Final Exam!

- Thursday May 3 in class
- Closed book, closed notes

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

❖ “Compiler Advances Double Computing Power Every 18 Years”
❖ Run your favorite set of benchmarks with your favorite state-of-the-art optimizing compiler. Run the benchmarks both with and without optimizations enabled. The ratio of those numbers represents the entirety of the contribution of compiler optimizations to speeding up those benchmarks. Let’s assume that this ratio is about 4X for typical real-world applications, and let’s further assume that compiler optimization work has been going on for about 36 years. Therefore, compiler optimization advances double computing power every 18 years. QED.

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
```

Example: DOALL parallelization

```
Core 1
C:1
X:1
C:2
X:2
C:4
X:4
C:3
X:3
C:5
X:5
C:6
X:6

Core 2
```

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
Affine/Linear Functions

- **f(i_1, i_2, ..., 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

```c
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
  \[ 1 \leq i \leq 10, 1 \leq i' \leq 10 \text{ and } i - 1 = i' \]
- For 2, we solve for \( i \) and \( i' \) in
  \[ 1 \leq i \leq 10, 1 \leq i' \leq 10, i = i' \text{ and } i \neq i' \] (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 dependence 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**

---

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++) {
       \[\ldots = *p;\]
       \[*q = \ldots\]
   }

Memory dependence profiling and speculative parallelization

DOALL Coverage – Provable and Profiled

Still not good enough!
2. Data dependences

```c
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;
}
```

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;
}

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
     Ŷ When to create a new dictionary in a compression scheme
**Sequential Program**

```
#define CUTOFF 100

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

**Parallel Program**

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

**2-Core Parallel Program**

```
dict = create_dict();
while((char = read(1))) {
    profitable = compress(char, dict)
    @YBRANCH(probability=.01)
    if (!profitable) {
        dict = restart(dict);
    }
} finish_dict(dict);
```

**64-Core Parallel Program**

```
dict = create_dict();
while((char = read(1))) {
    profitable = compress(char, dict)
    @YBRANCH(probability=.00001)
    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.gz**

```
#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++;
} finish_dict(dict);
```
Parallelization techniques must look inside function calls to expose operations that cause synchronization.

**197.parser**

**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.

---

<table>
<thead>
<tr>
<th>164.gzip</th>
<th>26</th>
<th>x</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>173.vpr</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>176.gcc</td>
<td>18</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>181.mcf</td>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>186.crafty</td>
<td>9</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>197.parser</td>
<td>3</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>0</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>254.gap</td>
<td>3</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>255.vortex</td>
<td>0</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>300.twolf</td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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)

\[
\text{for}(i=1; \ i<\text{N}; \ i++) \ // \ C \\
a[i] = a[i] + 1; \ // \ X
\]

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

\[
\text{while}(\text{ptr} = \text{ptr->next}) \ // \ LD \\
\text{ptr->val} = \text{ptr->val} + 1; \ // \ X
\]

Alternative Parallelization Approaches
Comparison: IMT, PMT, CMT

IMT

PMT

CMT

Comparison: IMT, PMT, CMT

IMT

PMT

CMT

Comparison: IMT, PMT, CMT

IMT

PMT

CMT

Comparison: IMT, PMT, CMT

IMT

PMT

CMT

Comparison: IMT, PMT, CMT

IMT

PMT

CMT
Comparison: IMT, PMT, CMT

Thread-local Recurrences ➔ Fast Execution

Core 1 Core 2

IMT

0 1 2 3 4 5
C:1 X:1 C:2 X:2 C:4 X:4 C:3 X:3 C:5 X:5 C:6 X:6

PMT

0 1 2 3 4 5

CMT

0 1 2 3 4 5
LD:1 X:1 LD:2 X:2 LD:3 X:3

Cross-thread Dependences ➔ Wide Applicability

Our Objective: Automatic Extraction of Pipeline Parallelism using DSWP

Decoupled Software Pipelining

Find English Sentences ➔ Parse Sentences (95%) ➔ Emit Results

PS-DSWP (Spec DOALL Middle Stage)
Decoupled Software Pipelining (DSWP)

```
while(node)
   ncost = doit(node);
   cost += ncost;
   node = node->next;
```

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
```

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;

Dependence Graph

DAG

SCC

A  D
B
C

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

Predictable Dependences

Why Speculation?

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

Dependence Graph

DAG

SCC

A  D
B
C

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

Predictable Dependences

Why Speculation?

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

Dependence Graph

DAG

SCC

A  D
B
C

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

Predictable Dependences
**Execution Paradigm**

![DAG and SCC diagram]

**Understanding PMT Performance**

![Core and thread diagram]

\[ T \propto \max(t_i) \]

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 and DAG diagram]
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
A4: while(cost < T && node)
B4: cost = consume(3);
C4: node = consume(1);
   produce({4,5,6},cost < T && node);
   if(!(cost < T && node))
   FLAG_MISSPEC();
Breaking False Memory Dependences

Adding Parallel Stages to DSWP

Thread Partitioning
Thread Partitioning: \( \text{DAG}_{\text{SCC}} \)

Thread Partitioning

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

Treated as sequential
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?
1. \( r_1 = 10 \)

1. \( r_1 = r_1 + 1 \)
2. \( r_2 = \text{MEM}[r_1] \)
3. \( r_2 = r_2 + 1 \)
4. \( \text{MEM}[r_1] = r_2 \)
5. Branch \( r_1 < 1000 \)

No register live outs

Loop-Level Parallelization: DOALL

1. \( r_1 = 10 \)

1. \( r_1 = r_1 + 1 \)
2. \( r_2 = \text{MEM}[r_1] \)
3. \( r_2 = r_2 + 1 \)
4. \( \text{MEM}[r_1] = r_2 \)
5. Branch \( r_1 < 1000 \)

1. \( r_1 = 9 \)

1. \( r_1 = r_1 + 2 \)
2. \( r_2 = \text{MEM}[r_1] \)
3. \( r_2 = r_2 + 1 \)
4. \( \text{MEM}[r_1] = r_2 \)
5. Branch \( r_1 < 999 \)

1. \( r_1 = 10 \)

1. \( r_1 = r_1 + 2 \)
2. \( r_2 = \text{MEM}[r_1] \)
3. \( r_2 = r_2 + 1 \)
4. \( \text{MEM}[r_1] = r_2 \)
5. Branch \( r_1 < 1000 \)

No register live outs

Another Example

1. \( r_1 = 10 \)

1. \( r_1 = r_1 + 1 \)
2. \( r_2 = \text{MEM}[r_1] \)
3. \( r_2 = r_2 + 1 \)
4. \( \text{MEM}[r_1] = r_2 \)
5. Branch \( r_2 == 10 \)

No register live outs
### Another Example

<table>
<thead>
<tr>
<th>r1 = 10</th>
<th>r1 = 9</th>
<th>r1 = 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = r1 + 1</td>
<td>r1 = r1 + 2</td>
<td>r1 = r1 + 2</td>
</tr>
<tr>
<td>r2 = MEM[r1]</td>
<td>r2 = MEM[r1]</td>
<td>r2 = MEM[r1]</td>
</tr>
<tr>
<td>r2 = r2 + 1</td>
<td>r2 = r2 + 1</td>
<td>r2 = r2 + 1</td>
</tr>
<tr>
<td>MEM[r1] = r2</td>
<td>MEM[r1] = r2</td>
<td>MEM[r1] = r2</td>
</tr>
<tr>
<td>Branch r2 == 10</td>
<td>Branch r2 == 10</td>
<td>Branch r2 == 10</td>
</tr>
</tbody>
</table>

No register live outs

### Speculation

<table>
<thead>
<tr>
<th>r1 = 9</th>
<th>r1 = 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = r1 + 2</td>
<td>r1 = r1 + 2</td>
</tr>
<tr>
<td>r2 = MEM[r1]</td>
<td>r2 = MEM[r1]</td>
</tr>
<tr>
<td>r2 = r2 + 1</td>
<td>r2 = r2 + 1</td>
</tr>
<tr>
<td>MEM[r1] = r2</td>
<td>MEM[r1] = r2</td>
</tr>
<tr>
<td>Branch r2 == 10</td>
<td>Branch r2 == 10</td>
</tr>
</tbody>
</table>

No register live outs

### Speculation, Commit, and Recovery

<table>
<thead>
<tr>
<th>r1 = 9</th>
<th>r2 = Receive{1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = r1 + 2</td>
<td>Branch r2 != 10</td>
</tr>
<tr>
<td>r2 = MEM[r1]</td>
<td>MEM[r1] = r2</td>
</tr>
<tr>
<td>r2 = r2 + 1</td>
<td>Branch r2 != 10</td>
</tr>
<tr>
<td>MEM[r1] = r2</td>
<td>MEM[r1] = r2</td>
</tr>
<tr>
<td>Send{1} r2</td>
<td>Branch r2 != 10</td>
</tr>
<tr>
<td>Jump</td>
<td>MEM[r1] = r2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r1 = 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = r1 + 2</td>
</tr>
<tr>
<td>r2 = r2 + 1</td>
</tr>
<tr>
<td>MEM[r1] = r2</td>
</tr>
<tr>
<td>Send{2} r2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r1 = 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = r1 + 2</td>
</tr>
<tr>
<td>r2 = r2 + 1</td>
</tr>
<tr>
<td>MEM[r1] = r2</td>
</tr>
<tr>
<td>Send{2} r2</td>
</tr>
</tbody>
</table>

| 1. Kill and Continue |

No register live outs
Difficult Dependences

1. r1 = Head

1. r1 = MEM[r1]
1. Branch r1 == 0
2. r2 = MEM[r1 + 4]
1. r3 = Work (r2)
2. Print ( r3 )
3. Jump

No register live outs

DOACROSS

1. r1 = Head

1. r1 = MEM[r1]
1. Branch r1 == 0
2. r2 = MEM[r1 + 4]
1. r3 = Work (r2)
2. Print ( r3 )
3. Jump

No register live outs

PS-DSWP

1. r1 = Head

1. r1 = MEM[r1]
1. Branch r1 == 0
2. r2 = MEM[r1 + 4]
1. r3 = Work (r2)
2. Print ( r3 )
3. Jump

No register live outs
Era of DIY:
- Multicore
- Reconfigurable
- GPUs
- Clusters

P6 SUPERSCALAR ARCHITECTURE (CIRCA 1994)
- Automatic Speculation
- Automatic Pipelining
- Parallel Resources
- Automatic Allocation/Scheduling

MULTICORE ARCHITECTURE (CIRCA 2010)
- Automatic Speculation
- Automatic Pipelining
- Parallel Resources
- Automatic Allocation/Scheduling
### Parallel Library Calls

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time</th>
</tr>
</thead>
</table>

### Realizable parallelism

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time</th>
</tr>
</thead>
</table>

### Credit: Jack Dongarra

---

**"Compiler Advances Double Computing Power Every 18 Years!"**

-- Proebsting’s Law

---

### "Compiler Advances Double Computing Power Every 18 Years!" -- Proebsting’s Law

- **CPU**
- **CPU95**
- **CPU2000**
- **CPU2006**

<table>
<thead>
<tr>
<th>Year</th>
<th>CPU</th>
<th>CPU95</th>
<th>CPU2000</th>
<th>CPU2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>1993</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1994</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1995</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1996</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1997</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2002</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2012</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Credit: Jack Dongarra**
Example
A: while (node) {
B:   node = node->next;
C:   res = work(node);
D:   write(res);
}

Program Dependence Graph

Spec-DOALL

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

Program Dependence Graph
Example
A: while (node) {
B:   node = node->next;
C:   res = work(node);
D:   write(res);
}

Program Dependence Graph

Spec-DOALL

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

Program Dependence Graph

Slowdown

Spec-DOACROSS

Throughput: 1 iter/cycle

Spec-DSWP

Throughput: 1 iter/cycle
Restoration of Trend

“Compiler Advances Double Computing Power Every 18 Years!”
-- Proebsting's Law

Era of DIY:
- Multicore
- Reconfigurable
- GPUs
- Clusters

Compiler technology inspired class of architectures?

CFGs and PCs