
Topic 14: Parallelism

COS 320

Compiling Techniques

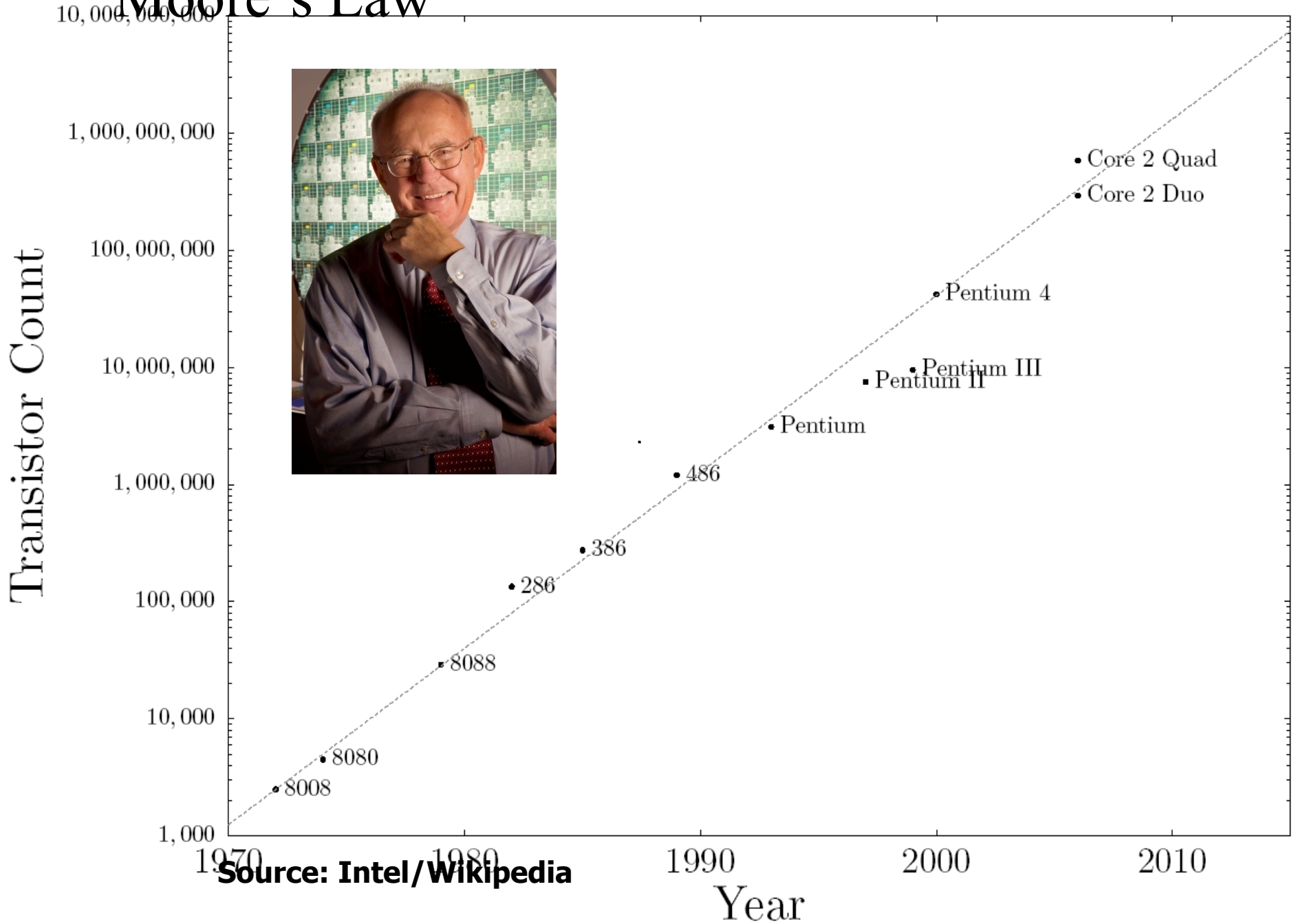
Princeton University
Spring 2015

Prof. David August

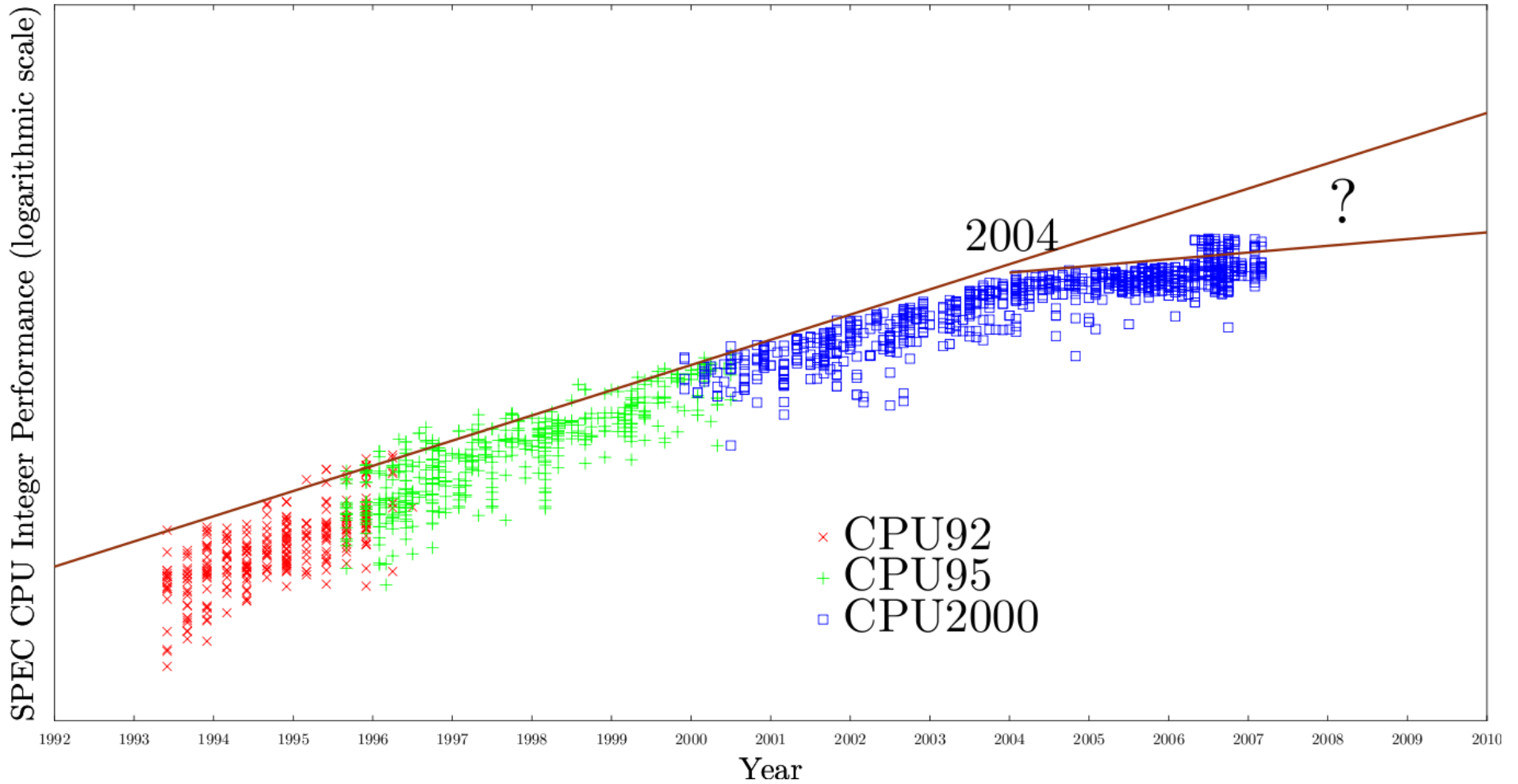
Final Exam!

- Friday May 22 at 1:30PM in FRIEND 006
- Closed book
- One Front/Back 8.5x11

Moore's Law



Single-Threaded Performance Not Improving

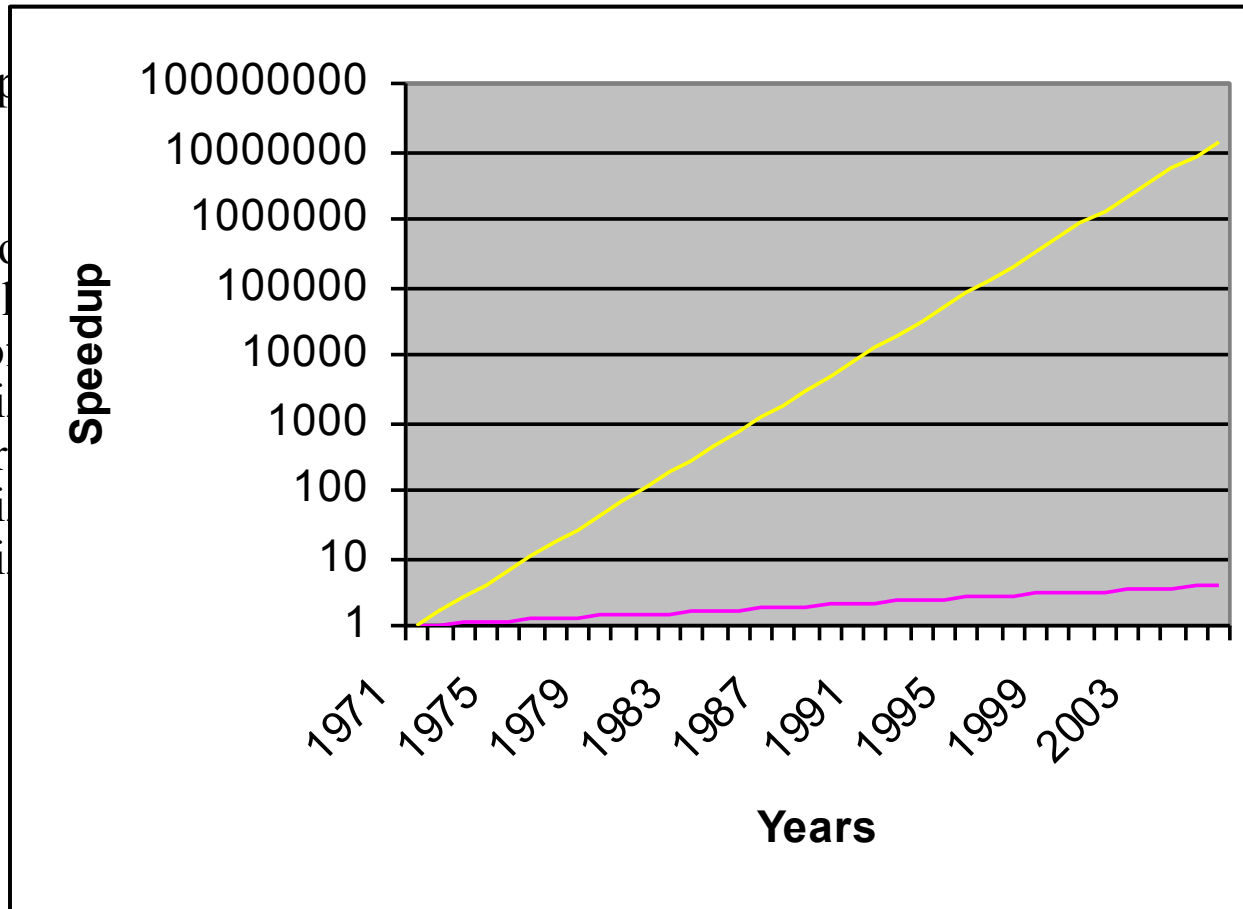


What about Parallel Programming? –or- What is Good About the Sequential Model?

- ❖ 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

- ❖ “Compilers are the Answer?”
- ❖ Run time compiler optimization ratio of 4X for optimization



optimizing
 bled. The
 compiler
 ratio is about
 piler
 piler

Conclusion – Compilers not about performance!

Are We Doomed?

A Step Back in Time: Old Skool
Parallelization

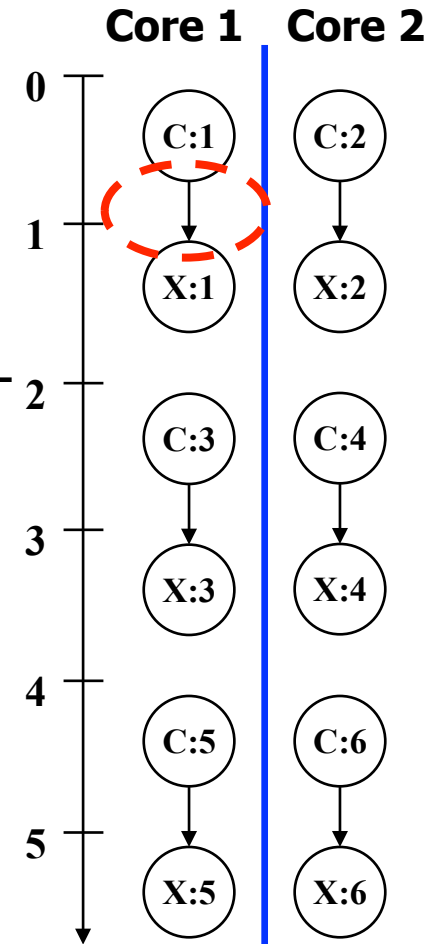
Parallelizing Loops In Scientific Applications

Scientific Codes (FORTRAN-like)

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

Independent
Multithreading
(IMT)

Example: DOALL
parallelization



What Information is Needed to Parallelize?

- ❖ Dependences within iterations are fine
- ❖ Identify the presence of cross-iteration data-dependences
 - » Traditional analysis is inadequate for parallelization. For instance, it does not distinguish between different executions of the same statement in a loop.
- ❖ Array dependence analysis enables optimization for parallelism in programs involving arrays.
 - » Determine pairs of iterations where there is a data dependence
 - » Want to know all dependences, not just yes/no

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

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

Affine/Linear Functions

- ❖ $f(i_1, i_2, \dots, i_n)$ is affine, if it can be expressed as a sum of a constant, plus constant multiples of the variables. i.e.

$$f = c_0 + \sum_{i=1}^n c_i x_i$$

- ❖ Array subscript expressions are usually affine functions involving loop induction variables.

- ❖ Examples:

» a[i]	affine
» a[i+j -1]	affine
» a[i*j]	non-linear, not affine
» a[2*i+1, i*j]	linear/non-linear, not affine
» a[b[i] + 1]	non linear (indexed subscript). not affine

Array Dependence Analysis

```
for (i = 1; i < 10; i++) {  
    X[i] = X[i-1]  
}
```

To find all the data dependences, we check if

1. $X[i-1]$ and $X[i]$ refer to the same location;
2. different instances of $X[i]$ refer to the same location.
 - » For 1, we solve for i and i' in
 $1 \leq i \leq 10, 1 \leq i' \leq 10$ and $i - 1 = i'$
 - » For 2, we solve for i and i' in
 $1 \leq i \leq 10, 1 \leq i' \leq 10, i = i'$ and $i \neq i'$ (between different dynamic accesses)

There is a dependence since there exist integer solutions to 1. e.g. $(i=2, i'=1)$, $(i=3, i'=2)$. 9 solutions exist.

There is no dependences among different instances of $X[i]$ because 2 has no solutions!

Array Dependence Analysis - Summary

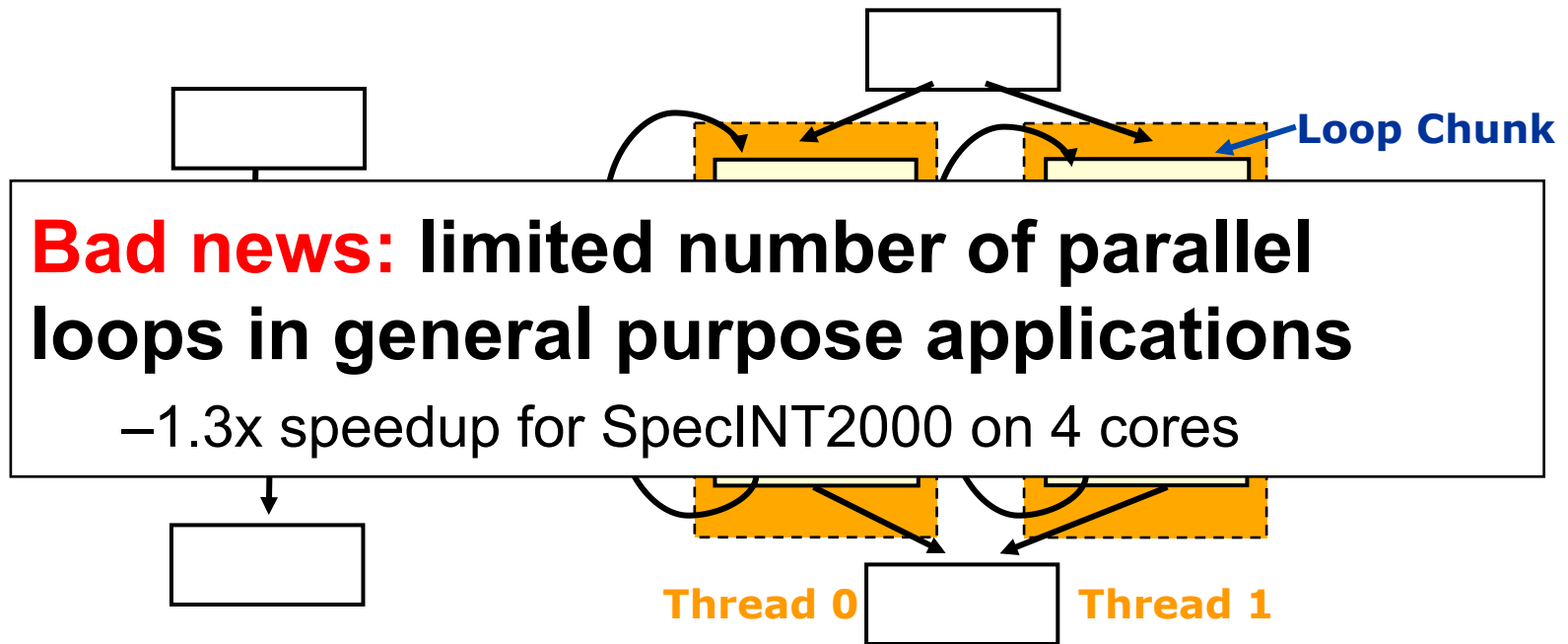
- ❖ Array data dependence basically requires finding integer solutions to a system (often refers to as dependence system) consisting of equalities and inequalities.
- ❖ Equalities are derived from array accesses.
- ❖ Inequalities from the loop bounds.
- ❖ It is an integer linear programming problem.
- ❖ ILP is an NP-Complete problem.
- ❖ Several Heuristics have been developed.
 - » Omega – U. Maryland

Loop Parallelization Using Affine Analysis Is Proven Technology

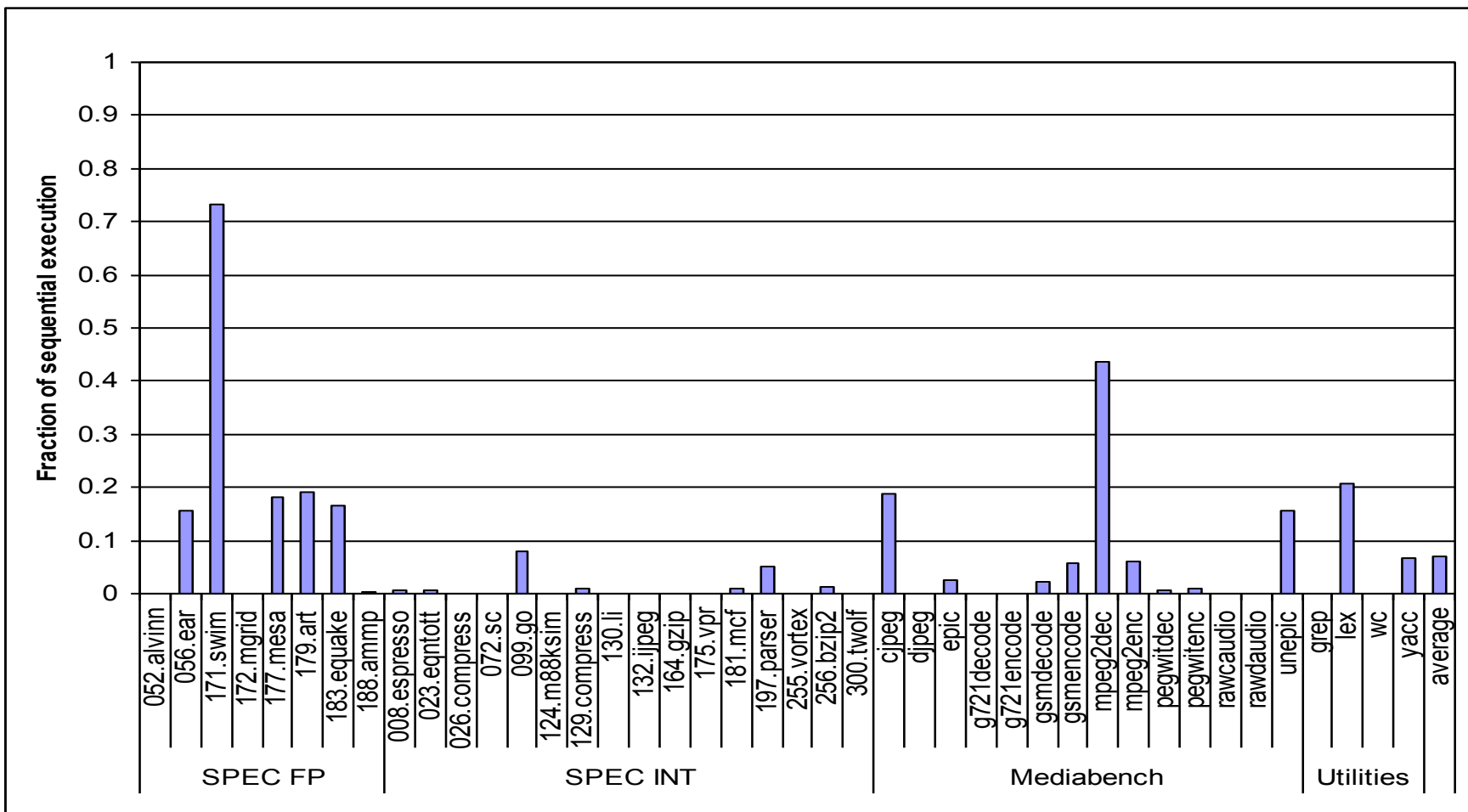
- ❖ DOALL Loop
 - » No loop carried dependences for a particular nest
 - » Loop interchange to move parallel loops to outer scopes
- ❖ Other forms of parallelization possible
 - » DOAcross, DOpipes
- ❖ Optimizing for the memory hierarchy
 - » Tiling, skewing, etc.
- ❖ Real compilers available – KAP, Portland Group, gcc
- ❖ For better information, see
 - » http://gcc.gnu.org/wiki/Graphite?action=AttachFile&do=get&target=graphite_lambda_tutorial.pdf

Back to the Present – Parallelizing C and C++ Programs

Loop Level Parallelization



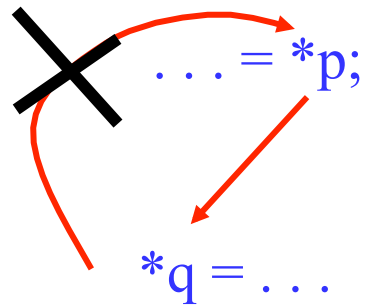
DOALL Loop Coverage



What's the Problem?

1. Memory dependence analysis

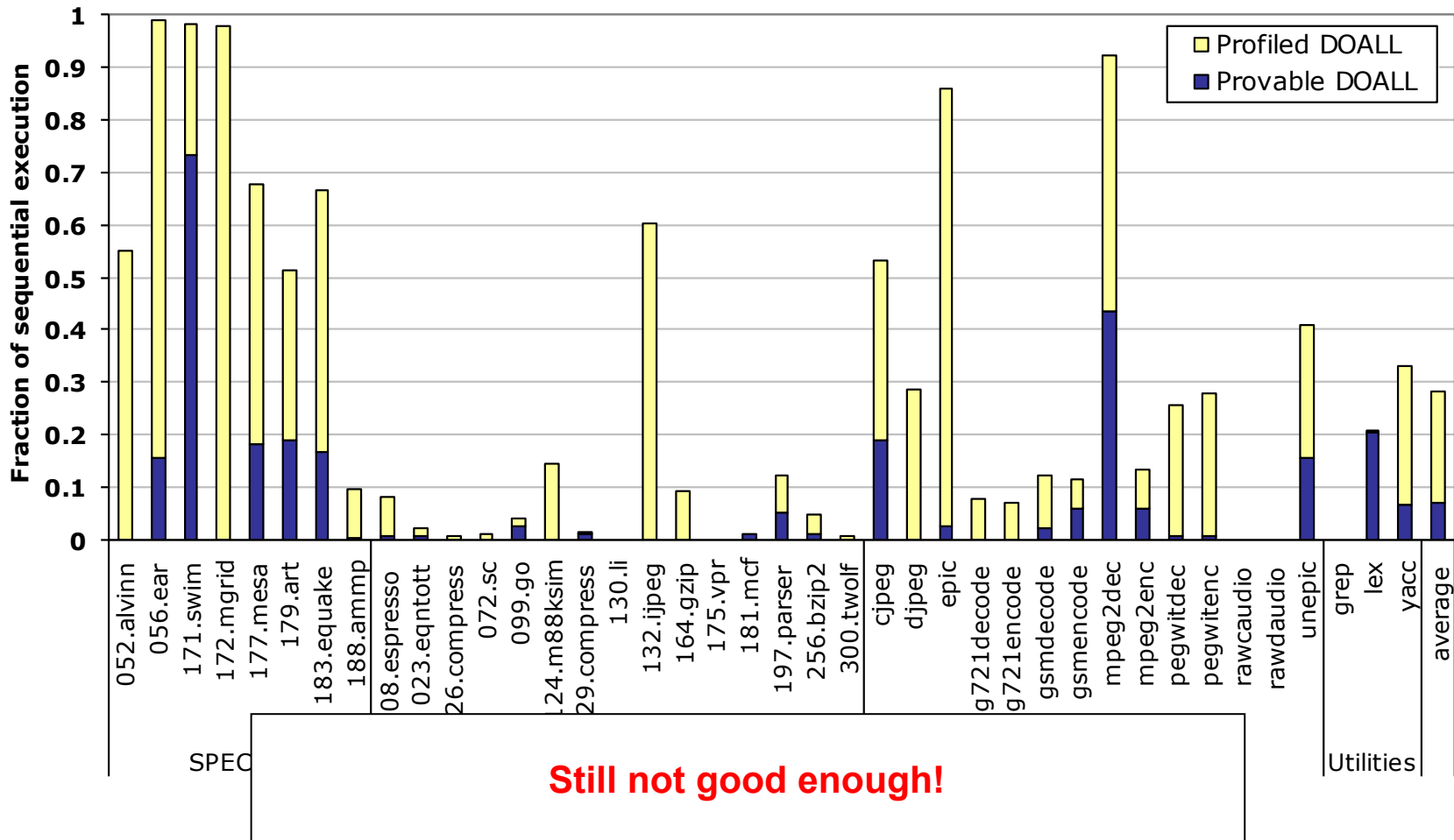
```
for (i=0; i<100; i++) {
```



```
}
```

➔ Memory dependence profiling
and speculative parallelization

DOALL Coverage – Provable and Profiled

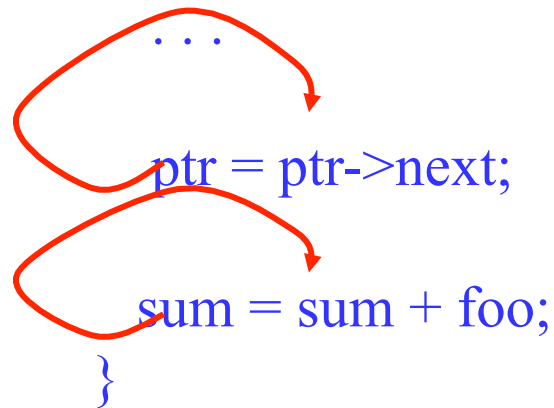


What's the Next Problem?

2. Data dependences

```
while (ptr != NULL) {
```

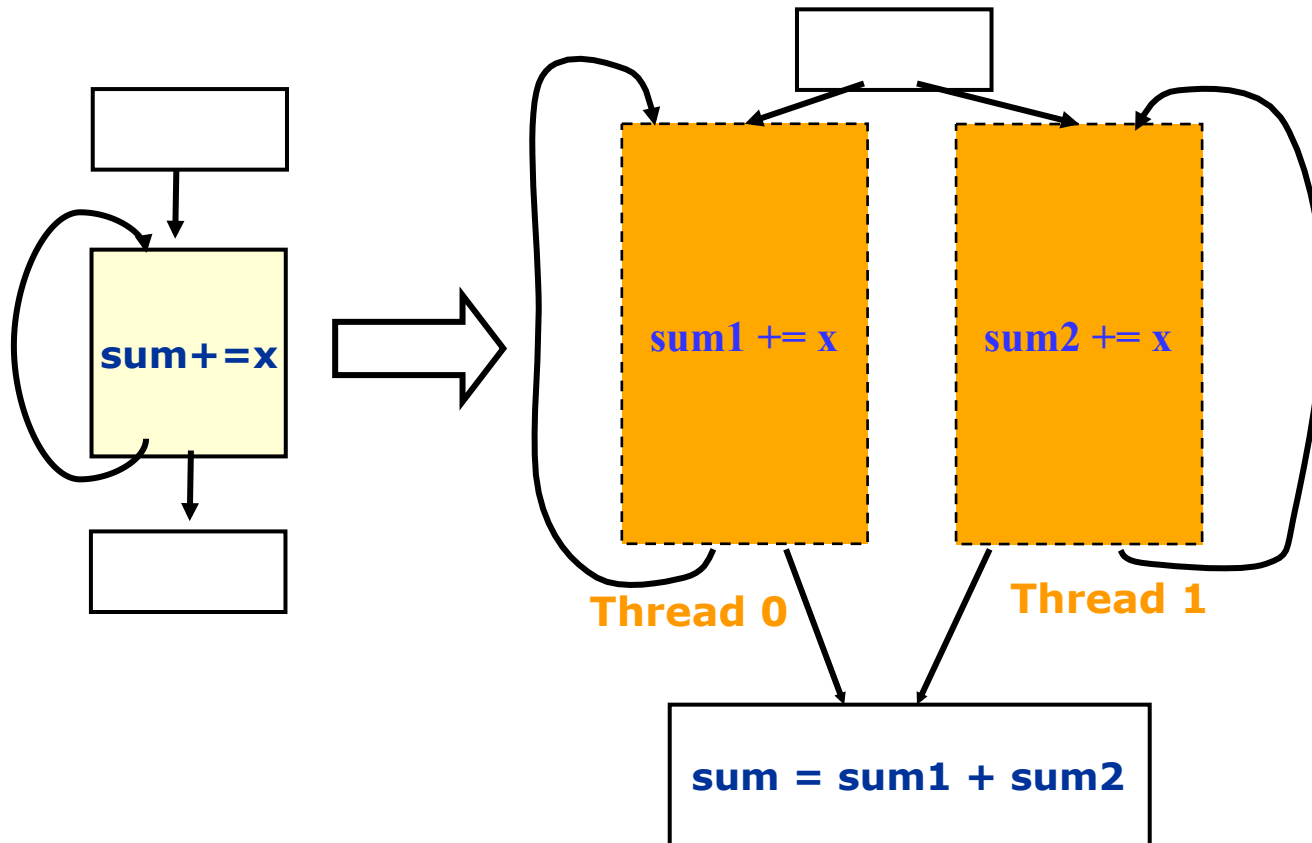
```
    ...  
    ptr = ptr->next;  
    sum = sum + foo;  
}
```



 Compiler transformations

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

Apply accumulator variable expansion!



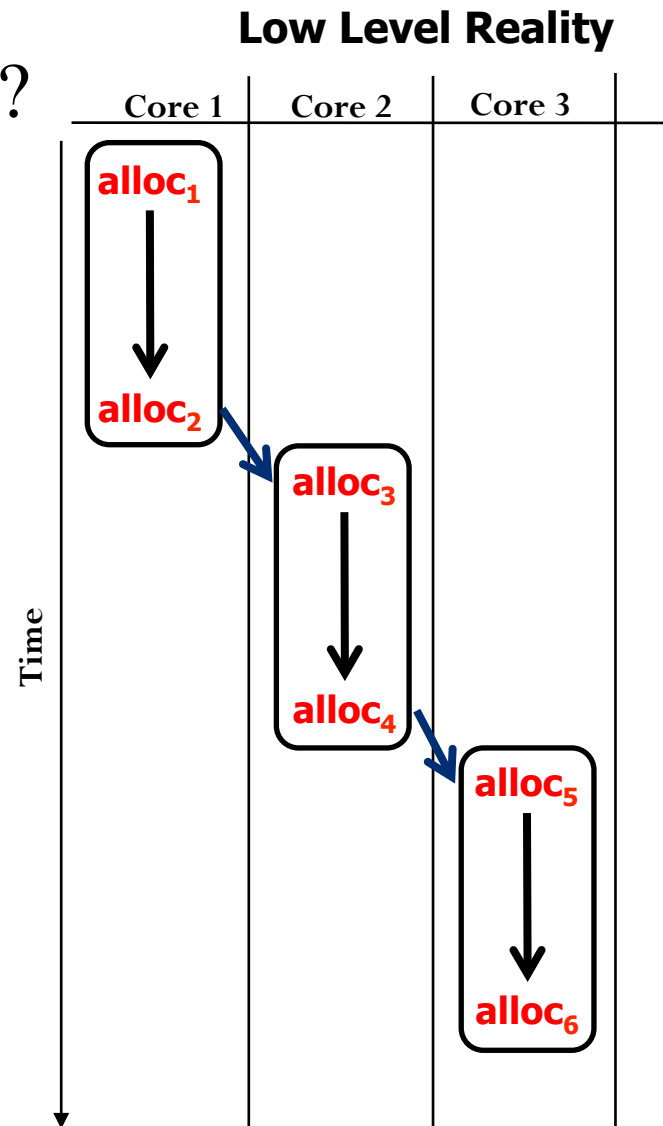
Data Dependences Inhibit Parallelization

- ❖ 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

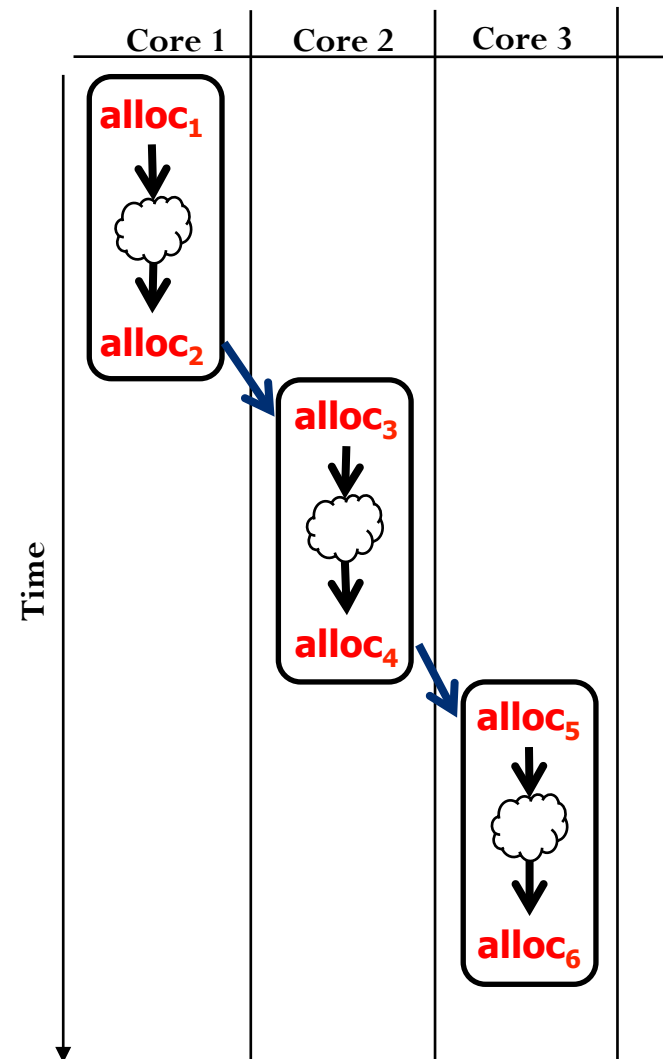
```
char *memory;  
  
void * alloc(int size);  
  
void * alloc(int size) {  
    void * ptr = memory;  
    memory = memory + size;  
    return ptr;  
}
```



Low Level Reality

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

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

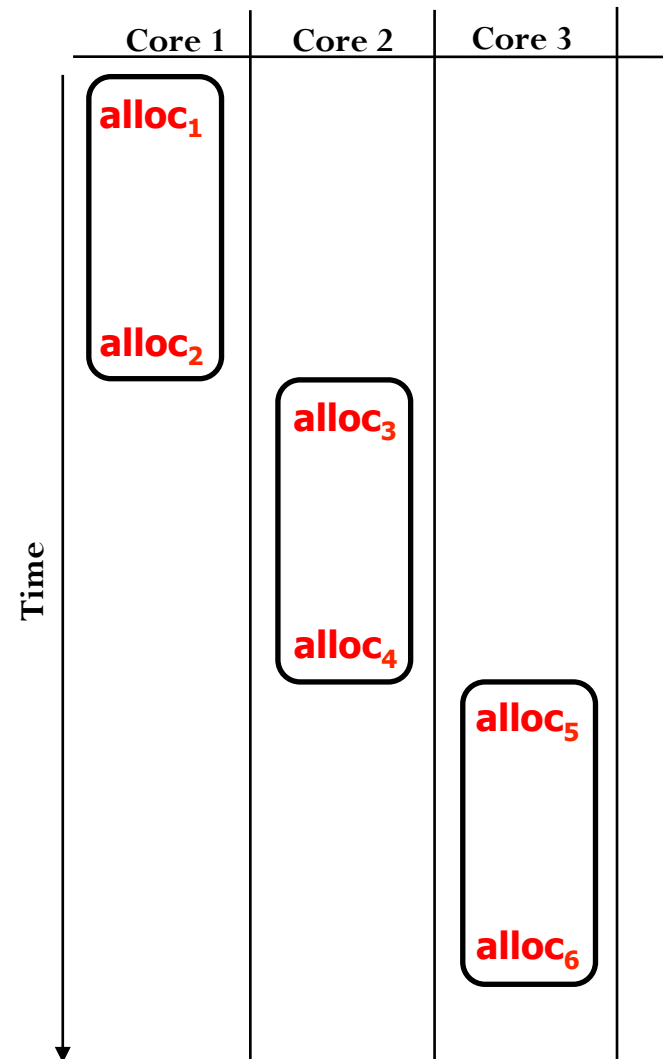
Low Level Reality

```
char *memory;
```

```
@Commutative
```

```
void * alloc(int size);
```

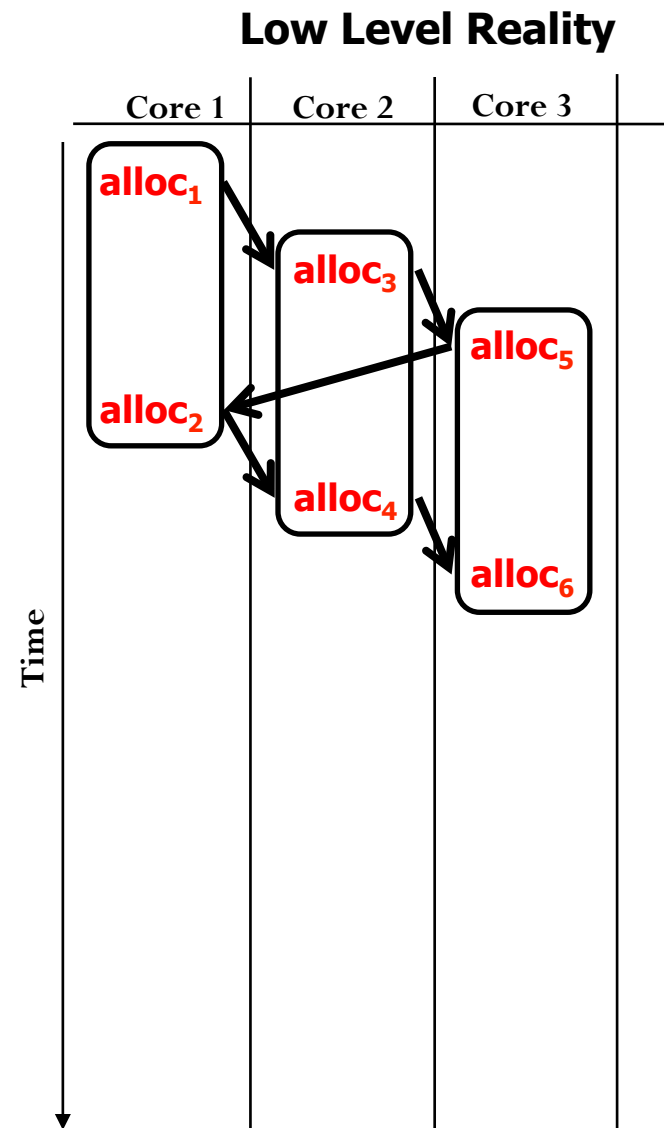
```
void * alloc(int size) {  
    void * ptr = memory;  
    memory = memory + size;  
    return ptr;  
}
```



```
char *memory;
```

```
@Commutative  
void * alloc(int size);
```

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



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
 - Ÿ When to create a new dictionary in a compression scheme

```

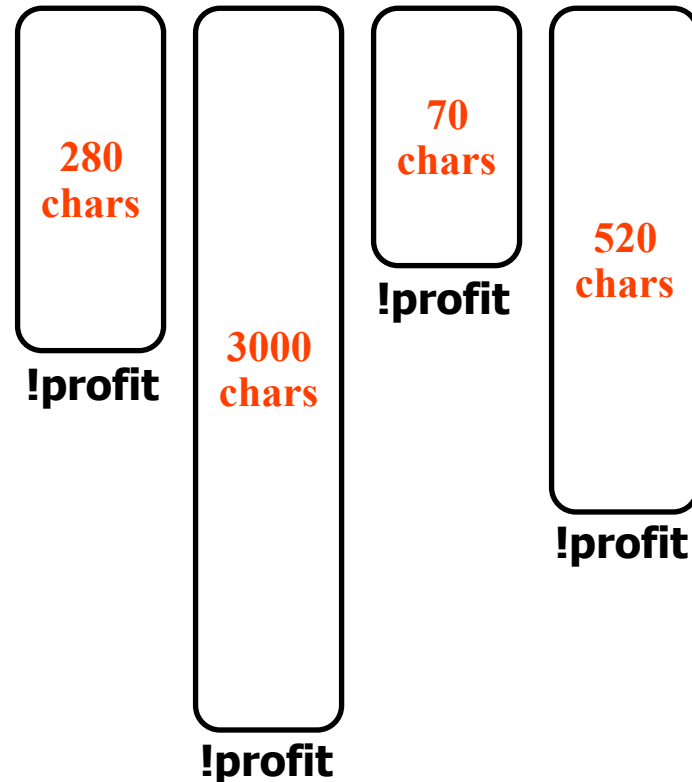
#define CUTOFF 100
dict = create_dict();
count = 0;
while (char = read(1)) {
    if (char == '\n') {
        profitable = compress(char, dict);
        compress(char, dict);

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

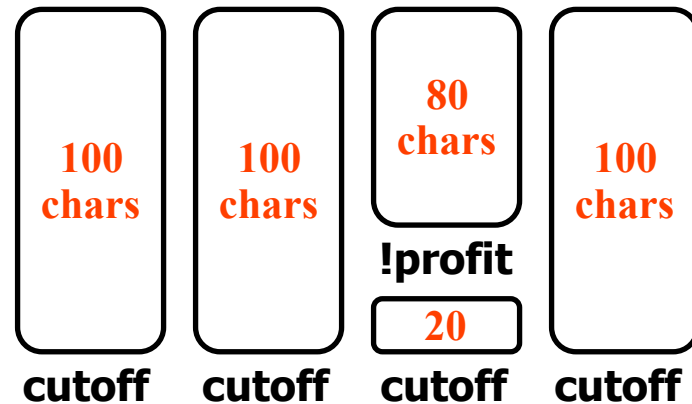
        count++;
    }
}
finish_dict(dict);

```

Sequential Program



Parallel Program



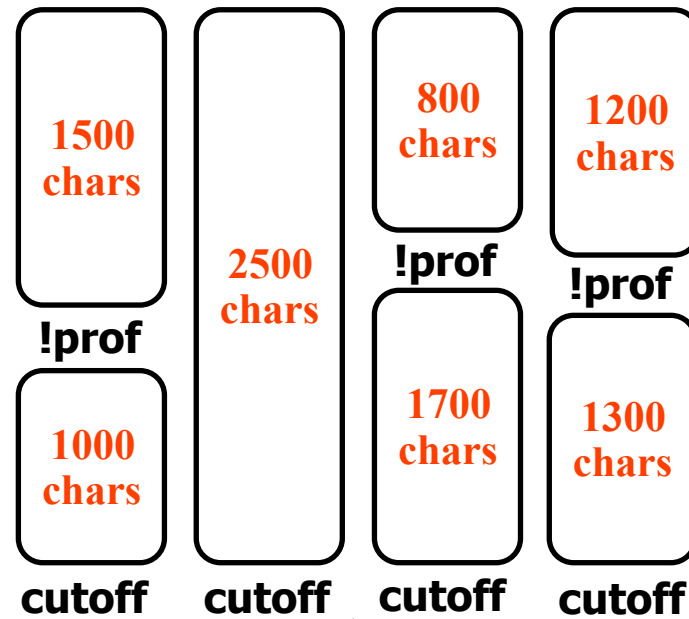
```

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

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

```

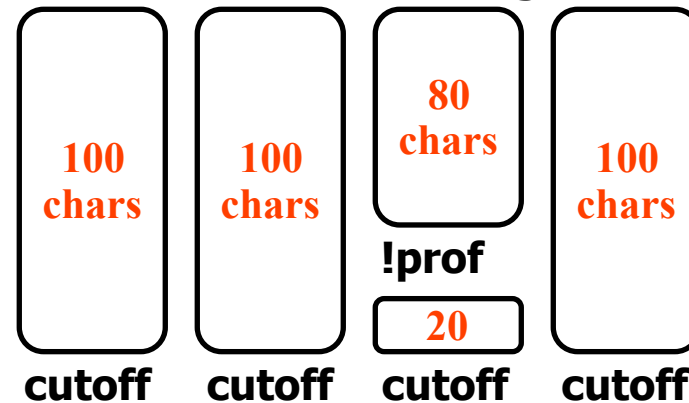
2-Core Parallel Program



**Reset every
2500 characters**



64-Core Parallel Program



**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.zip*

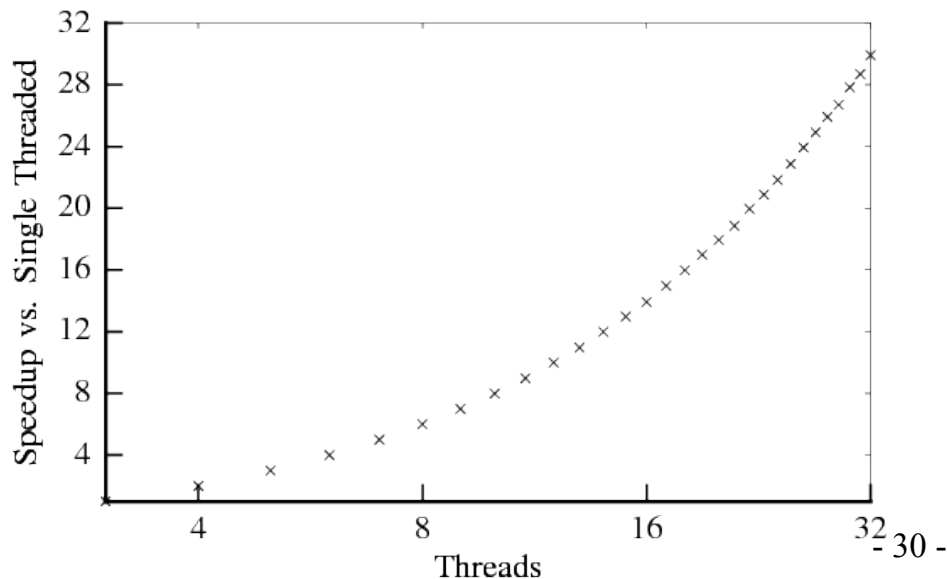
```
dict = create_dict();
while((char = read(1))) {
    profitable =
        compress(char, dict)

    @YBRANCH(probability=.00001)
    if(!compressable(&dict);
    } dict = restart(dict);
} }
finish_dict(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;
    }

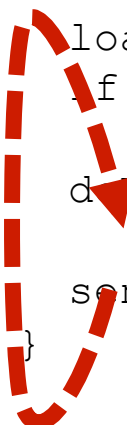
    count++;
}
finish_dict(dict);
```




256.bzip2

```
unsigned char *block;  
int last_written;
```


```
compressStream(in, out) {  
    while (True) {  
        loadAndRLEsource(in);  
        if (!last) break;  
        doReversibleTransform();  
        sendMTFValues(out);  
    }  
}
```



```
doReversibleTransform() {  
    .  
    sortIt();  
    ..  
}
```




```
sortIt() {  
    .  
    printf(...);  
    ..  
}
```



Parallelization techniques must look inside function calls to expose operations that cause synchronization.

197.parser

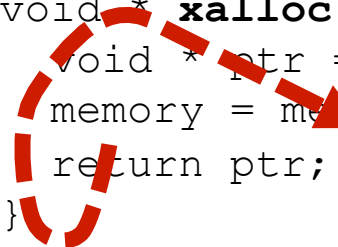
```
batch_process() {  
    while(True) {  
        sentence = read();  
        if (!sentence) break;  
        parse(sentence);  
        print(sentence);  
    }  
}
```



High-Level View:

Parsing a sentence is independent of any other sentence.

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



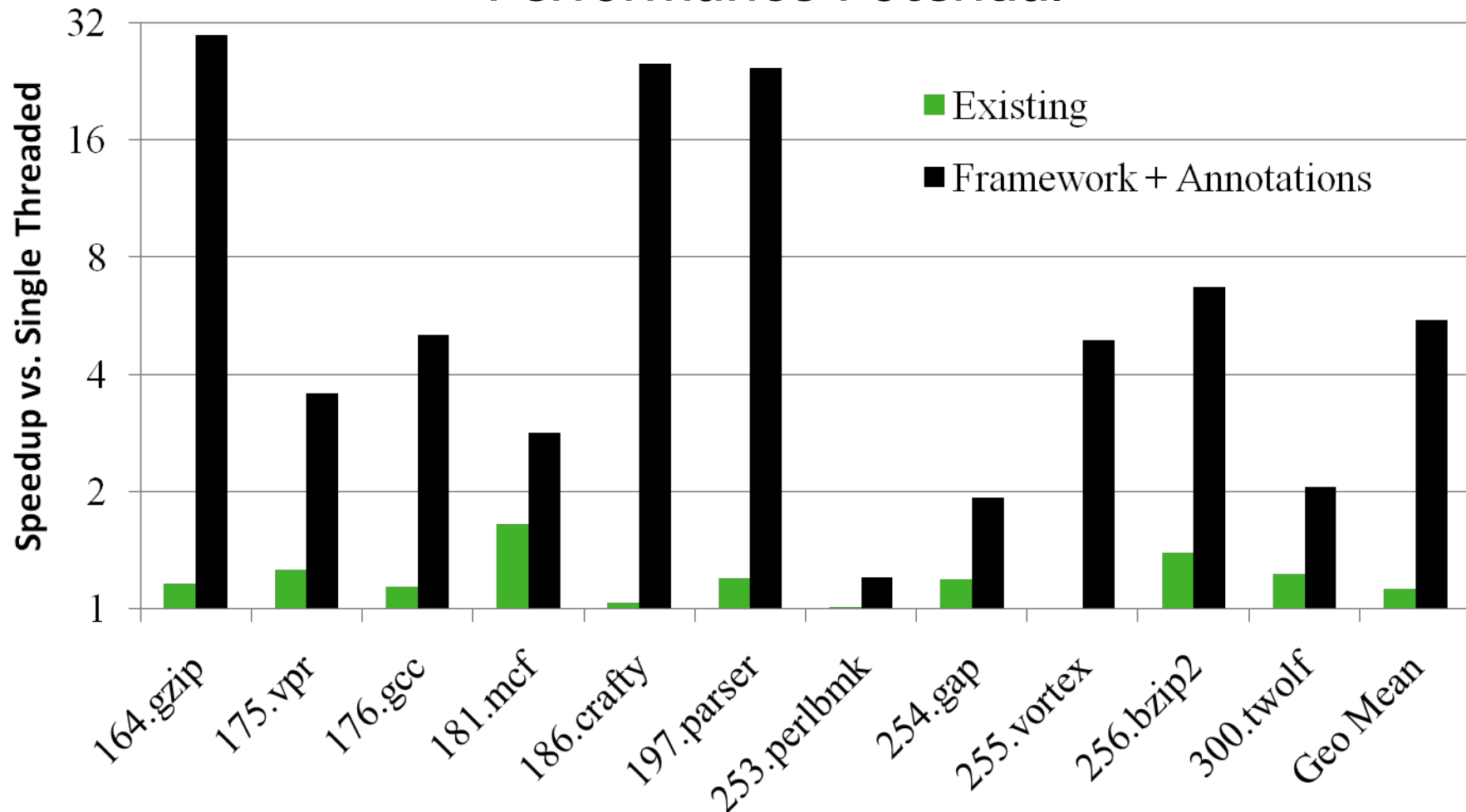
Low-Level Reality:

Implementation dependences inside functions called by *parse* lead to large sequential regions.

		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.perlbnk	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~~

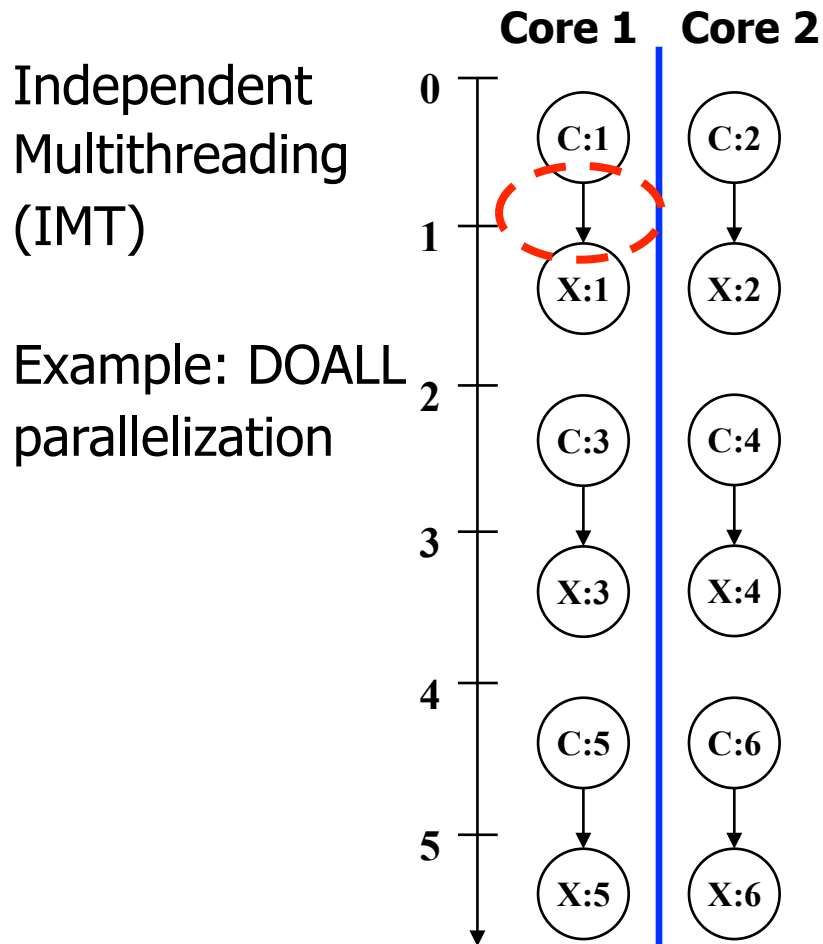
What About Non-Scientific Codes???

Scientific Codes (FORTRAN-like)

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

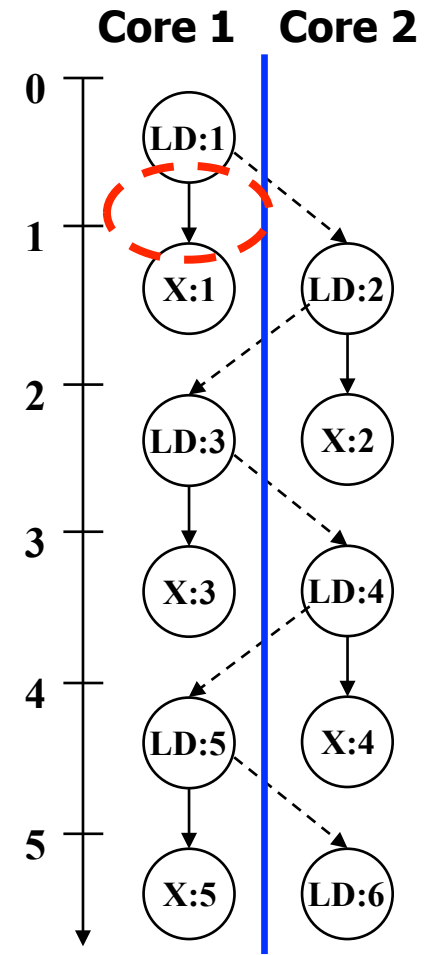
General-purpose Codes (legacy C/C++)

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



Cyclic Multithreading (CMT)

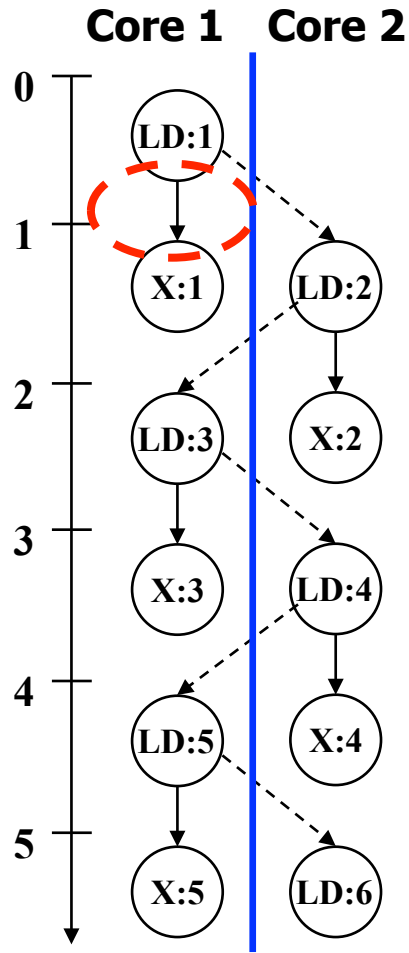
Example: DOACROSS [Cytron, ICPP 86]



Alternative Parallelization Approaches

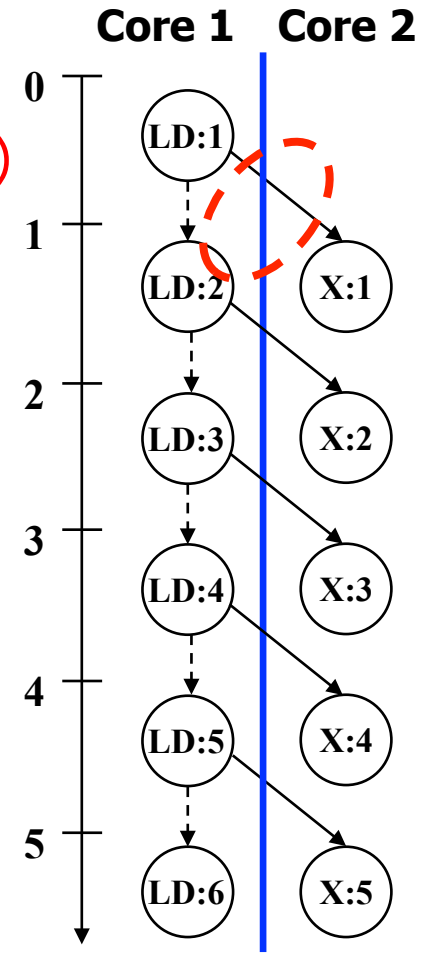
```
while(ptr = ptr->next)    // LD
    ptr->val = ptr->val + 1; // X
```

Cyclic
Multithreading
(CMT)

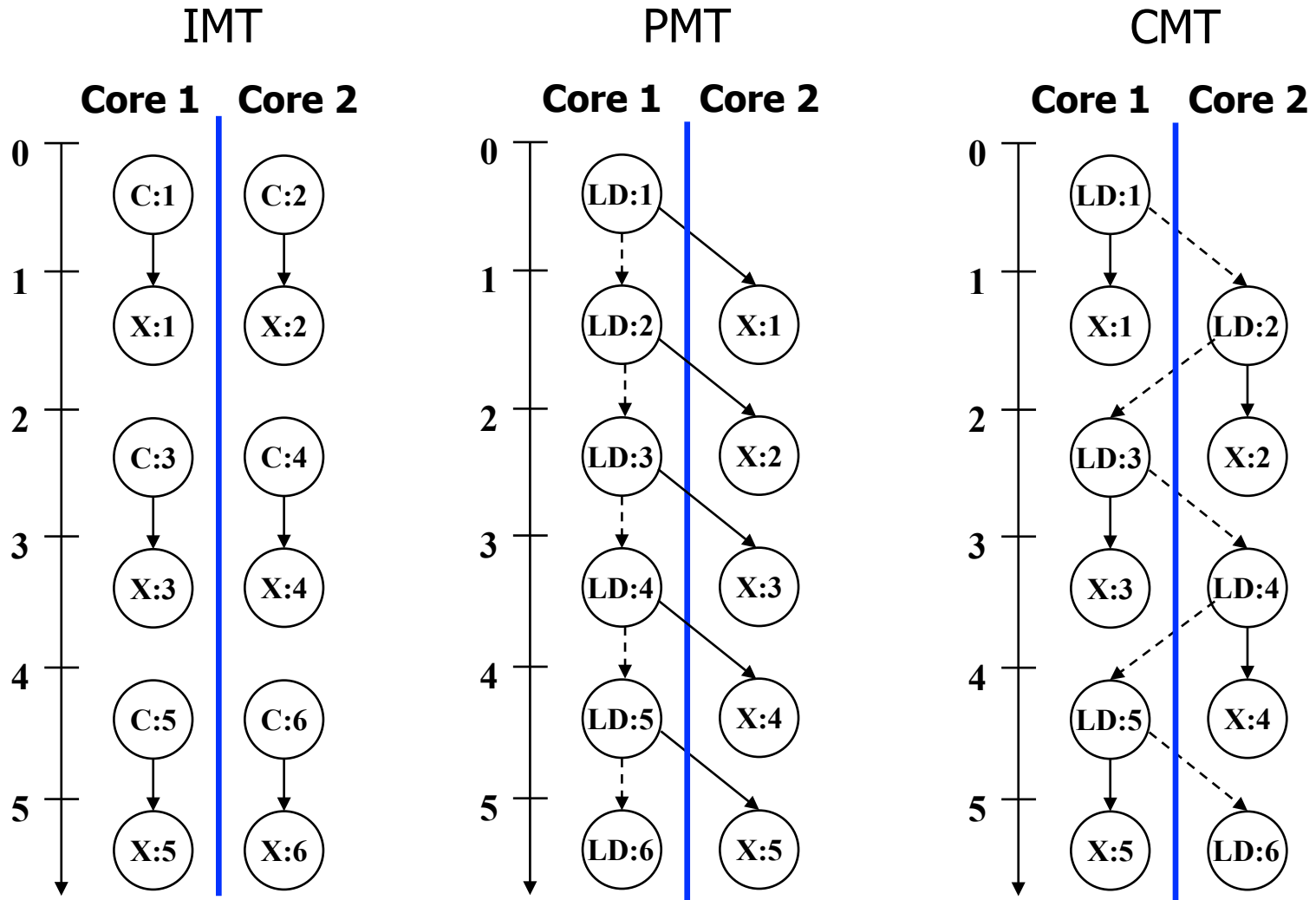


Pipelined
Multithreading (PMT)

Example: DSWP
[PACT 2004]

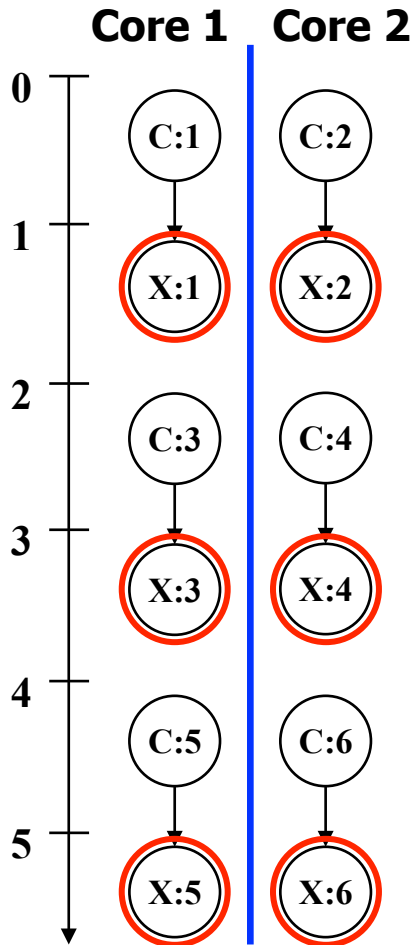


Comparison: IMT, PMT, CMT



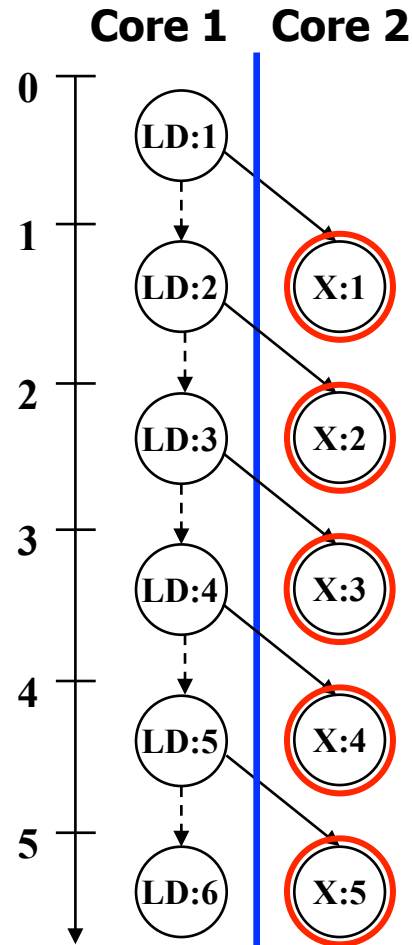
Comparison: IMT, PMT, CMT

IMT



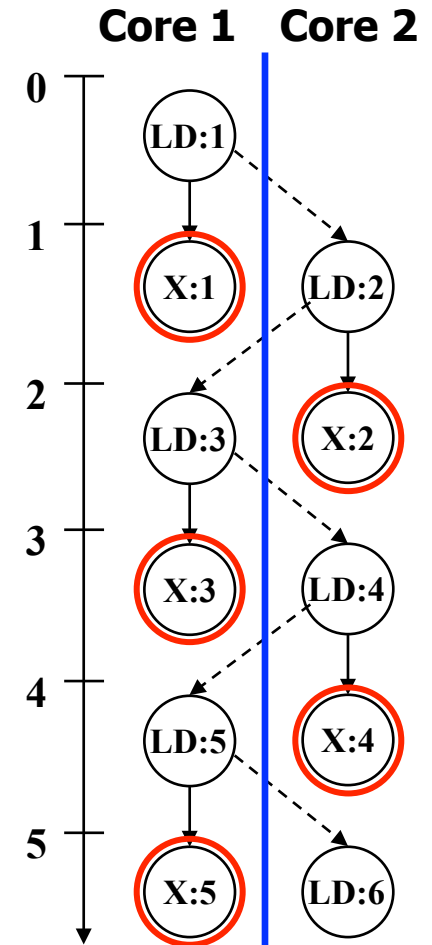
lat(comm) = 1: 1 iter/cycle

PMT



1 iter/cycle

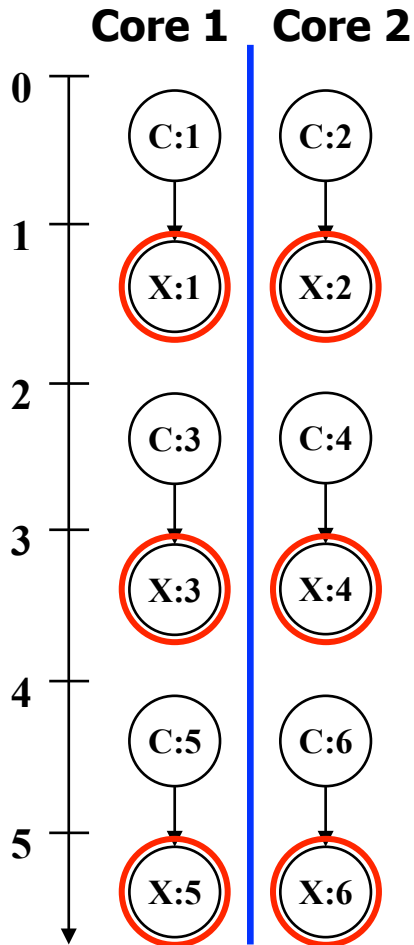
CMT



1 iter/cycle

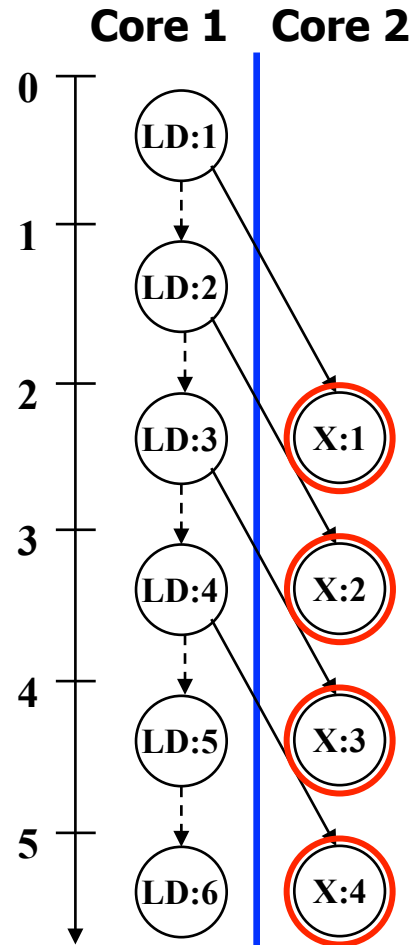
Comparison: IMT, PMT, CMT

IMT



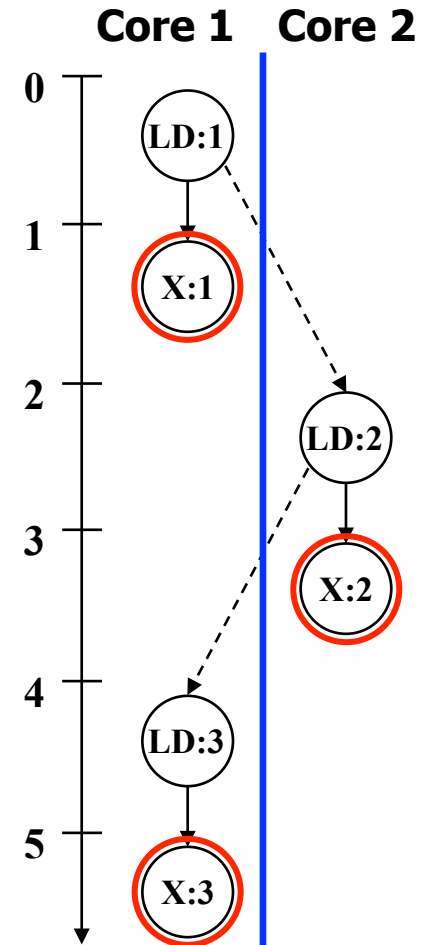
lat(comm) = 1: 1 iter/cycle
 lat(comm) = 2: 1 iter/cycle

PMT



1 iter/cycle
 1 iter/cycle

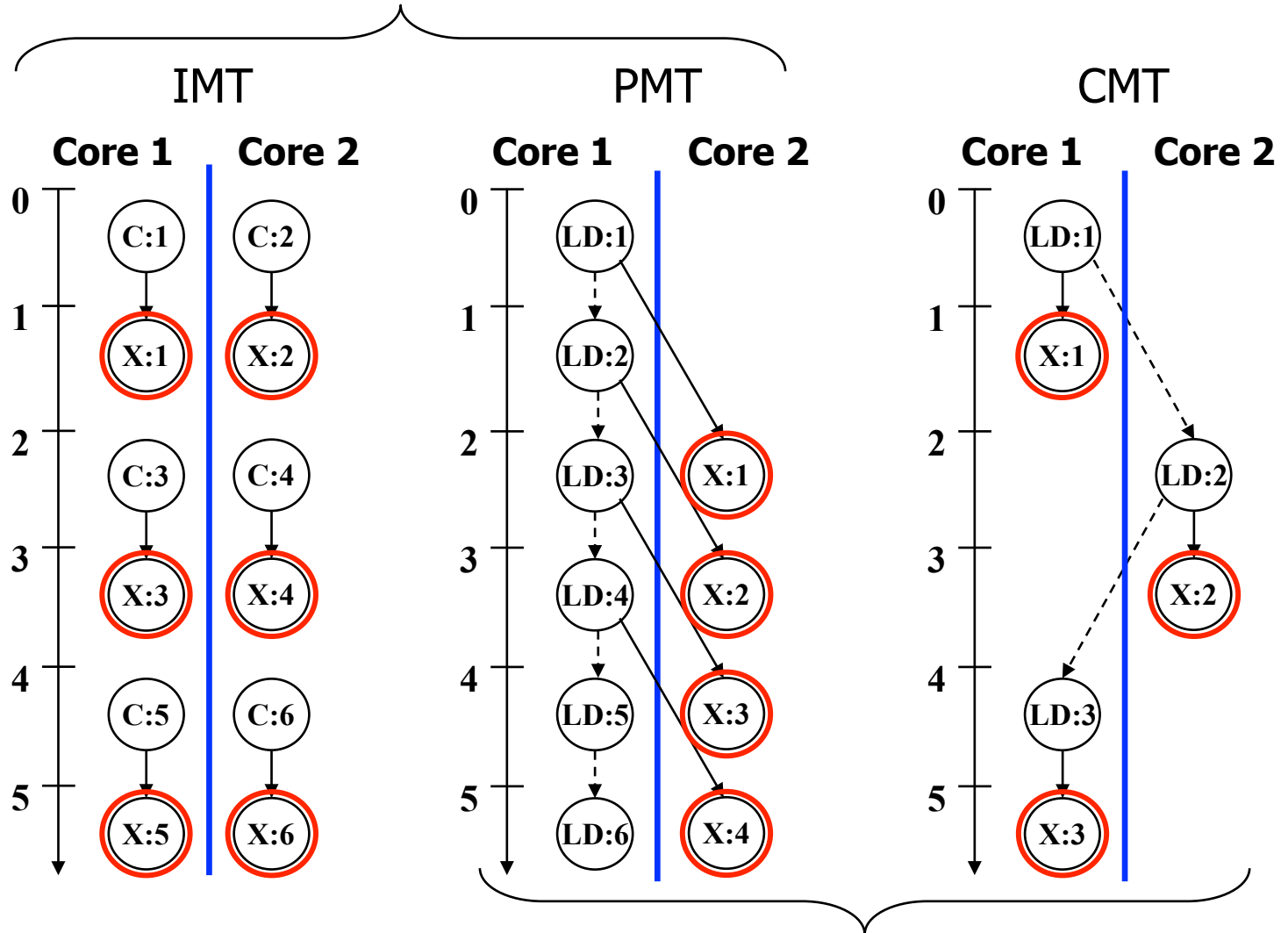
CMT



1 iter/cycle
 0.5 iter/cycle

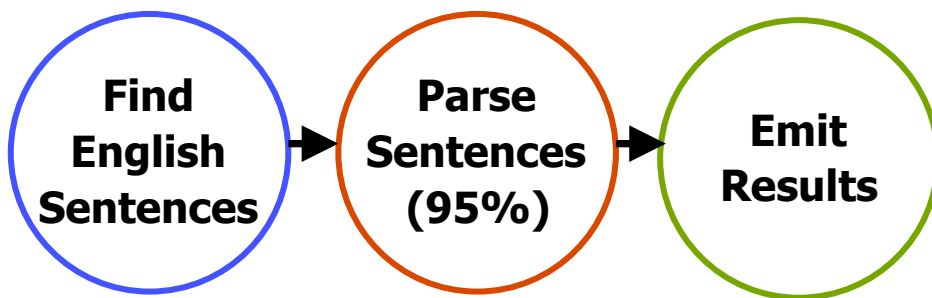
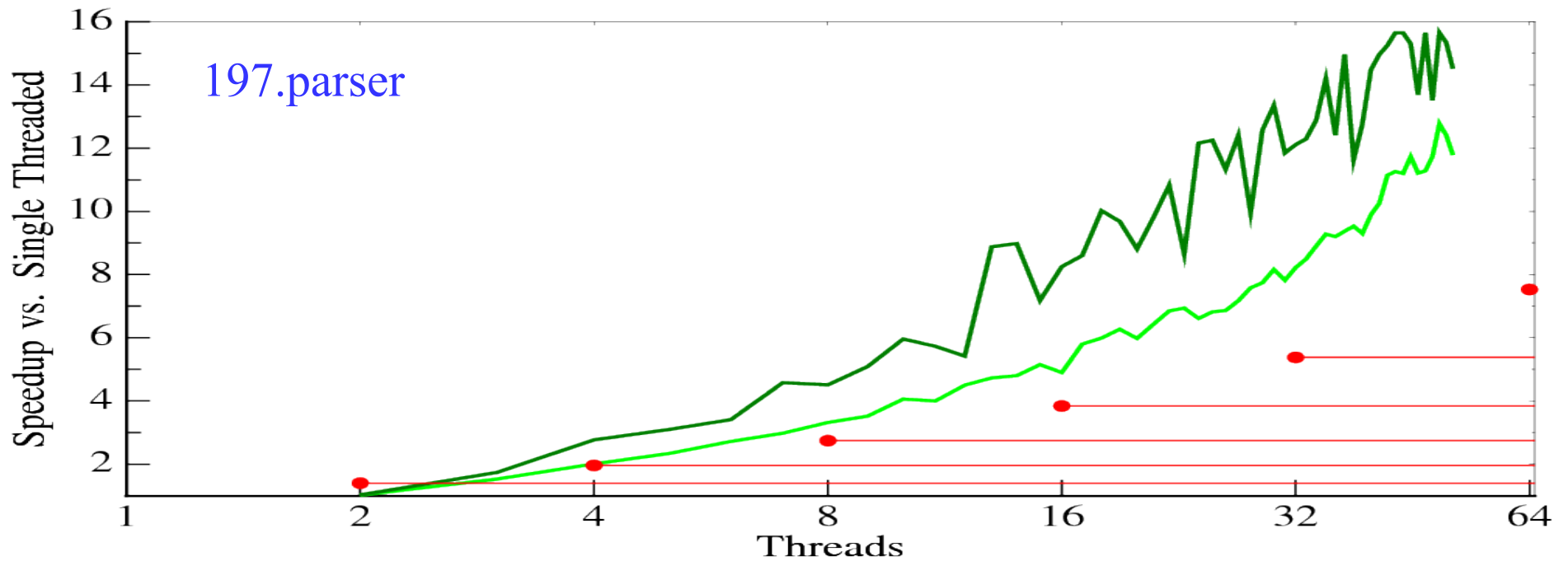
Comparison: IMT, PMT, CMT

Thread-local Recurrences → Fast Execution

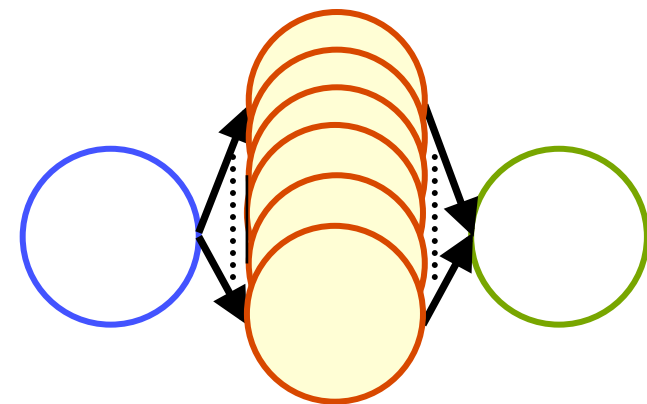


Cross-thread Dependences → Wide Applicability

Our Objective: Automatic Extraction of Pipeline Parallelism using DSWP



Decoupled Software Pipelining



PS-DSWP (Spec DOALL Middle Stage)

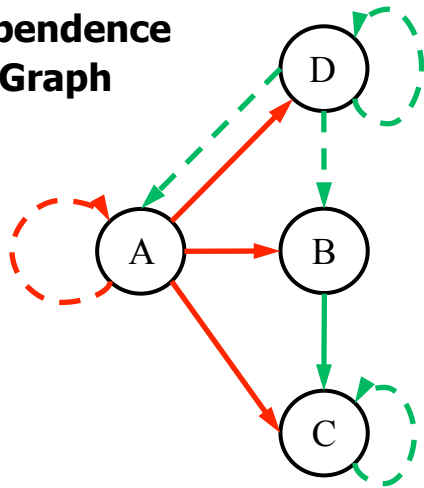
Decoupled Software Pipelining

Decoupled Software Pipelining (DSWP)

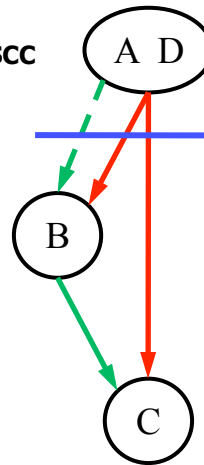
```

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

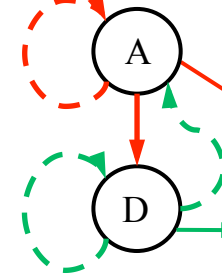
Dependence Graph



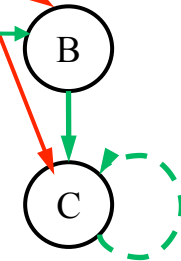
DAG_{scc}



Thread 1



Thread 2



register

control

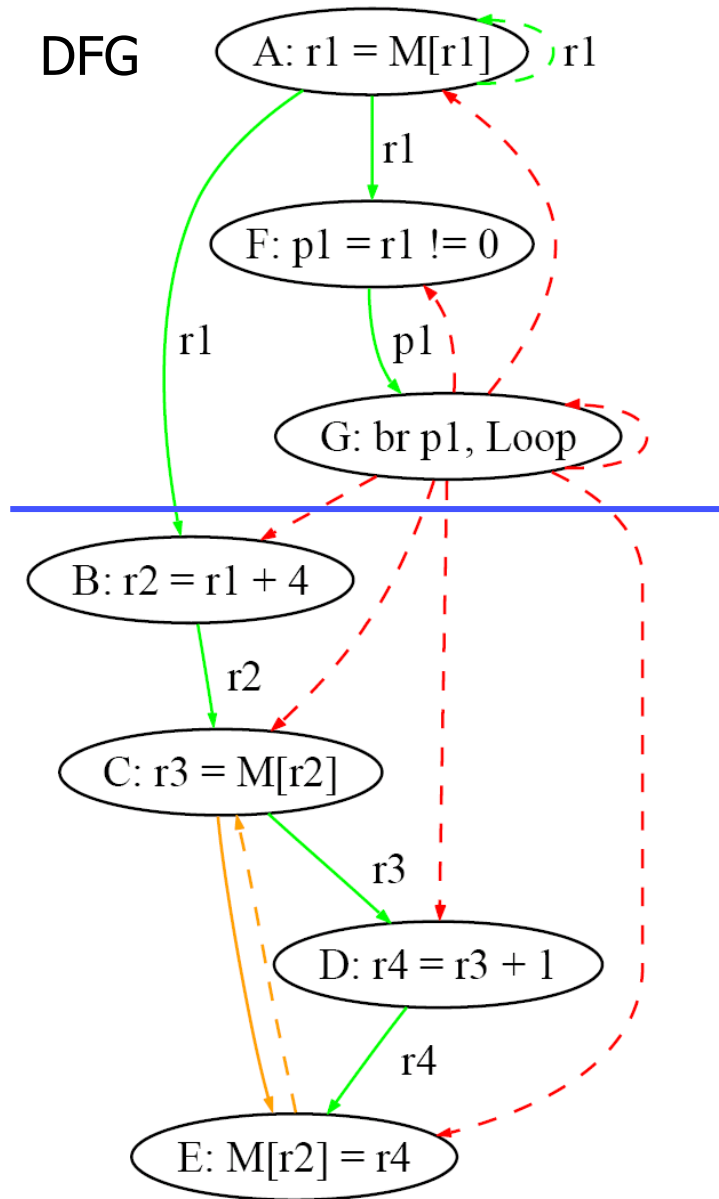
→ intra-iteration

- - → loop-carried

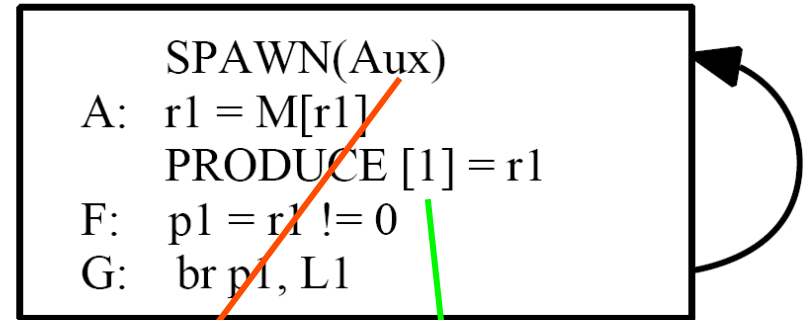
■ communication queue

Inter-thread communication latency is a one-time cost

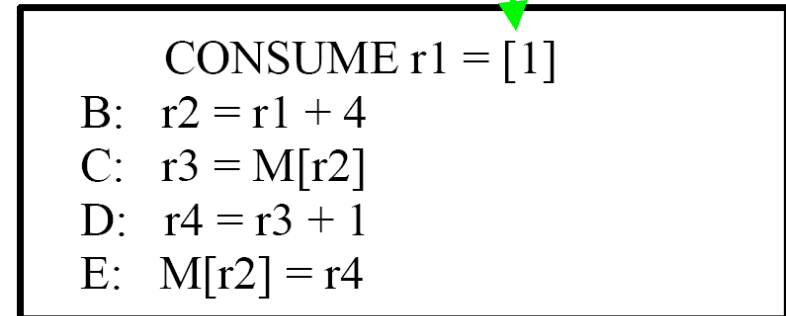
Implementing DSWP



L1:



Aux:



register

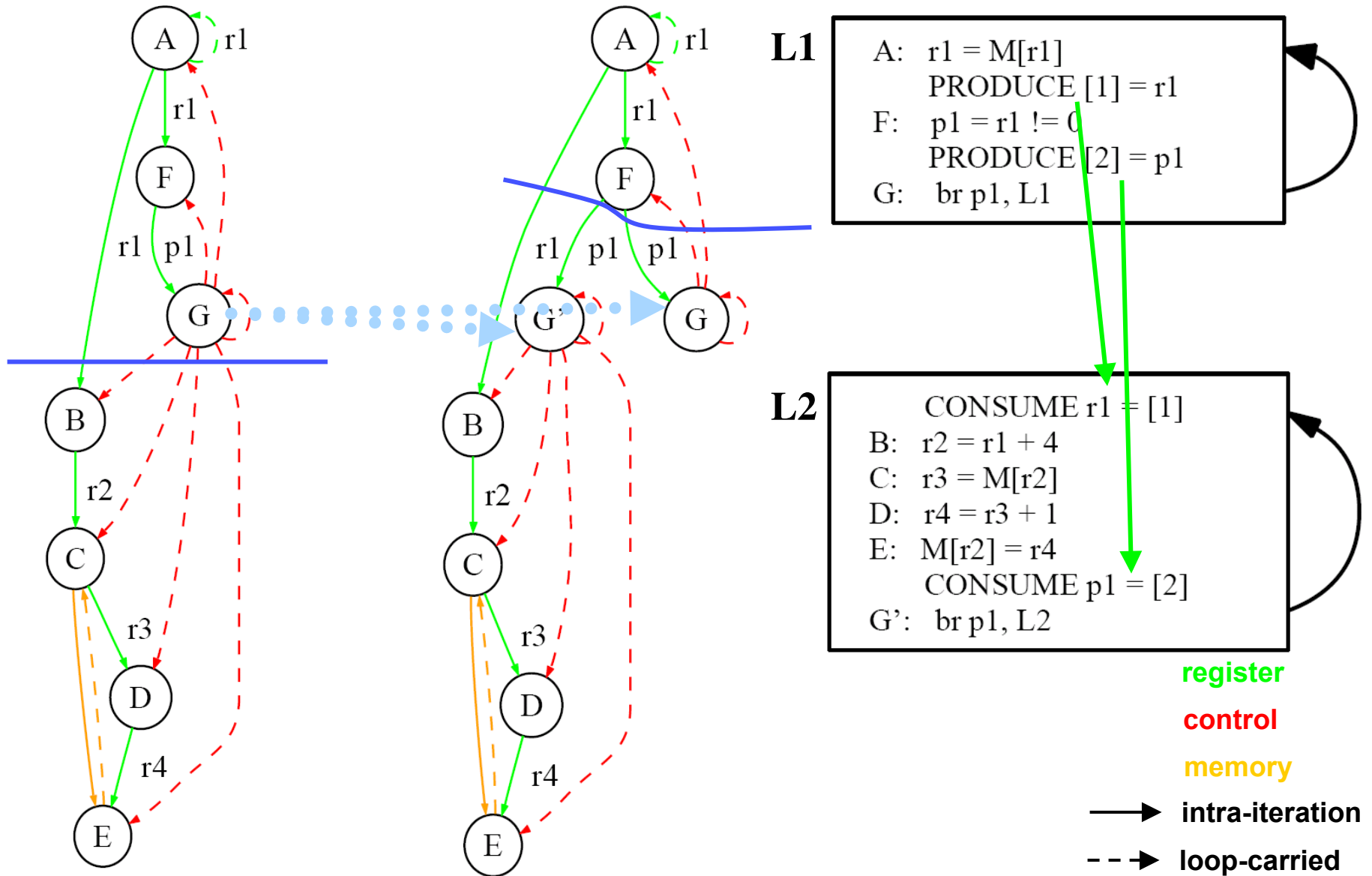
control

memory

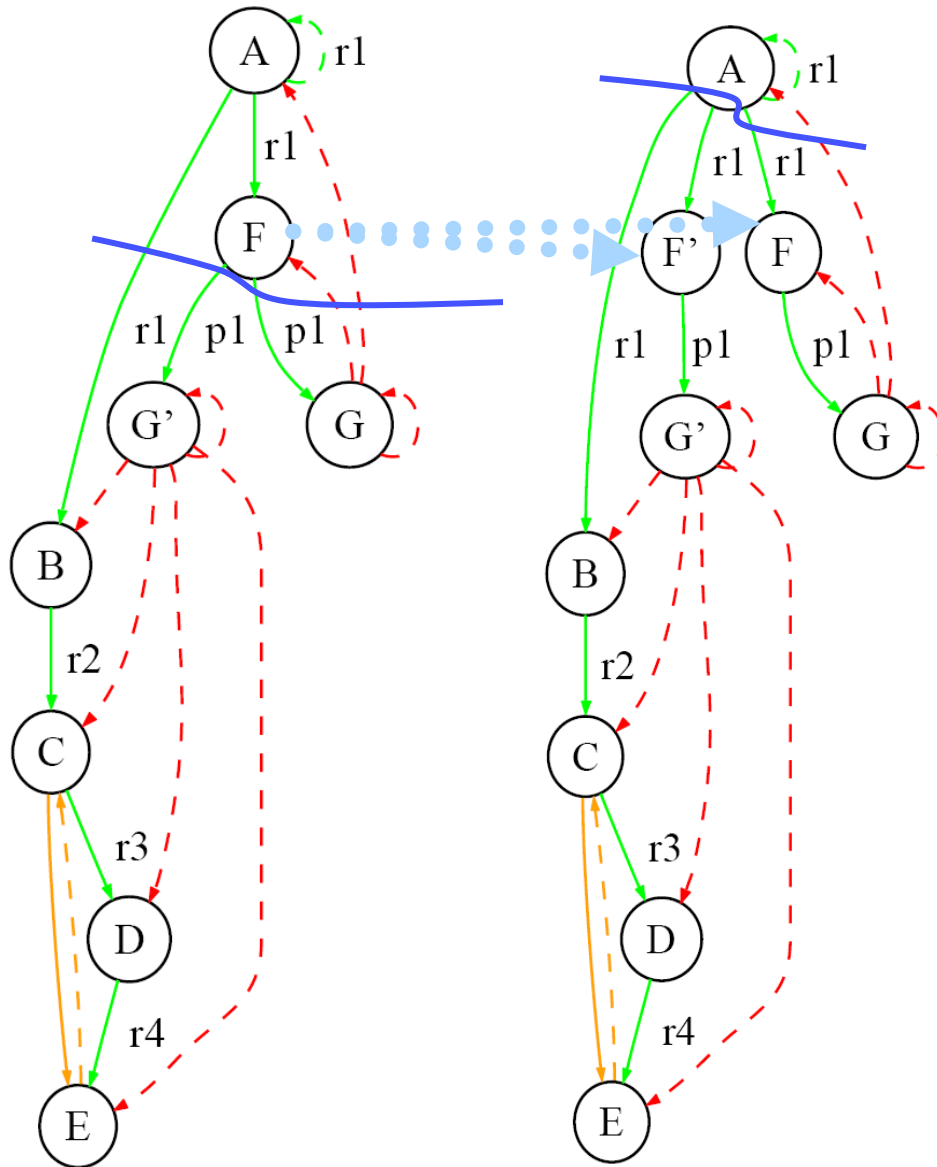
→ intra-iteration

- - -> loop-carried

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
    
```

register

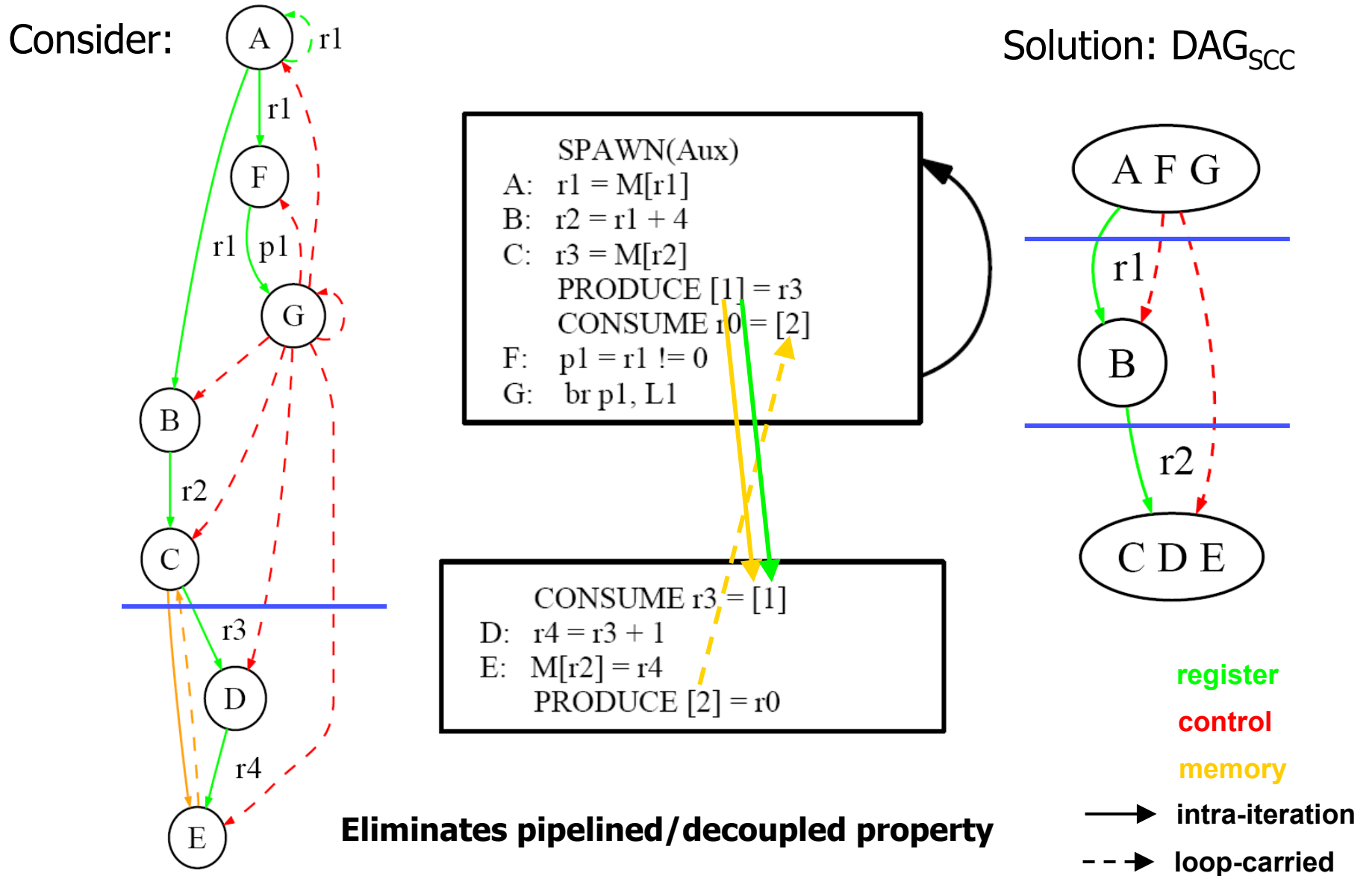
control

memory

→ intra-iteration

- - - → loop-carried

Constraint: Strongly Connected Components



2 Extensions to the Basic Transformation

- ❖ Speculation

- » Break statistically unlikely dependences
- » Form better-balanced pipelines

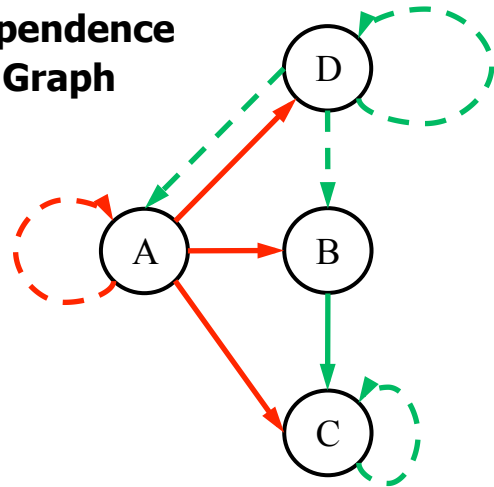
- ❖ Parallel Stages

- » 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



register

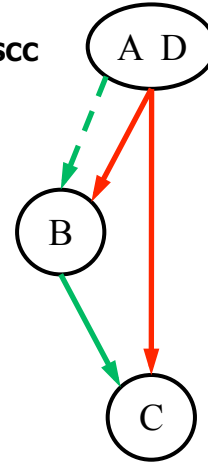
control

→ intra-iteration

- - → loop-carried

■ communication queue

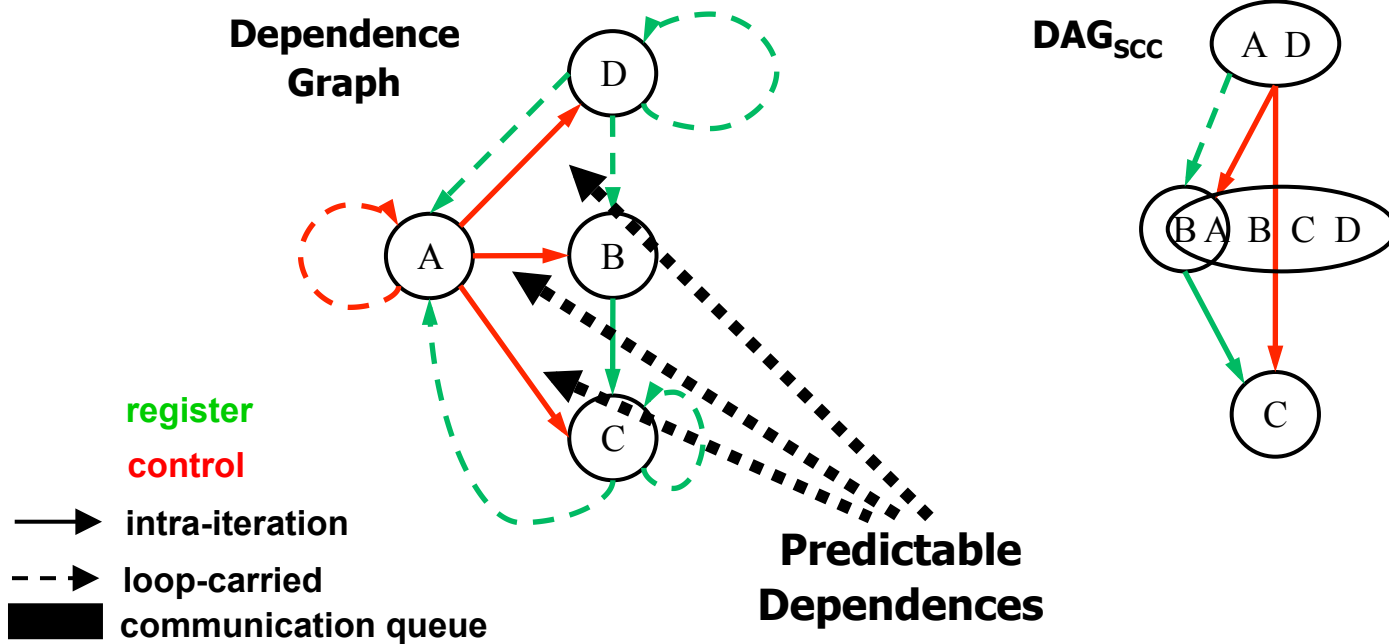
DAG_{scc}



Why Speculation?

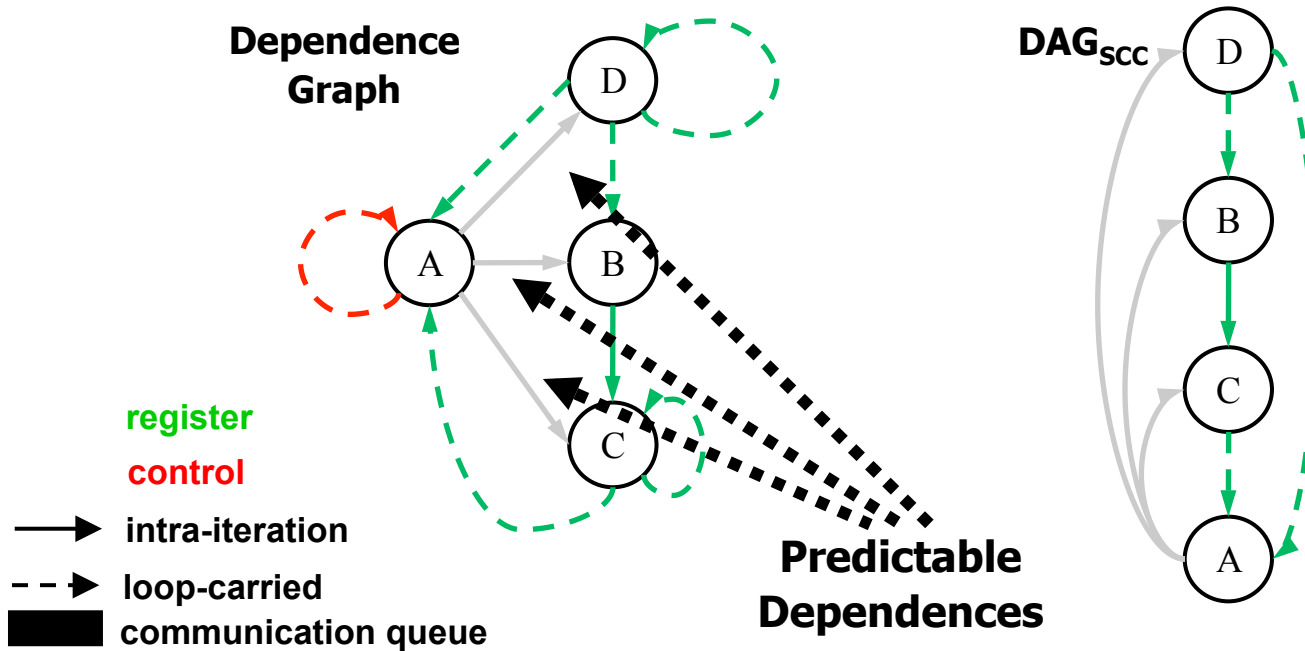
```

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

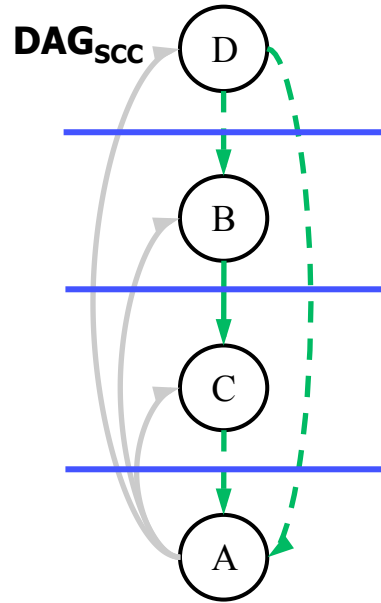


Why Speculation?

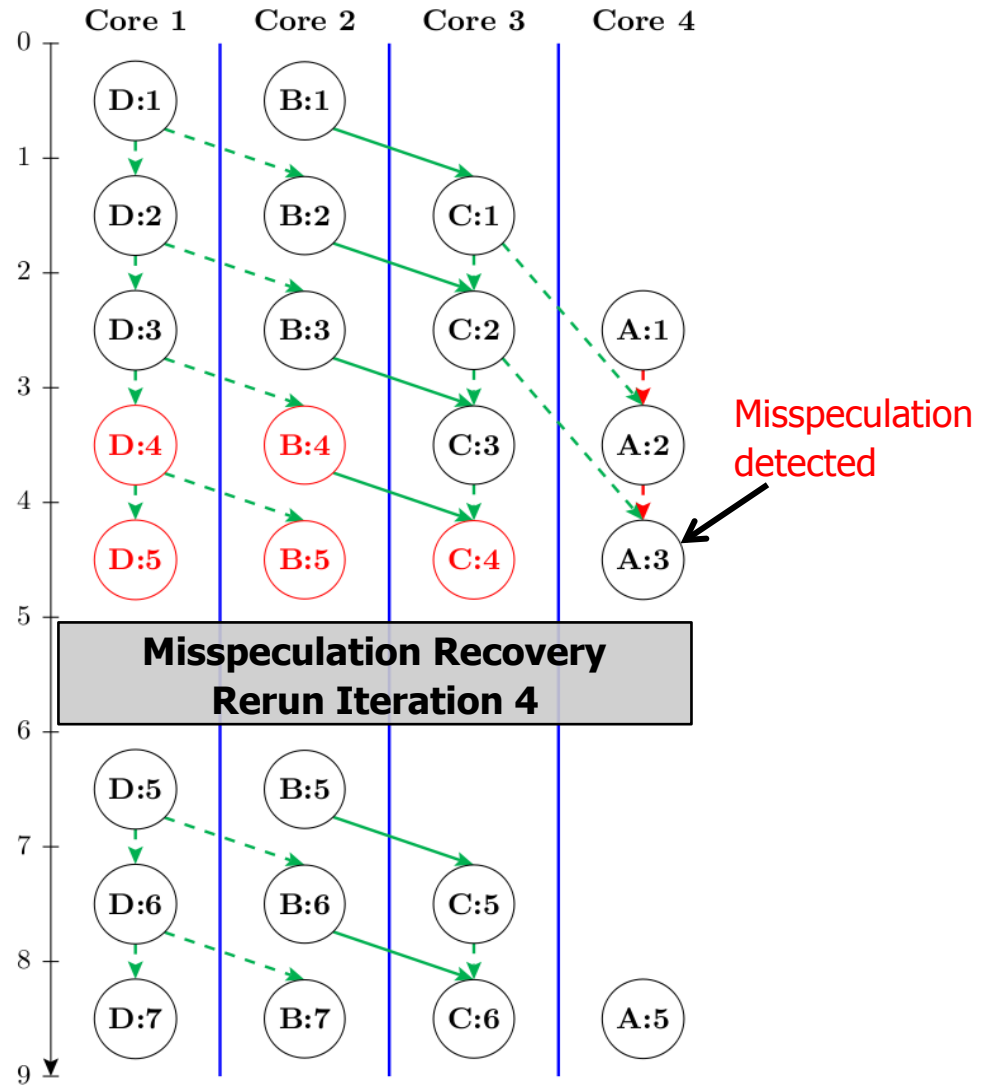
```
A: while(cost < T && node)
B:   ncost = doit(node);
C:   cost += ncost;
D:   node = node->next;
```



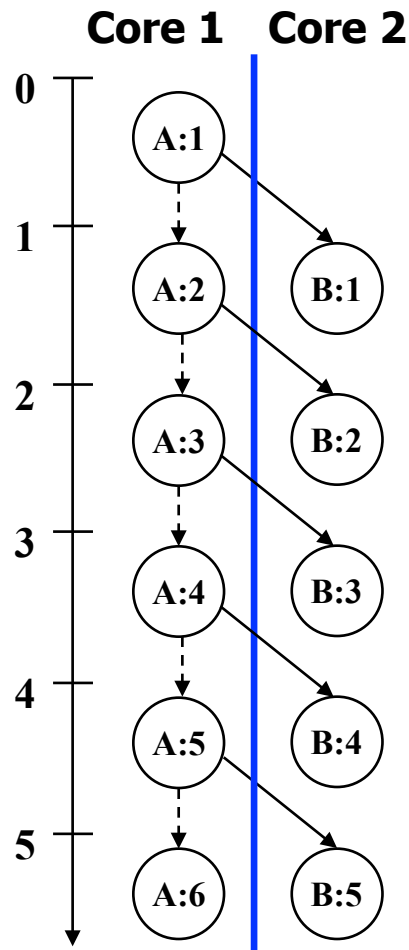
Execution Paradigm



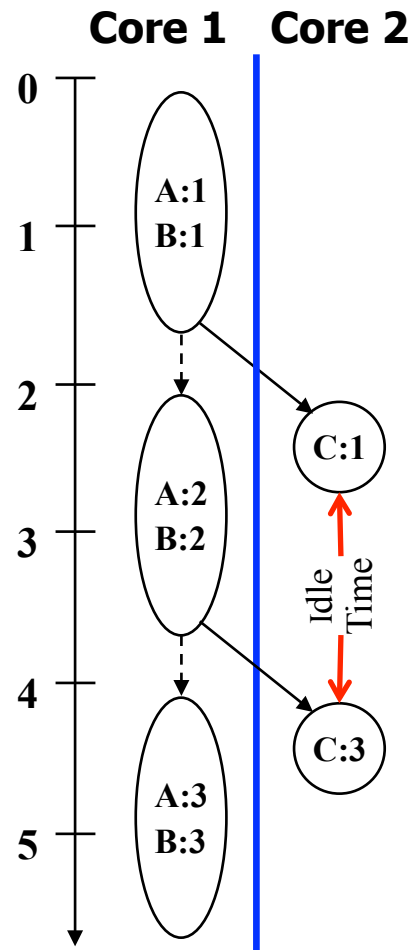
- register
- control
- intra-iteration
- - → loop-carried
- █ communication queue



Understanding PMT Performance



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



2 cycle/iter
0.5 iter/cycle

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

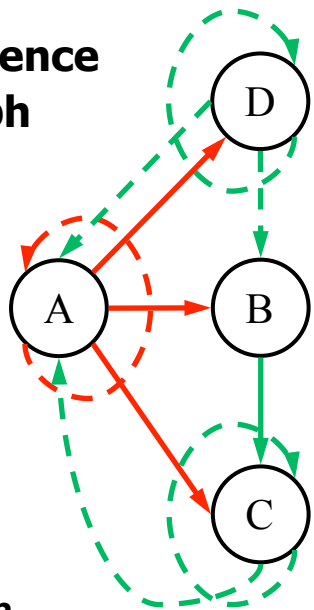
1. Rate t_i is at least as large as the longest dependence recurrence.
2. NP-hard to find longest recurrence.
3. Large loops make problem difficult in practice.

Selecting Dependences To Speculate

```

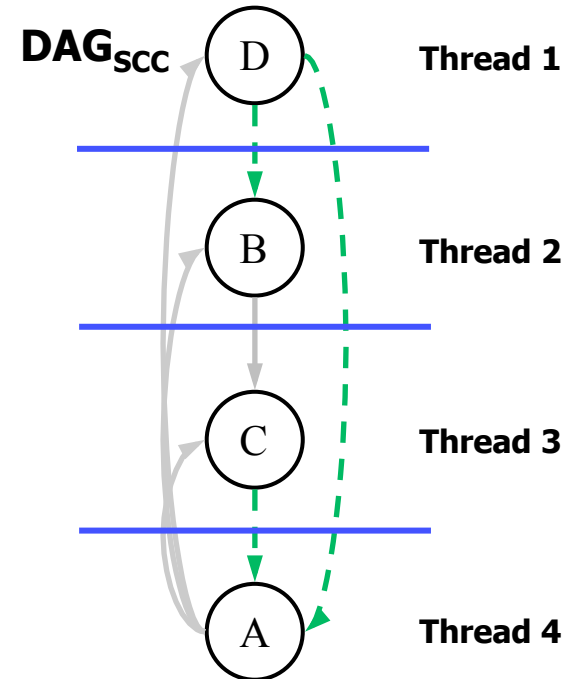
A: while(cost < T && node)
B:   ncost = doit(node);
C:   cost += ncost;
D:   node = node->next;
    
```

Dependence Graph

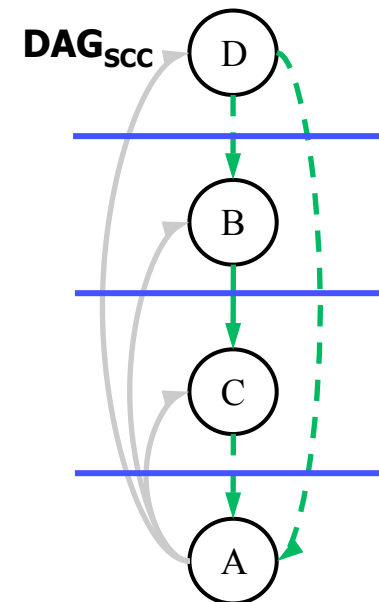
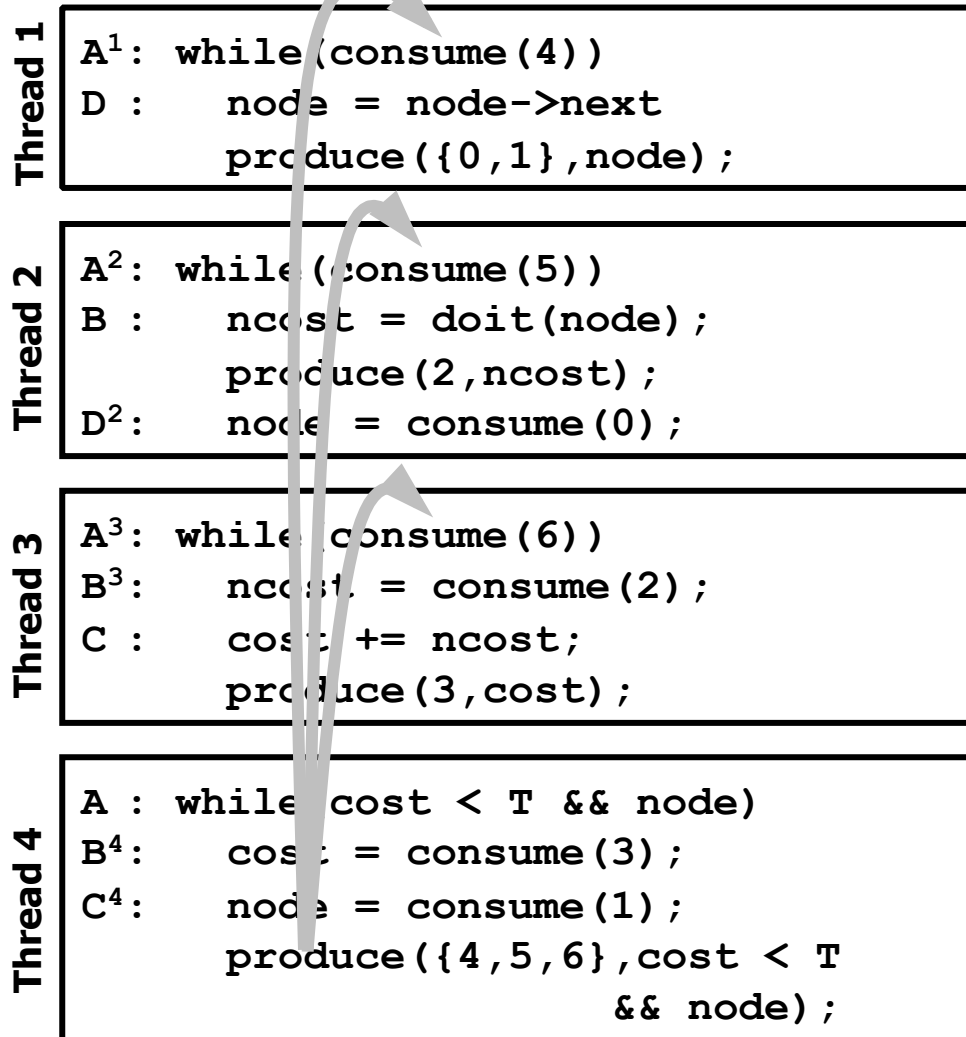


register
control

- intra-iteration
- - - → loop-carried
- communication queue

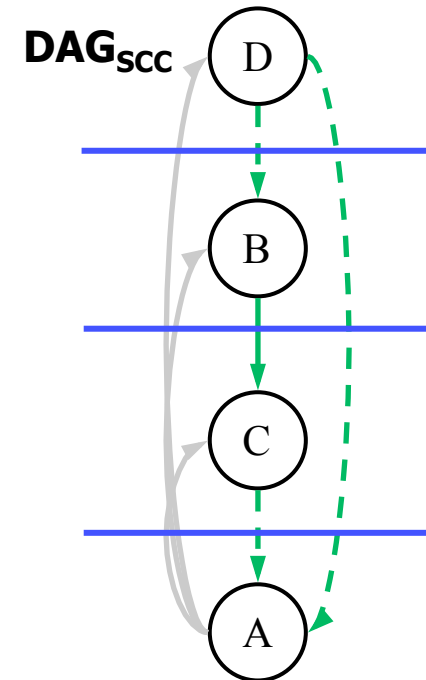


Detecting Misspeculation



Detecting Misspeculation

Thread 1	<pre> A¹: while (TRUE) D : node = node->next produce ({0, 1}, node); </pre>
Thread 2	<pre> A²: while (TRUE) B : ncost = doit(node); produce (2, ncost); D²: node = consume (0); </pre>
Thread 3	<pre> A³: while (TRUE) B³: ncost = consume (2); C : cost += ncost; produce (3, cost); </pre>
Thread 4	<pre> A : while (cost < T && node) B⁴: cost = consume (3); C⁴: node = consume (1); produce ({4, 5, 6}, cost < T && node); </pre>



Detecting Misspeculation

Thread 1

```
A1: while (TRUE)
D :   node = node->next
      produce ({0,1}, node);
```

Thread 2

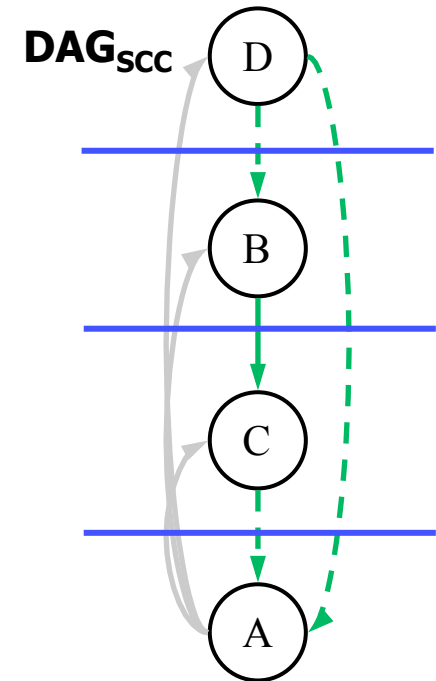
```
A2: while (TRUE)
B :   ncost = doit(node);
      produce (2, ncost);
D2:   node = consume (0);
```

Thread 3

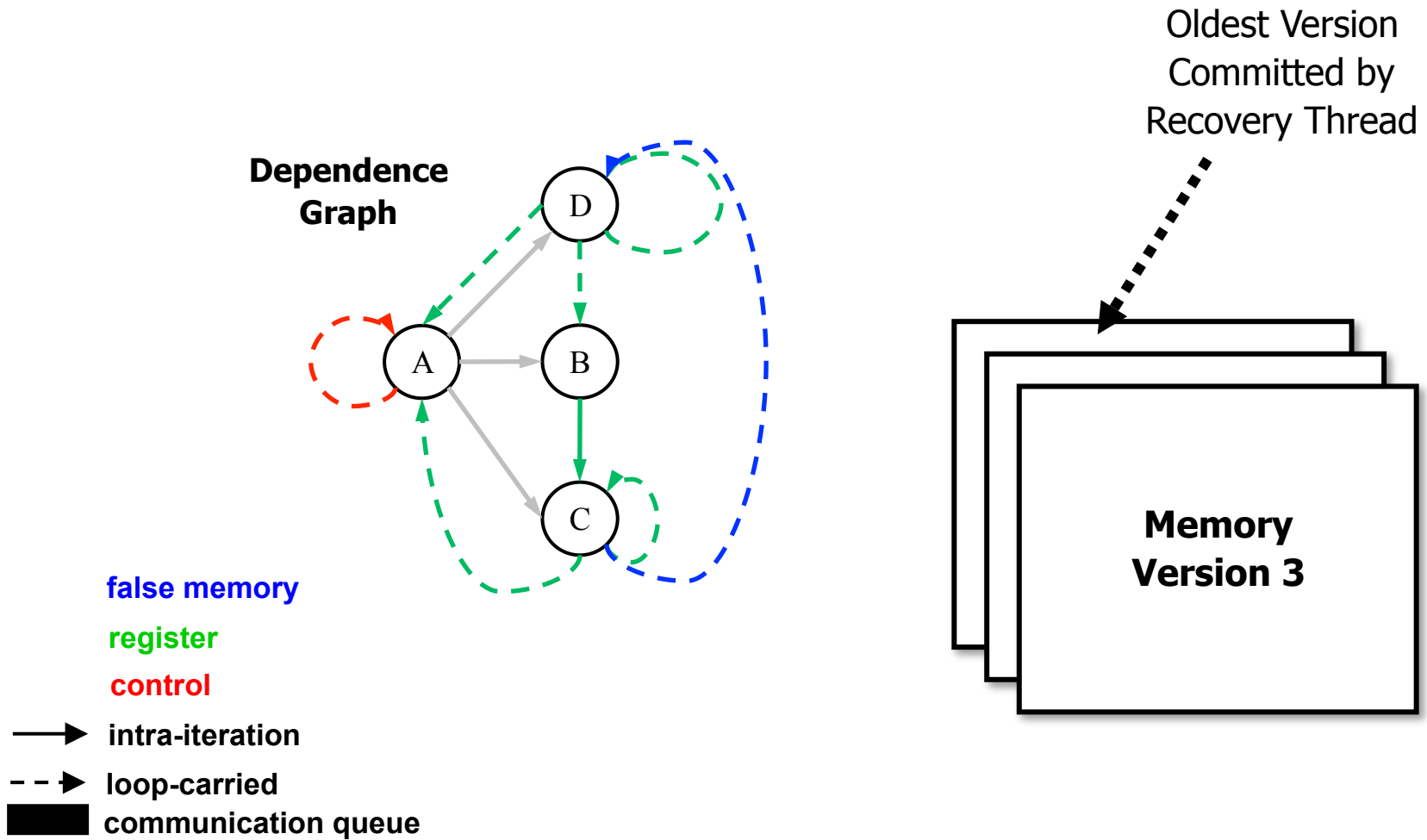
```
A3: while (TRUE)
B3:   ncost = consume (2);
C :   cost += ncost;
      produce (3, cost);
```

Thread 4

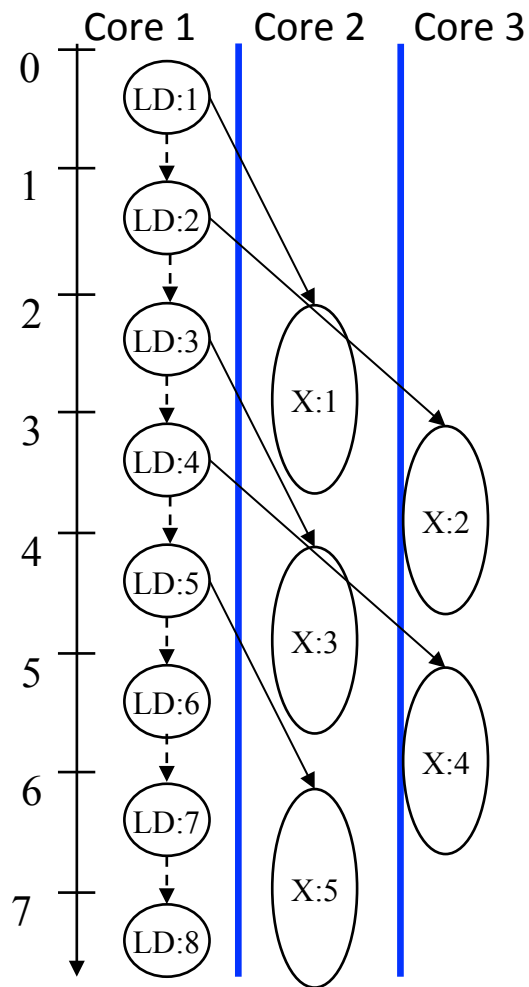
```
A : while (cost < T && node)
B4:   cost = consume (3);
C4:   node = consume (1);
      if (!(cost < T && node))
          FLAG_MISSPEC ();
```



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

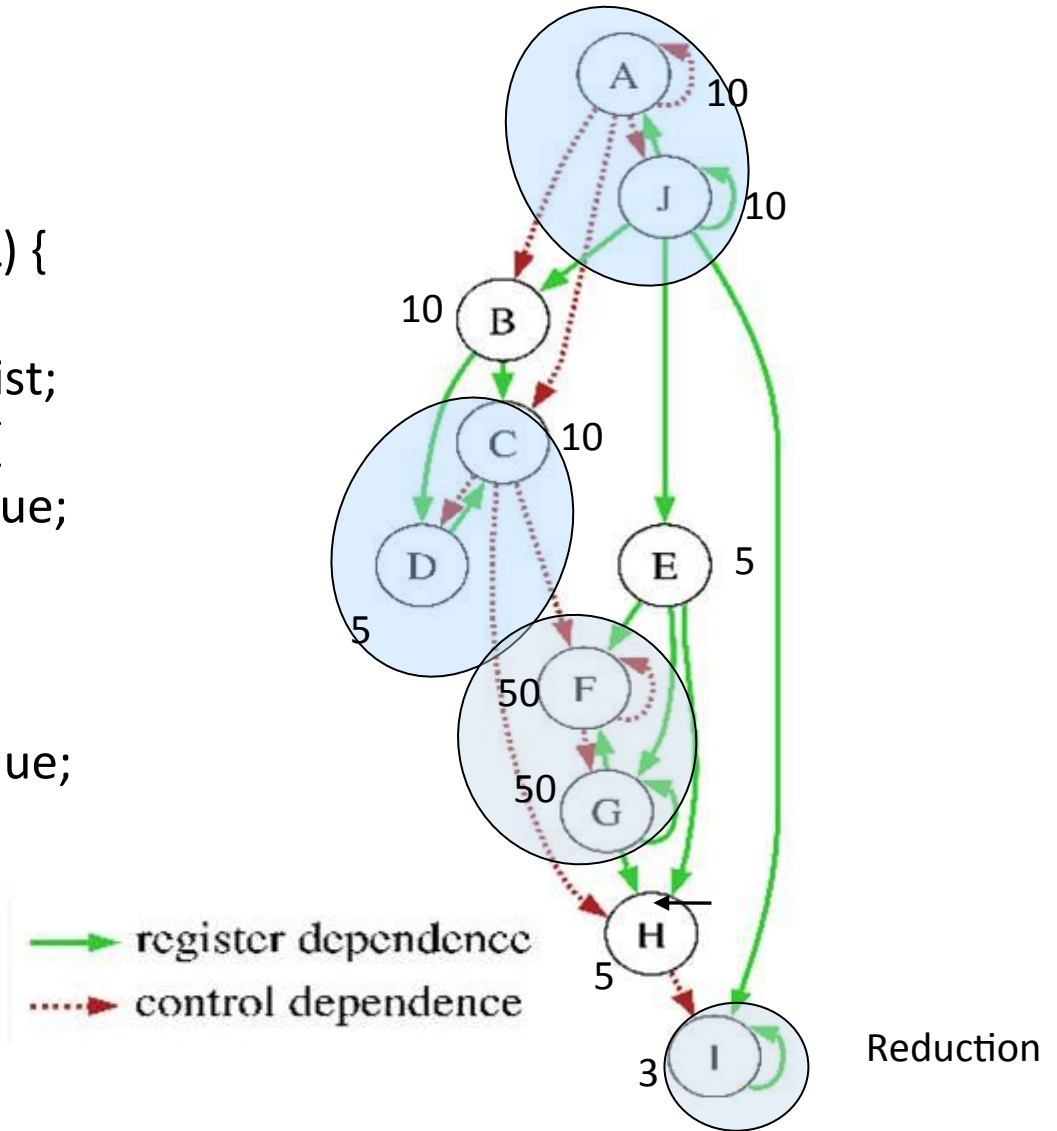
PS-DSWP: 1 iteration/cycle

Thread Partitioning

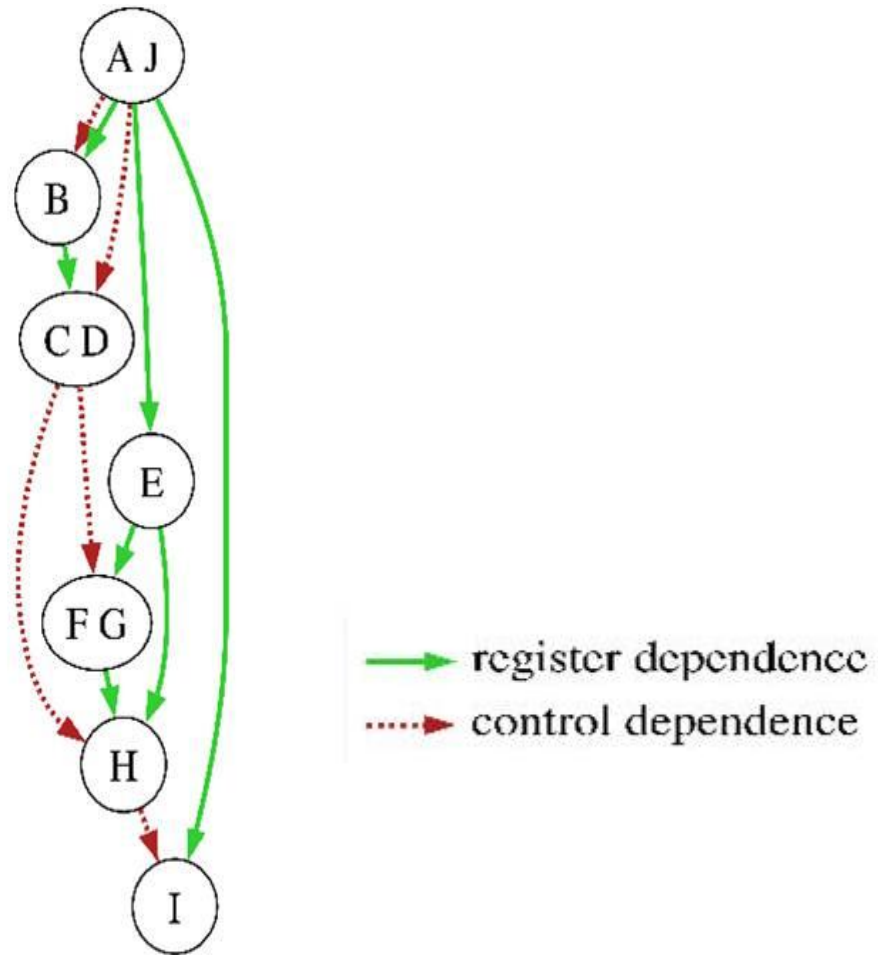
```

    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;
}

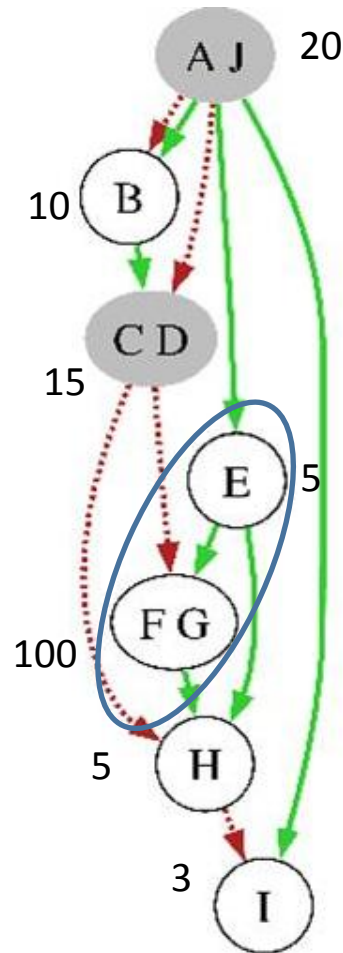
```



Thread Partitioning: DAG_{SCC}



Thread Partitioning



Merging Invariants

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

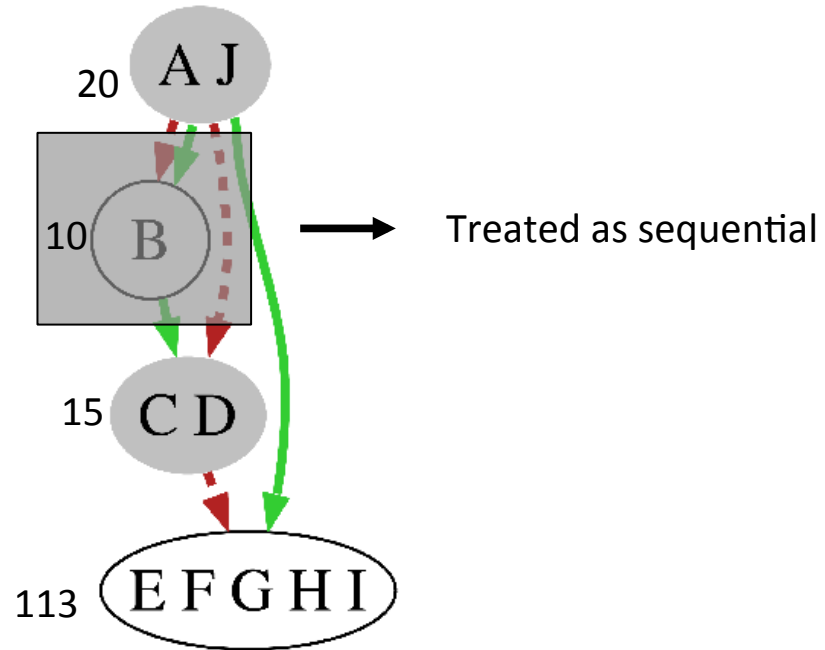
→ register dependence

⋯ control dependence

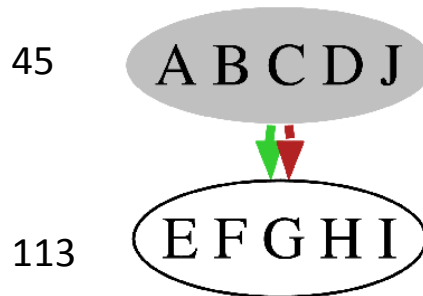
○ doall

● sequential

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?
 - » 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

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?

CFGs, PCs, and Cross-Iteration Deps

1. $r1 = 10$

1. $r1 = r1 + 1$

2. $r2 = \text{MEM}[r1]$

3. $r2 = r2 + 1$

4. $\text{MEM}[r1] = r2$

5. Branch $r1 < 1000$

No register live outs

Loop-Level Parallelization: DOALL

1. $r1 = 10$

1. $r1 = r1 + 1$

2. $r2 = \text{MEM}[r1]$

3. $r2 = r2 + 1$

4. $\text{MEM}[r1] = r2$

5. Branch $r1 < 1000$

1. $r1 = 9$

1. $r1 = r1 + 2$

2. $r2 = \text{MEM}[r1]$

3. $r2 = r2 + 1$

4. $\text{MEM}[r1] = r2$

5. Branch $r1 < 999$

1. $r1 = 10$

1. $r1 = r1 + 2$

2. $r2 = \text{MEM}[r1]$

3. $r2 = r2 + 1$

4. $\text{MEM}[r1] = r2$

5. Branch $r1 < 1000$

No register live outs

Another Example

1. $r1 = 10$

1. $r1 = r1 + 1$

2. $r2 = \text{MEM}[r1]$

3. $r2 = r2 + 1$

4. $\text{MEM}[r1] = r2$

5. Branch $r2 == 10$

No register live outs

Another Example

1. $r1 = 10$

1. $r1 = r1 + 1$

2. $r2 = \text{MEM}[r1]$

3. $r2 = r2 + 1$

4. $\text{MEM}[r1] = r2$

5. Branch $r2 == 10$

1. $r1 = 9$

1. $r1 = r1 + 2$

2. $r2 = \text{MEM}[r1]$

3. $r2 = r2 + 1$

4. $\text{MEM}[r1] = r2$

5. Branch $r2 == 10$

1. $r1 = 10$

1. $r1 = r1 + 2$

2. $r2 = \text{MEM}[r1]$

3. $r2 = r2 + 1$

4. $\text{MEM}[r1] = r2$

5. Branch $r2 == 10$

No register live outs

Speculation

1. $r1 = 9$

1. $r1 = 10$

1. $r1 = r1 + 2$

1. $r1 = r1 + 2$

2. $r2 = \text{MEM}[r1]$

2. $r2 = \text{MEM}[r1]$

3. $r2 = r2 + 1$

3. $r2 = r2 + 1$

4. $\text{MEM}[r1] = r2$

4. $\text{MEM}[r1] = r2$

5. Branch $r2 == 10$

5. Branch $r2 == 10$

No register live outs

Speculation, Commit, and Recovery

1. $r1 = 9$

1. $r1 = r1 + 2$
2. $r2 = \text{MEM}[r1]$
3. $r2 = r2 + 1$
4. $\text{Send}\{1\} r2$
5. Jump

1. $r2 = \text{Receive}\{1\}$
2. Branch $r2 \neq 10$
3. $\text{MEM}[r1] = r2$
4. $r2 = \text{Receive}\{2\}$
5. Branch $r2 \neq 10$
6. $\text{MEM}[r1] = r2$
7. Jump

1. $r1 = 10$

1. $r1 = r1 + 2$
2. $r2 = \text{MEM}[r1]$
3. $r2 = r2 + 1$
4. $\text{MEM}[r1] = r2$
5. Jump

No register live outs

1. Kill and Continue

Difficult Dependences

1. r1 = Head

1. r1 = MEM[r1]

2. Branch r1 == 0

3. r2 = MEM[r1 + 4]

4. r3 = Work (r2)

5. Print (r3)

6. Jump

No register live outs

DOACROSS

1. r1 = Head

1. r1 = MEM[r1]

2. Branch r1 == 0

3. r2 = MEM[r1 + 4]

4. r3 = Work (r2)

5. Print (r3)

6. Jump

No register live outs

PS-DSWP

1. r1 = Head

1. r1 = MEM[r1]

2. Branch r1 == 0

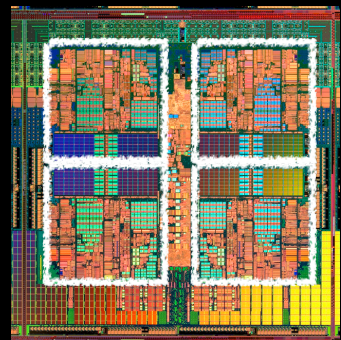
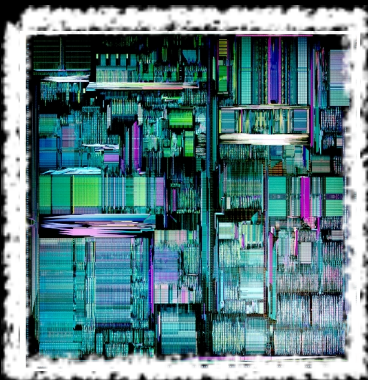
3. r2 = MEM[r1 + 4]

4. r3 = Work (r2)

5. Print (r3)

6. Jump

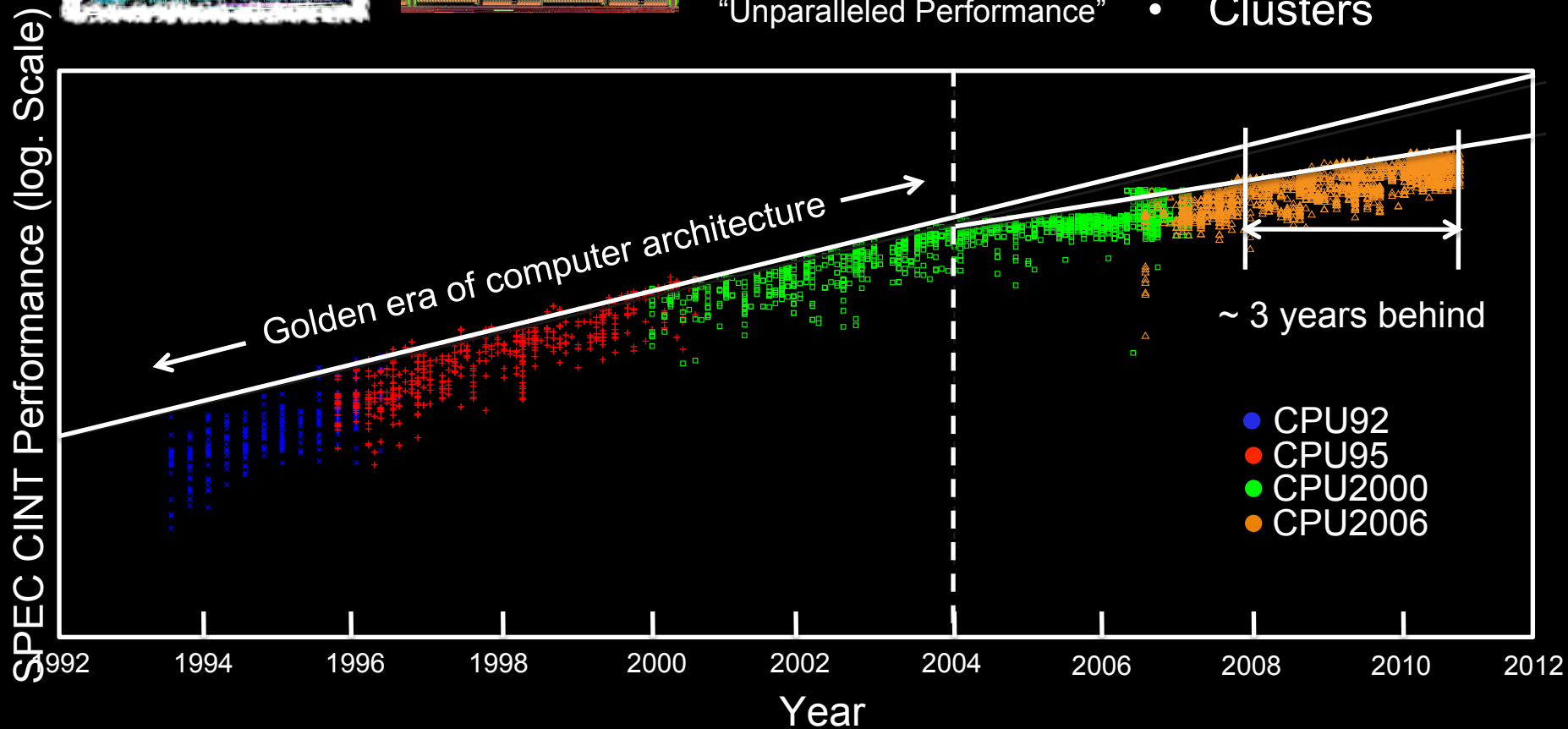
No register live outs



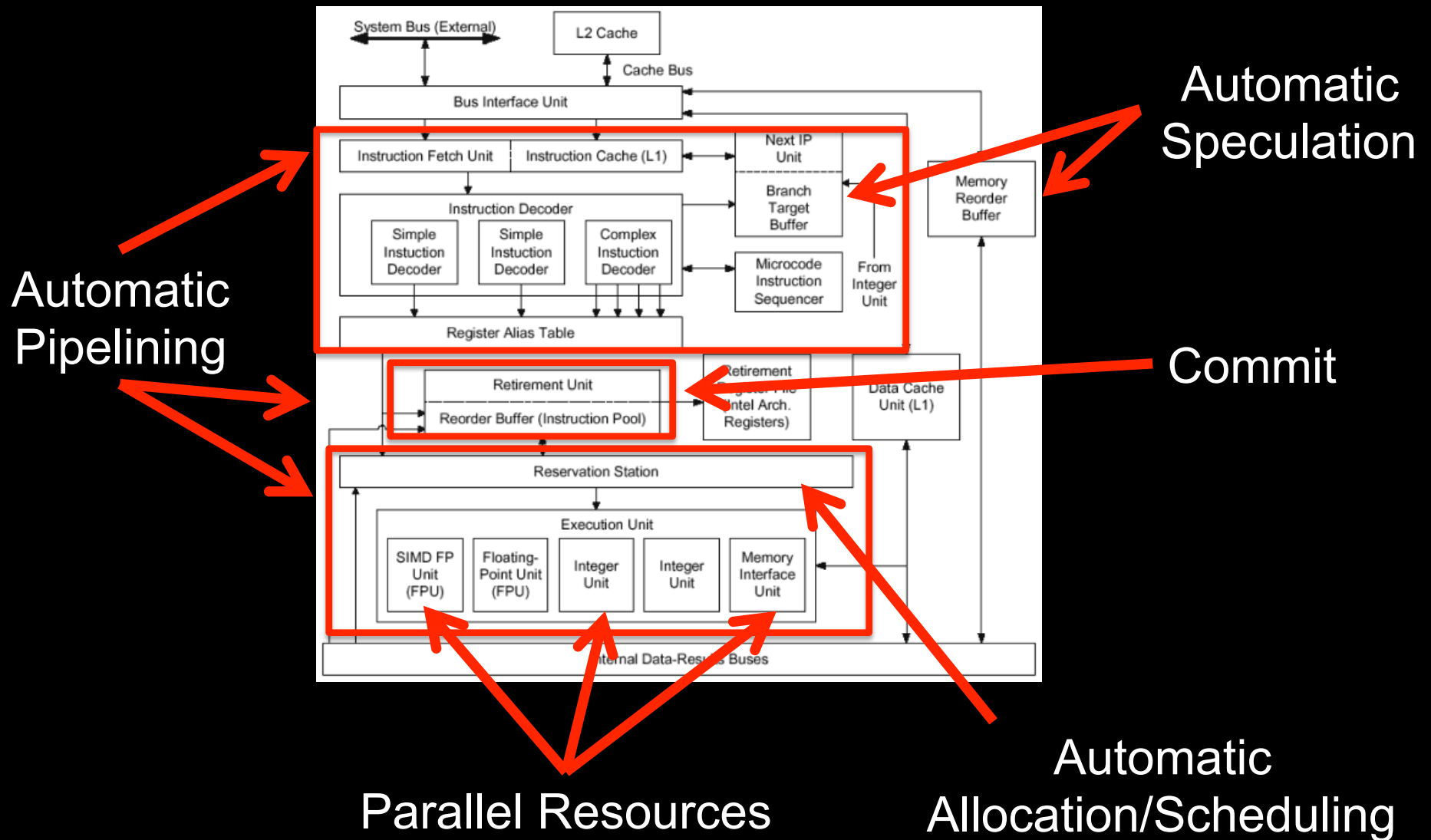
10-Core Intel Xeon
"Unparalleled Performance"

Era of DIY:

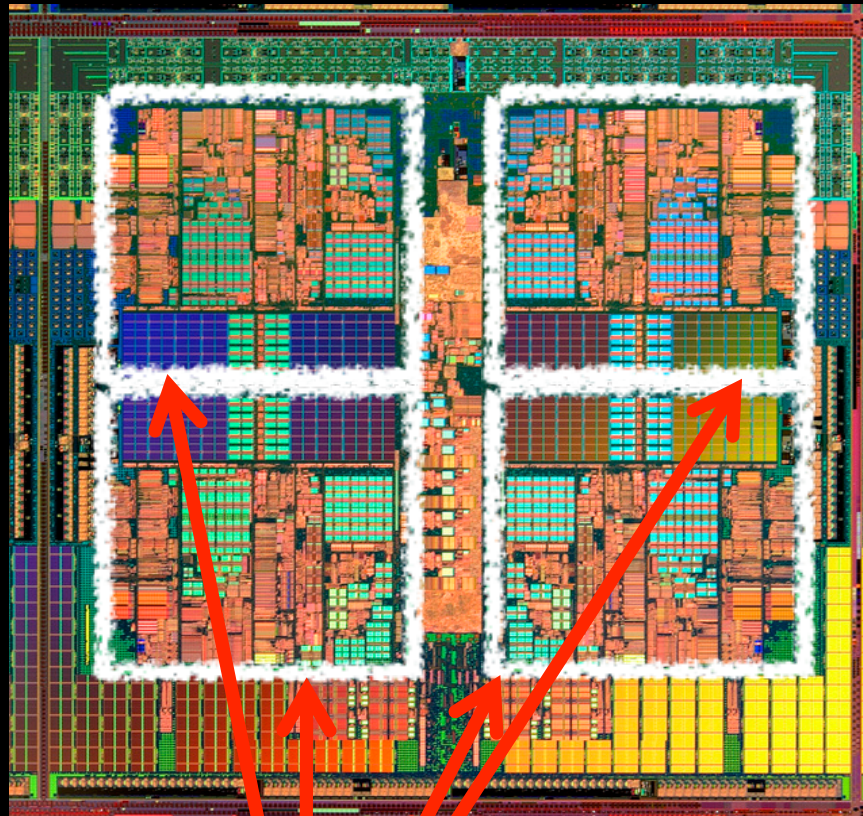
- Multicore
- Reconfigurable
- GPUs
- Clusters



P6 SUPERSCALAR ARCHITECTURE (CIRCA 1994)



MULTICORE ARCHITECTURE (CIRCA 2010)



~~Automatic
Pipelining~~

~~Automatic
Speculation~~

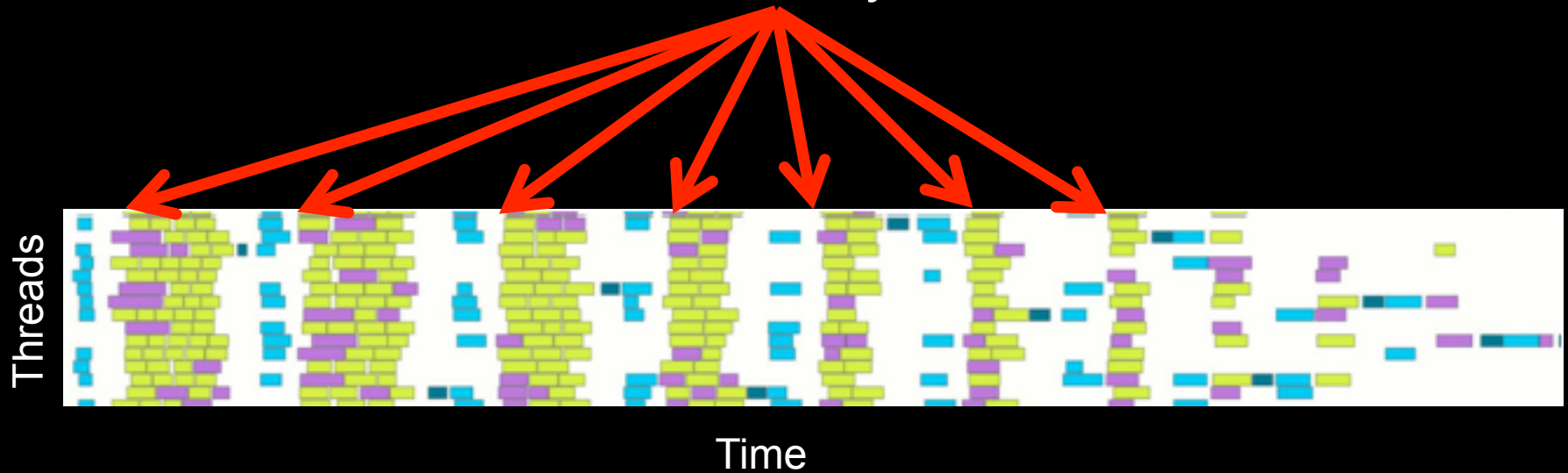
~~Commit~~

Parallel Resources

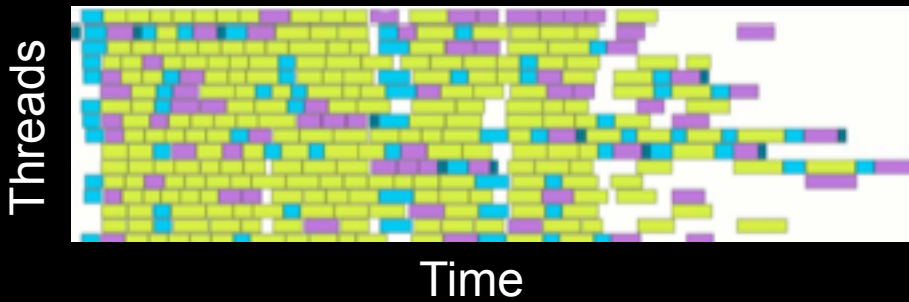
~~Automatic
Allocation/Scheduling~~

ABCPL	CORRELATE	GLU	Mentat	Parafrese2	
ACE	CPS	GUARD	Legion	Paralation	pC++
ACT++	CRL	HA _s L	Meta Chaos	Parallel-C++	SCHEDULE
Active messages	CSP	Haskell	Midway	Parallaxis	SciTL
Adl	Cthreads	HPC++	Millipede	ParC	POET
Adsmith	CUMULVS	JAVAR.	CparPar	ParLib++	SDDA.
ADDAP	DAGGER	HORUS	Mirage	ParLin	SHMEM
AFAPI	DAPPLE	HPC	MpC	Parmaes	SIMPLE
ALWAN	Data Parallel C	IMPACT	MOSIX	Parti	Sina
AM	DC++	ISIS.	Modula-P	pC	SISAL.
AMDC	DCE++	JAVAR	Modula-2*	pC++	distributed smalltalk
AppLeS	DDD	JADE	Multipol	PCN	SMI.
Amoeba	DICE.	Java RMI	MPI	PCP:	SONiC
ARTS	DIPC	javaPG	MPC++	PH	Split-C.
Athapascan-0b	DOLIB	JavaSpace	Munin	PEACE	SR
Aurora	DOME	JIDL	Nano-Threads	PCU	Sthreads
Automap	DOSMOS.	Joyce	NESL	PET	Strand.
bb_threads	DRL	Khoros	NetClasses++	PETSc	SUIF.
Blaze	DSM-Threads	Karma	Nexus	PENNY	Synergy
BSP	Ease .	KOAN/Fortran-S	Nimrod	Phosphorus	Telegraphos
BlockComm	ECO	LAM	NOW	POET.	SuperPascal
C*.	Eiffel	Lilac	Objective Linda	Polaris	TCGMSG.
"C* in C	Eilean	Linda	Occam	POOMA	Threads.h++.
C**	Emerald	JADA	Omega	POOL-T	TreadMarks
CarlOS	EPL	WWWinda	OpenMP	PRESTO	TRAPPER
Cashmere	Excalibur	ISETL-Linda	Orca	P-RIO	uC++
C4	Express	ParLin	OOF90	Prospero	UNITY
CC++	Falcon	Eilean	P++	Proteus	UC
Chu	Filaments	P4-Linda	P3L	QPC++	V
Charlotte	FM	Glenda	p4-Linda	PVM	ViC*
Charm	FLASH	POSYBL	Pablo	PSI	Visifold V-NUS
Charm++	The FORCE	Objective-Linda	PADE	PSDM	VPE
Cid	Fork	LiPS	PADRE	Quake	Win32 threads
Cilk	Fortran-M	Locust	Panda	Quark	WinPar
CM-Fortran	FX	Lparx	Papers	Quick Threads	WWWinda
Converse	GA	Lucid	AFAPI	Sage++	XENOOPS
Code	GAMMA	Maisie	Para++	SCANDAL	XPC
COOL	Glenda	Manifold	Paradigm	SAM	Zounds
					ZPL

Parallel Library Calls

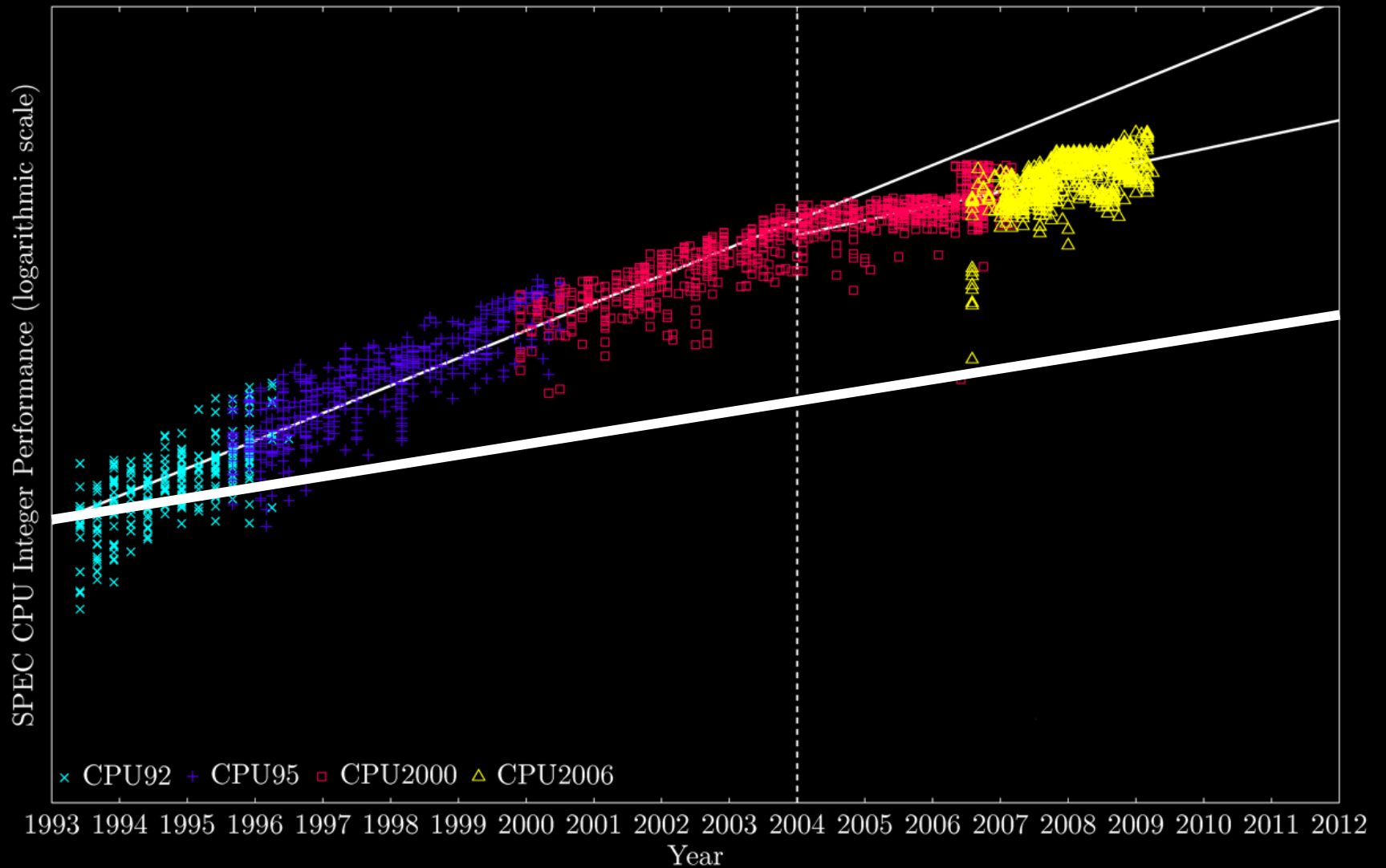


Realizable parallelism



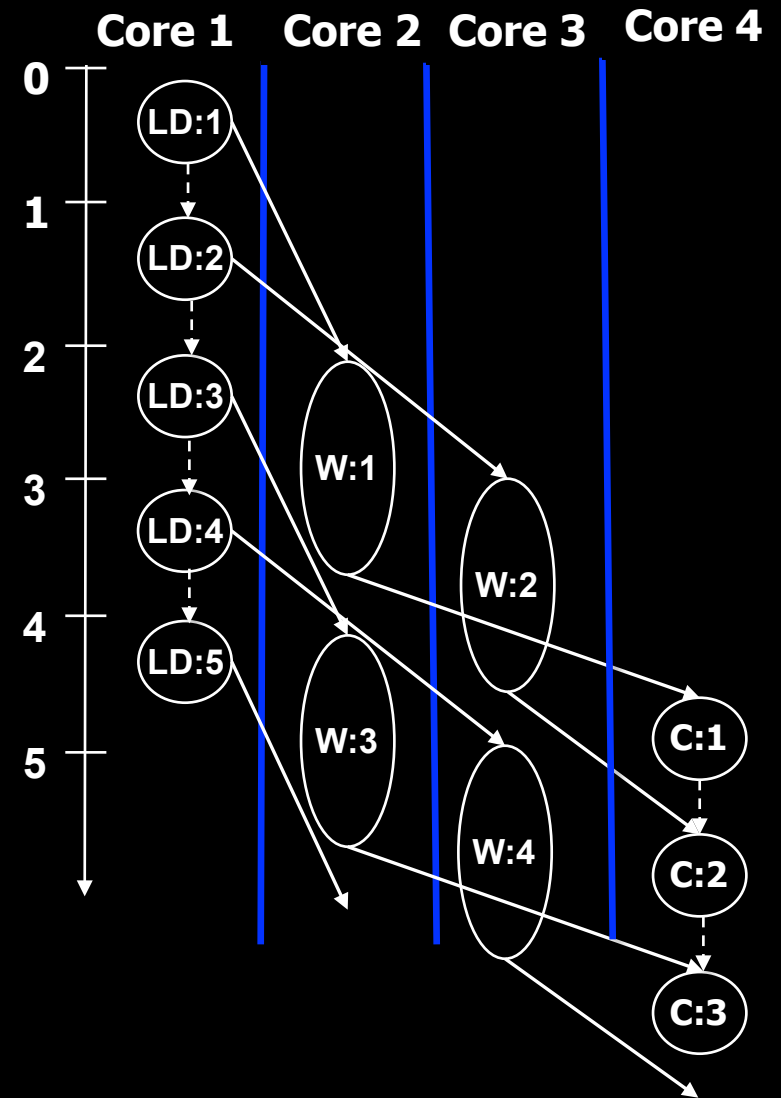
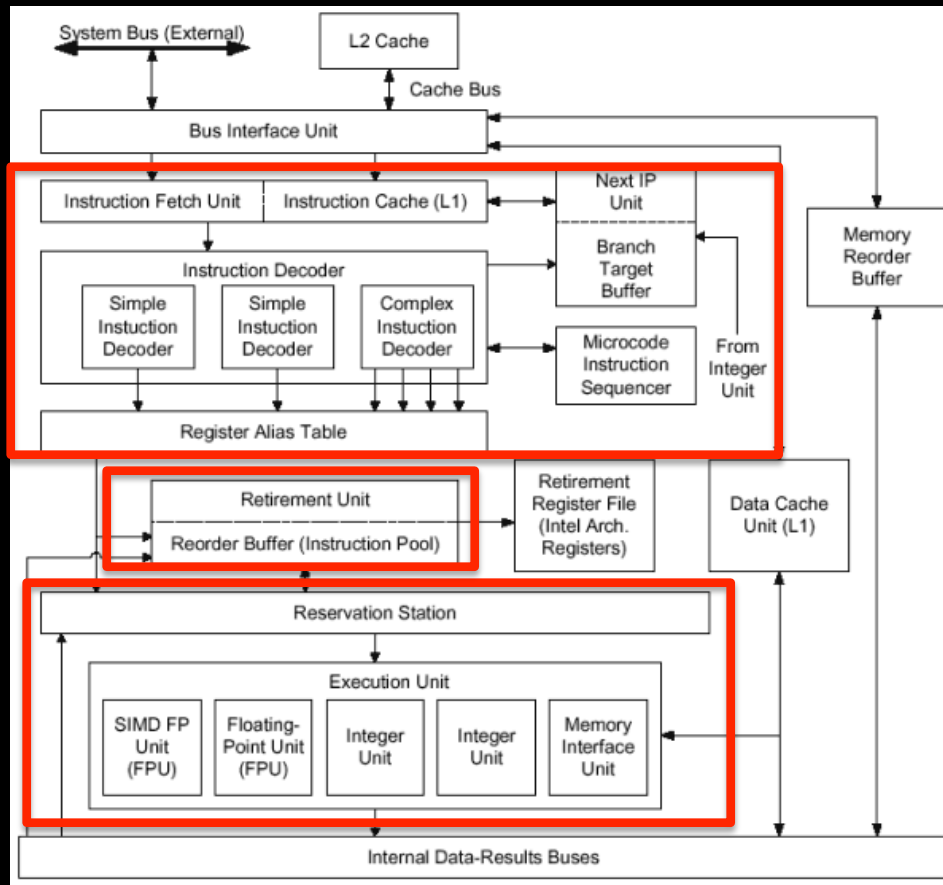
Credit: Jack Dongarra

“Compiler Advances Double Computing Power Every 18 Years!”
– Proebsting’s Law



P6 SUPERSCALAR ARCHITECTURE

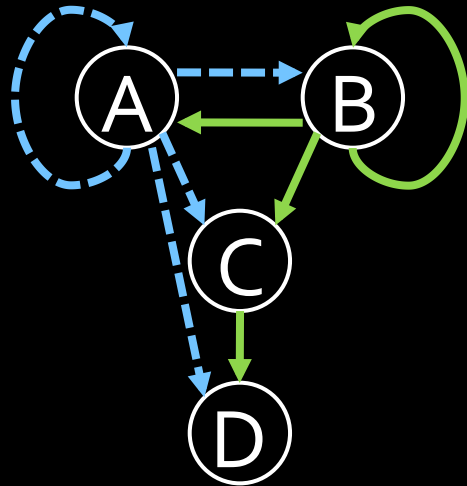
Spec-PS-DSWP



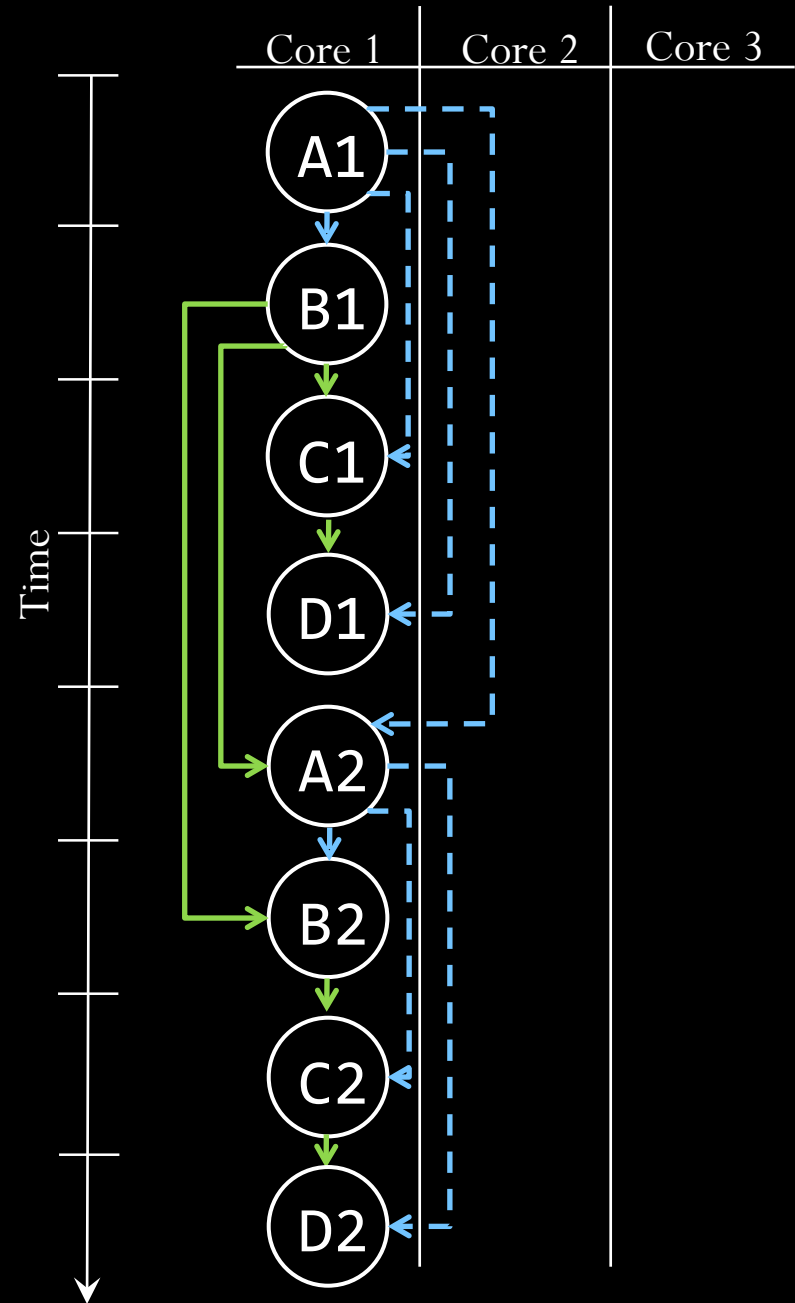
Example

```
A: while (node) {  
  B:   node = node->next;  
  C:   res = work(node);  
  D:   write(res);  
}
```

Program Dependence Graph



---> Control Dependence
—> Data Dependence

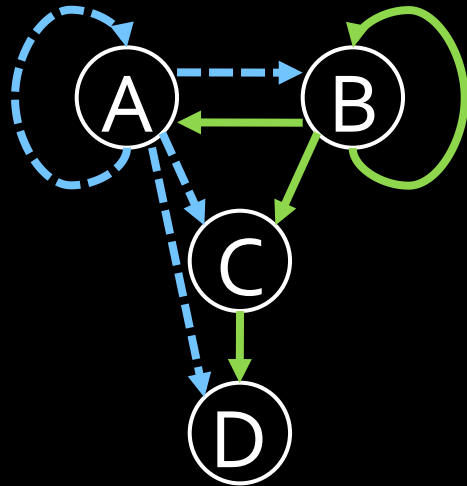


Spec-DOALL

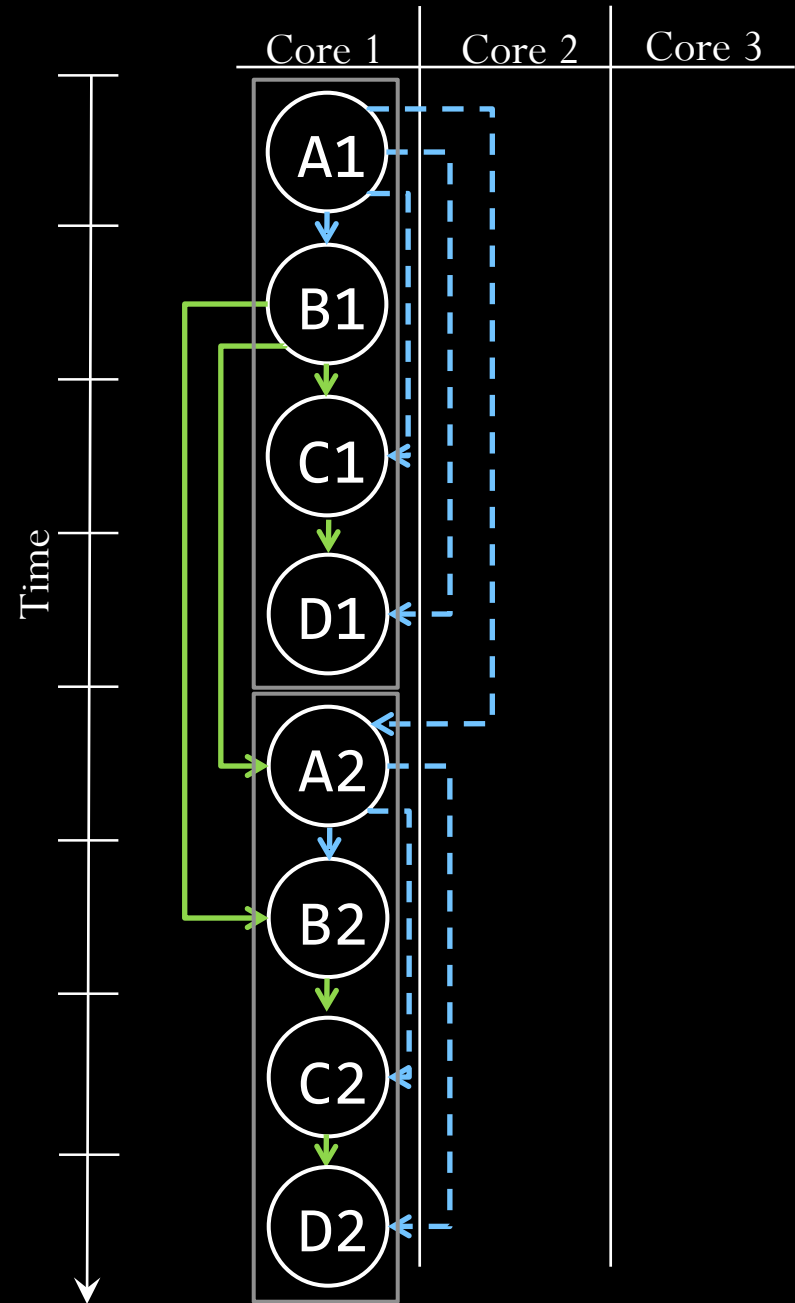
Example

```
A: while (node) {  
B:   node = node->next;  
C:   res = work(node);  
D:   write(res);  
}
```

Program Dependence Graph



---> Control Dependence
—> Data Dependence

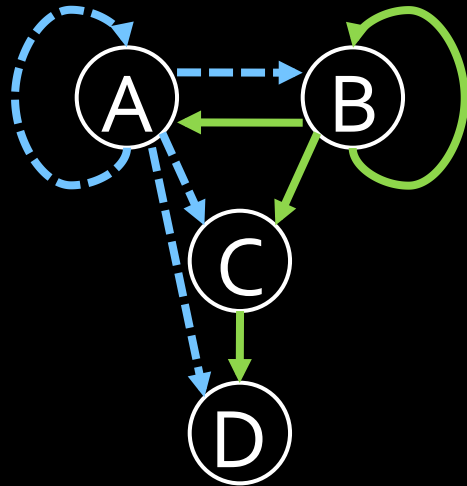


Spec-DOALL

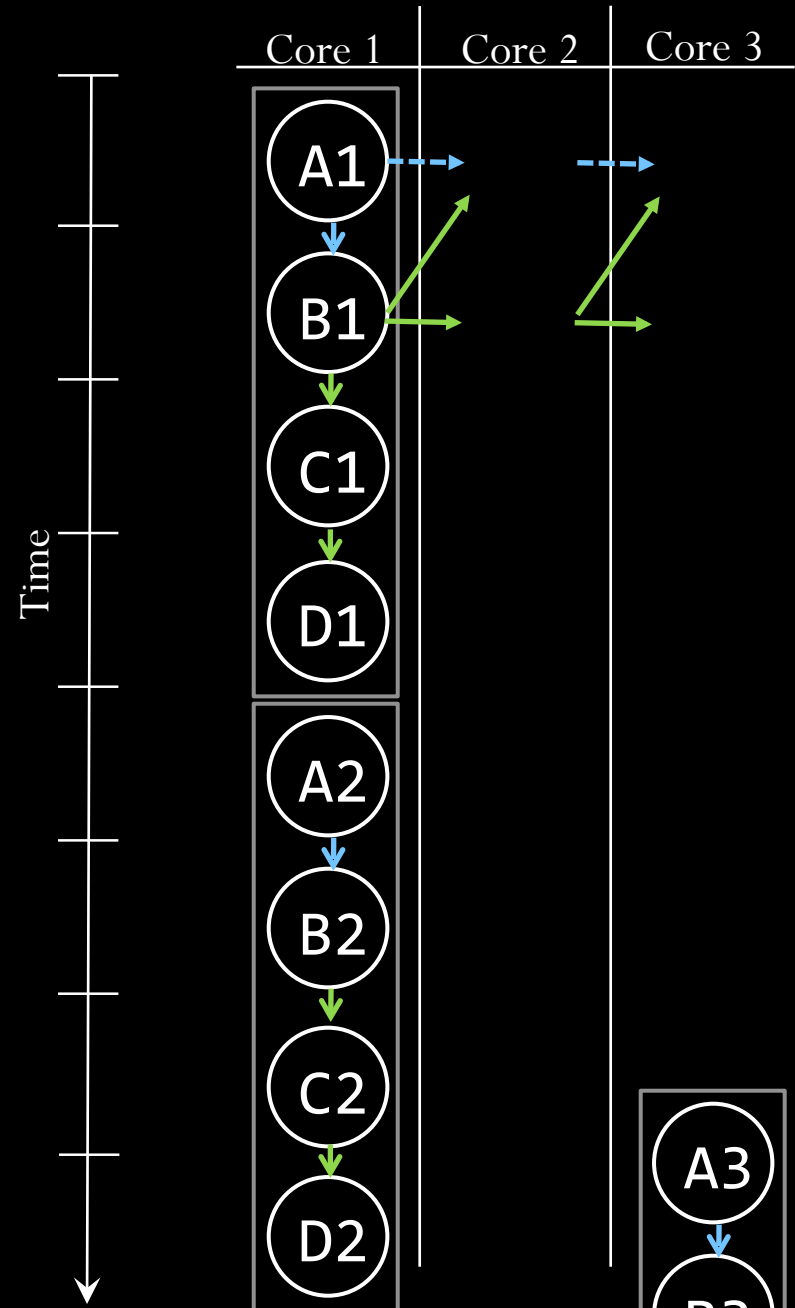
Example

```
A: while (node) {  
B:   node = node->next;  
C:   res = work(node);  
D:   write(res);  
}
```

Program Dependence Graph



---> Control Dependence
——> Data Dependence



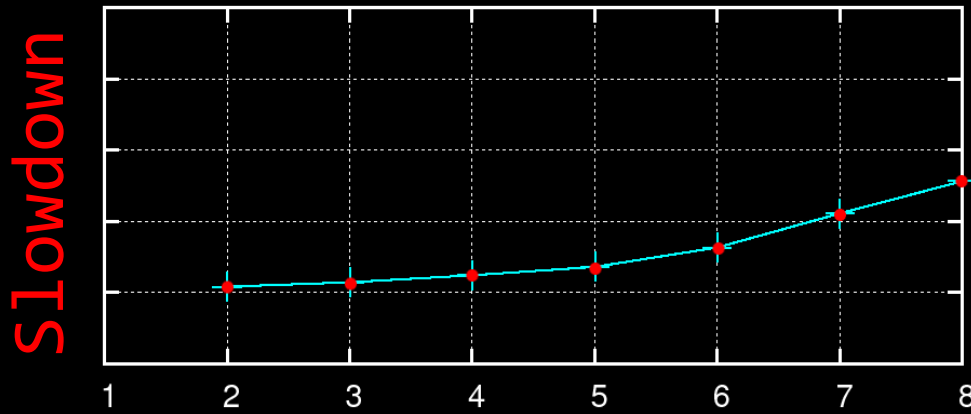
Spec-DOALL

Example

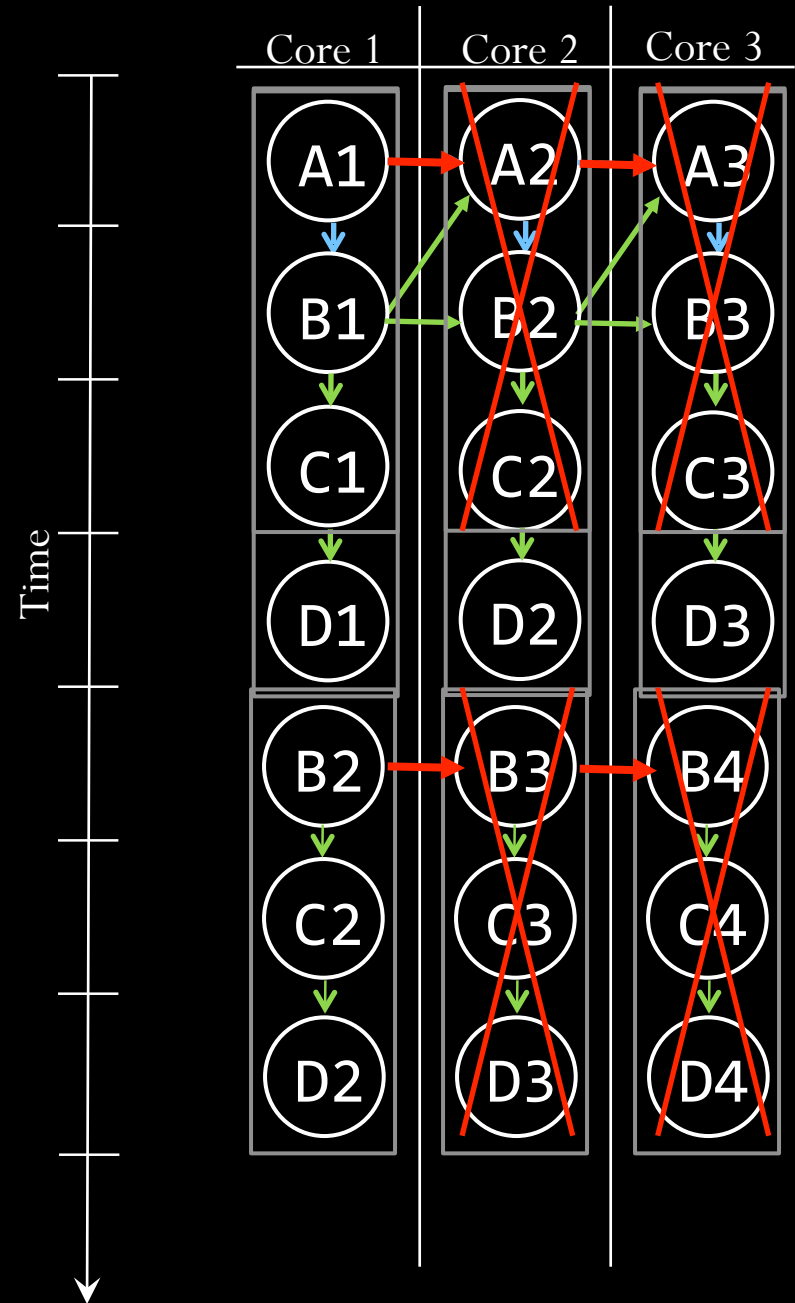
```

A: while (node) {
B:   node = node->next;
C:   res = work(node);
D:   write(res);
   }
    
```

Program Dependence Graph

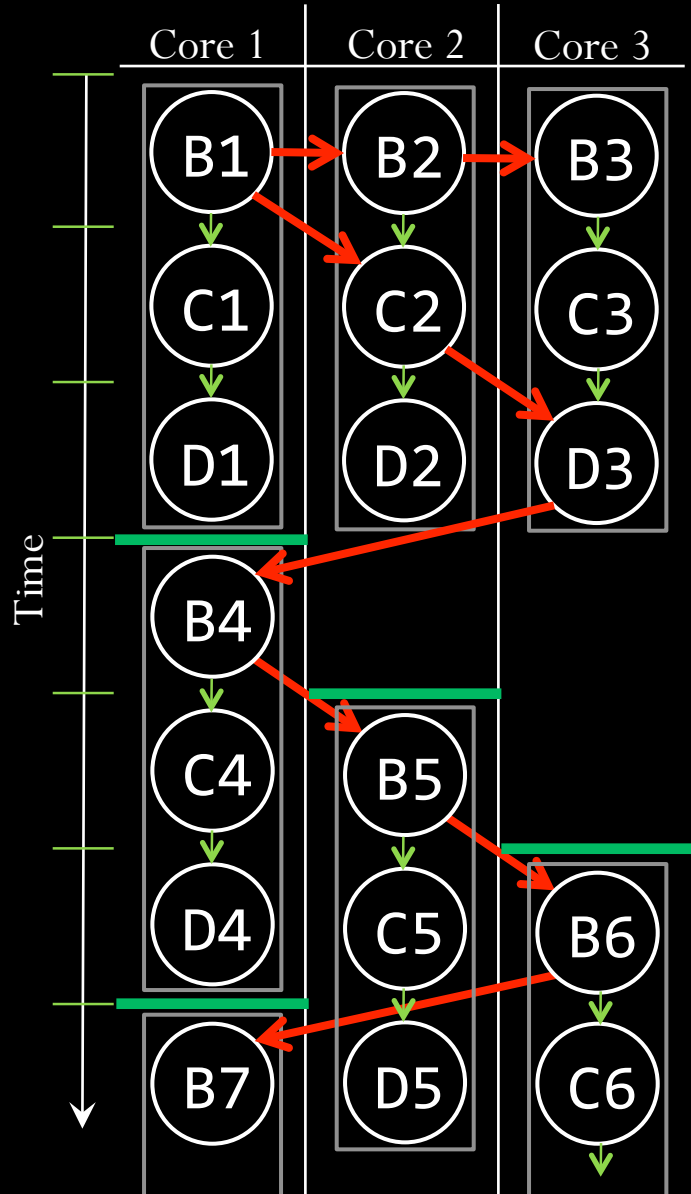


Control Dependence
Data Dependence



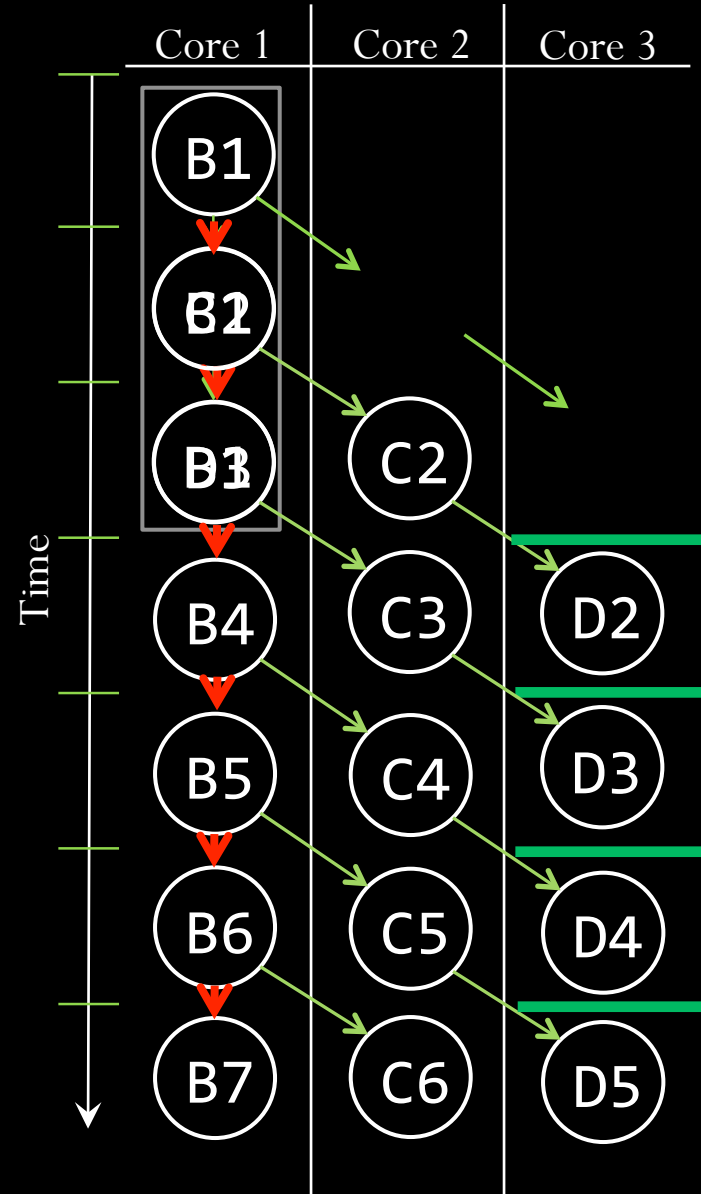
Spec-DOACROSS

Throughput: 1 iter/cycle



Spec-DSWP

Throughput: 1 iter/cycle



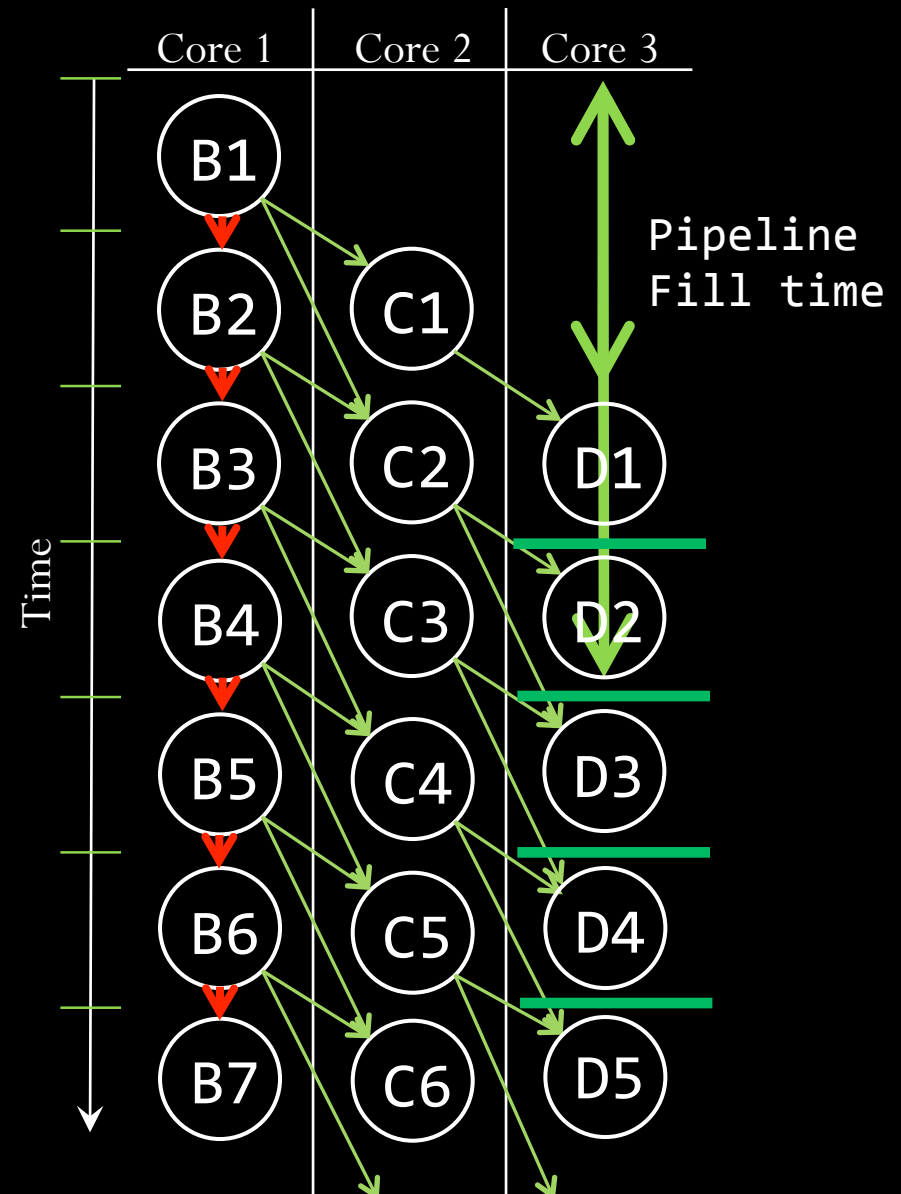
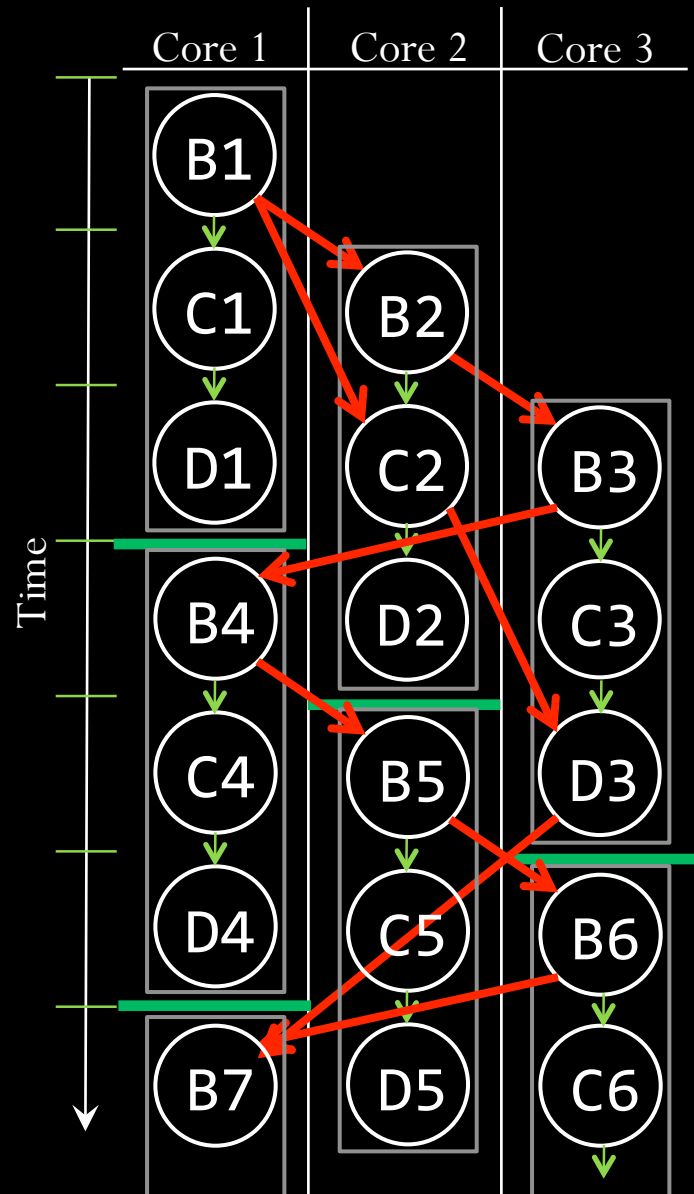
Comparison: Spec-DOACROSS and Spec-DSWP

Comm.Latency = 1: 1 iter/cycle

Comm.Latency = 1: 1 iter/cycle

Comm.Latency = 2: **0.5 iter/cycle**

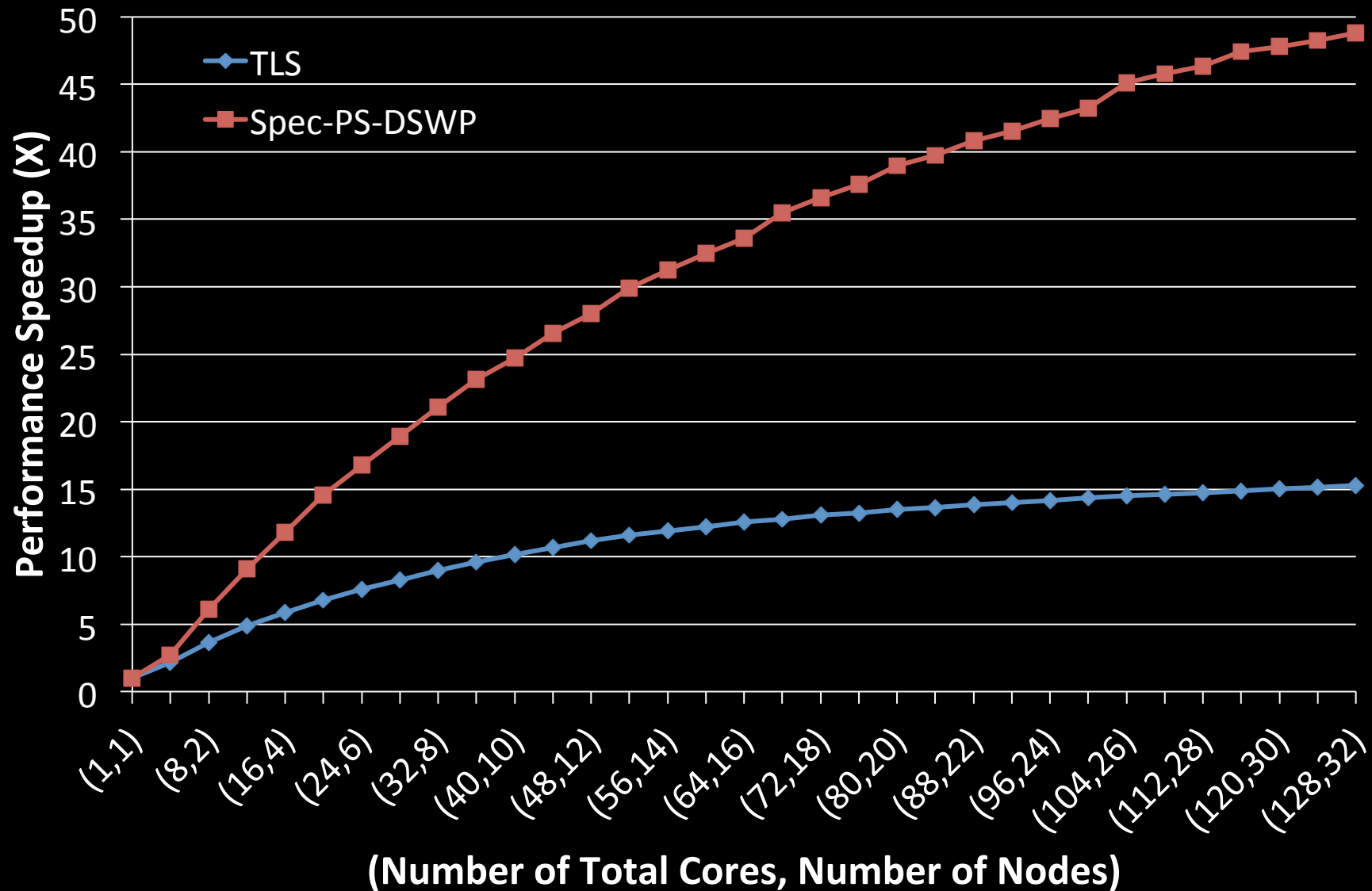
Comm.Latency = 2: **1 iter/cycle**

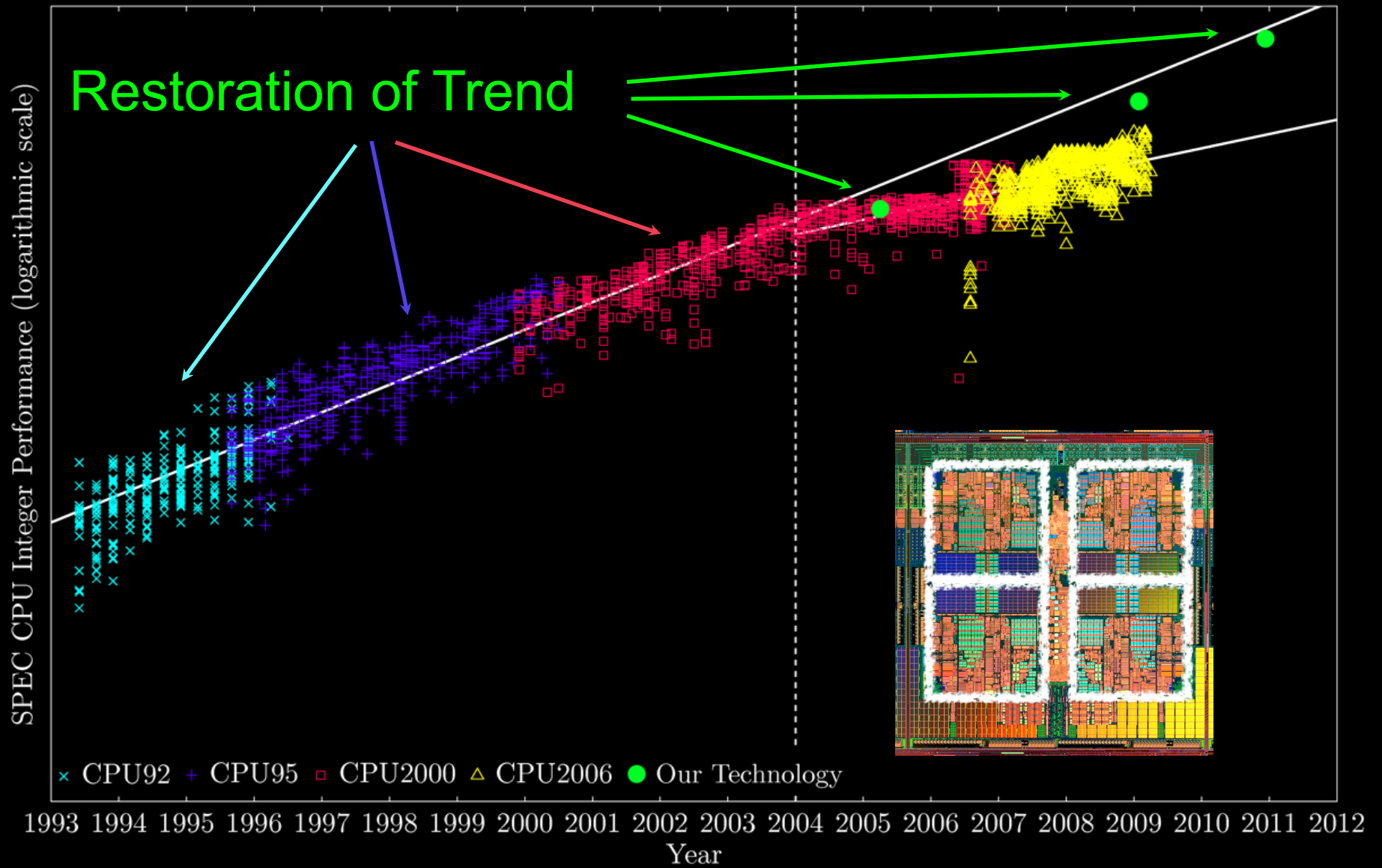


Spec-DOACROSS vs. Spec-DSWP

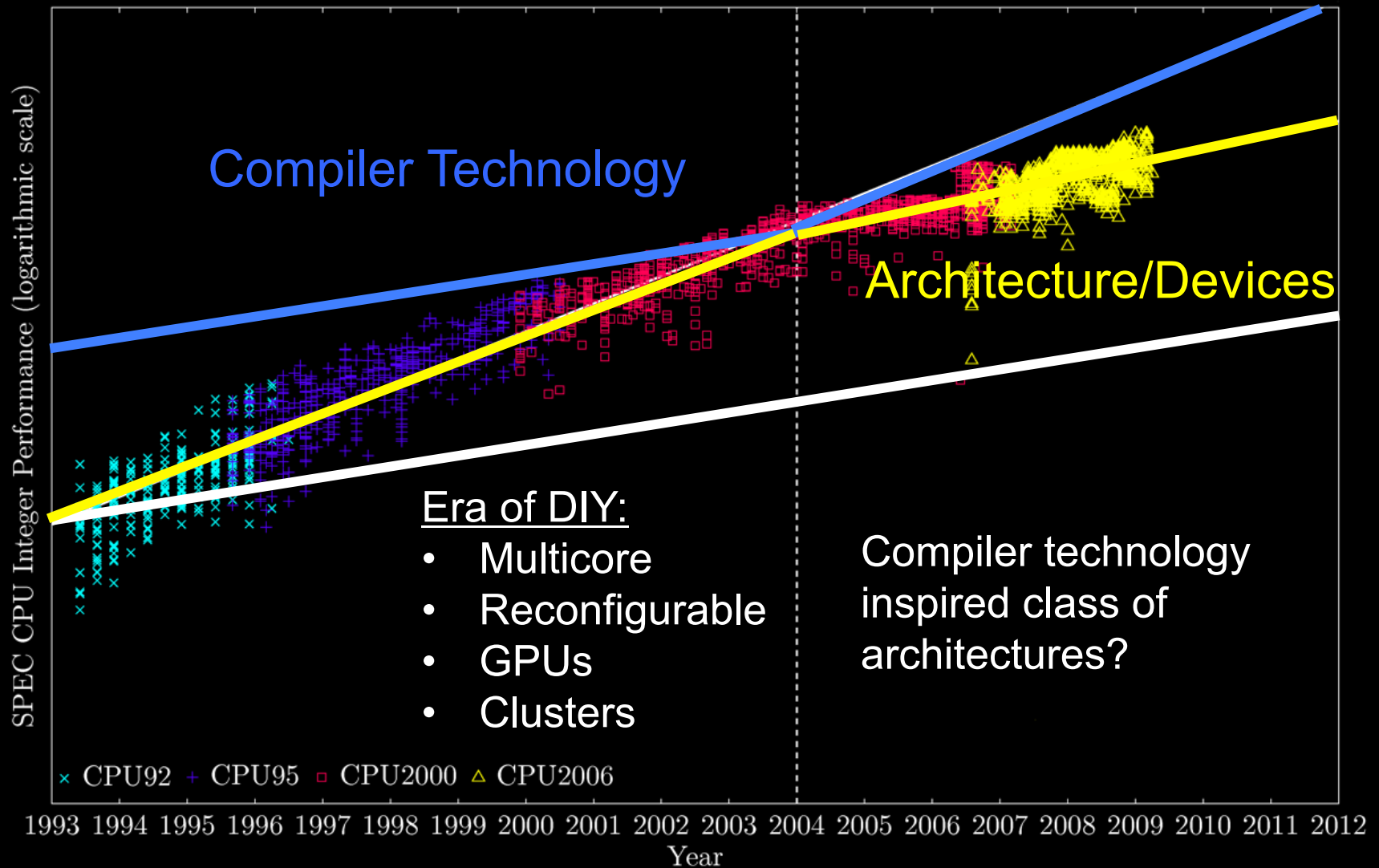
[MICRO 2010]

Geomean of 11 benchmarks on the same cluster





~~“Compiler Advances Double Computing Power Every 18 Years!”~~
~~– Probst’s Law~~



CFGs and PCs

Dependence Graphs and Parallelism



