Topic 14: Parallelism

COS 320

Compiling Techniques

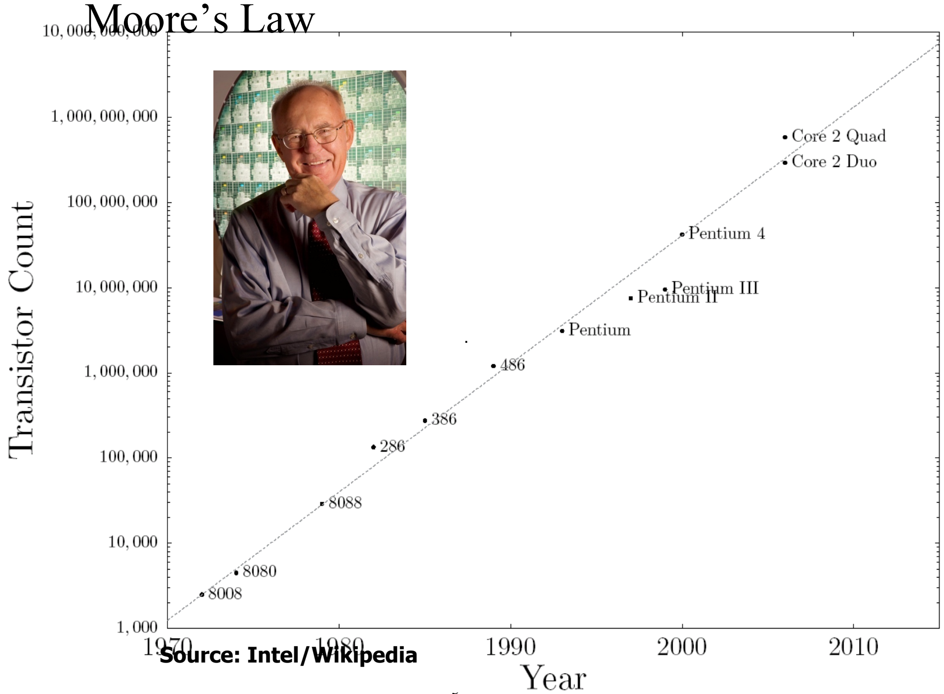
Princeton University
Spring 2015

Prof. David August

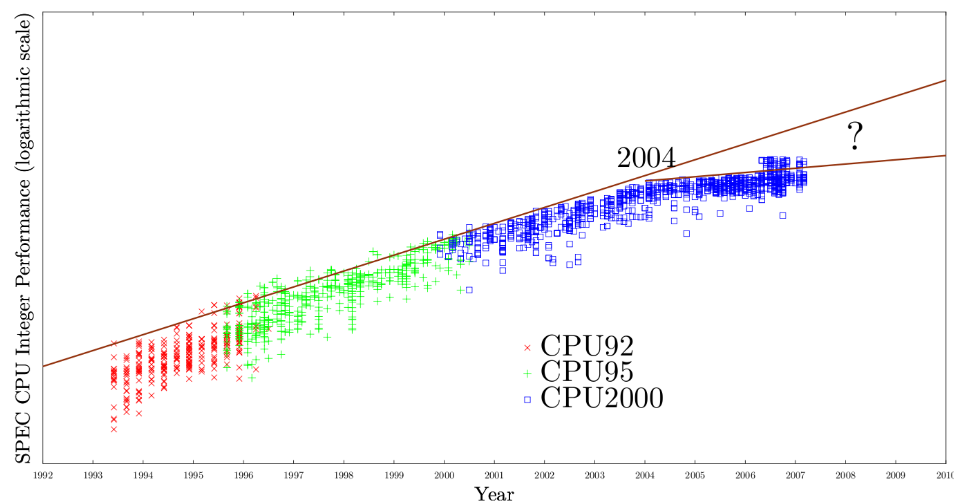
Final Exam!

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

Moore's Law



Single-Threaded Performance Not Improving

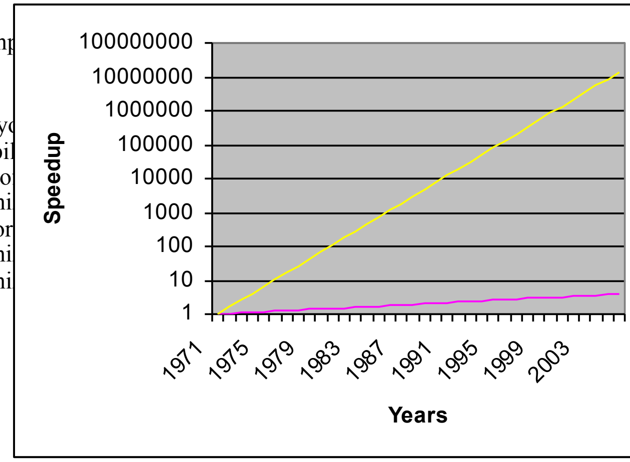


What about Parallel Programming? –or- What is Good About the Sequential Model?

- 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

- "Comp
- Run yo compil ratio o optimi 4X for optimi optimi ... ptimizing bled. The mpiler tio is about piler piler



Conclusion – Compilers not about performance!

Are We Doomed?

A Step Back in Time: Old Skool Parallelization

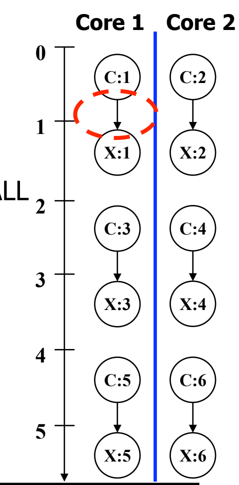
Parallelizing Loops In Scientific Applications

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($i_1, i_{2, \ldots,} i_{n)}$ is <u>affine</u>, 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
  $1 \le i \le 10$, $1 \le i' \le 10$ and $i - 1 = i'$

» For 2, we solve for i and i' in
  $1 \le i \le 10$, $1 \le i' \le 10$, $i = i'$ 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

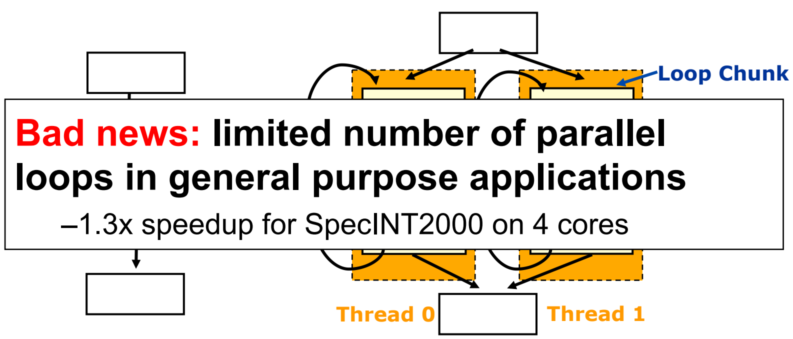
- Array data dependence basically requires finding integer solutions to a system (often refers to as dependence system) consisting of equalities and inequalities.
- Equalities are derived from array accesses.
- Inequalities from the loop bounds.
- It is an integer linear programming problem.
- ILP is 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, DOpipe
- 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

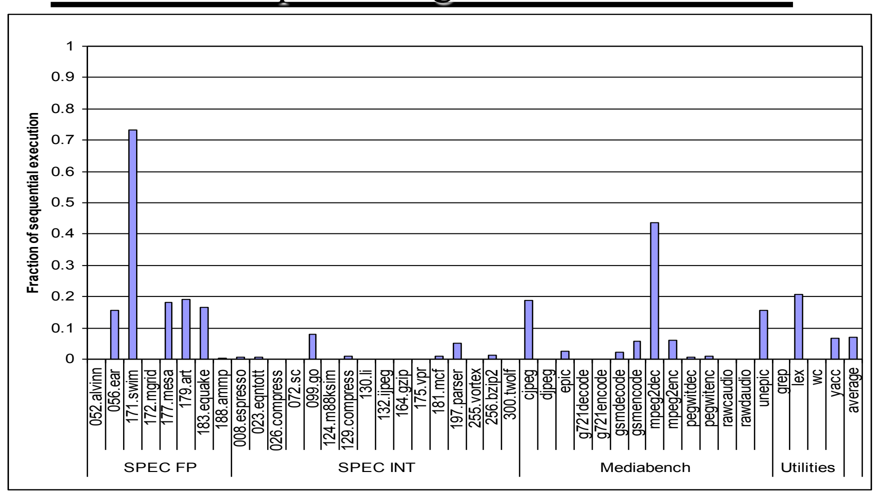
Loop Level Parallelization



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

–1.3x speedup for SpecINT2000 on 4 cores

DOALL Loop Coverage



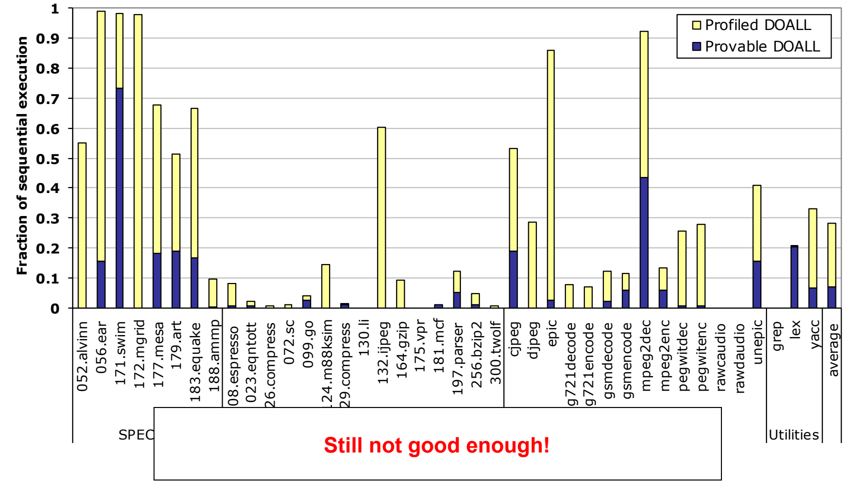
What's the Problem?

1. Memory dependence analysis

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

    . . . = *p;

    *q = . . .

}
```

➡ Memory dependence profiling and speculative parallelization

DOALL Coverage – Provable and Profiled



What's the Next Problem?

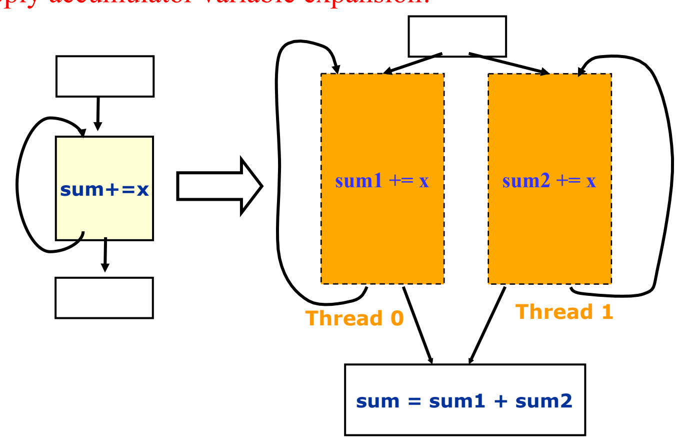
2. Data dependences

```
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

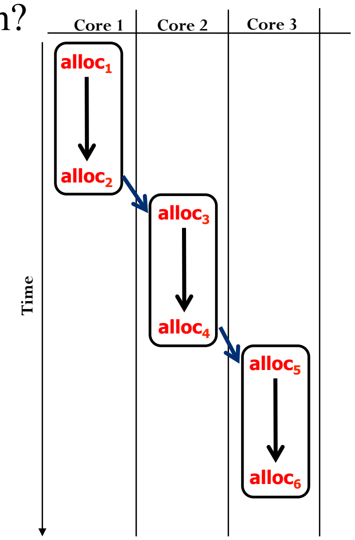
- Accumulator, induction, and min/max expansion only capture a small set of dependences
- 2 options
  - » 1) Break more dependences – New transformations
  - » 2) Parallelize in the presence of dependences – more than DOALL parallelization
- We will talk about both, but for now ignore this issue

What's the Next Problem?

3. C/C++ too restrictive

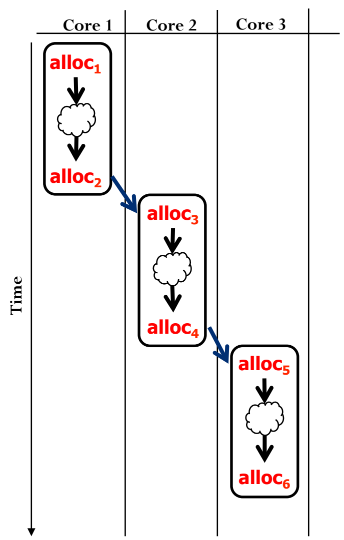
```
char *memory;

void * alloc(int size);

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

Low Level Reality



Low Level Reality

```
char *memory;

void * alloc(int size);

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

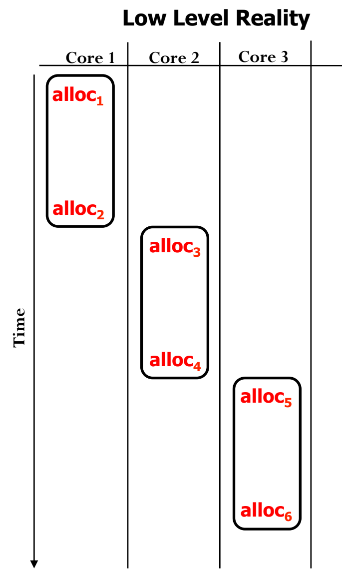


Loops cannot be parallelized even if computation is independent

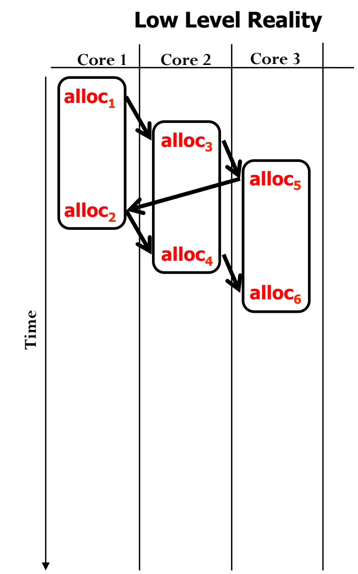
Commutative Extension

- 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

```
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;
}
```



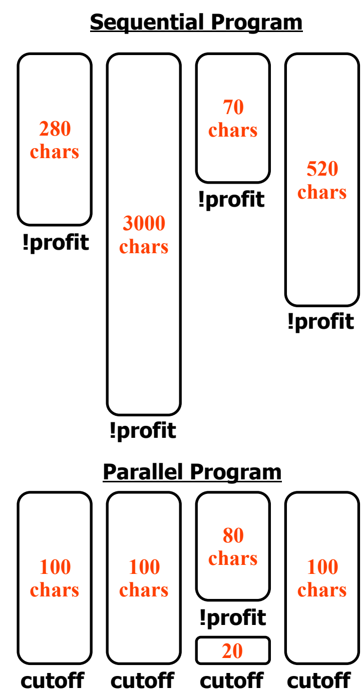
Implementation dependences should not cause serialization.

What is the Next Problem?

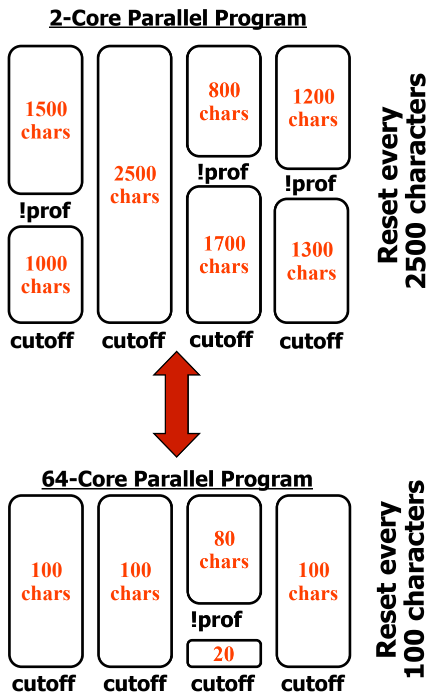
- 4. **C does not allow any prescribed non-determinism**
  - Thus sequential semantics must be assumed even though they not necessary
  - Restricts parallelism (useless dependences)
- Non-deterministic branch → programmer does not care about individual outcomes
  - They attach a probability to control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


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

```
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);
```



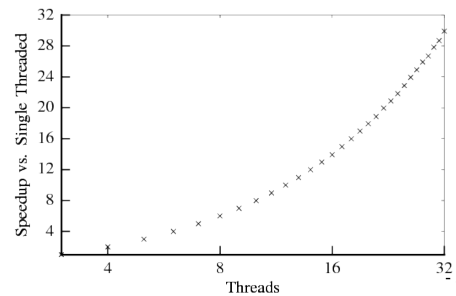
```
dict = create_dict();
while((char = read(1))) {
  profitable =
        compress(char, dict)

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



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)

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

```
char *memory;

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

High-Level View:

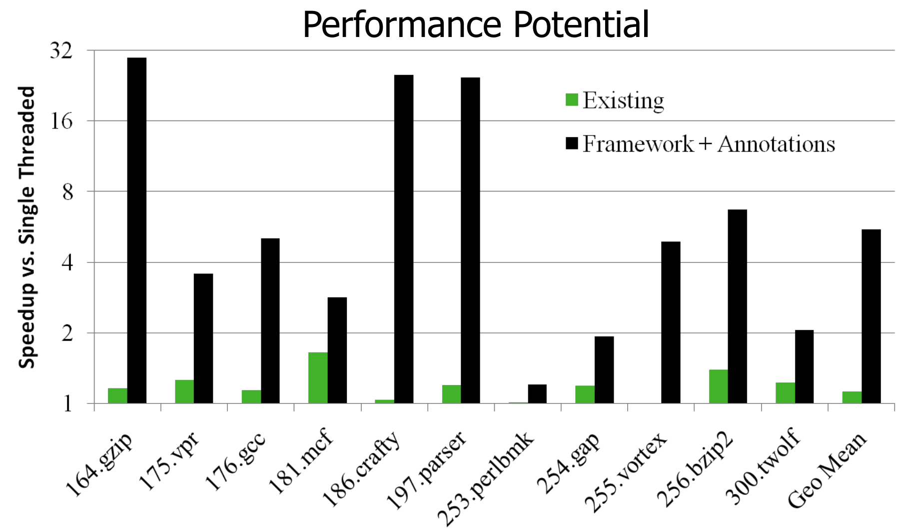
Parsing a sentence is independent of any other sentence.

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.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



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)

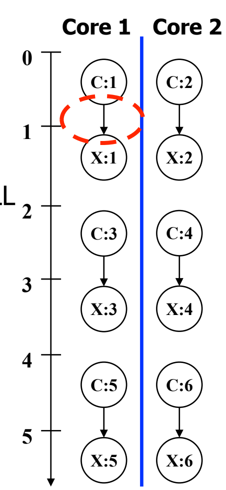
```
for(i=1; i<=N; i++)  // C
  a[i] = a[i] + 1;   // X
```

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

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

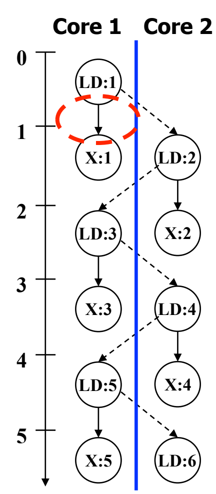
Independent Multithreading (IMT)

Example: DOALL parallelization



Cyclic Multithreading (CMT)

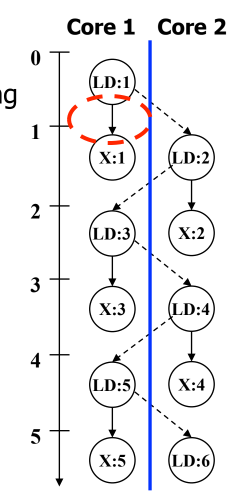
Example: DOACROSS [Cytron, ICPP 86]



Alternative Parallelization Approaches

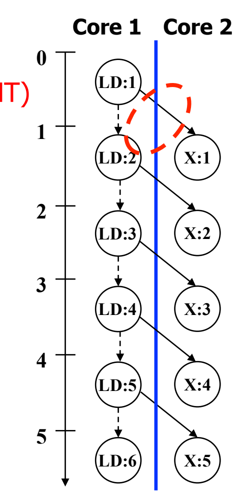
```
while(ptr = ptr->next)      // LD
  ptr->val = ptr->val + 1;  // X
```

Cyclic Multithreading (CMT)

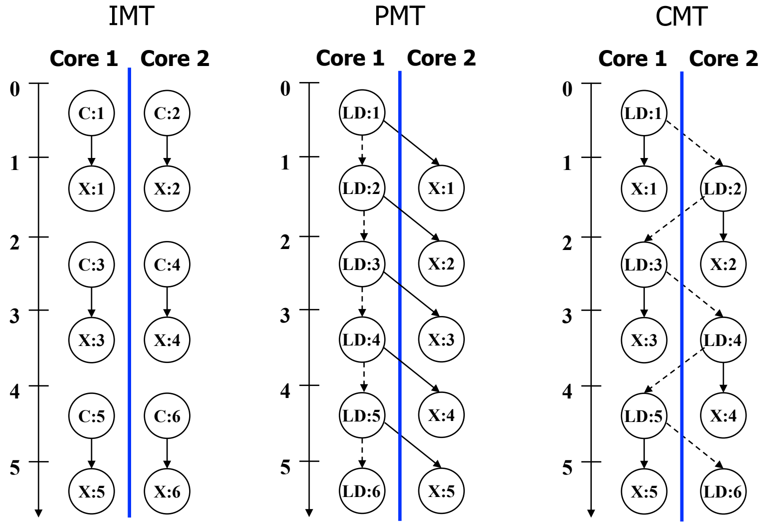


Pipelined Multithreading (PMT)

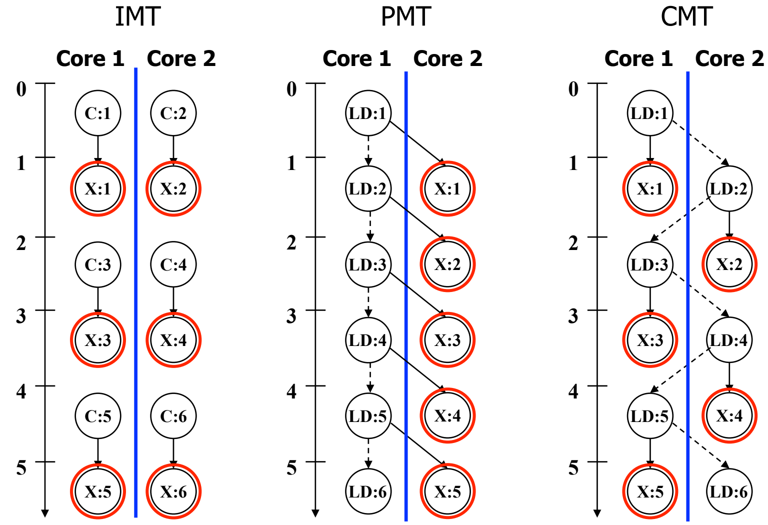
Example: DSWP **[PACT 2004]**



Comparison: IMT, PMT, CMT

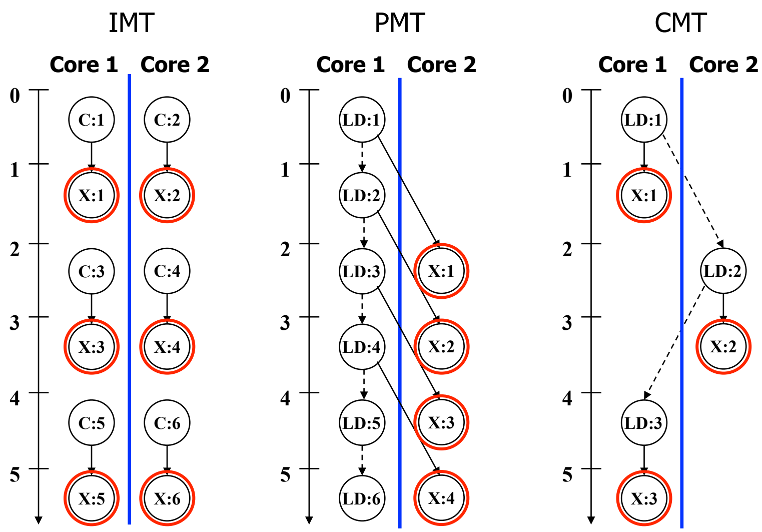


Comparison: IMT, PMT, CMT



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

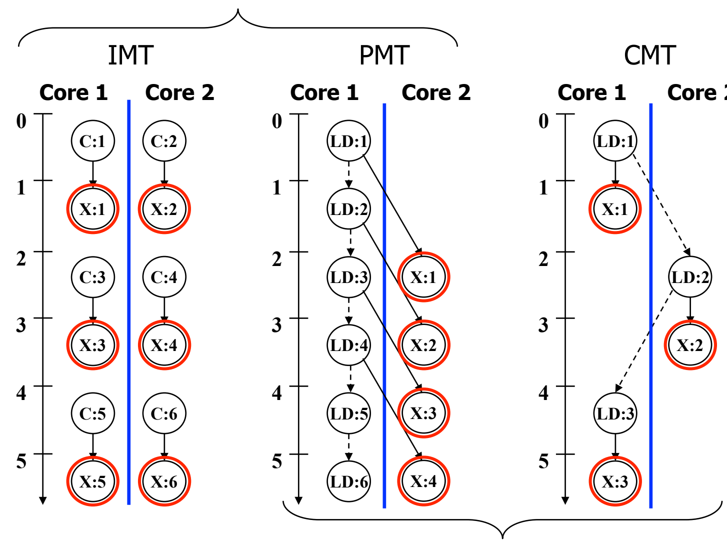
Comparison: IMT, PMT, CMT



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

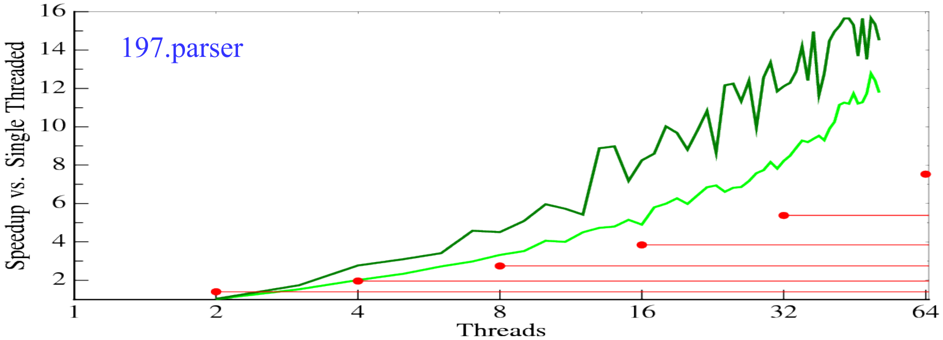
Comparison: IMT, PMT, CMT

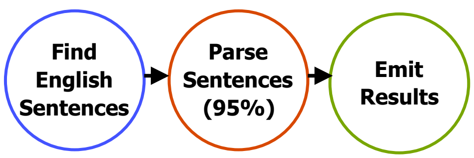
Thread-local Recurrences → Fast Execution



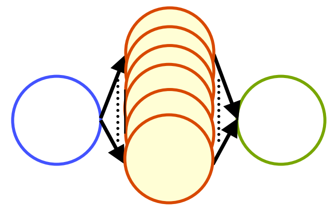
Cross-thread Dependences → Wide Applicability

Our Objective: Automatic Extraction of Pipeline Parallelism using DSWP





Decoupled Software Pipelining



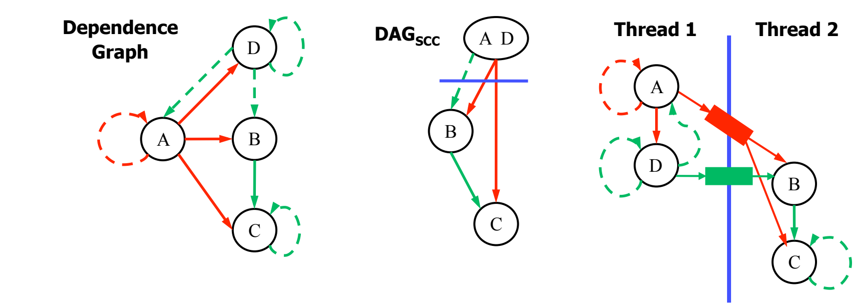
PS-DSWP (Spec DOALL Middle Stage)

Decoupled Software Pipelining

Decoupled Software Pipelining (DSWP)

[MICRO 2005]

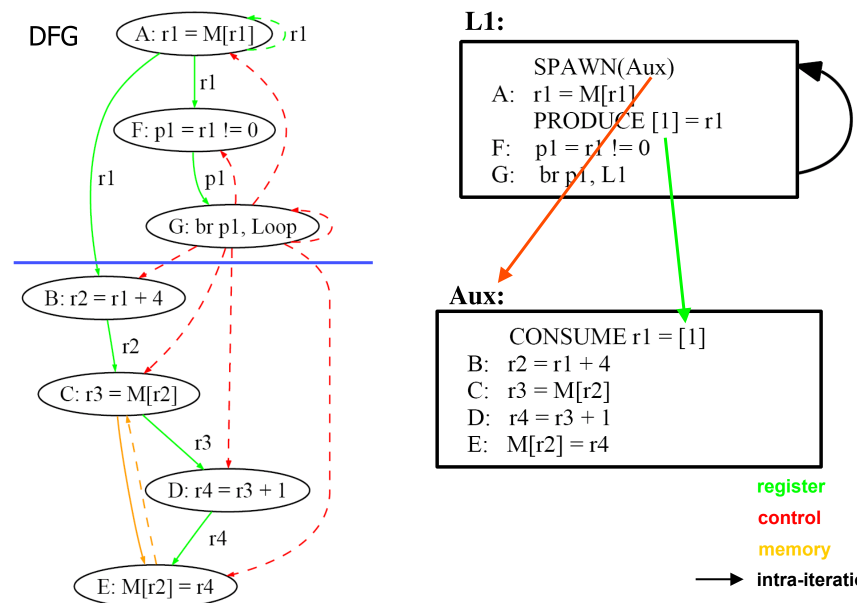
```
A: while(node)
B:    ncost = doit(node);
C:    cost += ncost;
D:    node = node->next;
```



register
control
intra-iteration
loop-carried
communication queue

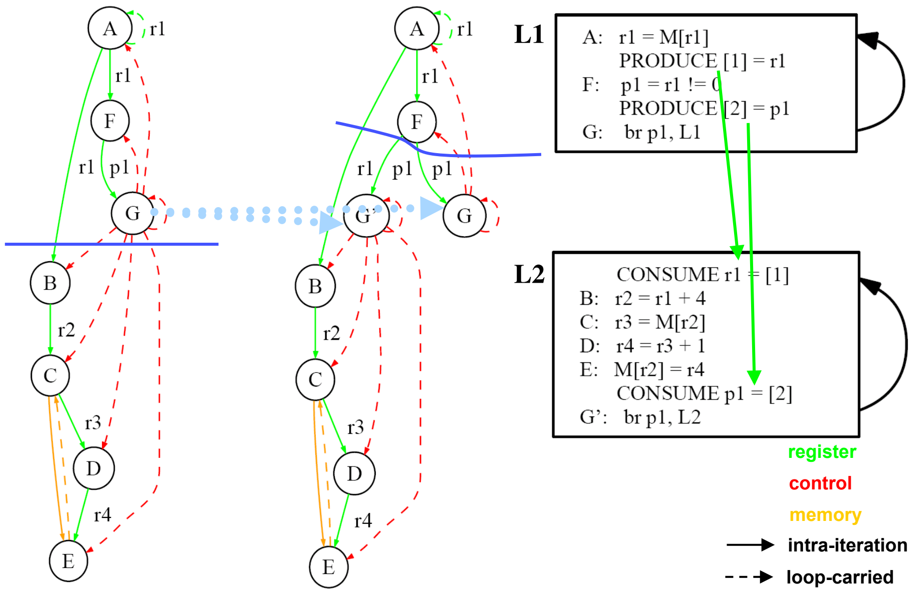
Inter-thread communication latency is a one-time cost

Implementing DSWP

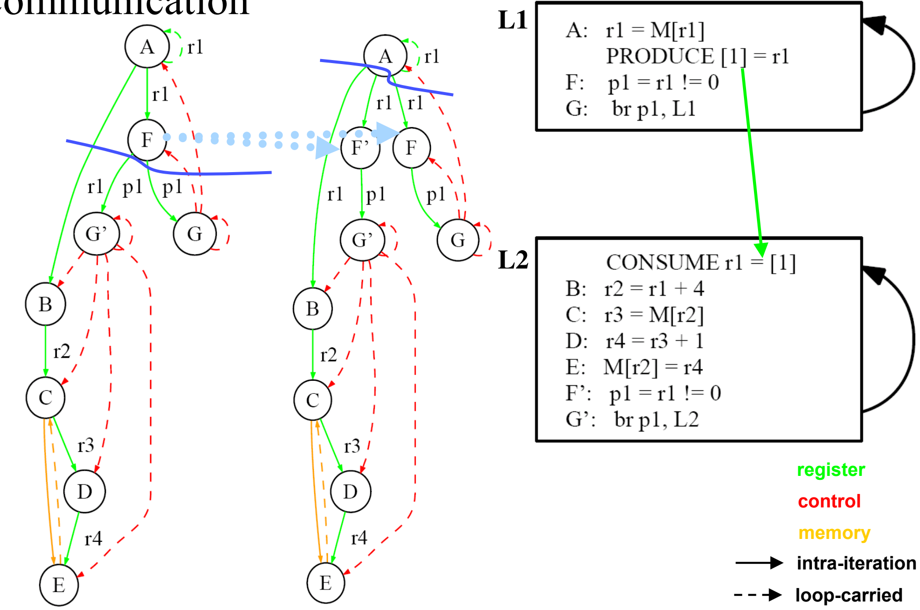


register
control
memory
intra-iteration
loop-carried

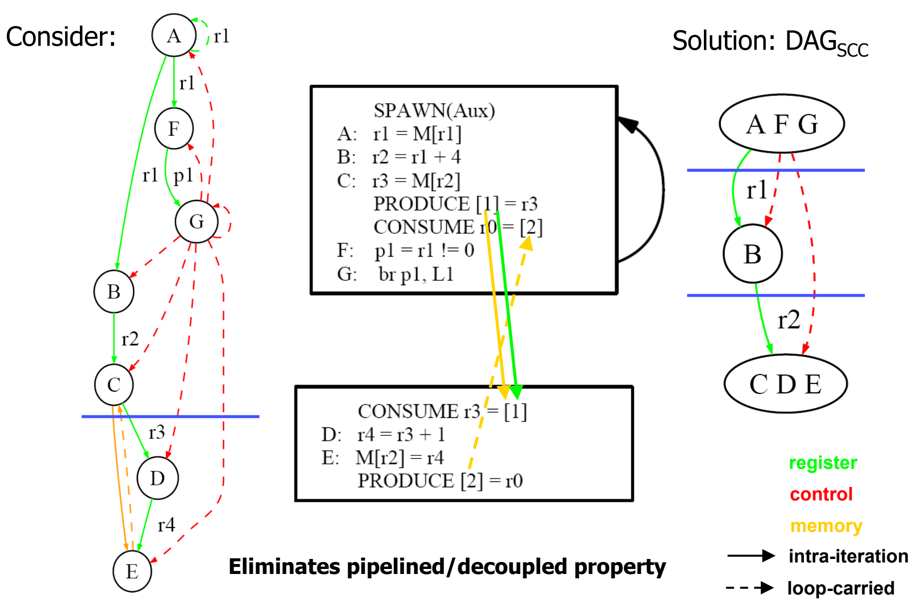
Optimization: Node Splitting To Eliminate Cross Thread Control



Optimization: Node Splitting To Reduce Communication



Constraint: Strongly Connected Components

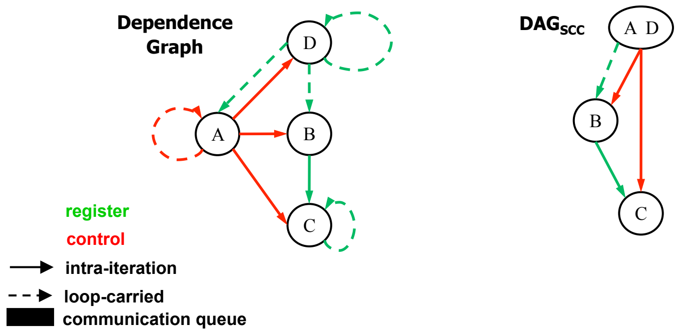


2 Extensions to the Basic Transformation

- 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

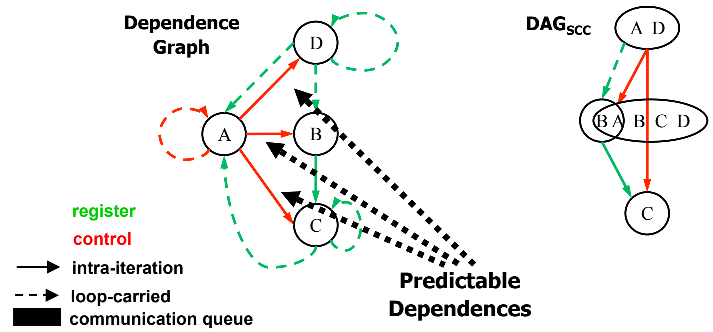
Why Speculation?

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



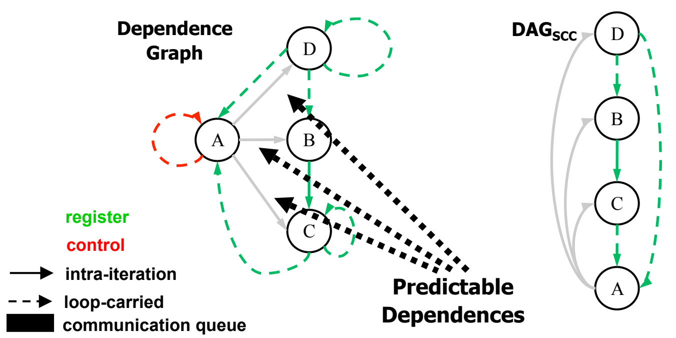
Why Speculation?

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

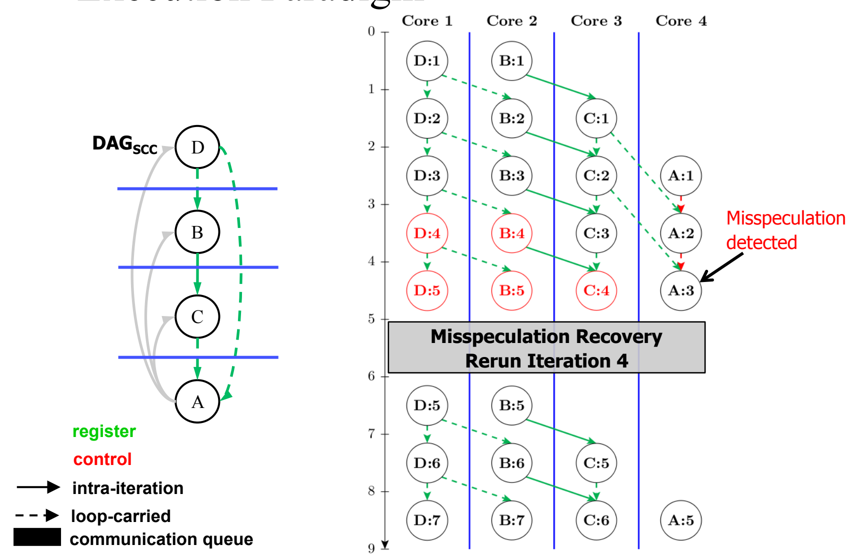


Why Speculation?

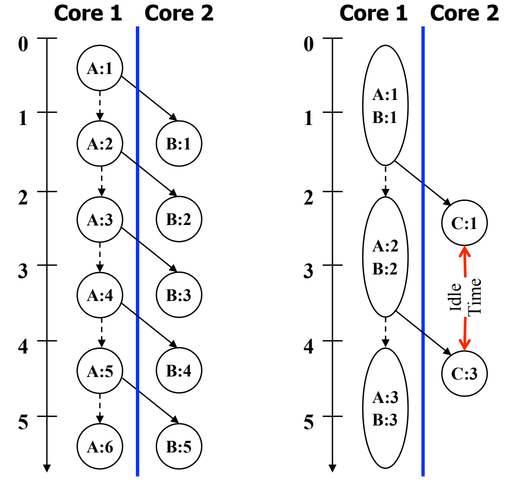
```
A: while(cost < T && node)
B:    ncost = doit(node);
C:    cost += ncost;
D:    node = node->next;
```



Execution Paradigm



Understanding PMT Performance



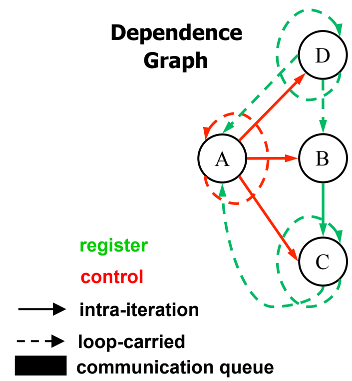
$$T \propto \max(t_i)$$

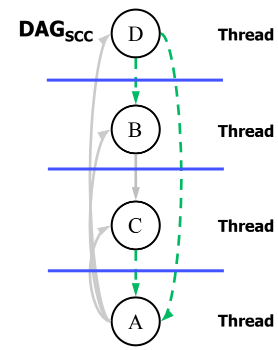
1. Rate t_i is at least as large as the longest dependence recurrence.
2. NP-hard to find longest recurrence.
3. Large loops make problem difficult in practice.

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;
```





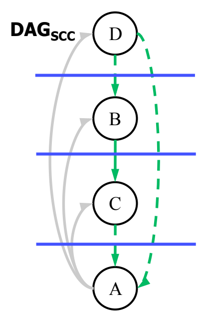
Detecting Misspeculation

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);
```



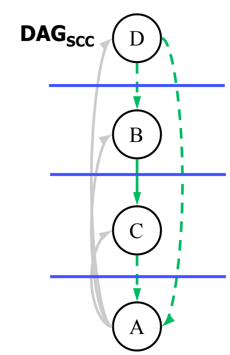
Detecting Misspeculation

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);
```



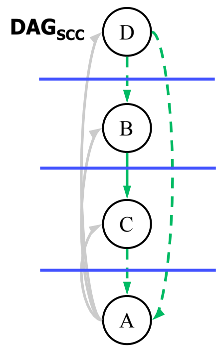
Detecting Misspeculation

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

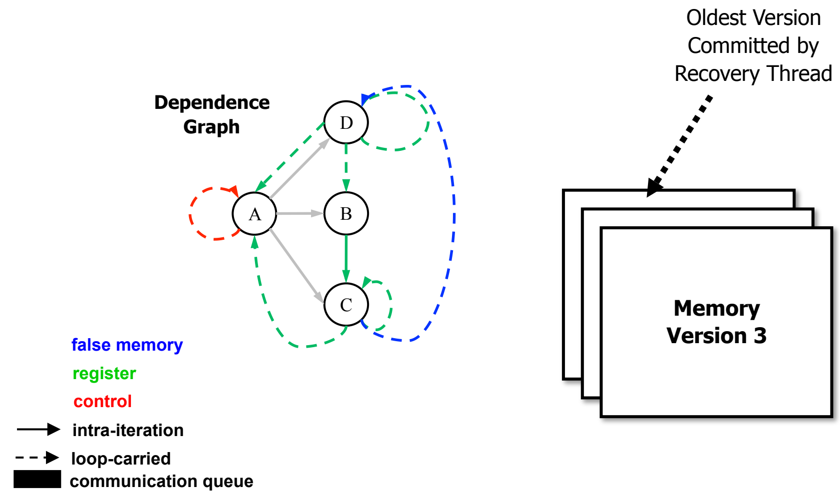
Thread 2
```
A²: while(TRUE)
B :    ncost = doit(node);
       produce(2,ncost);
D²:    node = consume(0);
```

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

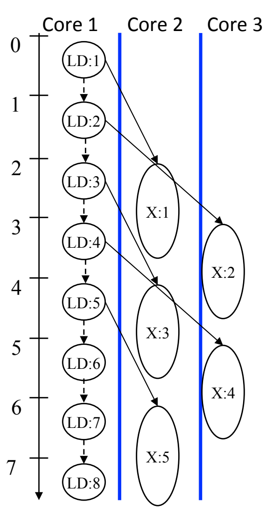
Thread 4
```
A : while(cost < T && node)
B⁴:    cost = consume(3);
C⁴:    node = consume(1);
       if(!(cost < T && node))
         FLAG_MISSPEC();
```



Breaking False Memory Dependences



Adding Parallel Stages to DSWP




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

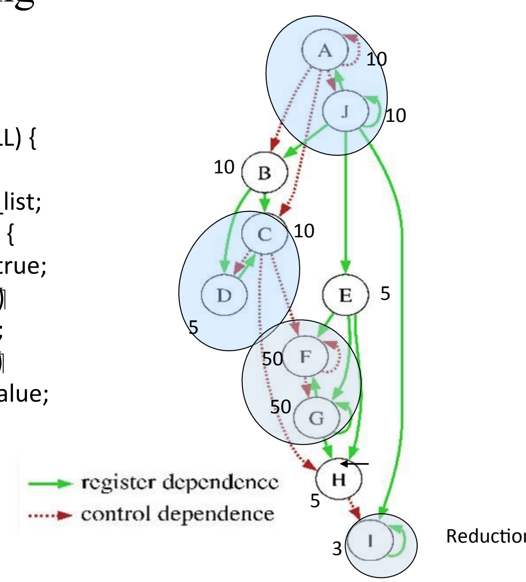
LD = 1 cycle
X = 2 cycles

Comm. Latency = 2 cycles

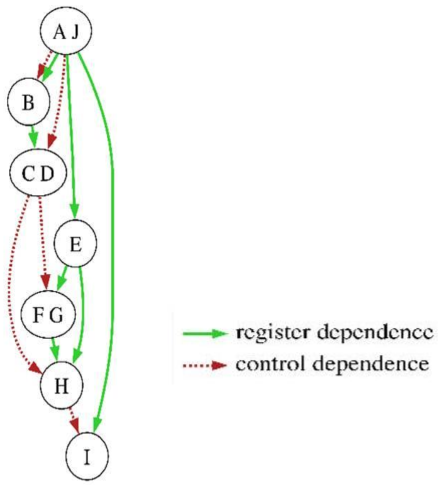
Throughput
DSWP: 1/2 iteration/cycle
DOACROSS: 1/2 iteration/cycle
PS-DSWP: 1 iteration/cycle

Thread Partitioning

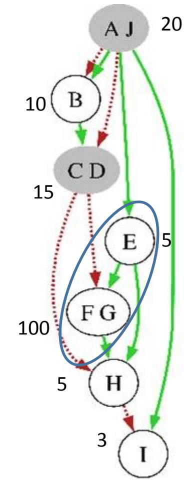
```
    p = list;
    sum = 0;
A:  while (p != NULL) {
B:    id = p->id;
E:      q = p->inner_list;
C:    if (!visited[id]) {
D:      visited[id] = true;
F:      while (foo(q))
G:        q = q->next;
H:      if (q != NULL)
I:        sum += p->value;
      }
J:    p = p->next;
    }
```



Thread Partitioning: DAG_{SCC}

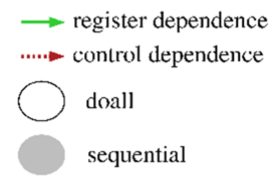


Thread Partitioning

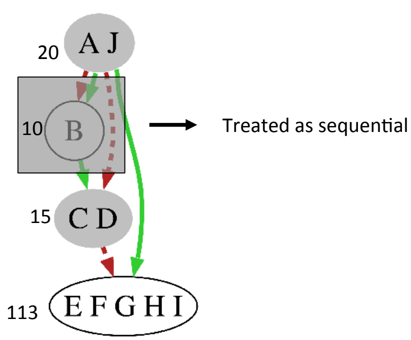


Merging Invariants

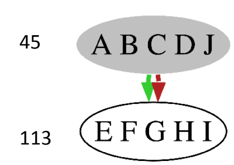
- No cycles
- No loop-carried dependence inside a doall node



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 = MEM[r1]
3. r2 = r2 + 1
4. MEM[r1] = r2
5. Branch r1 < 1000
```

No register live outs

Loop-Level Parallelization: DOALL

1. r1 = 10	1. r1 = 9	1. r1 = 10
1. r1 = r1 + 1 2. r2 = MEM[r1] 3. r2 = r2 + 1 4. MEM[r1] = r2 5. Branch r1 < 1000	1. r1 = r1 + 2 2. r2 = MEM[r1] 3. r2 = r2 + 1 4. MEM[r1] = r2 5. Branch r1 < 999	1. r1 = r1 + 2 2. r2 = MEM[r1] 3. r2 = r2 + 1 4. MEM[r1] = r2 5. Branch r1 < 1000

No register live outs

Another Example

```
1. r1 = 10
```

```
1. r1 = r1 + 1
2. r2 = MEM[r1]
3. r2 = r2 + 1
4. MEM[r1] = r2
5. Branch r2 == 10
```

No register live outs

Another Example

```
1. r1 = 10
```

```
1. r1 = r1 + 1
2. r2 = MEM[r1]
3. r2 = r2 + 1
4. MEM[r1] = r2
5. Branch r2 == 10
```

```
1. r1 = 9
```

```
1. r1 = r1 + 2
2. r2 = MEM[r1]
3. r2 = r2 + 1
4. MEM[r1] = r2
5. Branch r2 == 10
```

```
1. r1 = 10
```

```
1. r1 = r1 + 2
2. r2 = MEM[r1]
3. r2 = r2 + 1
4. MEM[r1] = r2
5. Branch r2 == 10
```

No register live outs

Speculation

```
1. r1 = 9
```

```
1. r1 = r1 + 2
2. r2 = MEM[r1]
3. r2 = r2 + 1
4. MEM[r1] = r2
5. Branch r2 == 10
```

```
1. r1 = 10
```

```
1. r1 = r1 + 2
2. r2 = MEM[r1]
3. r2 = r2 + 1
4. MEM[r1] = r2
5. Branch r2 == 10
```

No register live outs

Speculation, Commit, and Recovery

```
1. r1 = 9
```

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

```
1. r2 = Receive{1}
2. Branch r2 != 10
3. MEM[r1] = r2
4. r2 = Receive{2}
5. Branch r2 != 10
6. MEM[r1] = r2
7. Jump
```

```
1. r1 = 10
```

```
1. r1 = r1 + 2
2. r2 = MEM[r1]
3. r2 = r2 + 1
4. 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

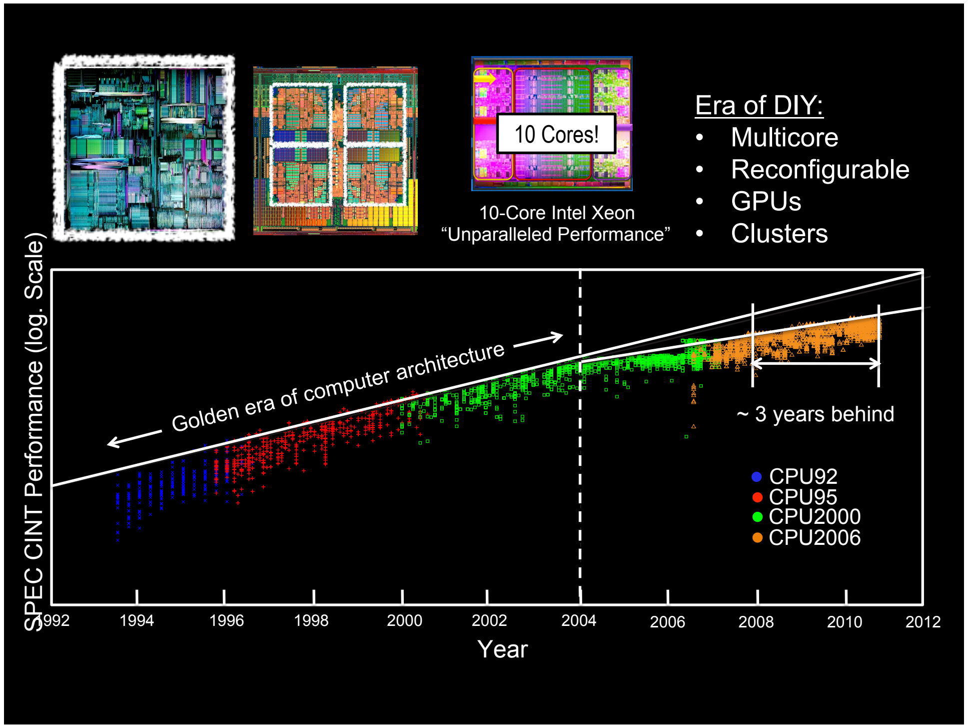
No register live outs

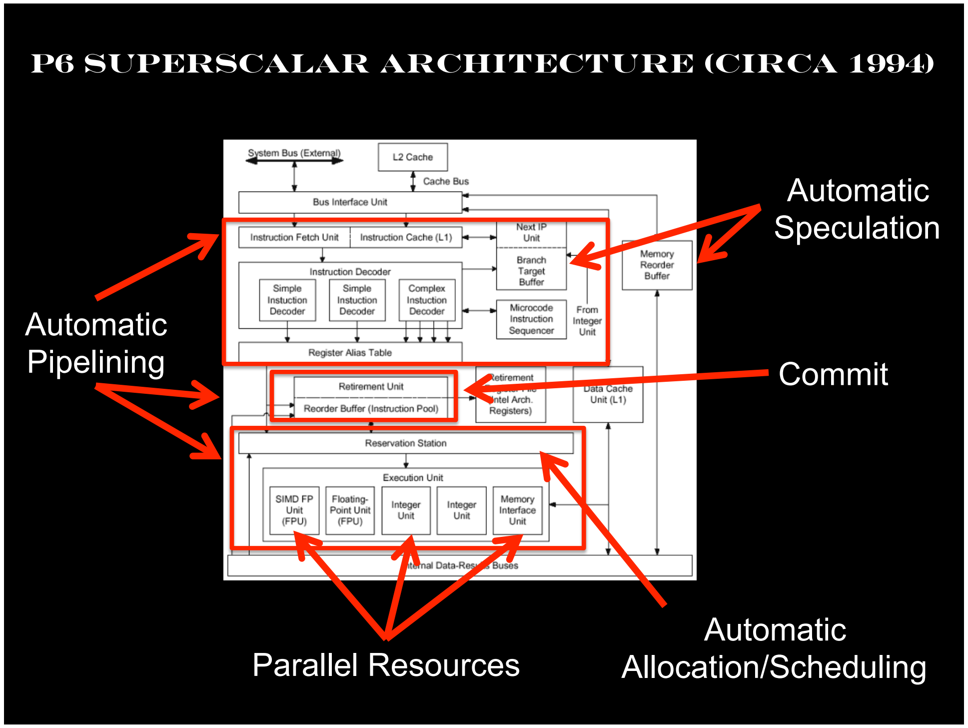
PS-DSWP

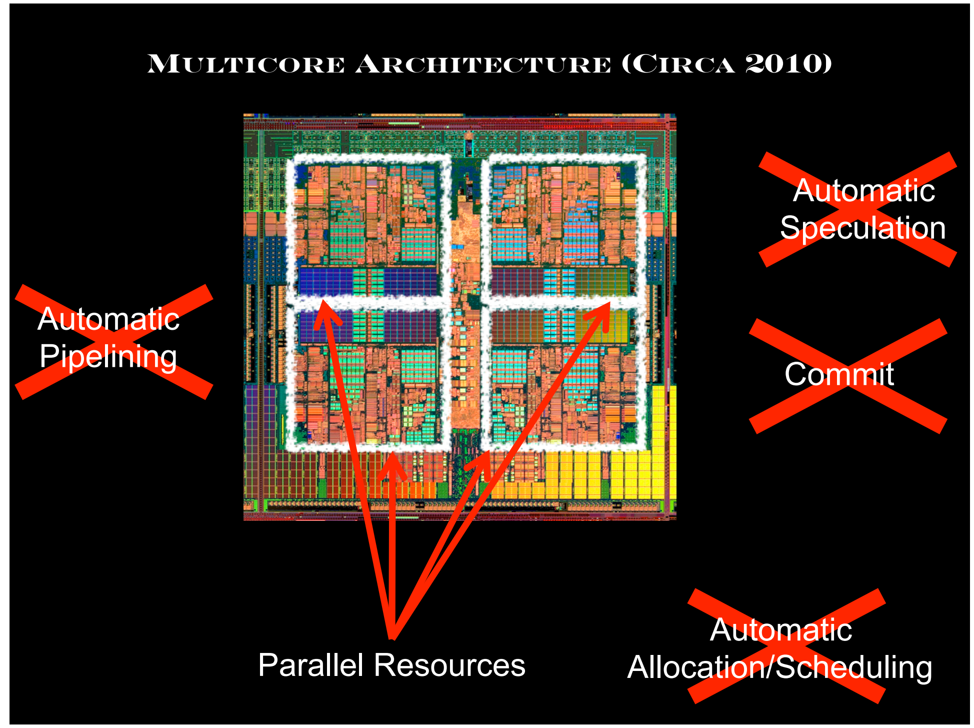
1. r1 = Head

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







ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL

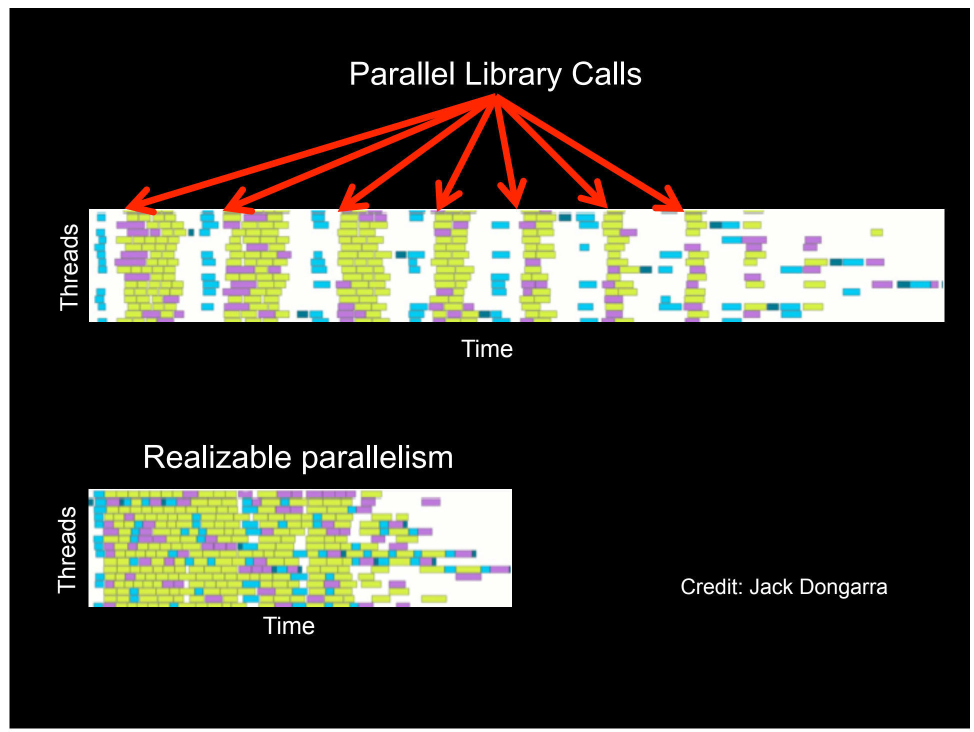
CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda

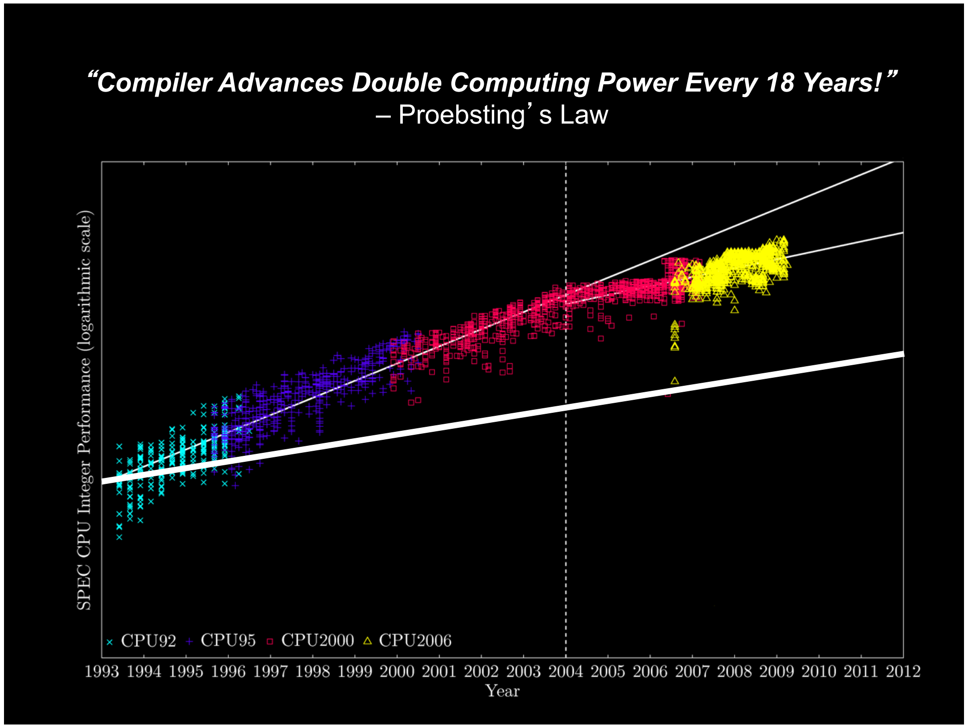
GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold

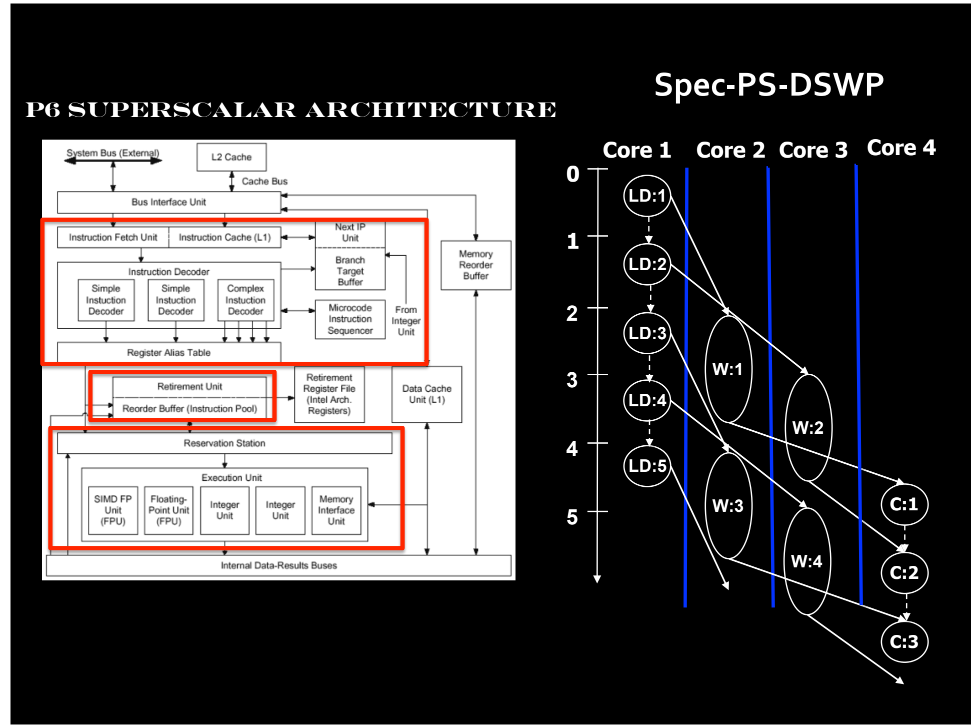
Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm

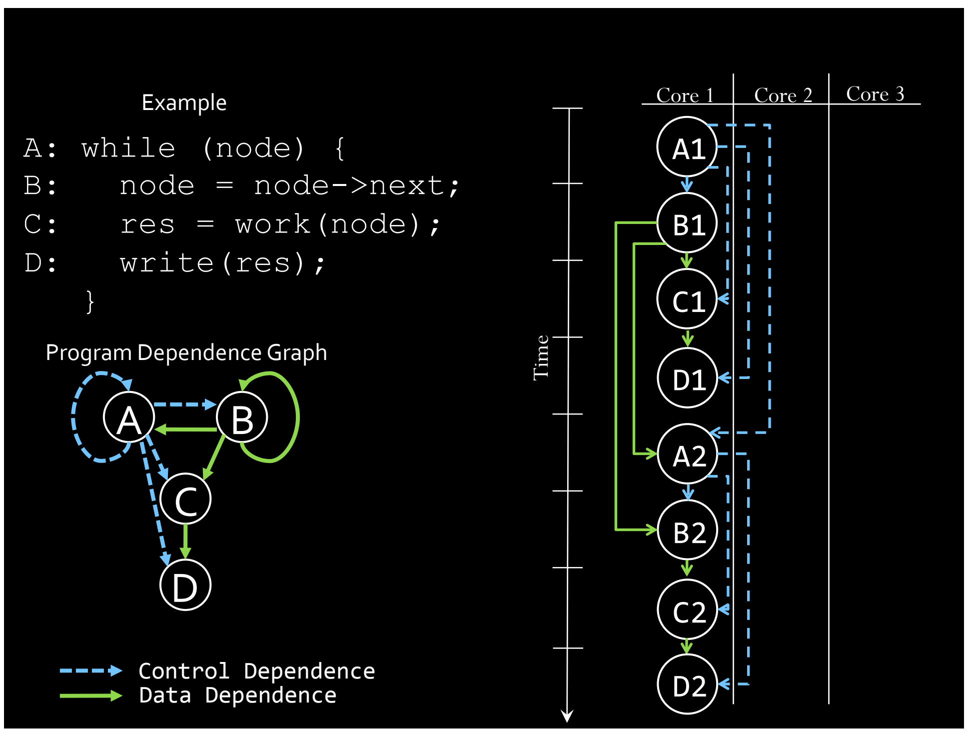
Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

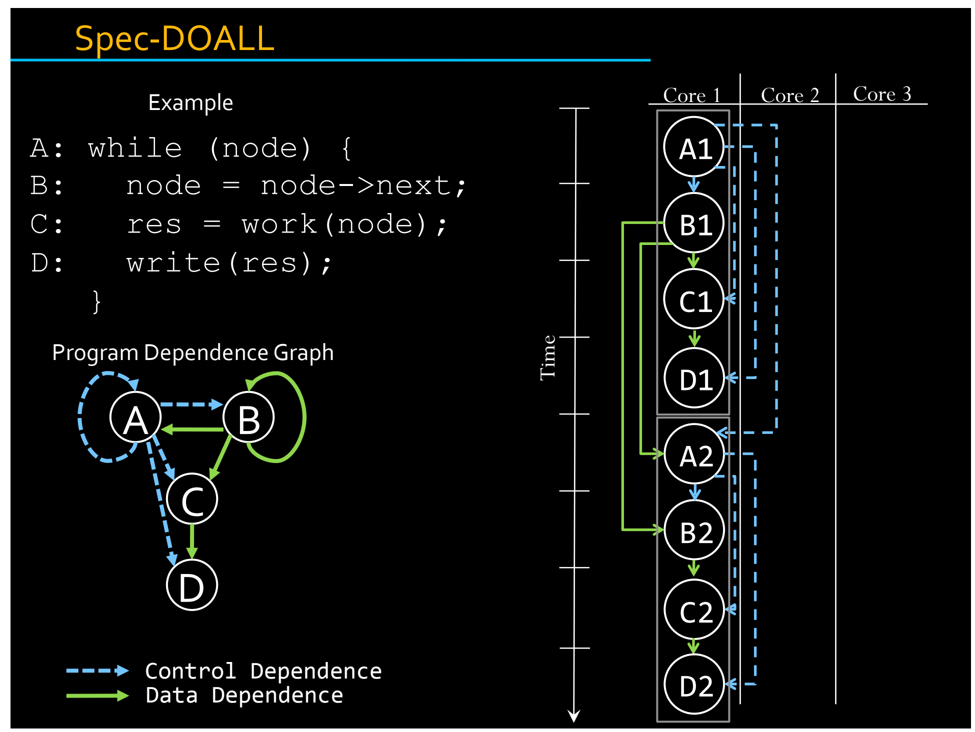
pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

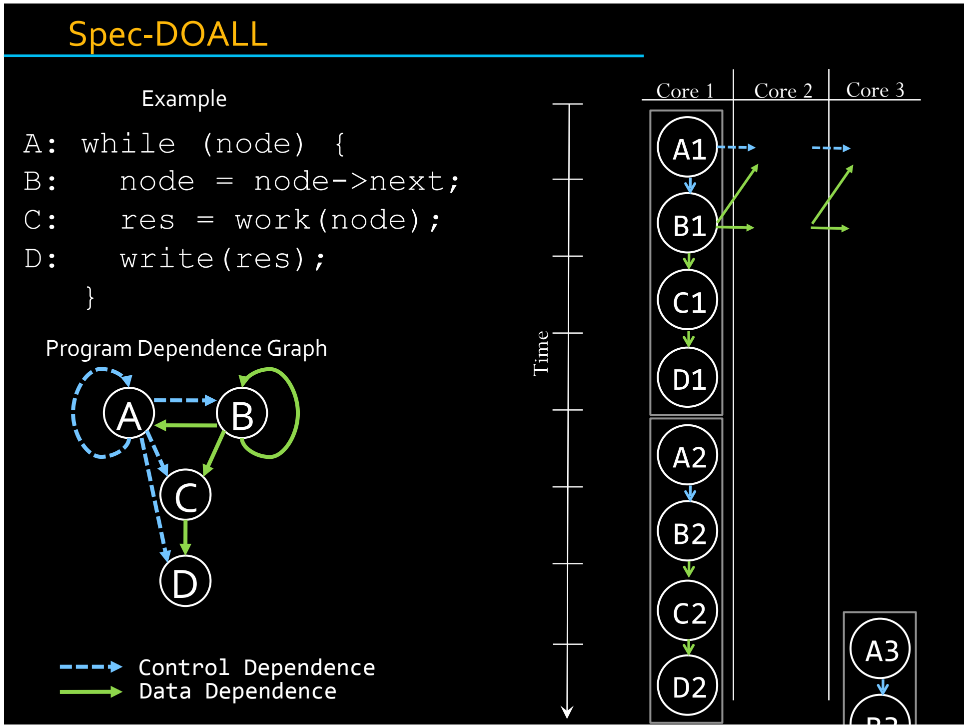


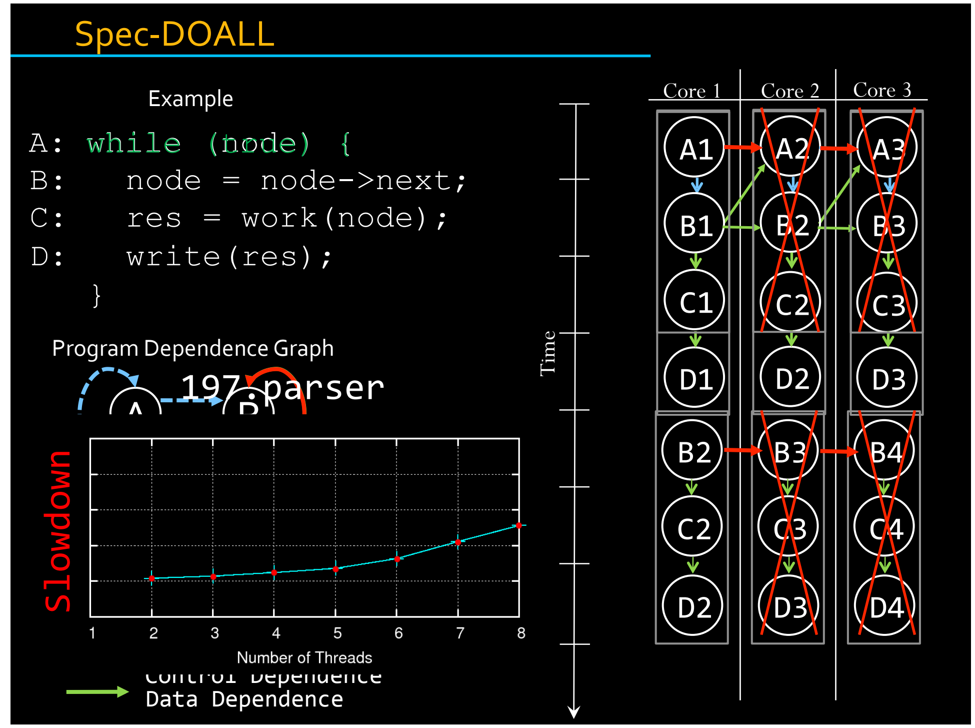

"Compiler Advances Double Computing Power Every 18 Years!"
– Proebsting's Law
SPEC CPU Integer Performance (logarithmic scale)
CPU92 CPU95 CPU2000 CPU2006
1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012
Year

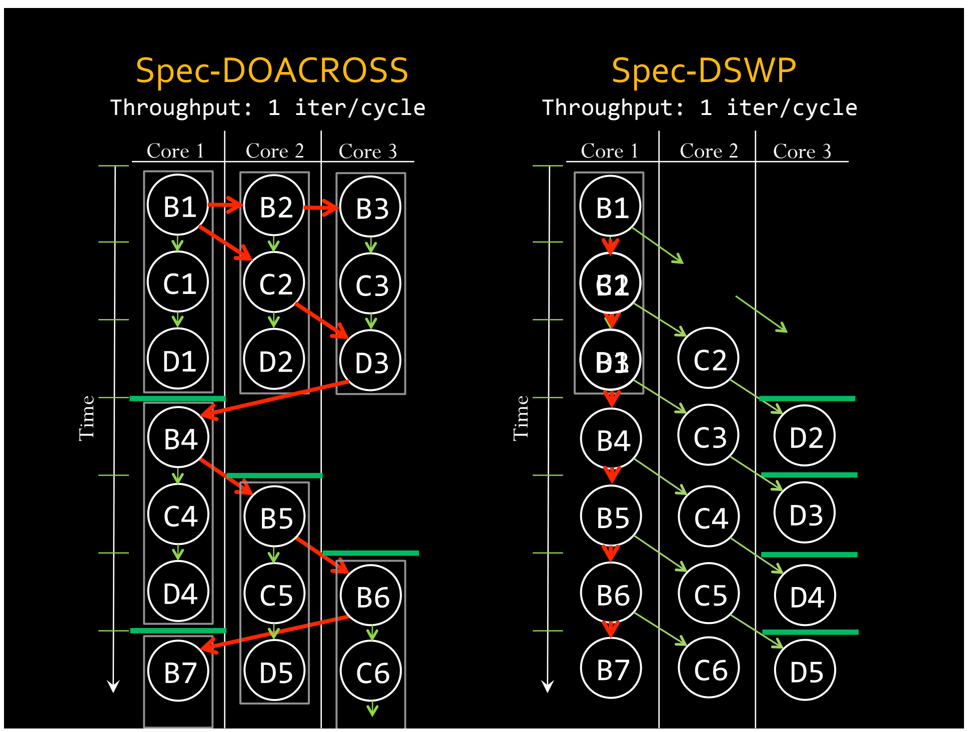

P6 SUPERSCALAR ARCHITECTURE
System Bus (External)
L2 Cache
Cache Bus
Bus Interface Unit
Instruction Fetch Unit
Instruction Cache (L1)
Next IP Unit
Branch Target Buffer
Memory Reorder Buffer
Instruction Decoder
Simple Instuction Decoder
Simple Instuction Decoder
Complex Instuction Decoder
Microcode Instruction Sequencer
From Integer Unit
Register Alias Table
Retirement Unit
Reorder Buffer (Instruction Pool)
Retirement Register File (Intel Arch. Registers)
Data Cache Unit (L1)
Reservation Station
Execution Unit
SIMD FP Unit (FPU)
Floating-Point Unit (FPU)
Integer Unit
Integer Unit
Memory Interface Unit
Internal Data-Results Buses
Spec-PS-DSWP
Core 1
Core 2
Core 3
Core 4
0
1
2
3
4
5
LD:1
LD:2
LD:3
LD:4
LD:5
W:1
W:2
W:3
W:4
C:1
C:2
C:3

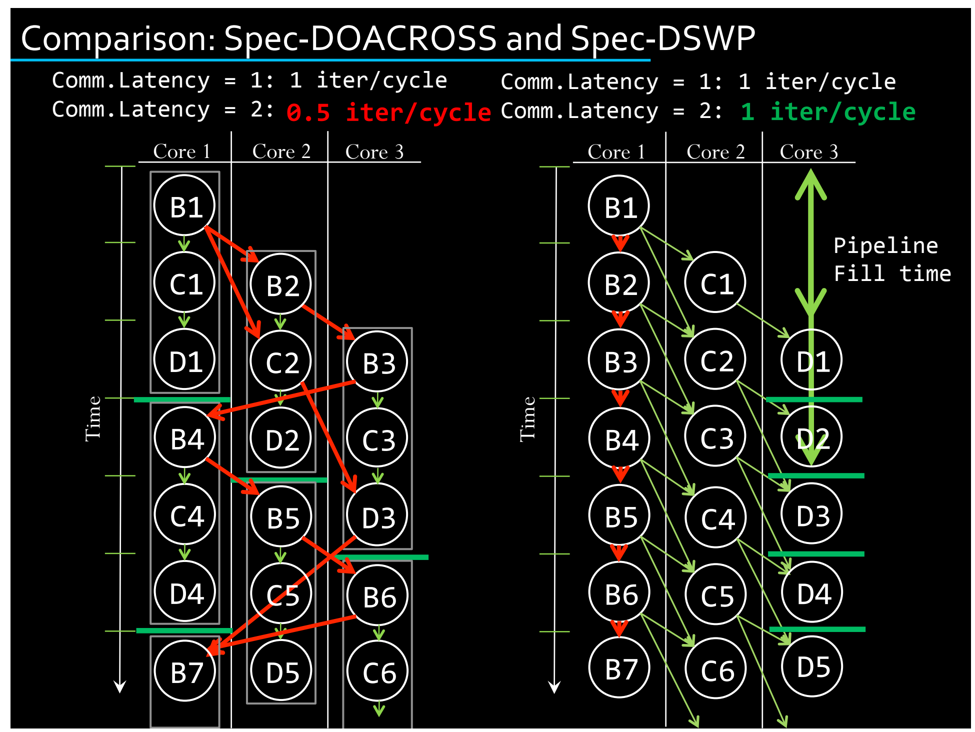

Example
A: while (node) {
B: node = node->next;
C: res = work(node);
D: write(res);
}
Program Dependence Graph
A
B
C
D
Control Dependence
Data Dependence
Time
Core 1
Core 2
Core 3
A1
B1
C1
D1
A2
B2
C2
D2

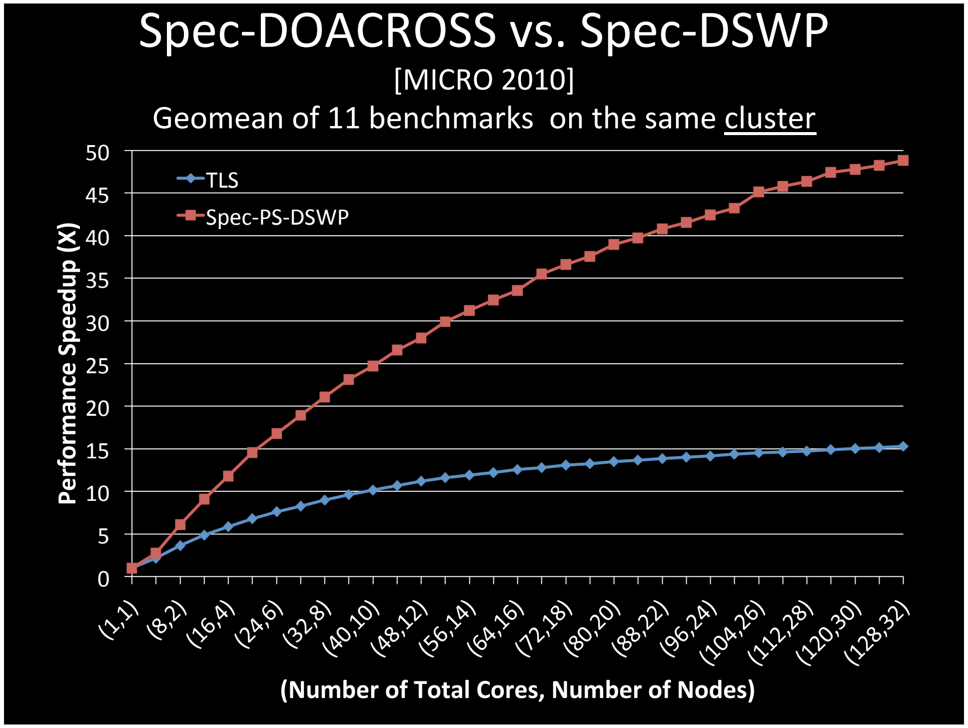

Spec-DOALL
Example
A: while (node) {
B: node = node->next;
C: res = work(node);
D: write(res);
}
Program Dependence Graph
A
B
C
D
Control Dependence
Data Dependence
Time
Core 1
Core 2
Core 3
A1
B1
C1
D1
A2
B2
C2
D2

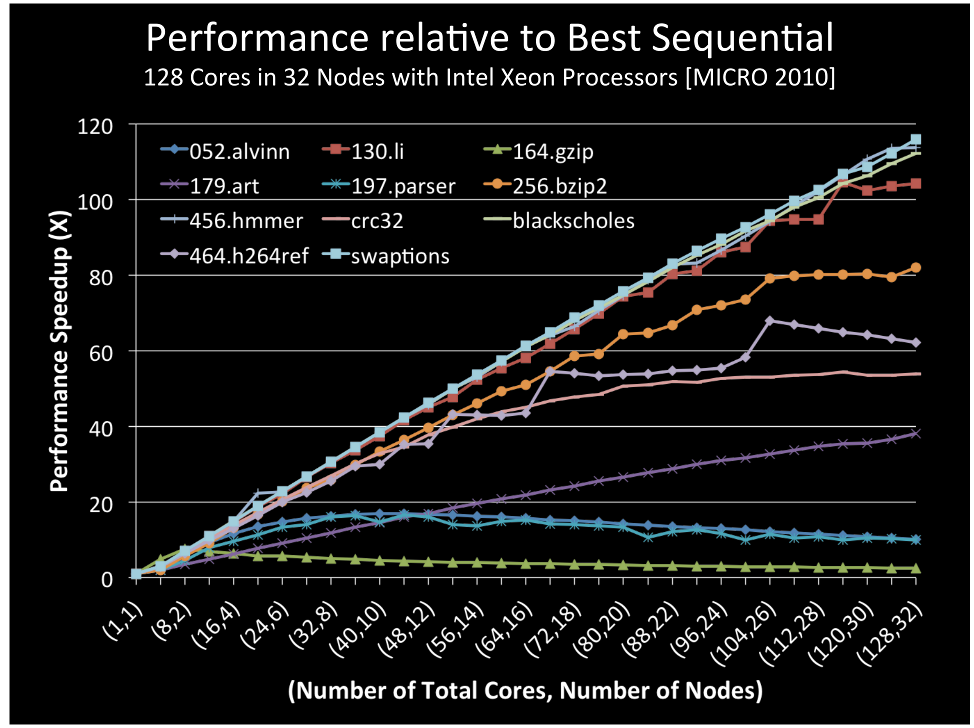

Spec-DOALL
Example
A: while (node) {
B: node = node->next;
C: res = work(node);
D: write(res);
}
Program Dependence Graph
A
B
C
D
Control Dependence
Data Dependence
Time
Core 1
Core 2
Core 3
A1
B1
C1
D1
A2
B2
C2
D2
A3

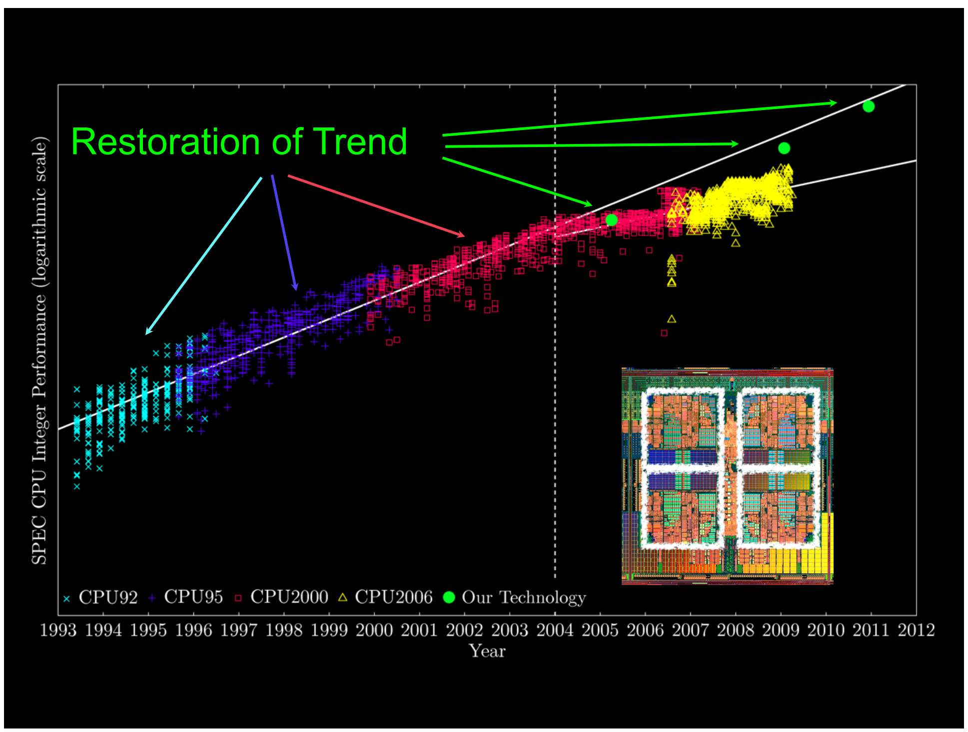

Spec-DOALL
Example
A: while (node) {
B: node = node->next;
C: res = work(node);
D: write(res);
}
Program Dependence Graph
197.parser
Slowdown
1
2
3
4
5
6
7
8
Number of Threads
Data Dependence
Time
Core 1
Core 2
Core 3
A1
B1
C1
D1
A2
B2
C2
D2
A3
B3
C3
D3
B4
C4
D4

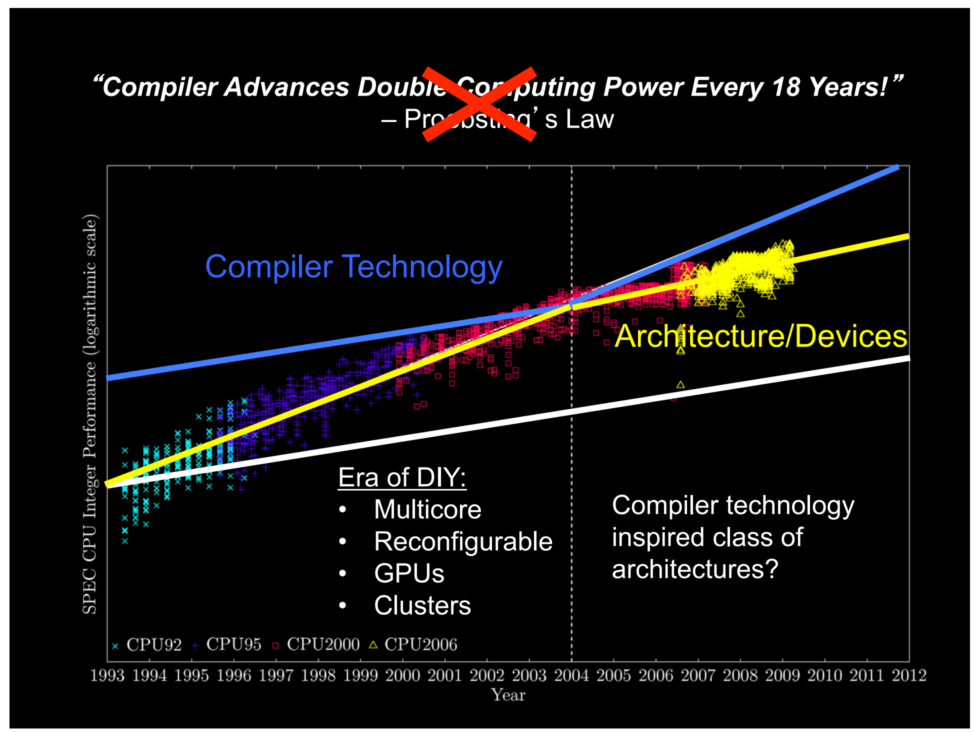

Spec-DOACROSS
Throughput: 1 iter/cycle
Spec-DSWP
Throughput: 1 iter/cycle
Core 1
Core 2
Core 3
Time
B1
C1
D1
B2
C2
D2
B3
C3
D3
B4
C4
D4
B5
C5
D5
B6
C6
B7
D4
D5
C6


Comparison: Spec-DOACROSS and Spec-DSWP
Comm.Latency = 1: 1 iter/cycle
Comm.Latency = 2: 0.5 iter/cycle
Comm.Latency = 1: 1 iter/cycle
Comm.Latency = 2: 1 iter/cycle
Core 1
Core 2
Core 3
Time
Pipeline
Fill time
B1
C1
D1
B2
C2
D2
B3
C3
D3
B4
C4
D4
B5
C5
D5
B6
C6
B7


Spec-DOACROSS vs. Spec-DSWP
[MICRO 2010]
Geomean of 11 benchmarks on the same cluster
TLS
Spec-PS-DSWP
Performance Speedup (X)
0
5
10
15
20
25
30
35
40
45
50
(1,1)
(8,2)
(16,4)
(24,6)
(32,8)
(40,10)
(48,12)
(56,14)
(64,16)
(72,18)
(80,20)
(88,22)
(96,24)
(104,26)
(112,28)
(120,30)
(128,32)
(Number of Total Cores, Number of Nodes)


Performance relative to Best Sequential
128 Cores in 32 Nodes with Intel Xeon Processors [MICRO 2010]
052.alvinn
130.li
164.gzip
179.art
197.parser
256.bzip2
456.hmmer
crc32
blackscholes
464.h264ref
swaptions
Performance Speedup (X)
0
20
40
60
80
100
120
(1,1)
(8,2)
(16,4)
(24,6)
(32,8)
(40,10)
(48,12)
(56,14)
(64,16)
(72,18)
(80,20)
(88,22)
(96,24)
(104,26)
(112,28)
(120,30)
(128,32)
(Number of Total Cores, Number of Nodes)


Restoration of Trend
SPEC CPU Integer Performance (logarithmic scale)
CPU92
CPU95
CPU2000
CPU2006
Our Technology
1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012
Year


"Compiler Advances Double Computing Power Every 18 Years!"
– Proebsting's Law
Compiler Technology
Architecture/Devices
SPEC CPU Integer Performance (logarithmic scale)
Era of DIY:
• Multicore
• Reconfigurable
• GPUs
• Clusters
Compiler technology
inspired class of
architectures?
CPU92
CPU95
CPU2000
CPU2006
1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012
Year

CFGs and PCs

93

Dependence Graphs and Parallelism

94

95

96