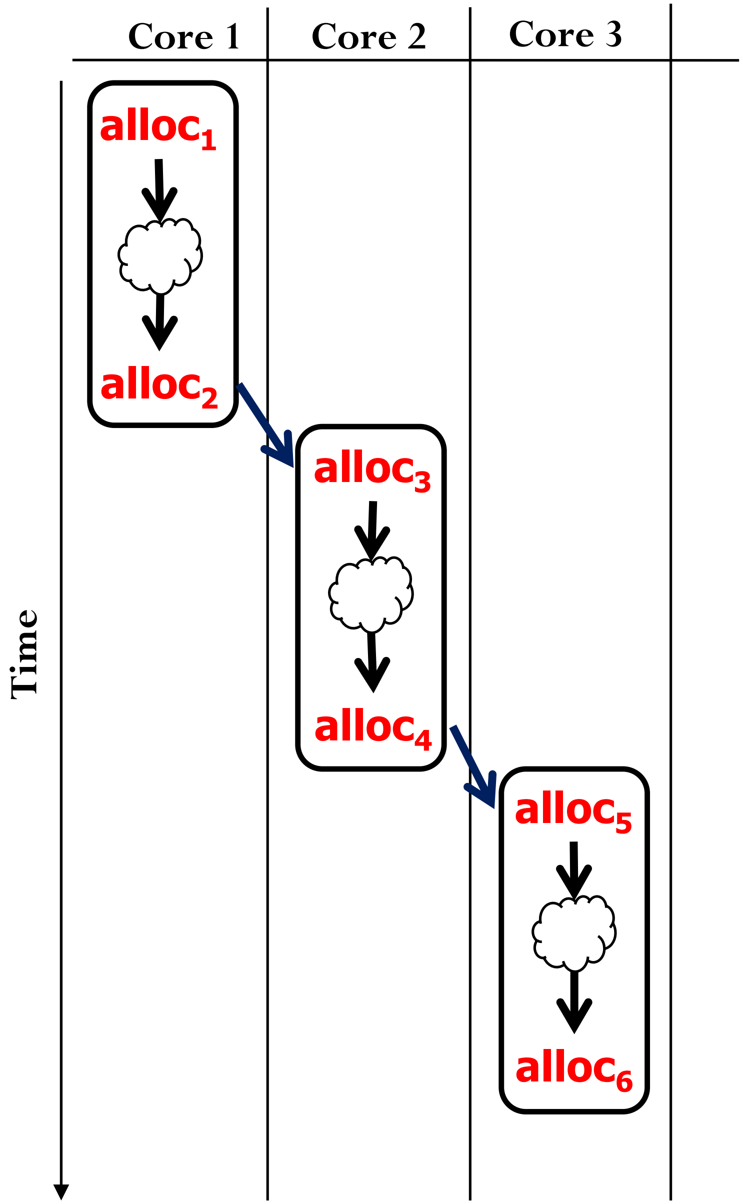
```
char *memory;

void * alloc(int size);

void * alloc(int size) {
  void * ptr = memory;
  memory = memory + size;
  return ptr;
}
```

Low Level Reality



```
char *memory;

void * alloc(int size);

void * alloc(int size) {
  void * ptr = memory;
  memory = memory + size;
  return ptr;
}
```

Low Level Reality



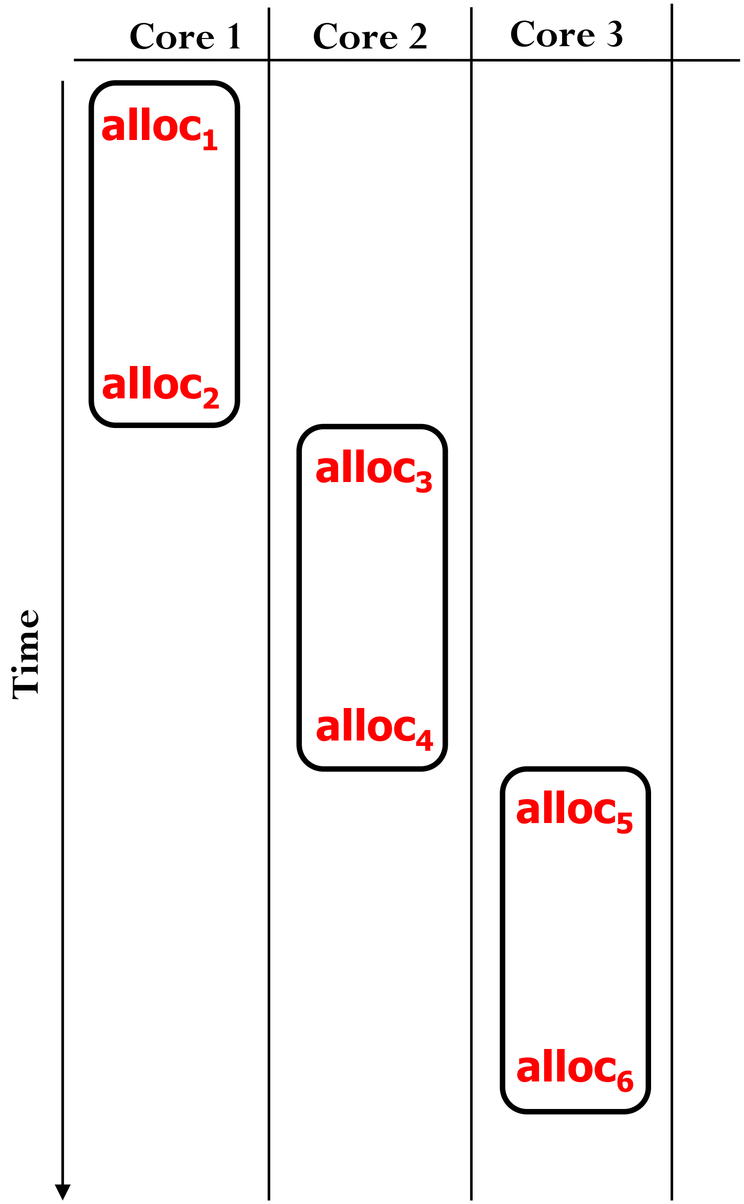
Loops cannot be parallelized even if computation is independent

Commutative Extension

- Interchangeable call sites
  - Programmer doesn’t care about the order that a particular function is called
  - Multiple different orders are all defined as correct
  - Impossible to express in C
- Prime example is memory allocation routine
  - Programmer does not care which address is returned on each call, just that the proper space is provided
- Enables compiler to break dependences that flow from 1 invocation to next forcing sequential behavior

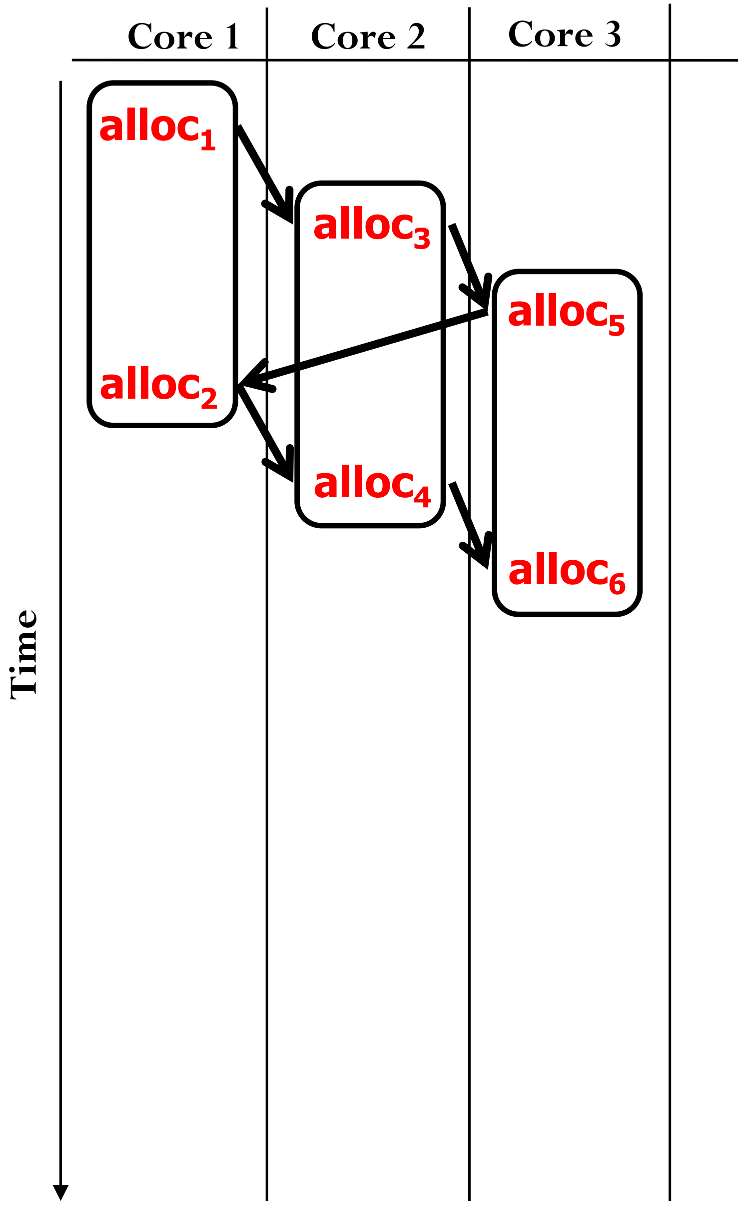
Low Level Reality

```
char *memory;

@Commutative
void * alloc(int size);

void * alloc(int size) {
  void * ptr = memory;
  memory = memory + size;
  return ptr;
}
```



```
char *memory;

@Commutative
void * alloc(int size);

void * alloc(int size) {
  void * ptr = memory;
  memory = memory + size;
  return ptr;
}
```

Low Level Reality



Implementation dependences should not cause serialization.

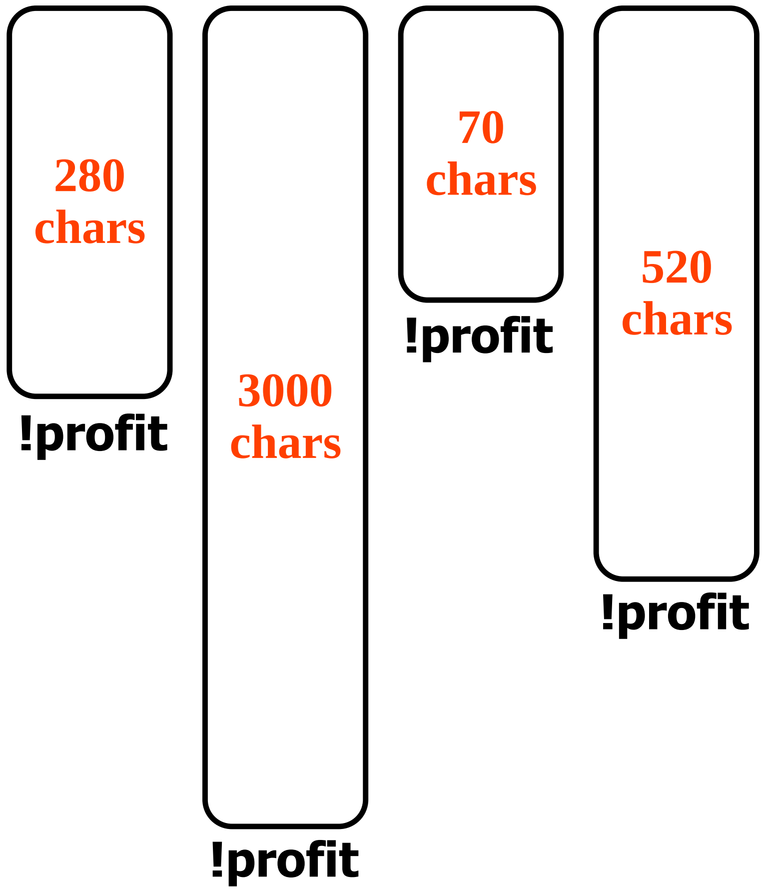
What is the Next Problem?

- 4. **C does not allow any prescribed non-determinism**
  - Thus sequential semantics must be assumed even though they not necessary
  - Restricts parallelism (useless dependences)
- Non-deterministic branch → programmer does not care about individual outcomes
  - They attach a probability to control how statistically often the branch should take
  - Allow compiler to tradeoff ‘quality’ (e.g., compression rates) for performance
    - When to create a new dictionary in a compression scheme

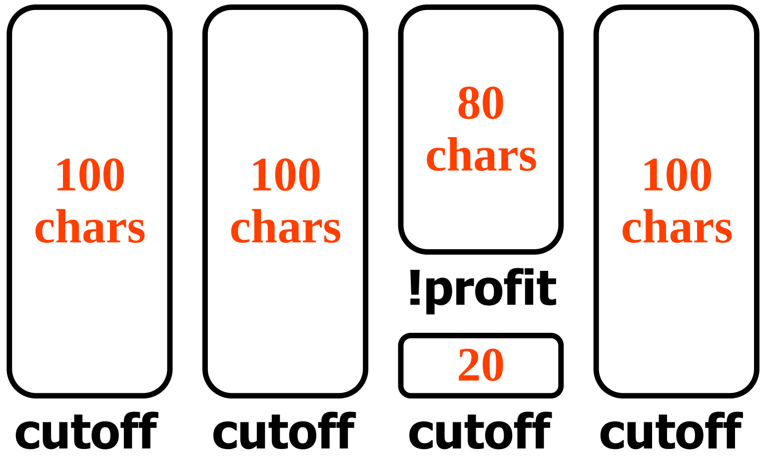
```
dict = create_dict();
while((char = read(1))) {
   profitable =
       compress(char, dict)
   if (!profitable) {
       dict=restart(dict);
   }
}
finish_dict(dict);
```

```
#define CUTOFF 100
dict = create_dict();
count = 0;
while((char==read(1))) {
   profitable =
       compress(char, dict)
   if (!profitable)
       dict=restart(dict);
   if (count == CUTOFF){
       dict=restart(dict);
       count=0;
   }

   count++;
}
finish_dict(dict);
```

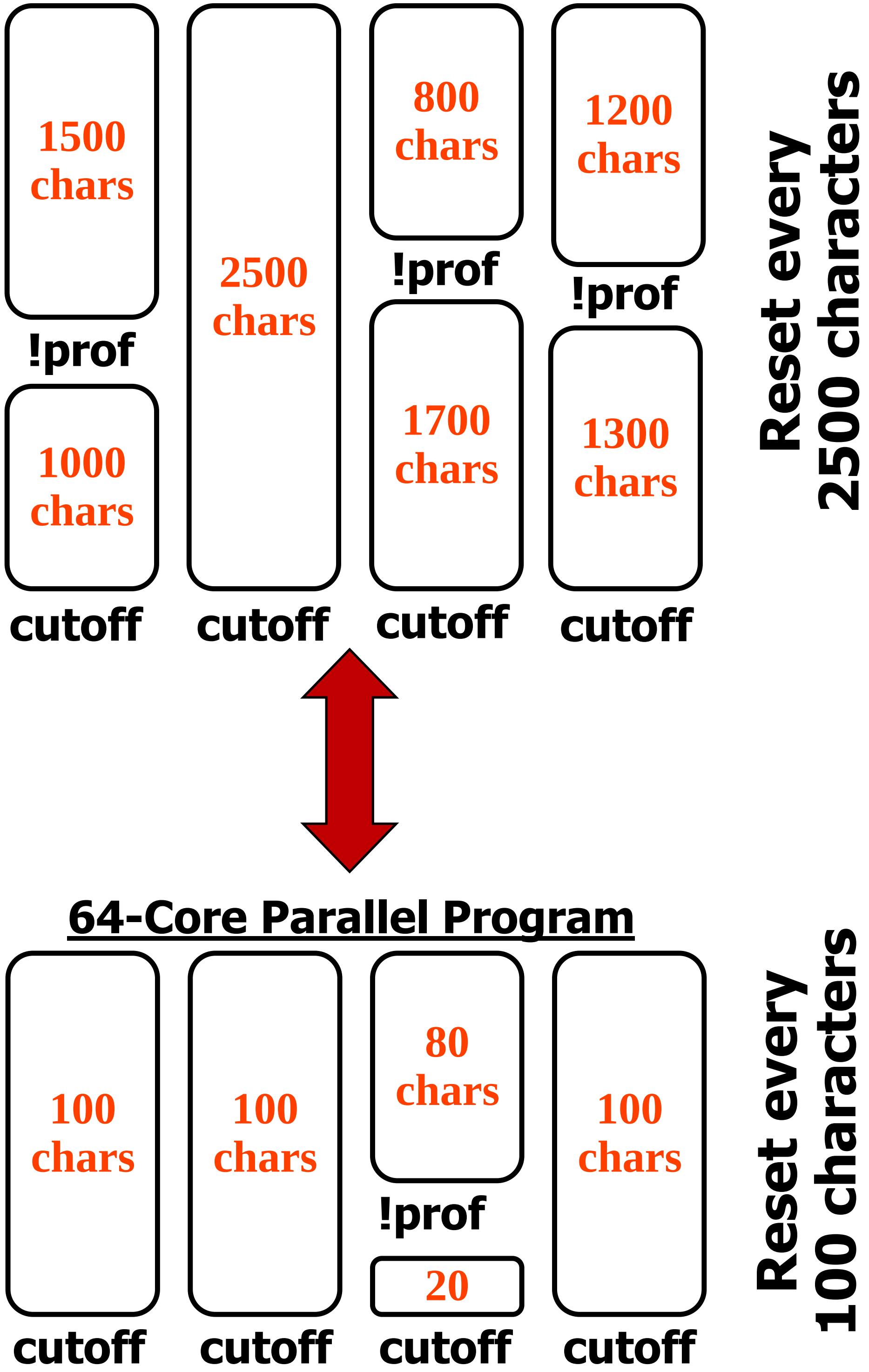
Sequential Program



Parallel Program



```
dict = create_dict();
while((char = read(1))) {
  profitable =
       compress(char, dict)

  @YBRANCH(probability=.01)
  if (!profitable) {
    dict = restart(dict);
  }
}
finish_dict(dict);
```



2-Core Parallel Program

1500 chars
!prof
1000 chars
cutoff

2500 chars
cutoff

800 chars
!prof
1700 chars
cutoff

1200 chars
!prof
1300 chars
cutoff

Reset every 2500 characters

64-Core Parallel Program

100 chars
cutoff

100 chars
cutoff

80 chars
!prof
20
cutoff

100 chars
cutoff

Reset every 100 characters

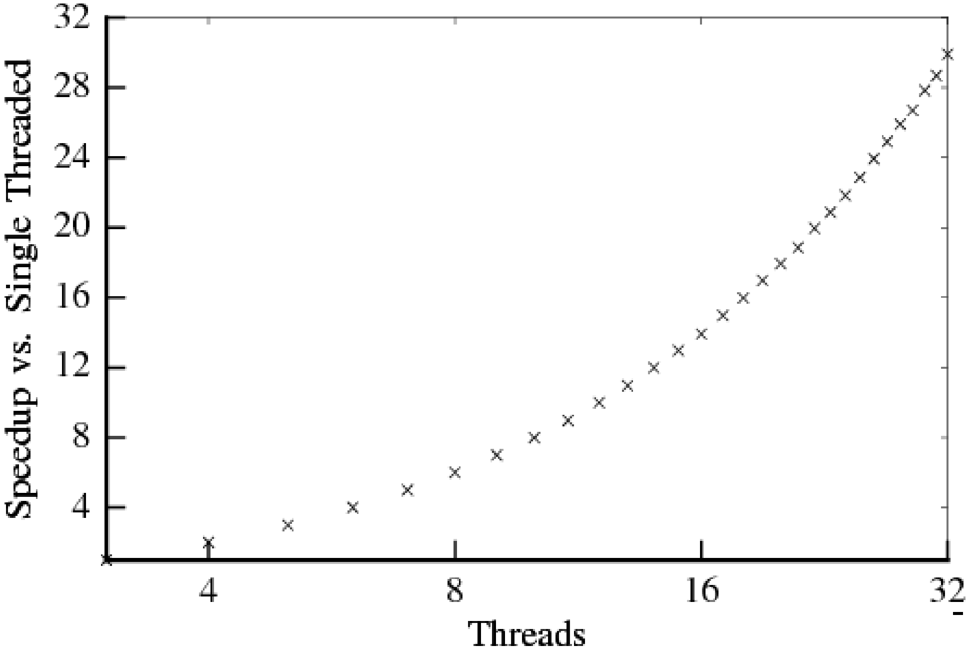
Compilers are best situated to make the tradeoff between output quality and performance

Capturing Output/Performance Tradeoff: *Y-Branches in 164.gzip*

```
dict = create_dict();
while((char = read(1))) {
  profitable =
      compress(char, dict)

  if (!profitable) {
    dict = restart(dict);
  }
}
finish_dict(dict);
```

```
dict = create_dict();
while((char = read(1))) {
  profitable =
      compress(char, dict)

  @YBRANCH(probability=.00001)
  if (!profitable) {
    dict = restart(dict);
  }
}
finish_dict(dict);
```

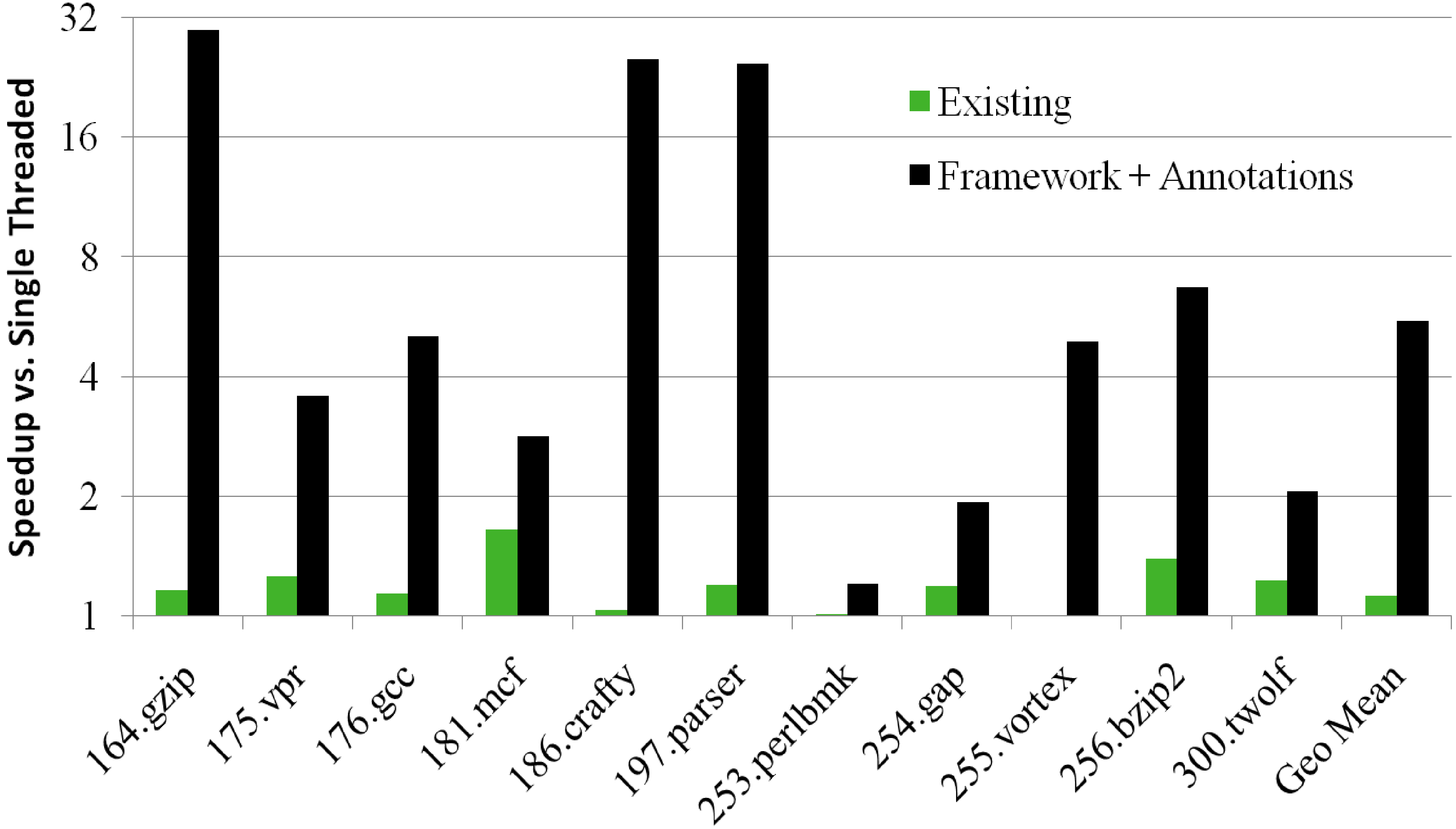
```
#define CUTOFF 100000
dict = create_dict();
count = 0;
while((char = read(1))) {
  profitable =
      compress(char, dict)

  if (!profitable)
    dict=restart(dict);
  if (count == CUTOFF){
    dict=restart(dict);
    count=0;
  }

  count++;
}
finish_dict(dict);
```



	LoC Changed	Increased Scope	Commutative	Y-Branch	Nested Parallel	Iter. Inv. Value Spec.	Loop Alias Spec.	Programmer Mod.
164.gzip	26	x		x				x
175.vpr	1		x			x	x	
176.gcc	18	x	x				x	x
181.mcf	0				x			
186.crafty	9	x	x		x	x	x	
197.parser	3	x	x					
253.perlbmk	0	x				x	x	
254.gap	3	x	x				x	
255.vortex	0	x				x	x	
256.bzip2	0	x					x	
300.twolf	1	x	x				x	

Modified only 60 LOC out of ~500,000 LOC

Performance Potential



What prevents the automatic extraction of parallelism?

~~**Lack of an Aggressive Compilation Framework**~~

~~**Sequential Programming Model**~~

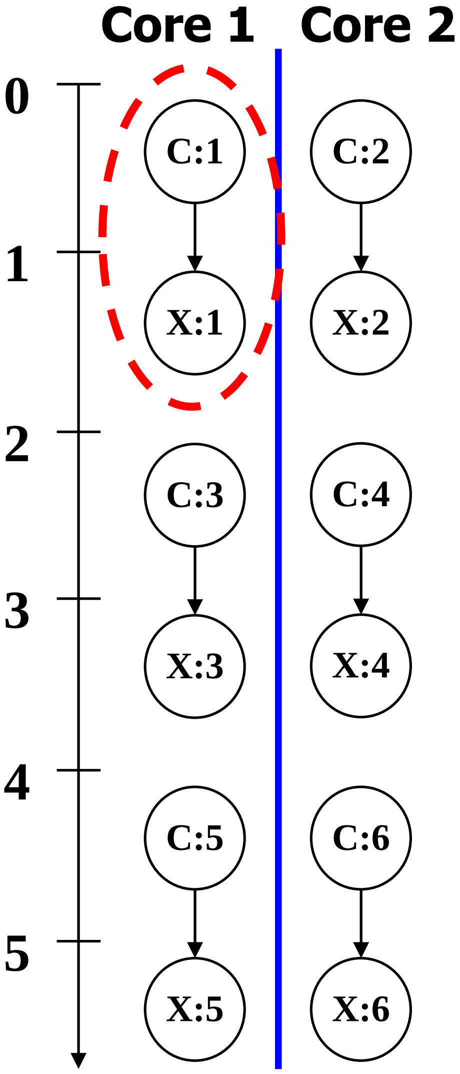
What About Non-Scientific Codes???

Scientific Codes (FORTRAN-like)

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

Independent Multithreading (IMT)

Example: DOALL parallelization

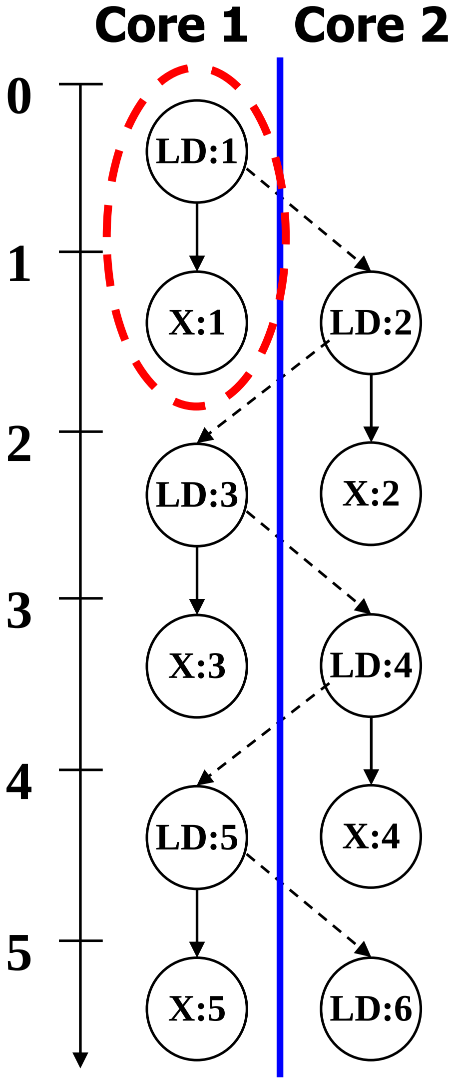


General-purpose Codes (legacy C/C++)

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

Cyclic Multithreading (CMT)

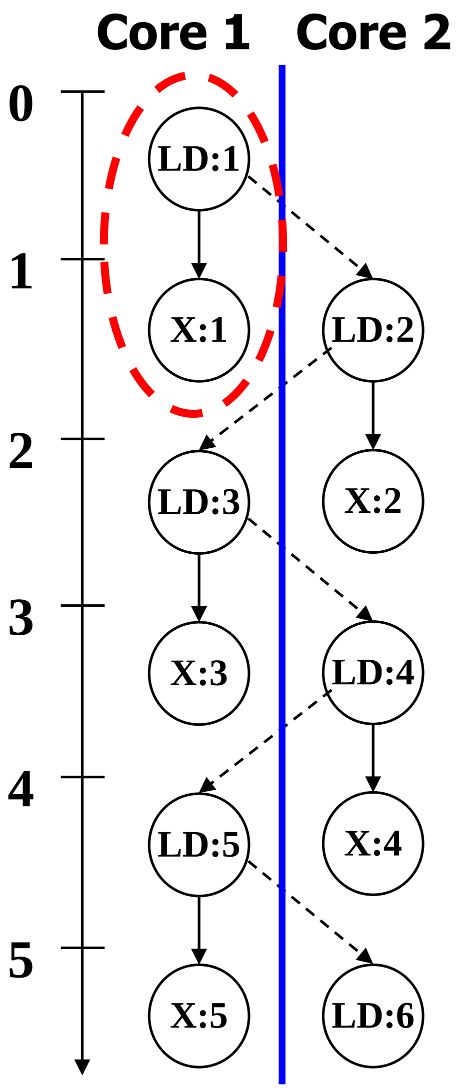
Example: DOACROSS [Cytron, ICPP 86]



Alternative Parallelization Approaches

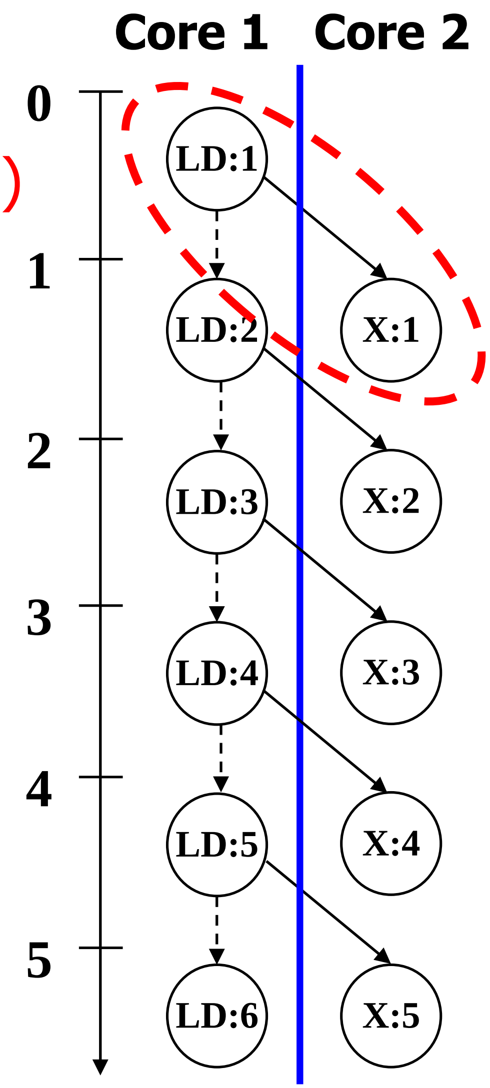
```
while(ptr = ptr->next)     // LD
  ptr->val = ptr->val + 1; // X
```

Cyclic Multithreading (CMT)

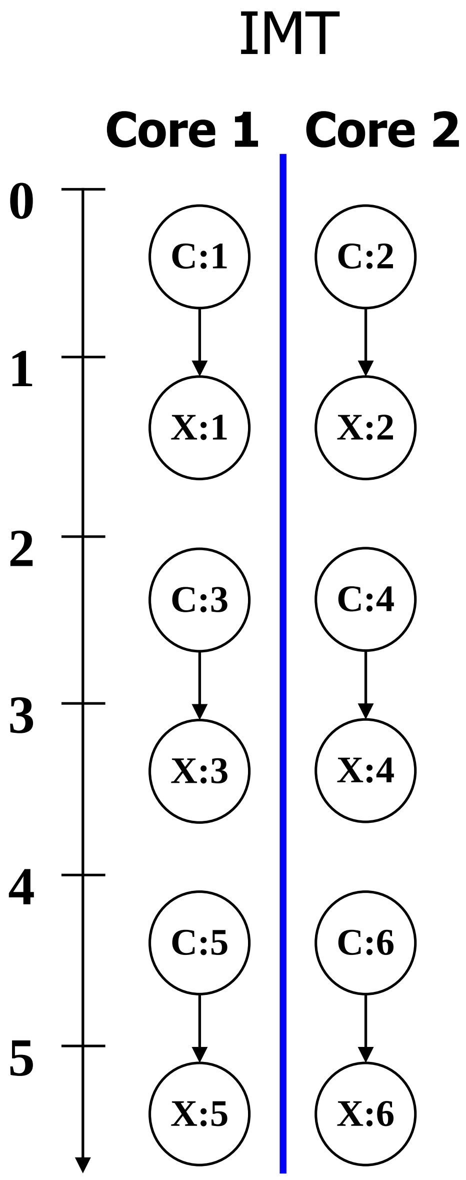


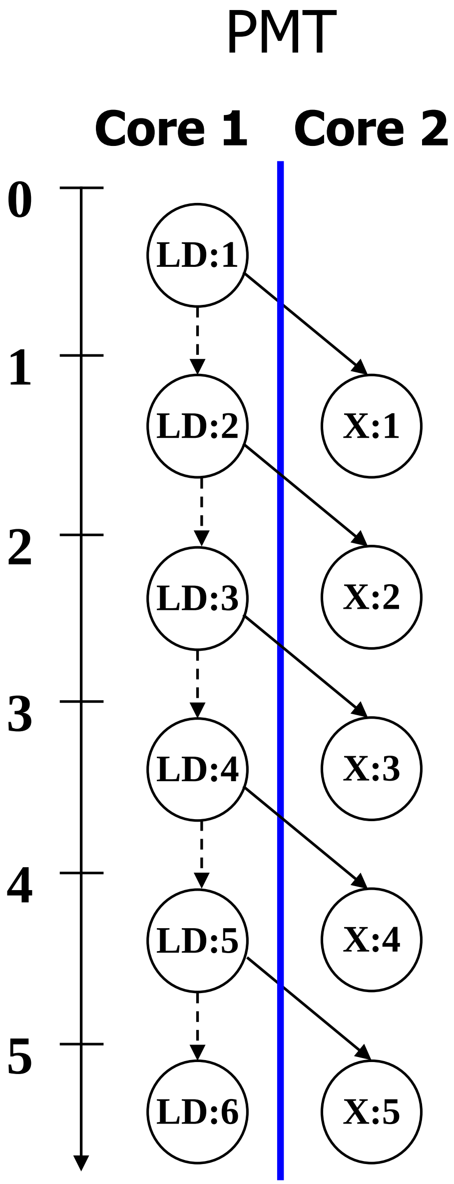
Pipelined Multithreading (PMT)

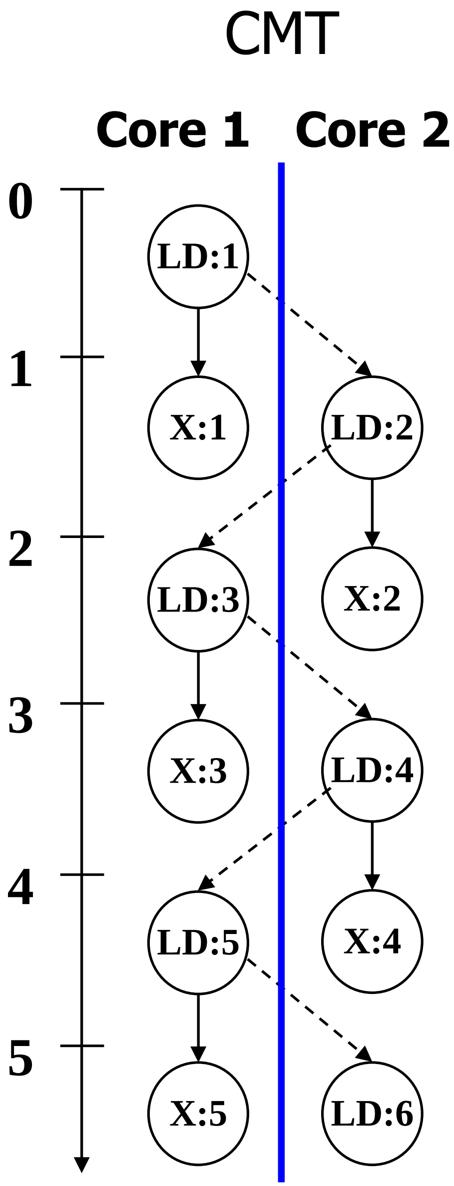
Example: DSWP **[PACT 2004]**



Comparison: IMT, PMT, CMT

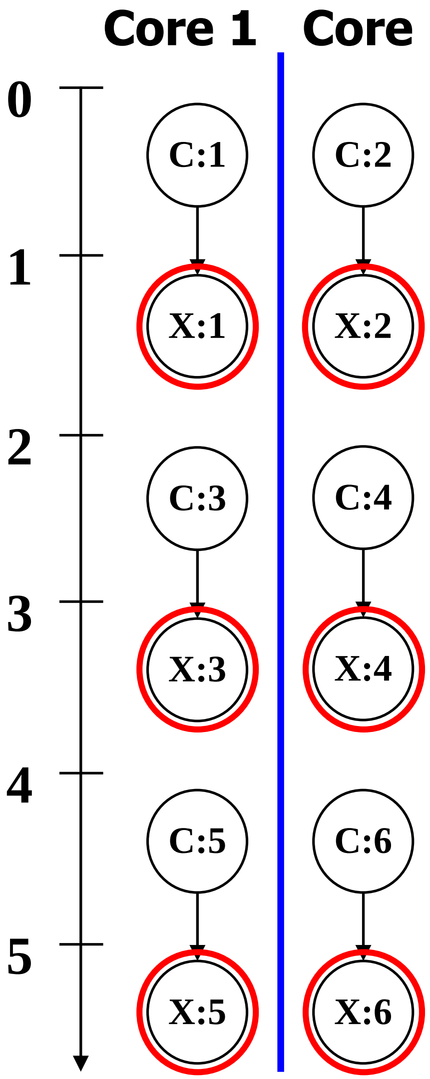






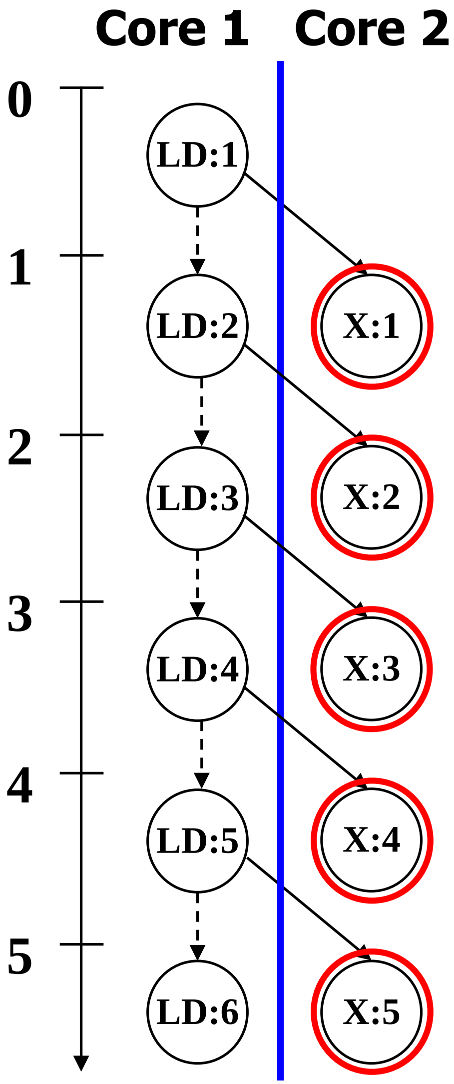
Comparison: IMT, PMT, CMT

IMT



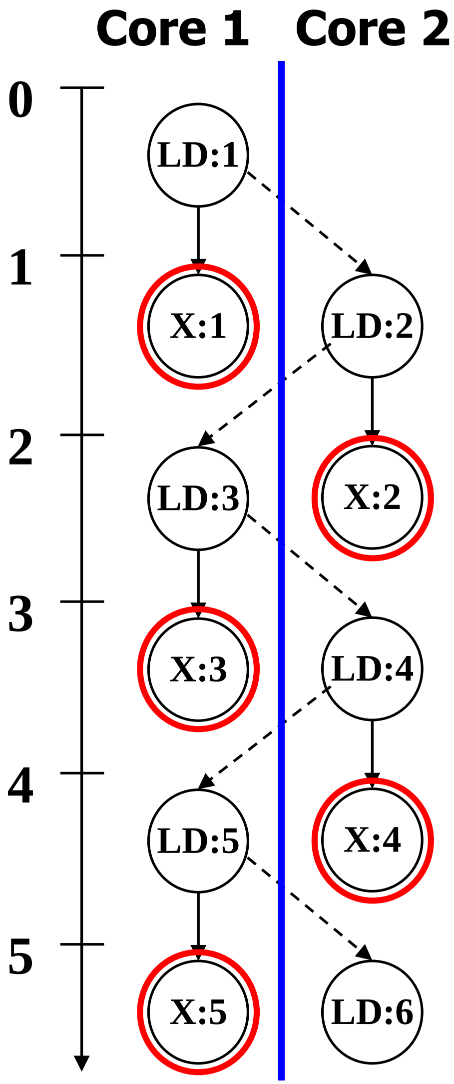
lat(comm) = 1: 1 iter/cycle

PMT



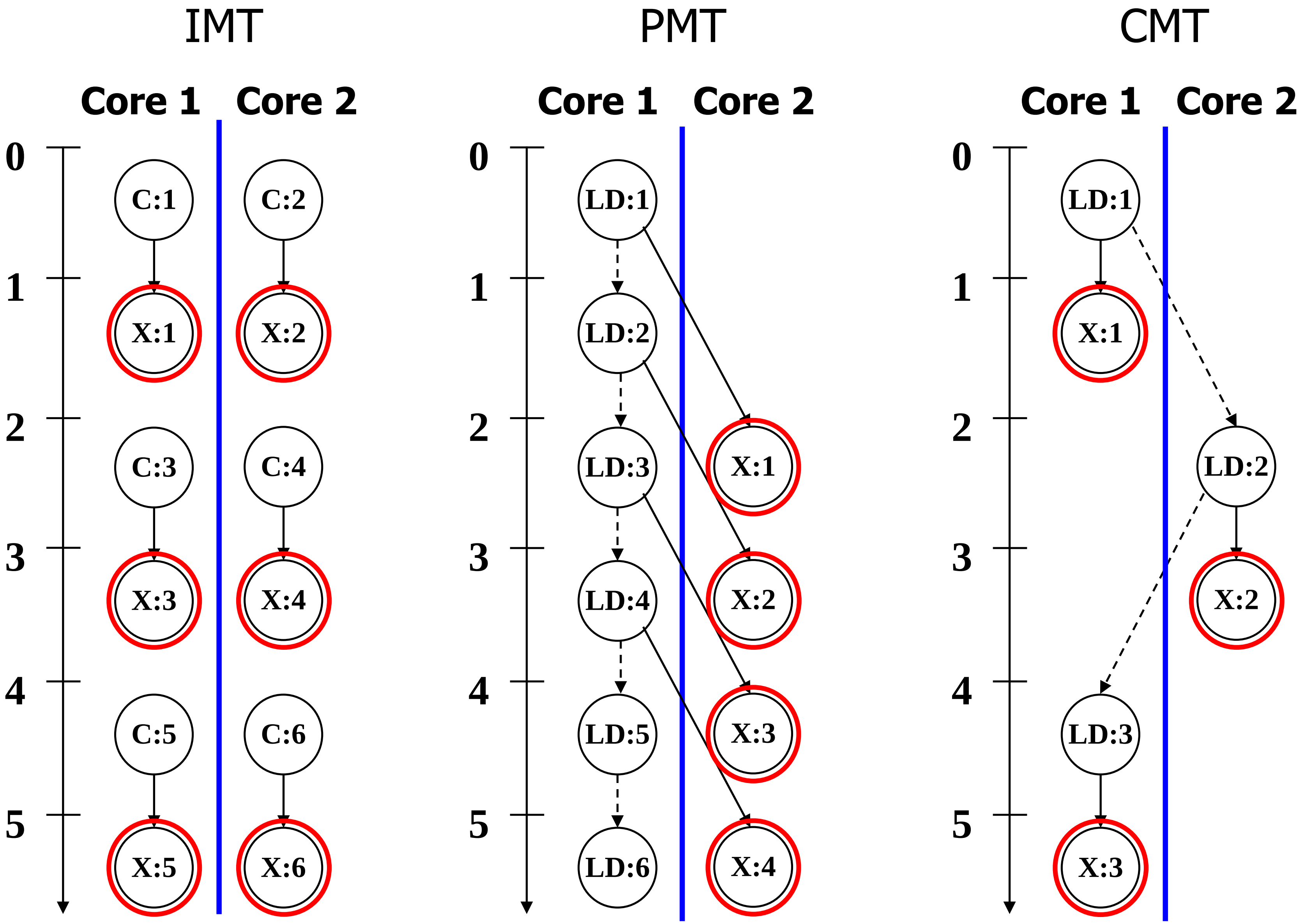
1 iter/cycle

CMT



1 iter/cycle

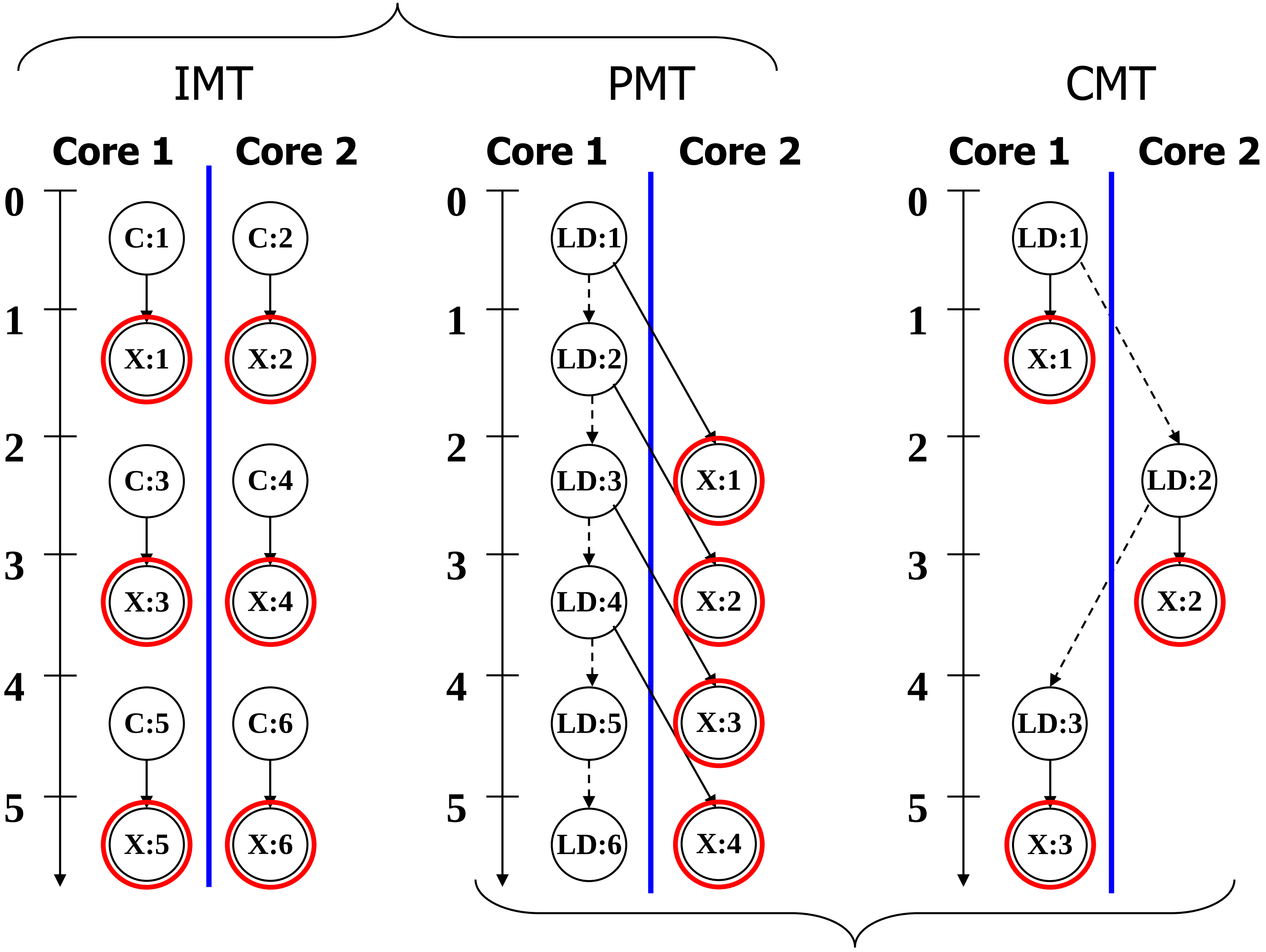
Comparison: IMT, PMT, CMT



	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

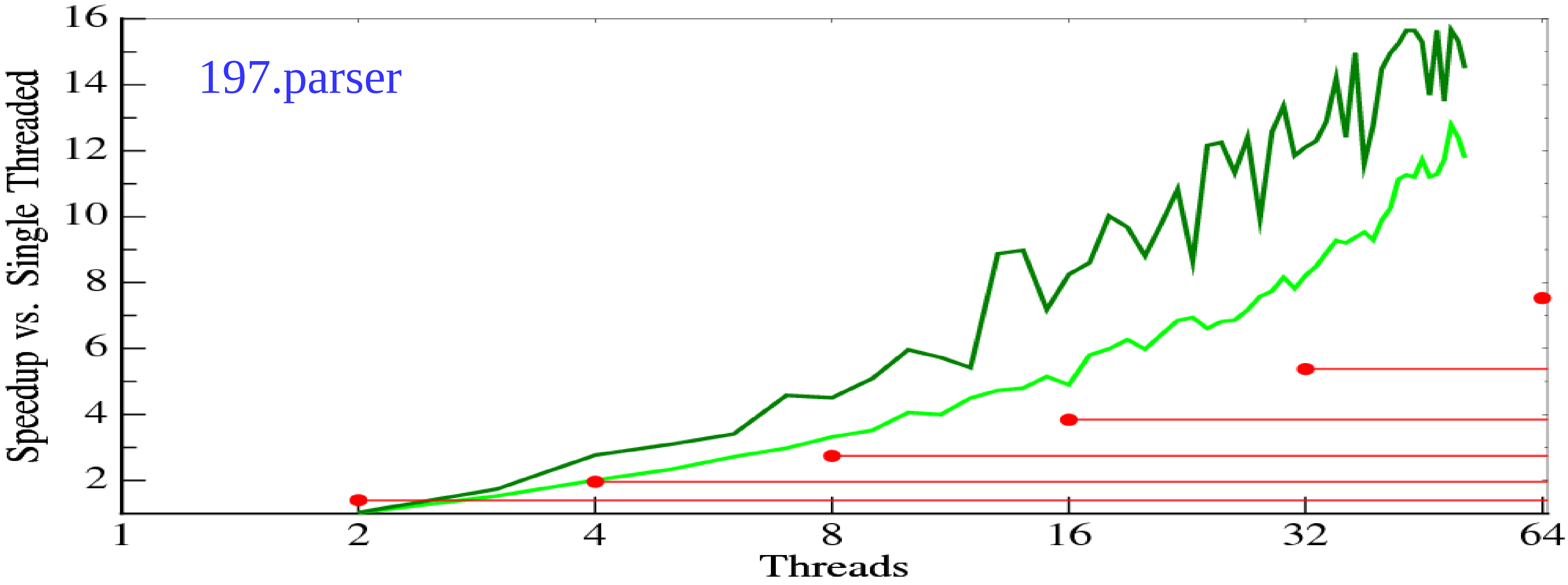
Comparison: IMT, PMT, CMT

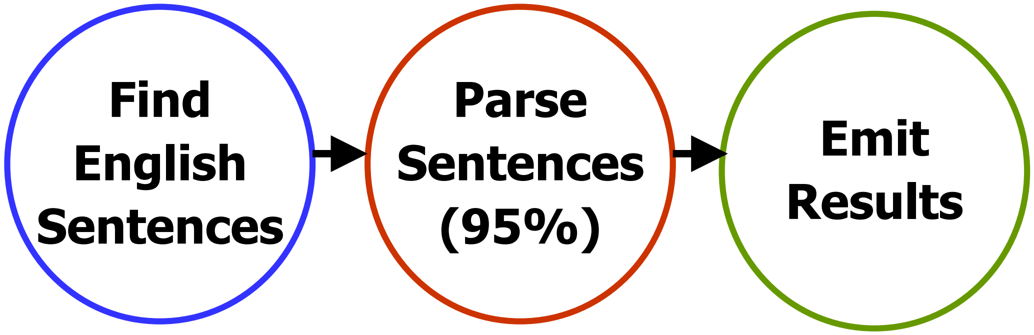
Thread-local Recurrences → Fast Execution



Cross-thread Dependences → Wide Applicability

Our Objective: Automatic Extraction of Pipeline Parallelism using DSWP



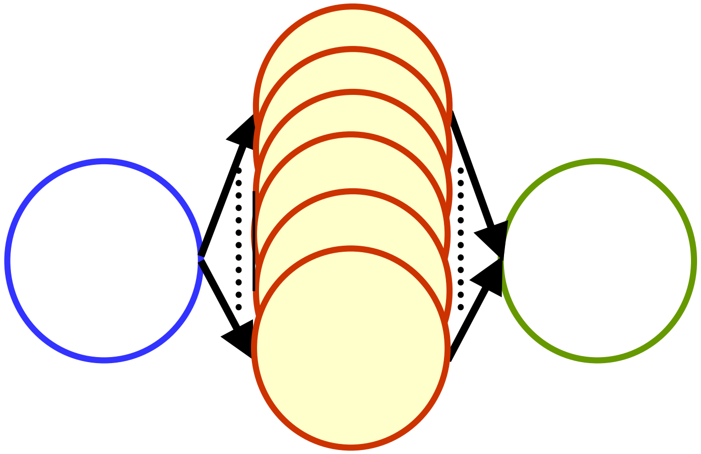


Find English Sentences

Parse Sentences (95%)

Emit Results

Decoupled Software Pipelining



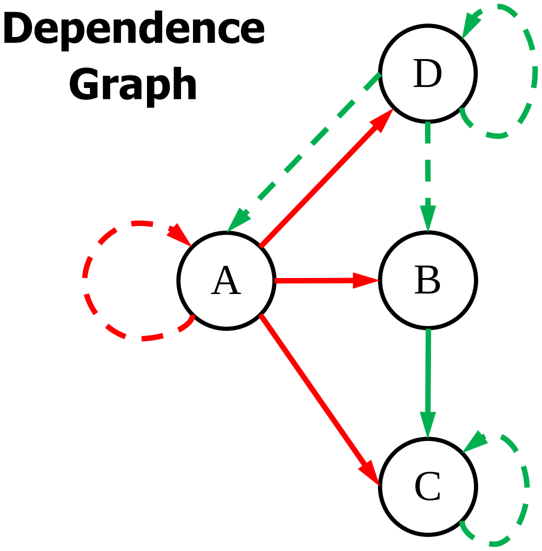
PS-DSWP (Spec DOALL Middle Stage)

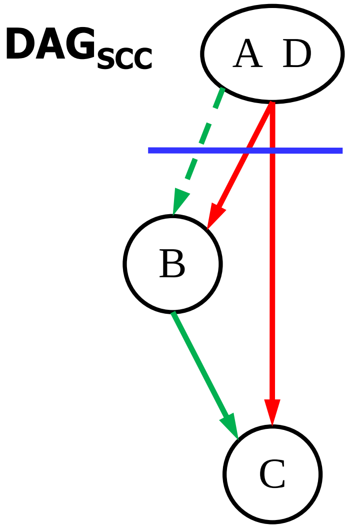
Decoupled Software Pipelining

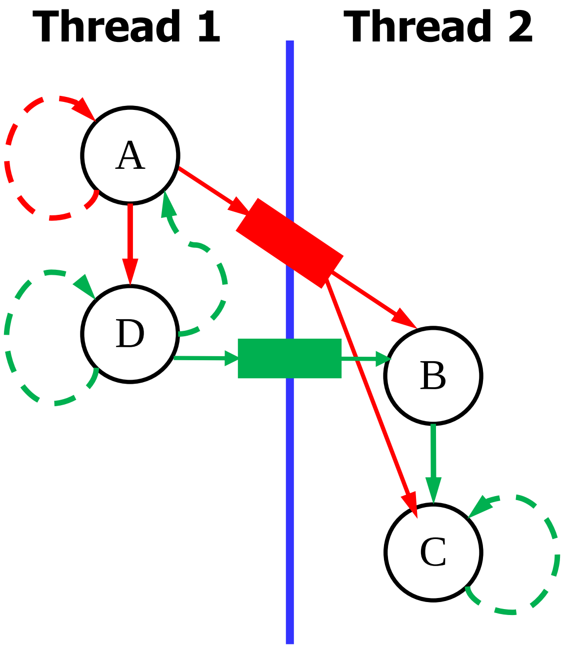
[MICRO 2005]

Decoupled Software Pipelining (DSWP)

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







register

control

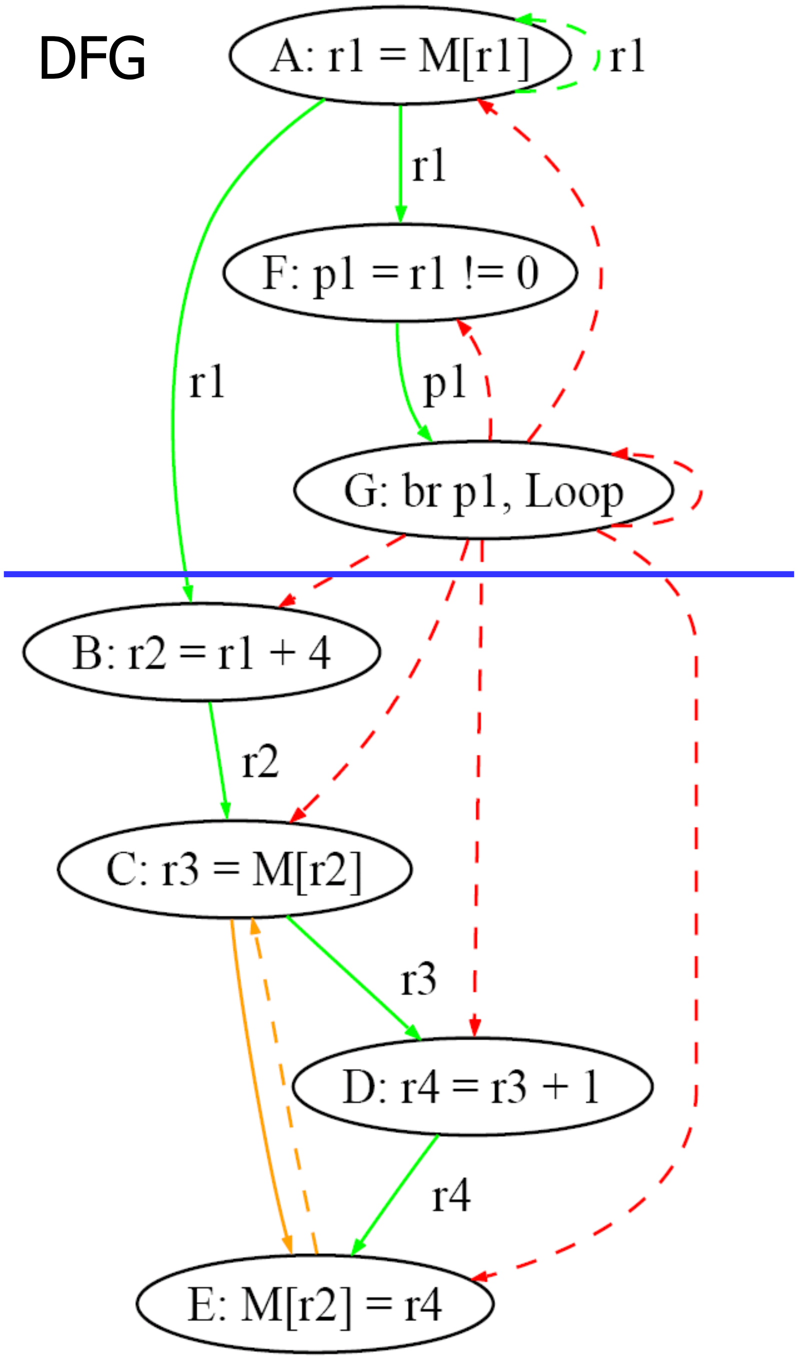
→ intra-iteration

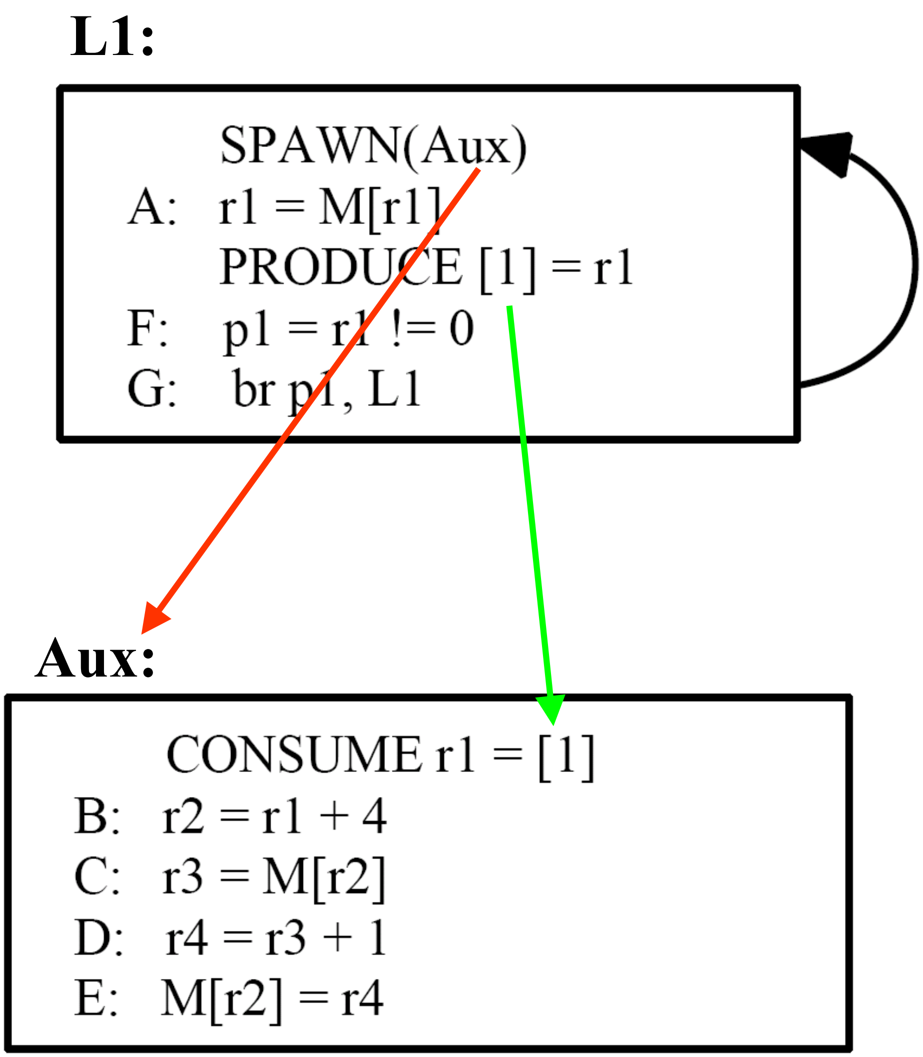
- - -▶ loop-carried

▬ communication queue

Inter-thread communication latency is a one-time cost

Implementing DSWP





L1:

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

Aux:

```
      CONSUME r1 = [1]
B:    r2 = r1 + 4
C:    r3 = M[r2]
D:    r4 = r3 + 1
E:    M[r2] = r4
```

register

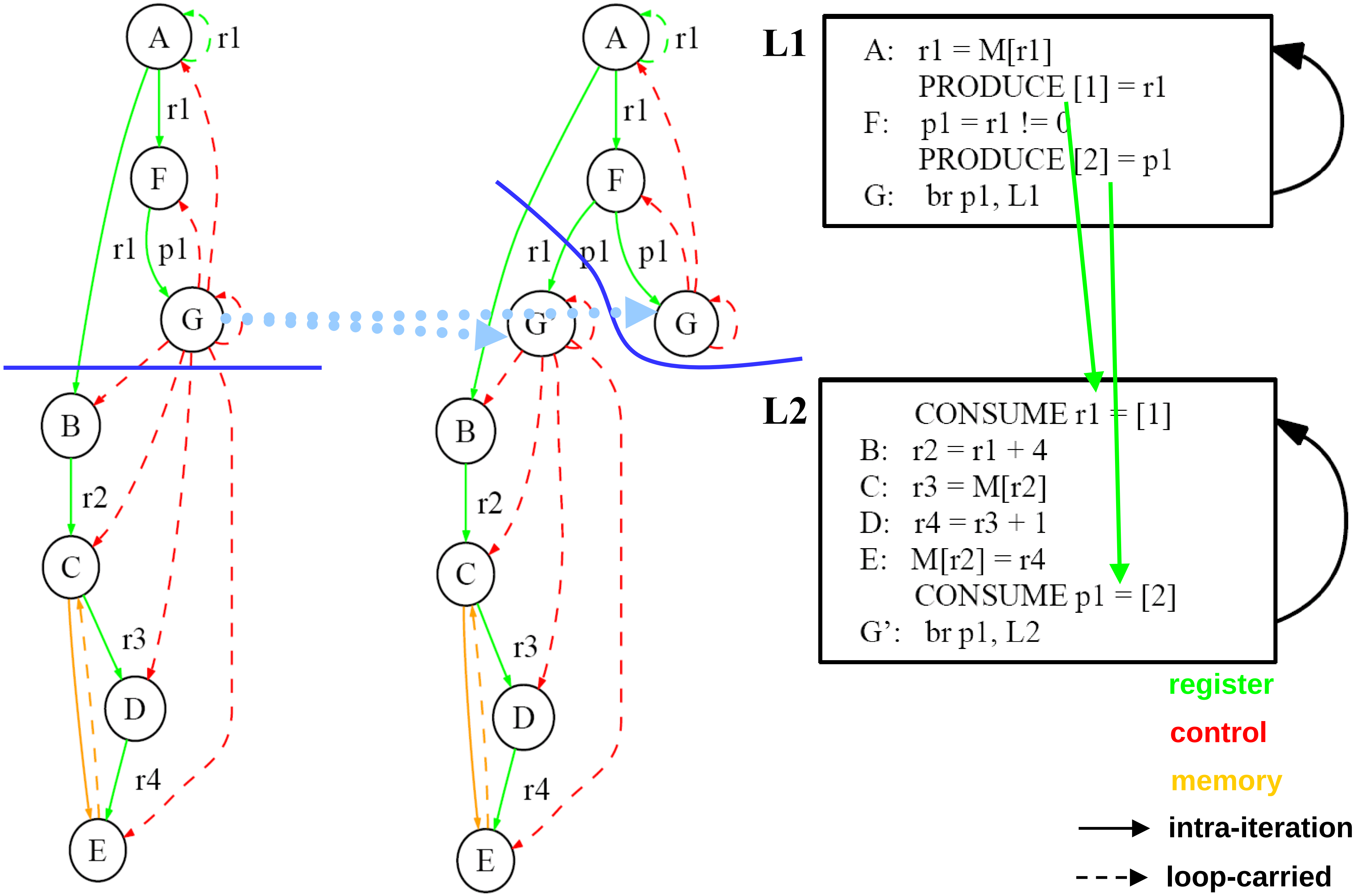
control

memory

⟶ intra-iteration

- - -▶ loop-carried

Optimization: Node Splitting
To Eliminate Cross Thread Control



L1

```
A:   r1 = M[r1]
     PRODUCE [1] = r1
F:   p1 = r1 != 0
     PRODUCE [2] = p1
G:   br p1, L1
```

L2

```
     CONSUME r1 = [1]
B:   r2 = r1 + 4
C:   r3 = M[r2]
D:   r4 = r3 + 1
E:   M[r2] = r4
     CONSUME p1 = [2]
G':  br p1, L2
```

register

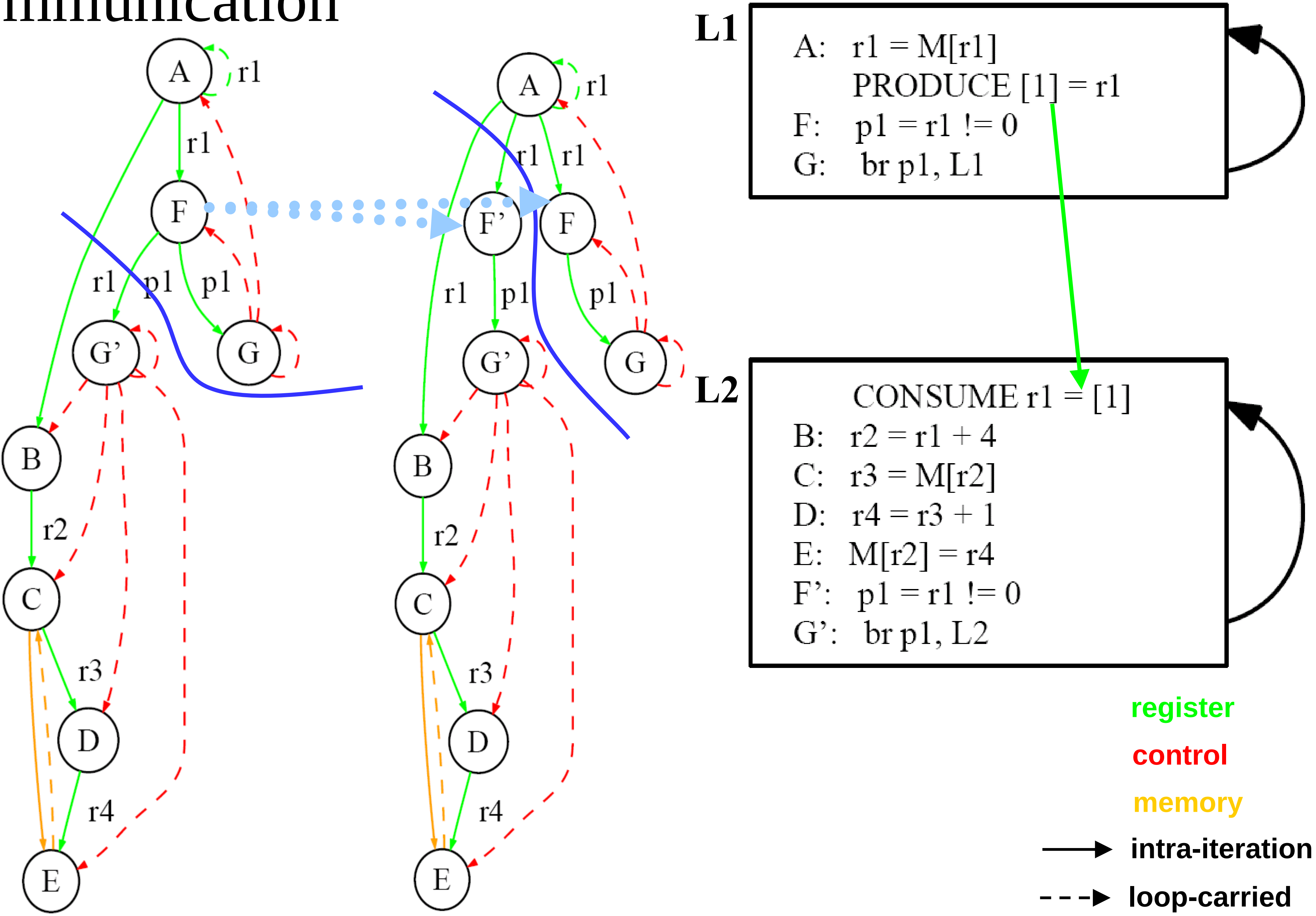
control

memory

→ intra-iteration

- - → loop-carried

Optimization: Node Splitting To Reduce Communication



L1

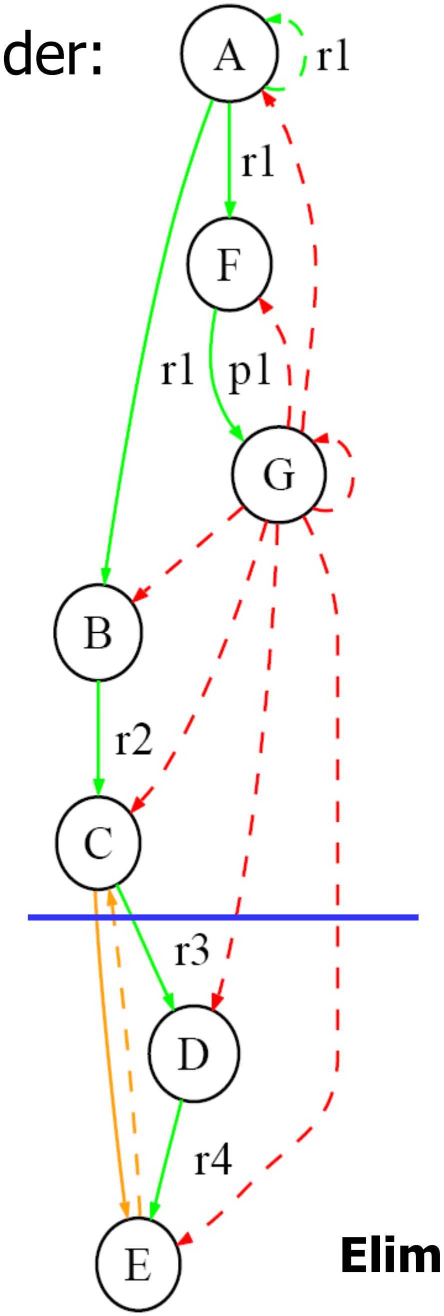
```
A:  r1 = M[r1]
    PRODUCE [1] = r1
F:  p1 = r1 != 0
G:  br p1, L1
```

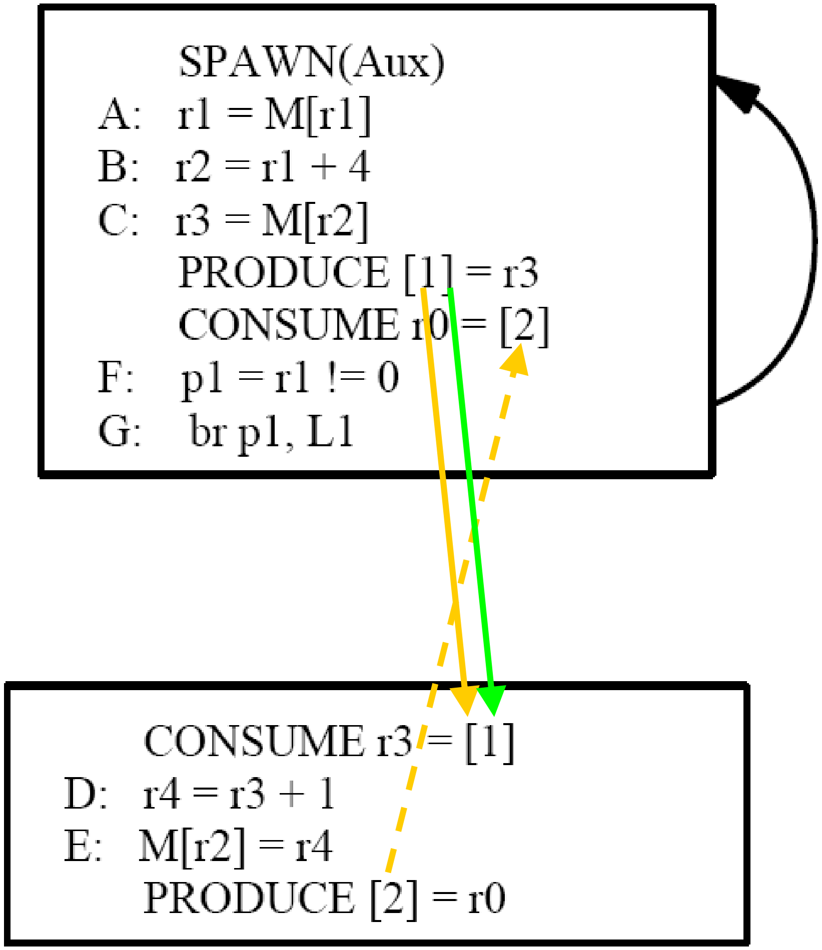
L2

```
    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:

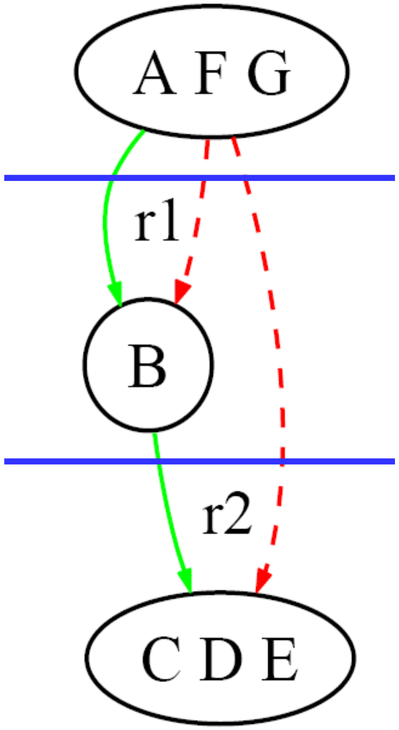




SPAWN(Aux)
A: r1 = M[r1]
B: r2 = r1 + 4
C: r3 = M[r2]
PRODUCE [1] = r3
CONSUME r0 = [2]
F: p1 = r1 != 0
G: br p1, L1

CONSUME r3 = [1]
D: r4 = r3 + 1
E: M[r2] = r4
PRODUCE [2] = r0

Solution: DAG_{SCC}



register

control

memory

→ intra-iteration

- - -> loop-carried

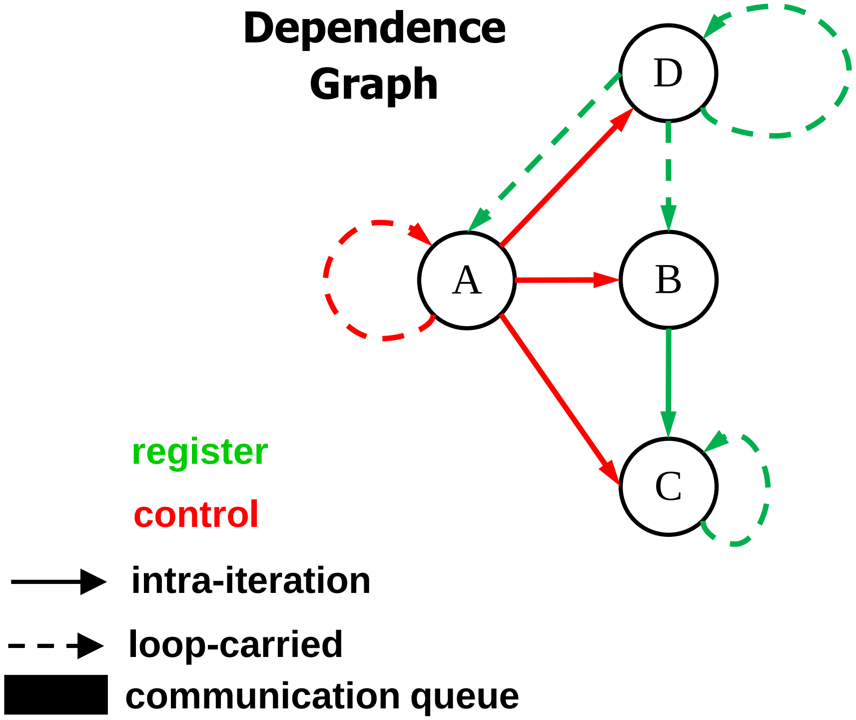
Eliminates pipelined/decoupled property

2 Extensions to the Basic Transformation

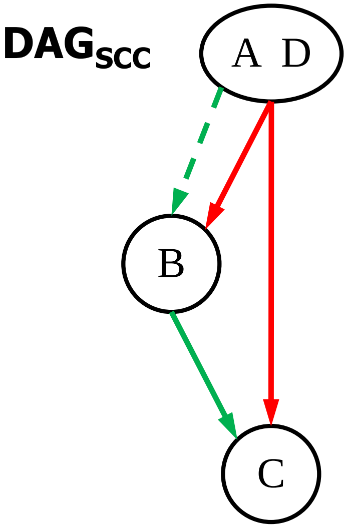
- Speculation
  - Break statistically unlikely dependences
  - Form better-balanced pipelines
- Parallel Stages
  - Execute multiple copies of certain “large” stages
  - Stages that contain inner loops perfect candidates

Why Speculation?

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



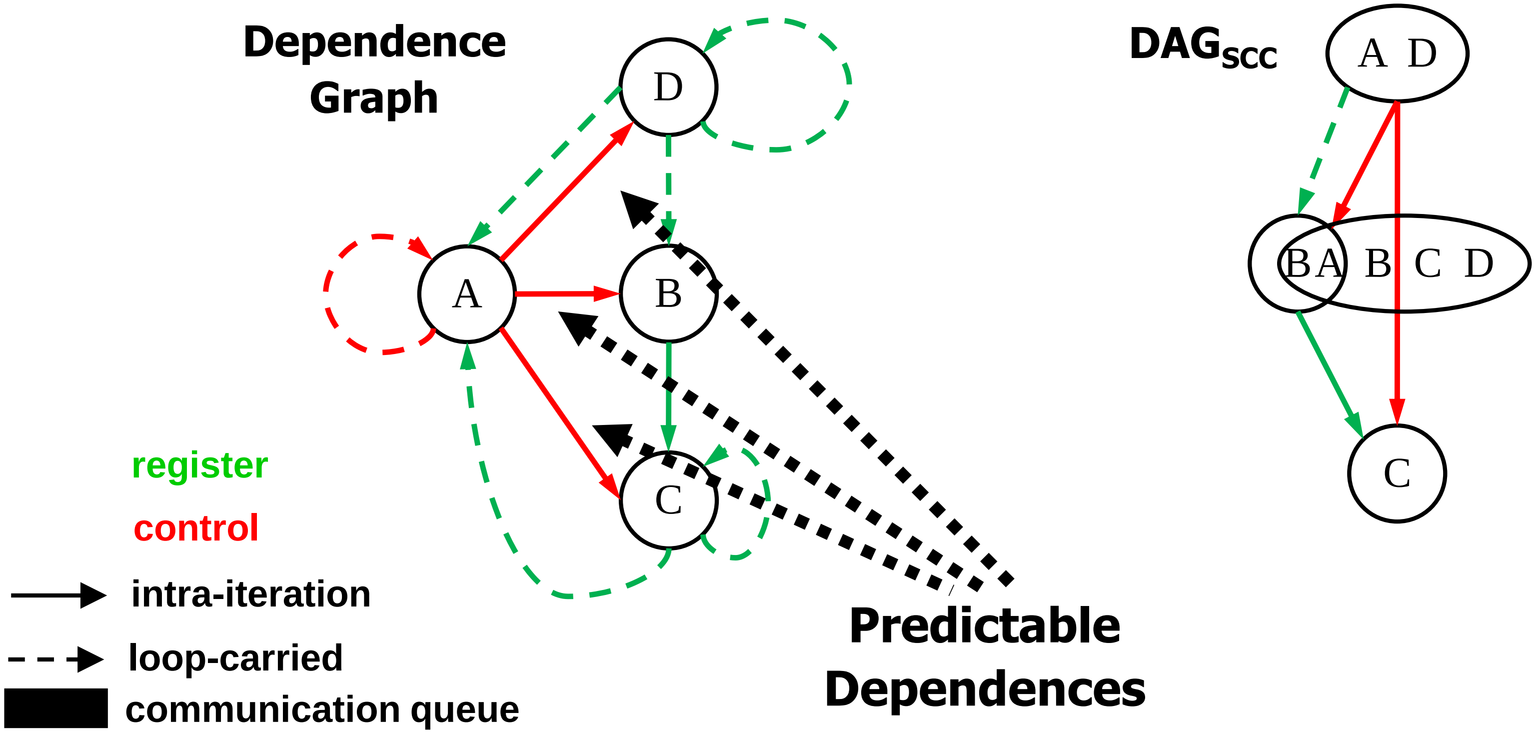
Dependence Graph



DAG$_{SCC}$

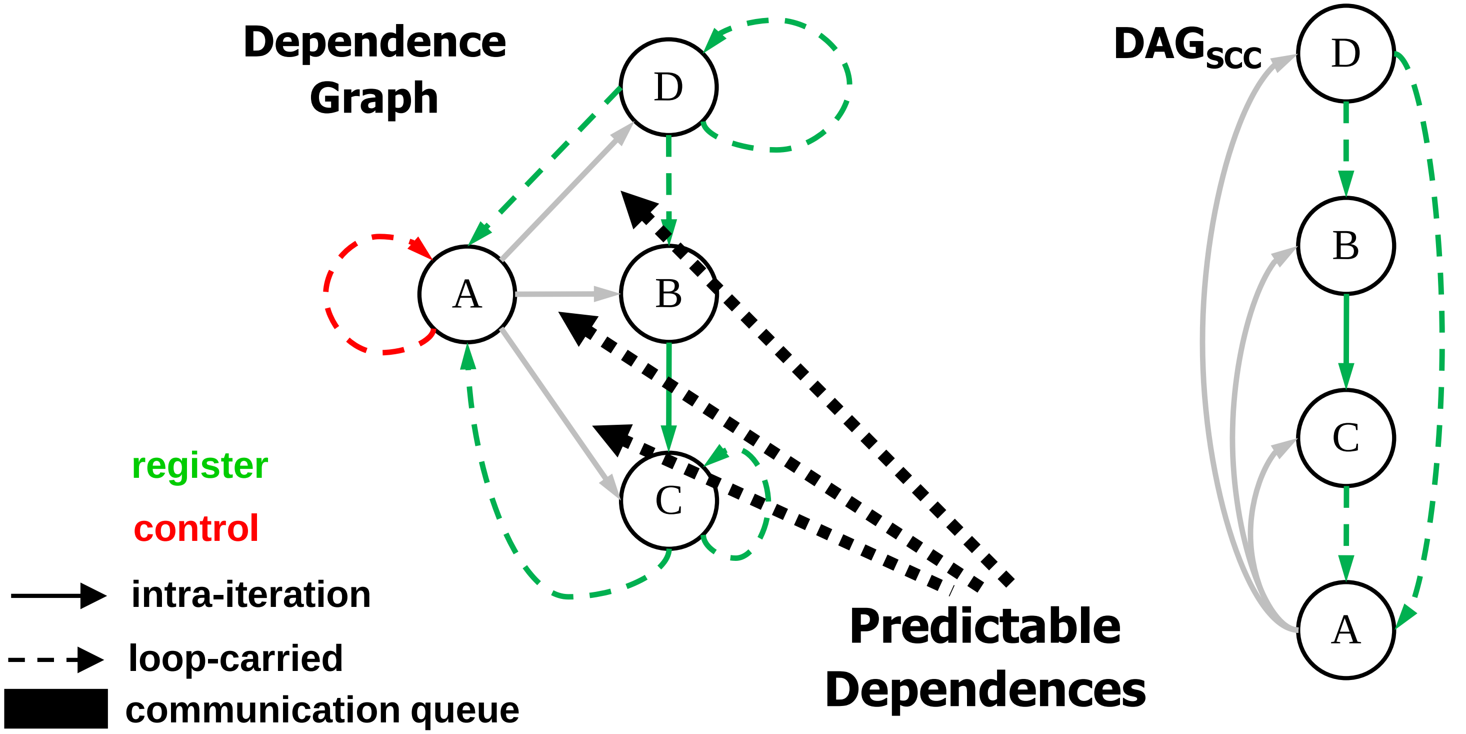
Why Speculation?

```
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;
```



Dependence Graph

DAG$_{SCC}$

register

control

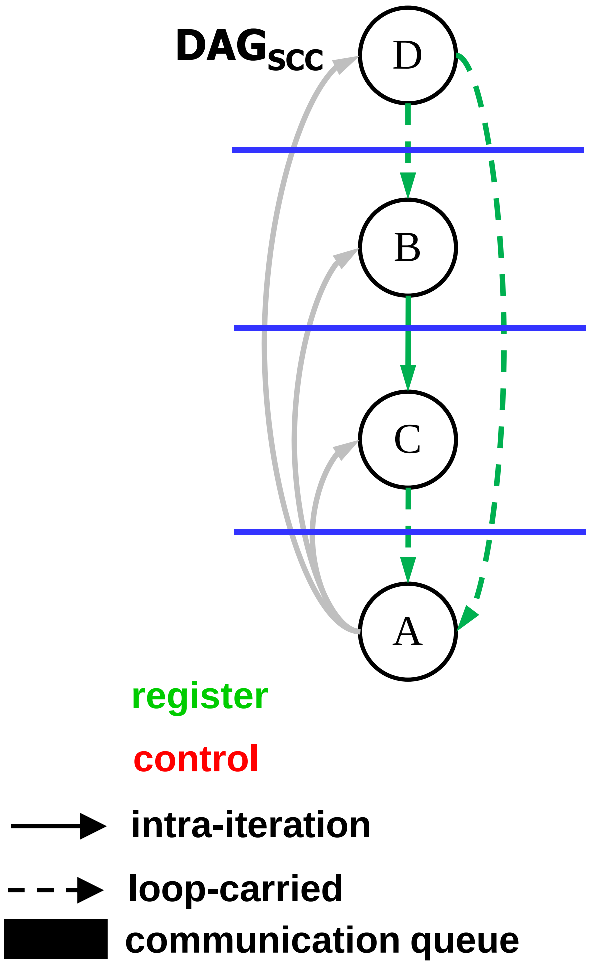
intra-iteration

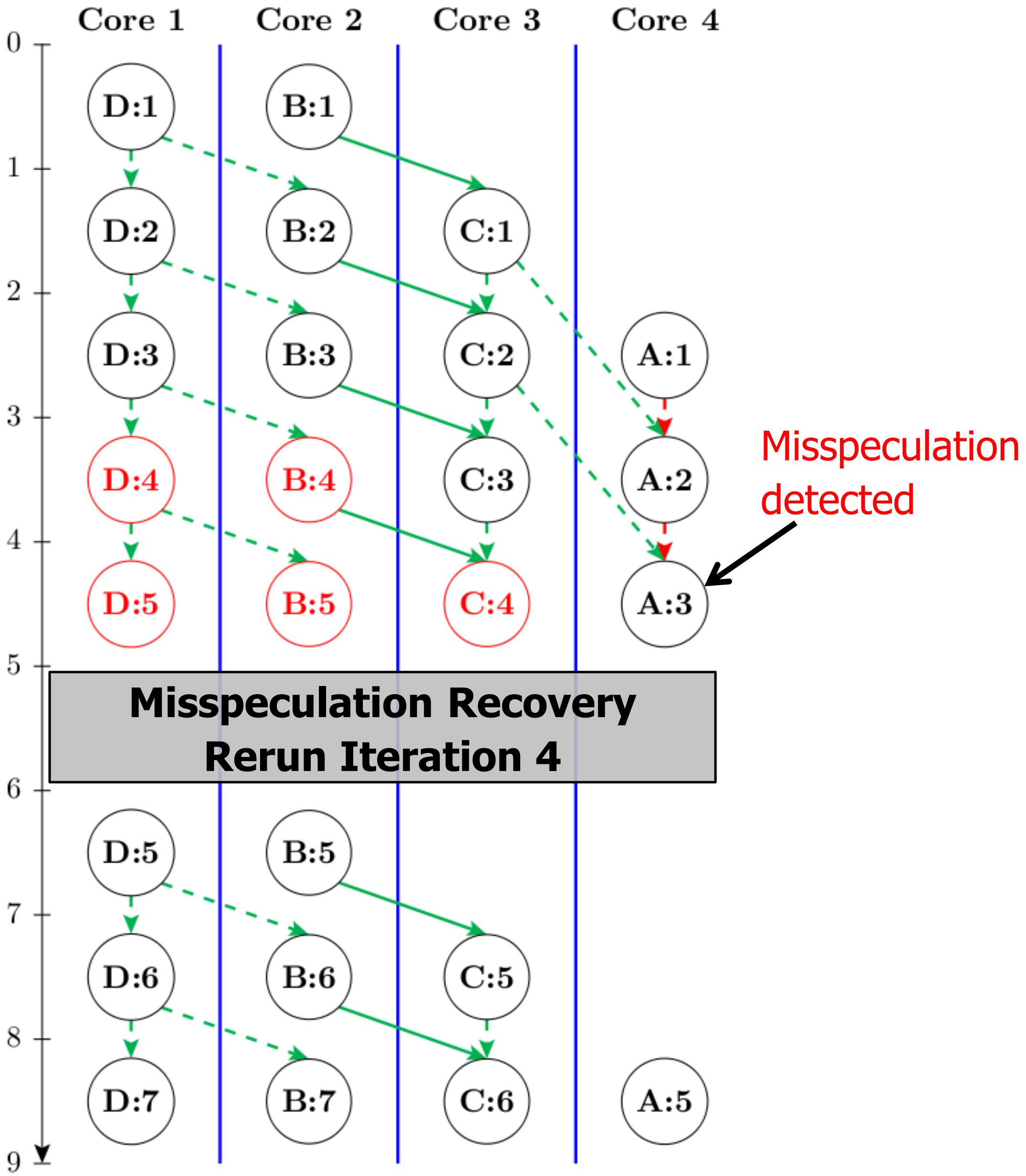
loop-carried

communication queue

Predictable Dependences

Execution Paradigm

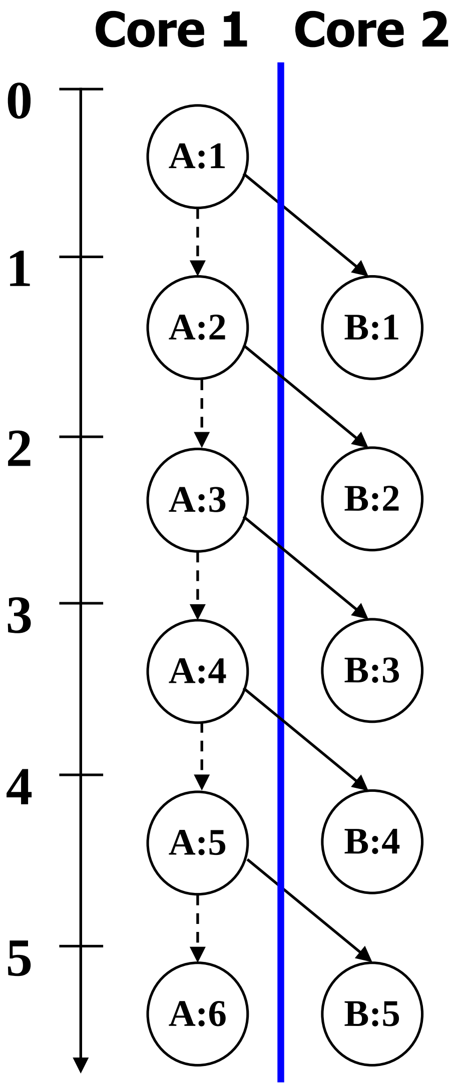


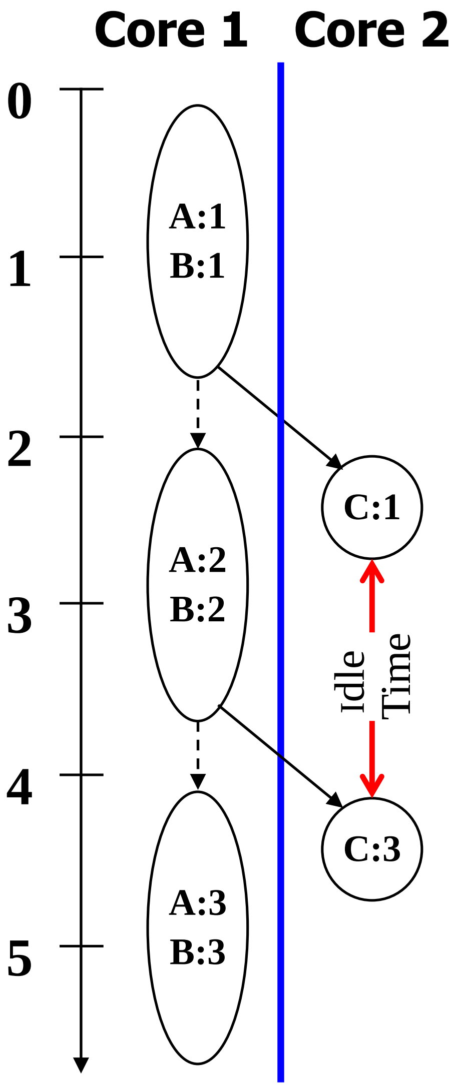


Misspeculation detected

Misspeculation Recovery
Rerun Iteration 4

Understanding PMT Performance





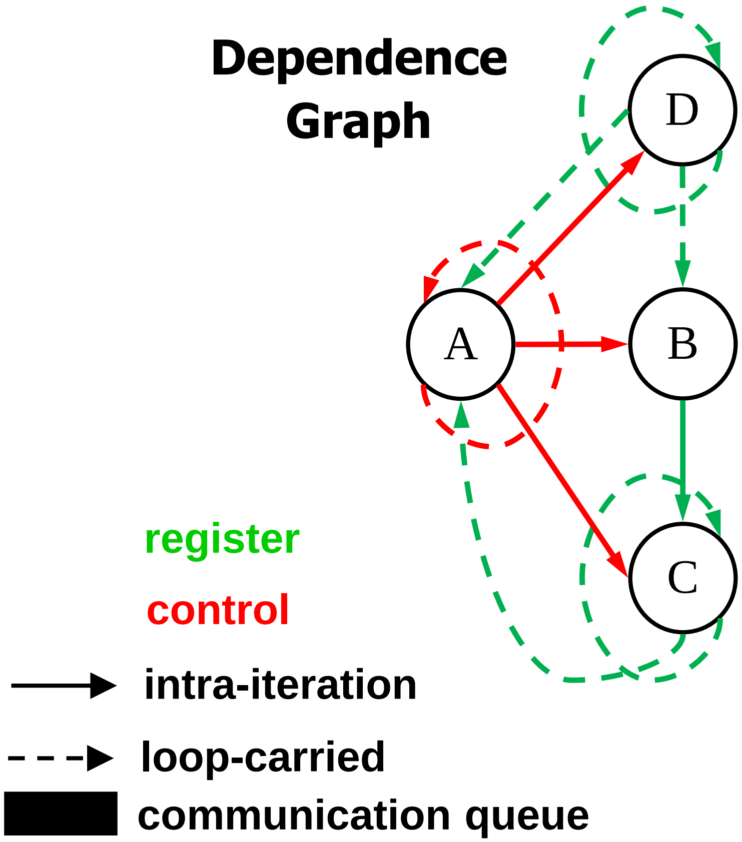
$$T \propto \max(t_i)$$

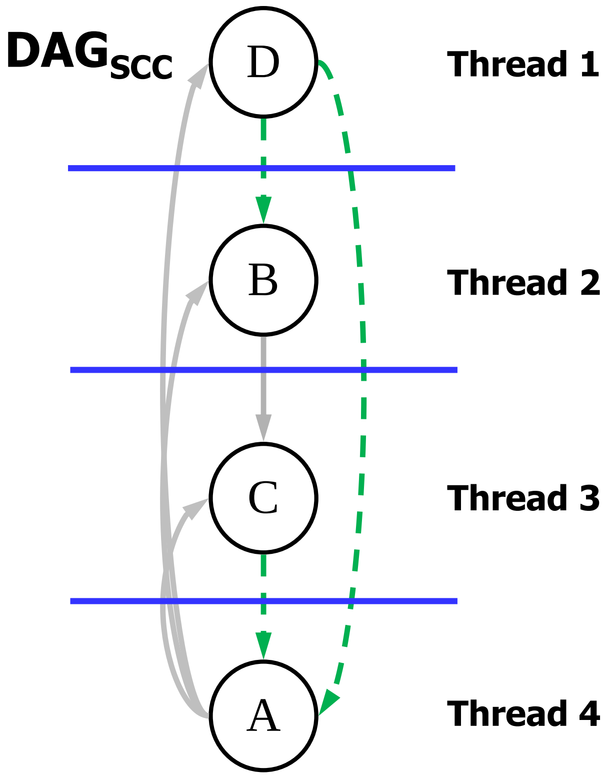
1. Rate t_i is at least as large as the longest dependence recurrence.
2. NP-hard to find longest recurrence.
3. Large loops make problem difficult in practice.

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

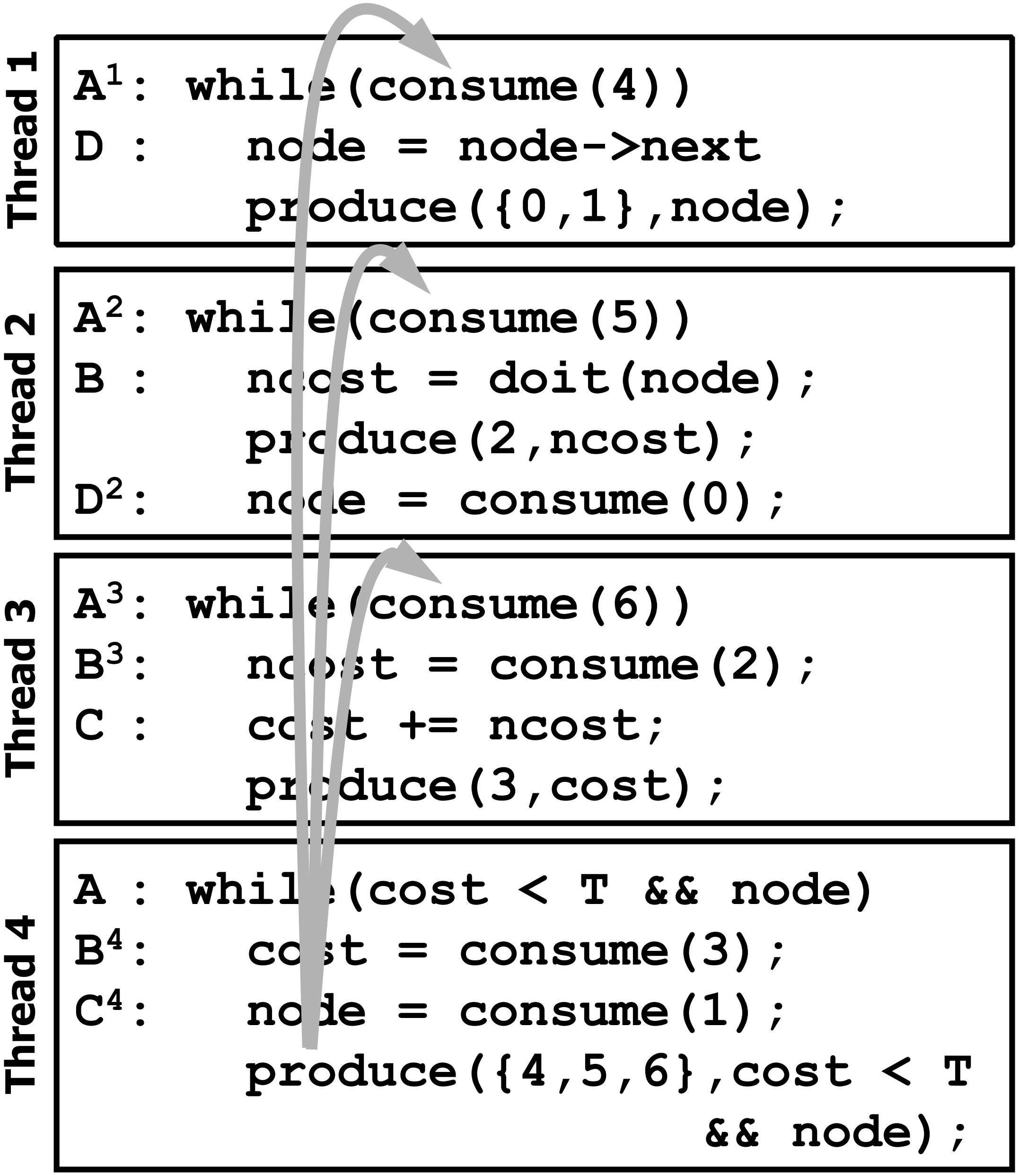
Selecting Dependences To Speculate

```
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

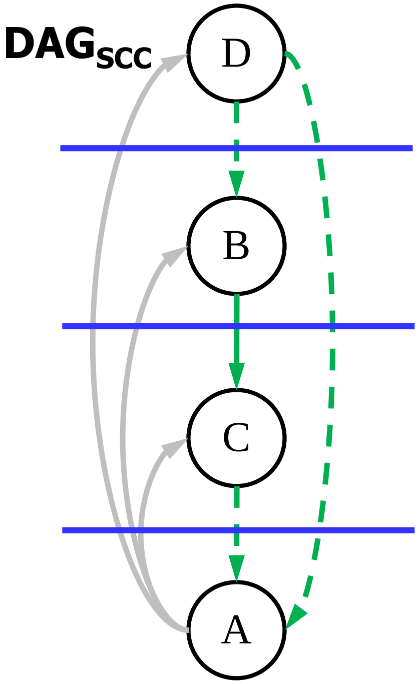
```
A^2: while(consume(5))
B  :   ncost = doit(node);
       produce(2,ncost);
D^2:   node = consume(0);
```

Thread 3

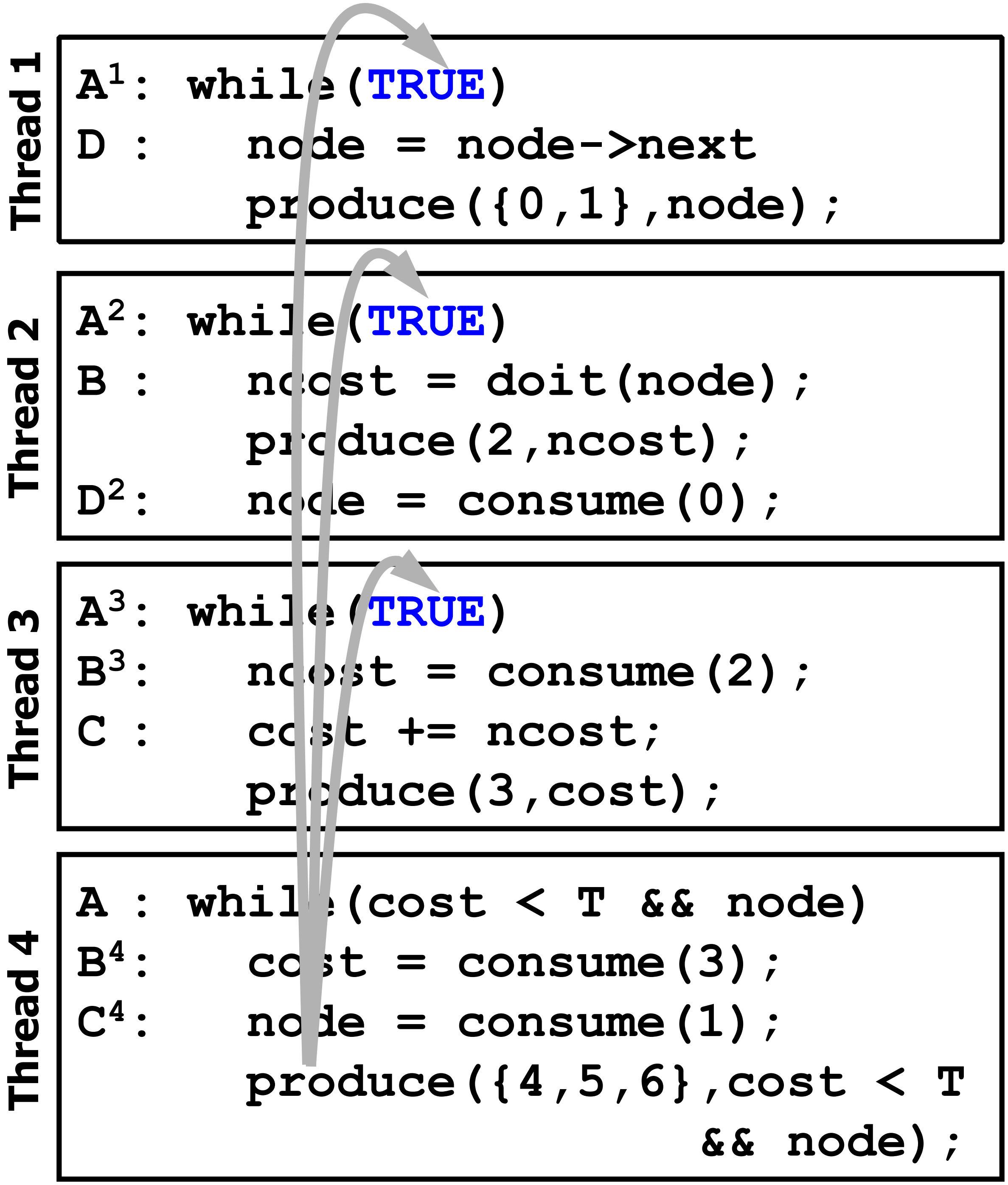
```
A^3: while(consume(6))
B^3:   ncost = consume(2);
C  :   cost += ncost;
       produce(3,cost);
```

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

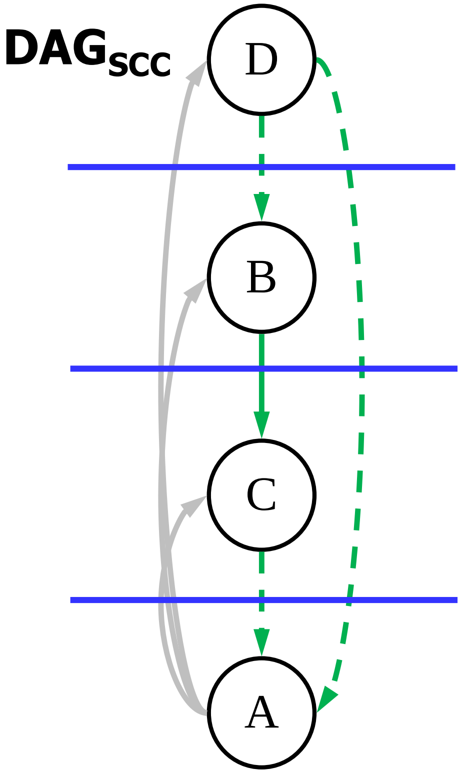
```
A^2: while(TRUE)
B  :    ncost = doit(node);
        produce(2,ncost);
D^2:    node = consume(0);
```

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);
```



Detecting Misspeculation

Thread 1

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

Thread 2

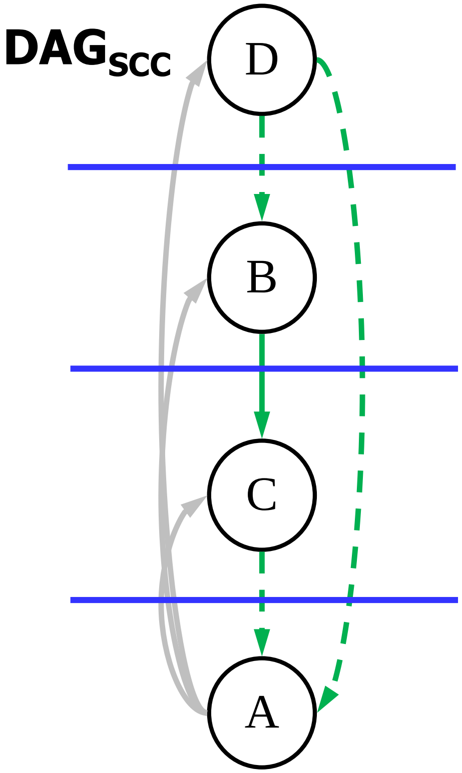
```
A^2: while(TRUE)
B :    ncost = doit(node);
       produce(2,ncost);
D^2:   node = consume(0);
```

Thread 3

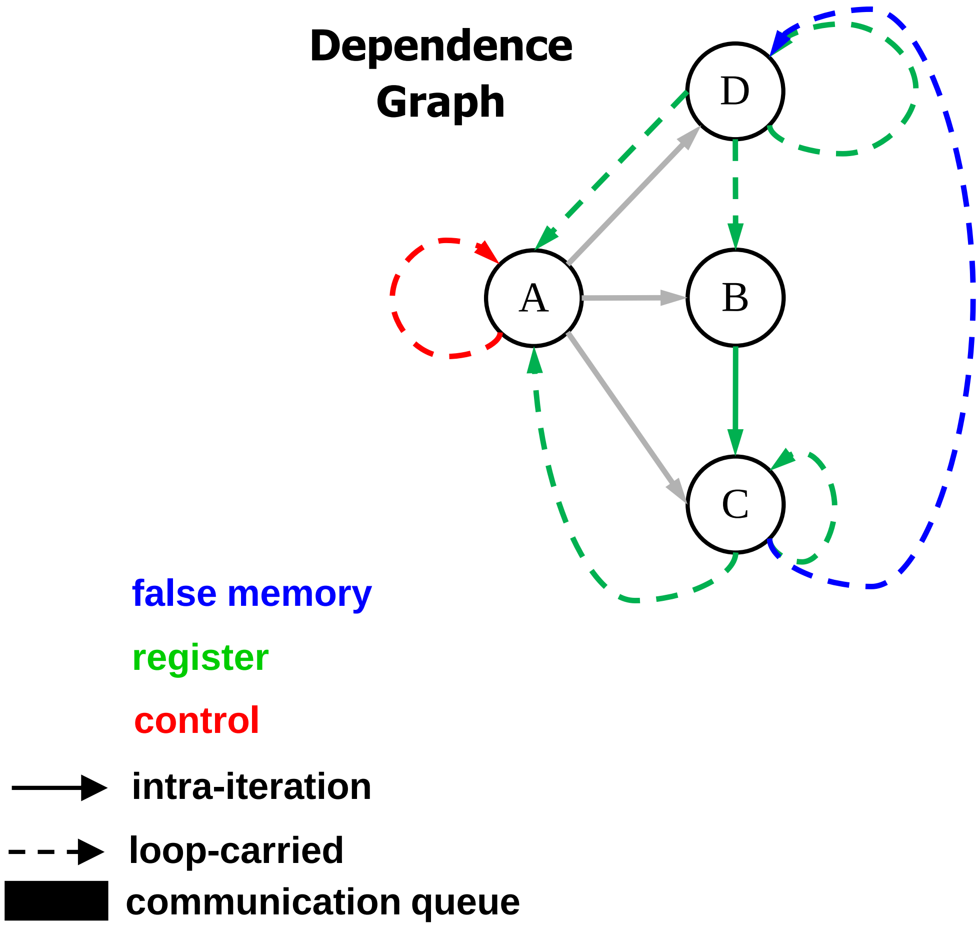
```
A^3: while(TRUE)
B^3:   ncost = consume(2);
C :    cost += ncost;
       produce(3,cost);
```

Thread 4

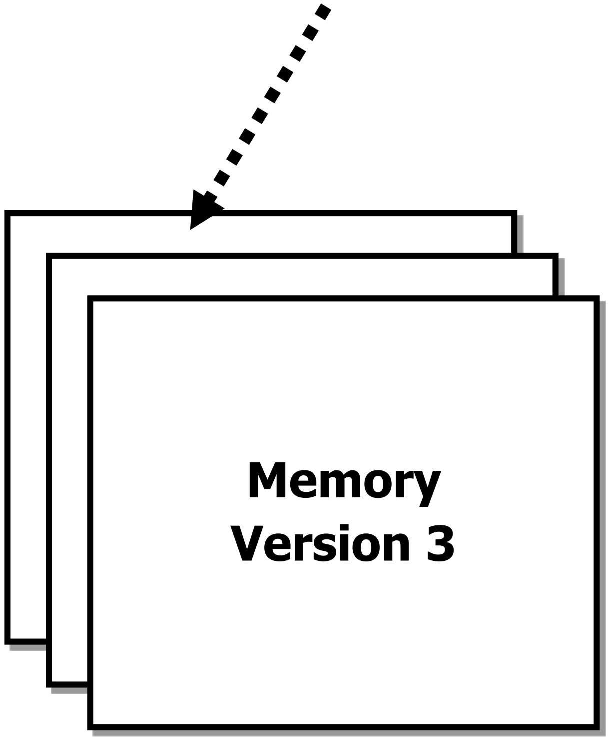
```
A : while(cost < T && node)
B^4:   cost = consume(3);
C^4:   node = consume(1);
       if(!(cost < T && node))
          FLAG_MISSPEC();
```



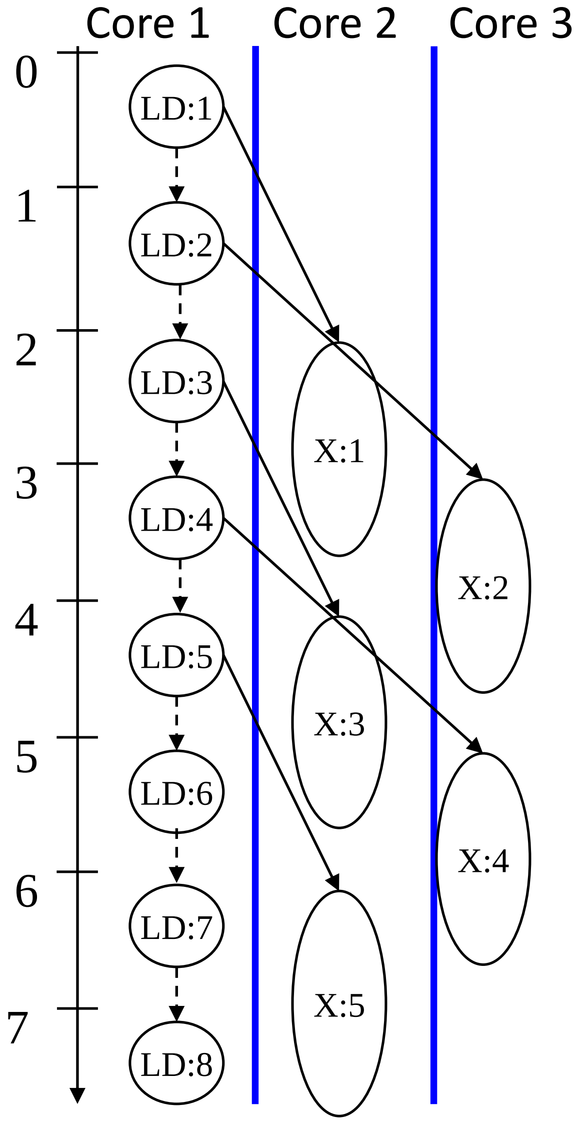
Breaking False Memory Dependences



Oldest Version Committed by Recovery Thread



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

Things to Think About – Speculation

- How do you decide what dependences to speculate?
  - Look solely at profile data?
  - How do you ensure enough profile coverage?
  - What about code structure?
  - What if you are wrong? Undo speculation decisions at run-time?
- How do you manage speculation in a pipeline?
  - Traditional definition of a transaction is broken
  - Transaction execution spread out across multiple cores

Things to Think About 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, or based on what else is running on the processor?
- How many cores can DSWP realistically scale to?