EECS 583 – Class 16
Research Topic 1
Automatic Parallelization

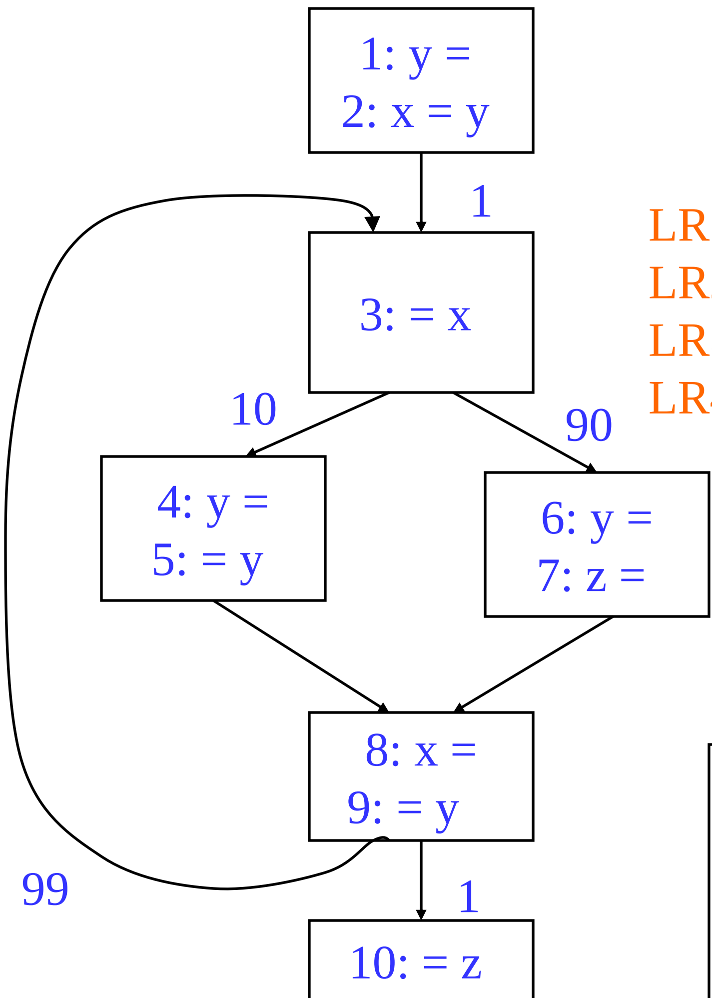
University of Michigan

November 7, 2011

Announcements + Reading Material

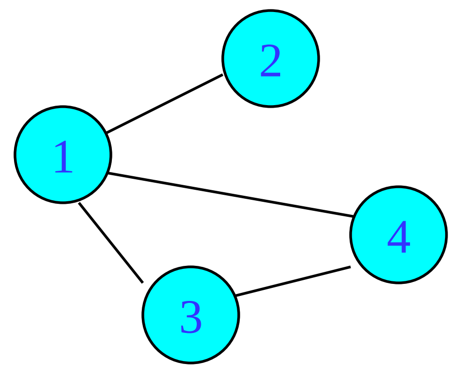
- Midterm exam: Mon Nov 14 in class (Next Monday)
  - I will post 2 practice exams by tonight!
  - We'll talk more about the exam next class
- 1st paper review due today!
  - Copy file to andrew.eecs.umich.edu:/y/submit
  - Put uniquename_classXX.txt
- Today's class reading
  - "Revisiting the Sequential Programming Model for Multi-Core," M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August, *Proc 40th IEEE/ACM International Symposium on Microarchitecture*, December 2007.
- Next class reading
  - "Automatic Thread Extraction with Decoupled Software Pipelining," G. Ottoni, R. Rangan, A. Stoler, and D. I. August, *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, Nov. 2005.

Class Problem from Last Time – Answer



do a 2-coloring
- compute cost matrix
- draw interference graph
- color graph

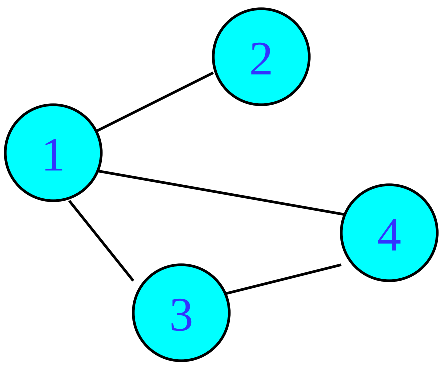
LR1(x) = {2,3,4,5,6,7,8,9}
LR2(y) = {1,2}
LR3(y) = {4,5,6,7,8,9}
LR4(z) = {3,4,5,6,7,8,9,10}



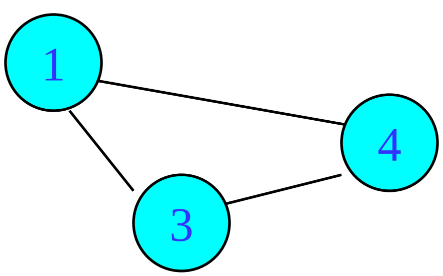
Interference graph

	1	2	3	4
cost	201	2	210	91
nbors	3	1	2	2
c/n	67	2	105	45.5

Class Problem Answer (continued)

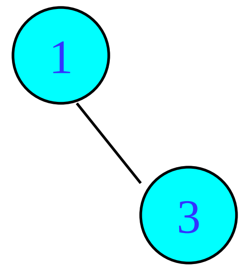


1. Remove all nodes degree < 2, remove node 2



stack
2

2. Cannot remove any nodes, so choose node 4 to spill



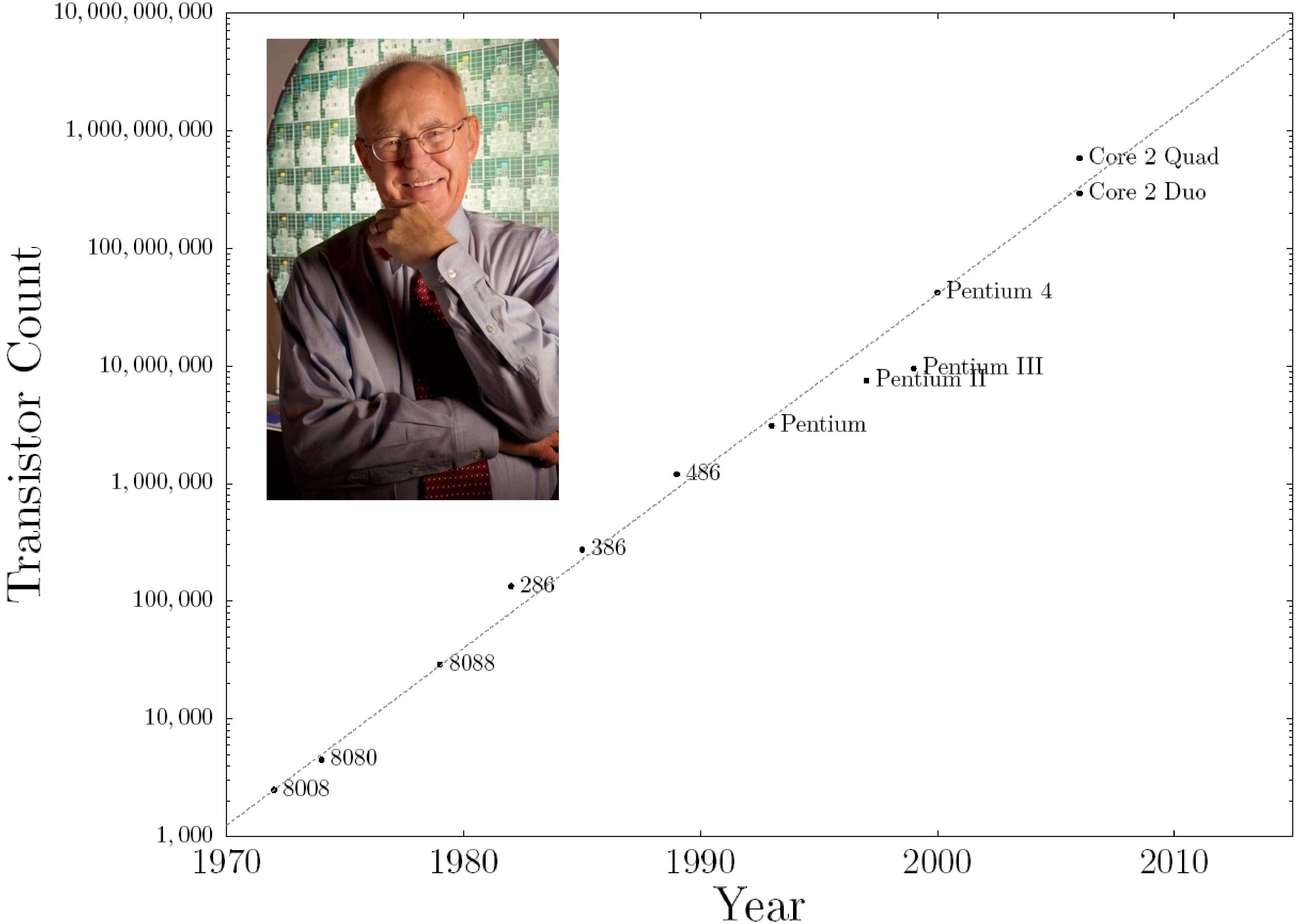
stack
4 (spill)
2

3. Remove all nodes degree < 2, remove 1 and 3

stack
1
3
4 (spill)
2

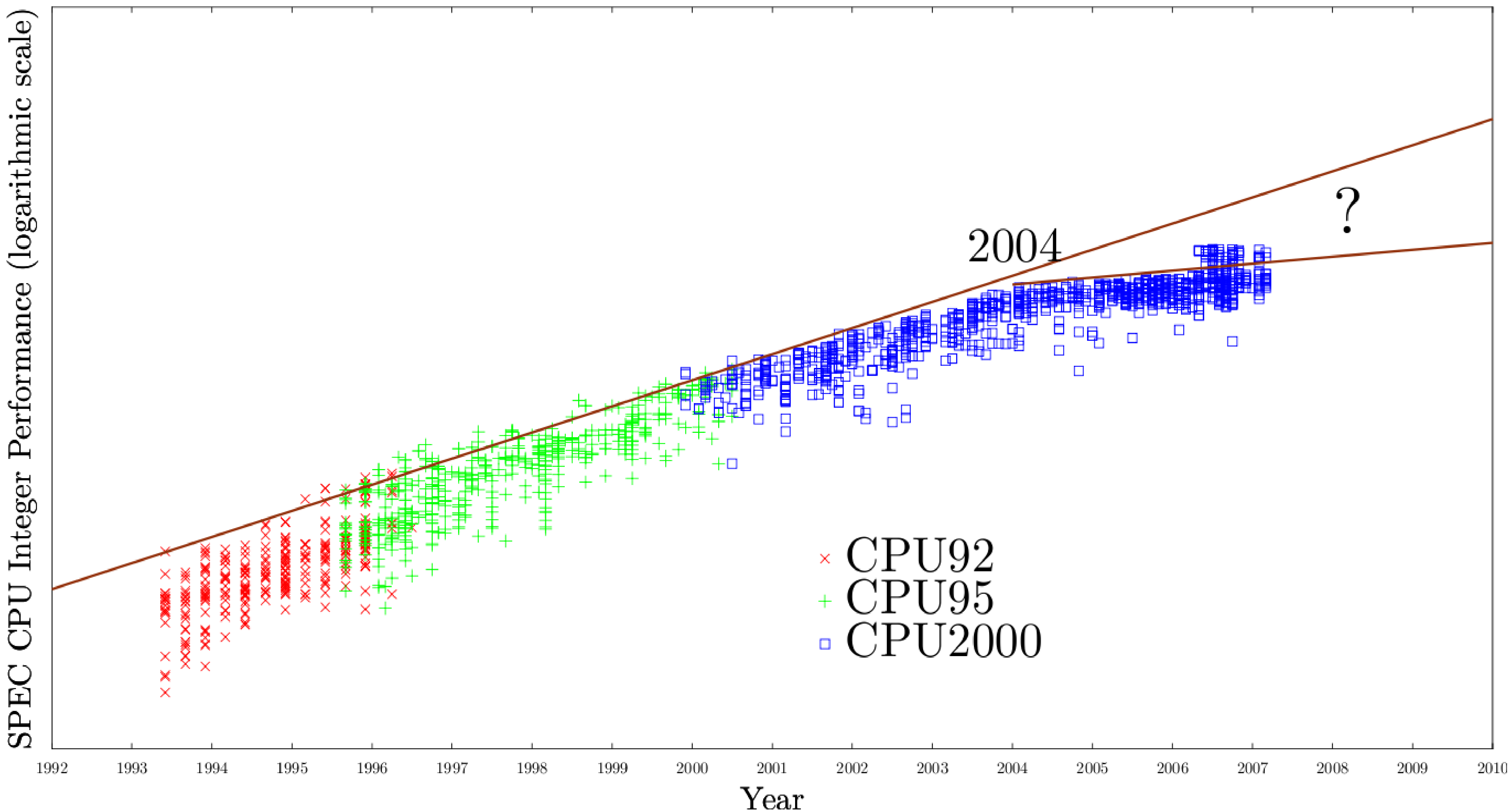
4. Assign colors: 1 = red, 3 = blue, 4 = spill, 2 = blue

Moore’s Law



Source: Intel/Wikipedia

Single-Threaded Performance Not Improving



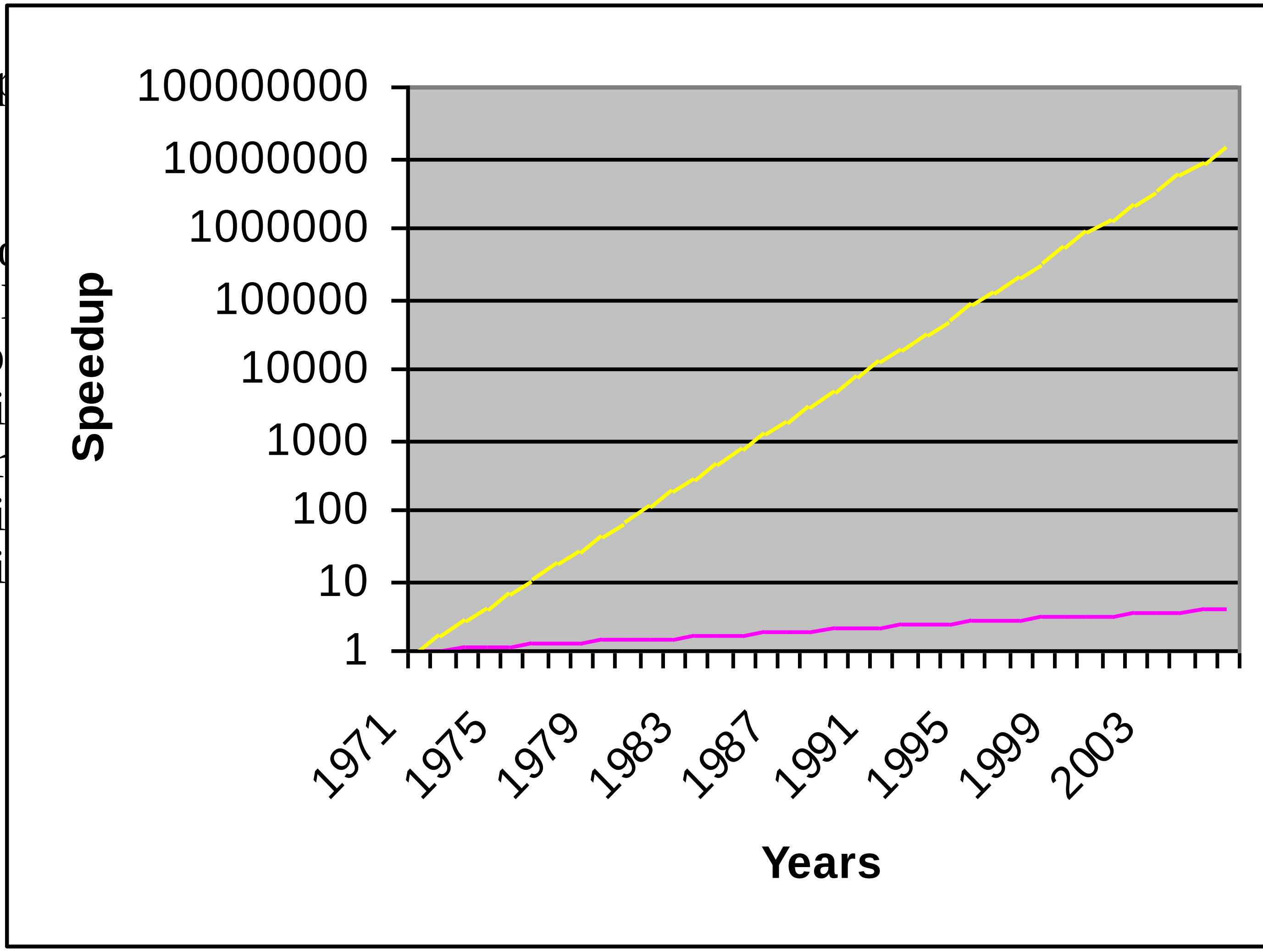
What about Parallel Programming? –or- What is Good About the Sequential Model?

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

Compilers are the Answer? - Proebsting's Law

- "Comp

- Run yo ... optimizing compil ... bled. The ratio o ... ompiler optimi ... atio is about 4X for ... piler optimi ... npiler optimi



Conclusion – Compilers not about performance!

What Do the Experts Say?

"That isn't to say we are parallelizing arbitrary C code, that's a fool's errand!" – **Richard Lethin, Reservoir Labs**

"Compiler can't determine a tree from a graph…" **– Burton Smith, MSR**

"Compilers can't determine dependences without type information. Even then…" **– Burton Smith**

"Decades of automatic parallelization work has been a failure…" **– James Larus, MSR**

"All that icky pointer chasing code…"

– Tim Mattson, Intel

Are We Doomed?

A Step Back in Time: Old Skool Parallelization

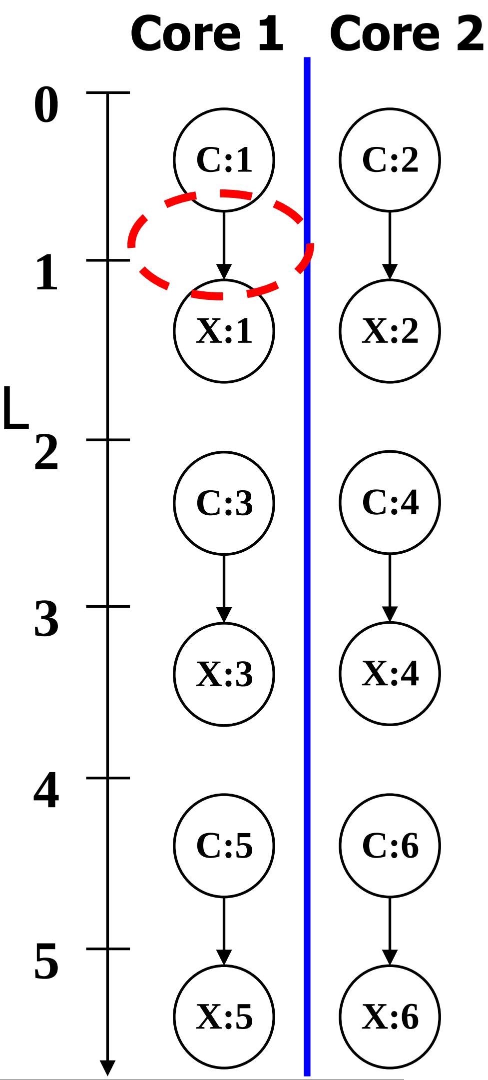
Parallelizing Loops In Scientific Applications

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 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

Iteration Space

- Iteration space is the set of iterations, whose ID's are given by the values held by the loop index variables.

```
for (i = 2; i <= 100;  i= i+3)
        Z[i] = 0;
```

  IS = {2, 5, 8, 11, … , 98} – the set contains the value of the loop index *i* at each iteration of the loop.

- The iteration space can be normalized. Prior loop is:

  for (i^n = 0; i^n <= 32; i^n ++)

  Z[2 + 3* i^n] = 0;

In general, $i^n = (i – lowerBound) / i_{step}$

Iteration Space (continued)

- How about nested loops?

```
for (i = 3; i <= 7;  i++)
  for (j = 6; j >= 2; j = j – 2 )
       Z[i, j] = Z[i, j+2] + 1
```

The iteration space is given by the set of vectors:
{[3,6], [3,4], [3,2], [4,6], [4,4], [4,2], [5,6], [5,4], [5,2], [6,6], [6,4], [6,2], [7,6], [7,4], [7,2]}

Question: Rewrite the loop using normalized iteration vectors?

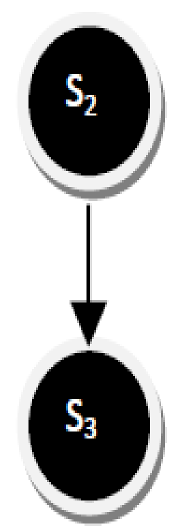
- Normalized form

```
for (i = 0; i <= 4;  i++)
  for (j = 0; j <= 2; j++ )
       Z[3 + i, 6 - j*2] = Z[3 + i, 6 - j*2+2] + 1
```

Dependence Graph

- 3 dependence types
  - » Flow dependence (true dependence)
    - A variable assigned in one statement is used subsequently in another statement.
  - » Anti-dependence
    - A variable is used in one statement and reassigned in a subsequently executed statement.
  - » Output dependence
    - A variable is assigned in one statement and subsequently reassigned in another statement.

- Graph can be drawn to show data dependence between statements within a loop.



S_1: for (i = 2; i<= 5; ++i){

S_2: X[i] = Y[i] + Z[i]

S_3: A[i] = X[i-1] + 1

}

i=2 → i=3 → i=4 → i=5

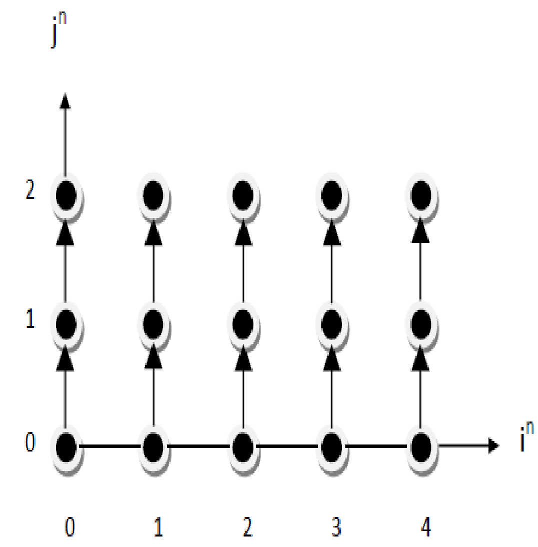
S_2: X[2] X[3] X[4] X[5]

S_3: X[1] ↘ X[2] ↘ X[3] ↘ X[4]

Iteration Space Dependence Graph

```
for (i = 3; i <= 7;  i++)
    for (j = 6; j >= 2; j = j – 2 )
        Z[i, j] = Z[i, j+2] + 1
```

- Iteration space dependence graph (normalized)



Array Dependence Analysis

- Consider two static accesses A in a *d*-deep loop nest and A' in a *d'*-deep loop nest respectively defined as

  $$A = <\mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b}> \text{ and } A' = <\mathbf{F}', \mathbf{f}', \mathbf{B}', \mathbf{b}'>$$

- A and A' are data dependent if
  - $\mathbf{Bi} \geq \mathbf{0}$; $\mathbf{B'i'} \geq \mathbf{0}$ and
  - $\mathbf{Fi} + \mathbf{f} = \mathbf{F'i'} + \mathbf{f'}$
  - (and $\mathbf{i} \neq \mathbf{i'}$ for dependencies between instances of the same static access)

Array Dependence Analysis (continued)

```
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

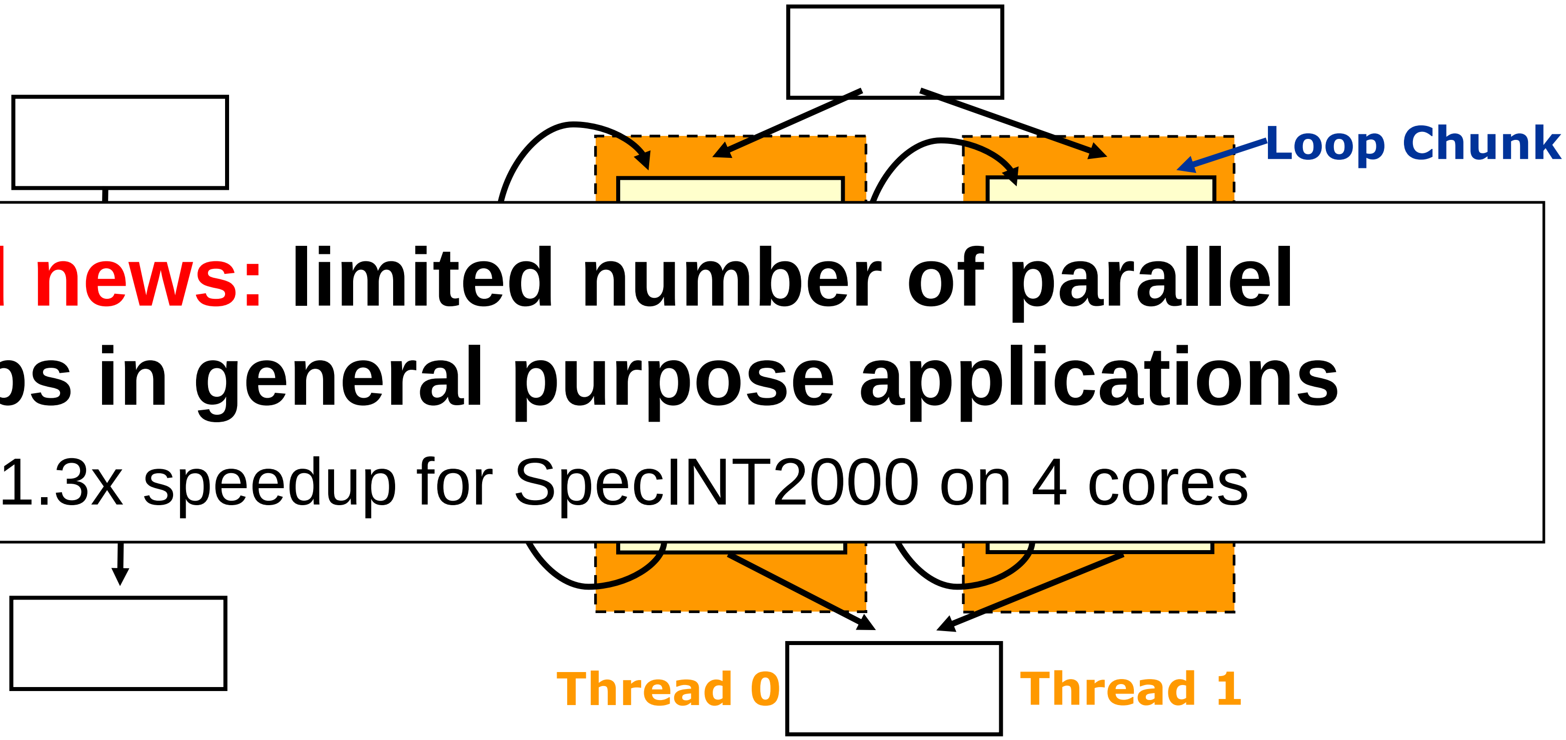
- Array data dependence basically requires finding integer solutions to a system (often refers to as dependence system) consisting of equalities and inequalities.
- Equalities are derived from array accesses.
- Inequalities from the loop bounds.
- It is an integer linear programming problem.
- ILP is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

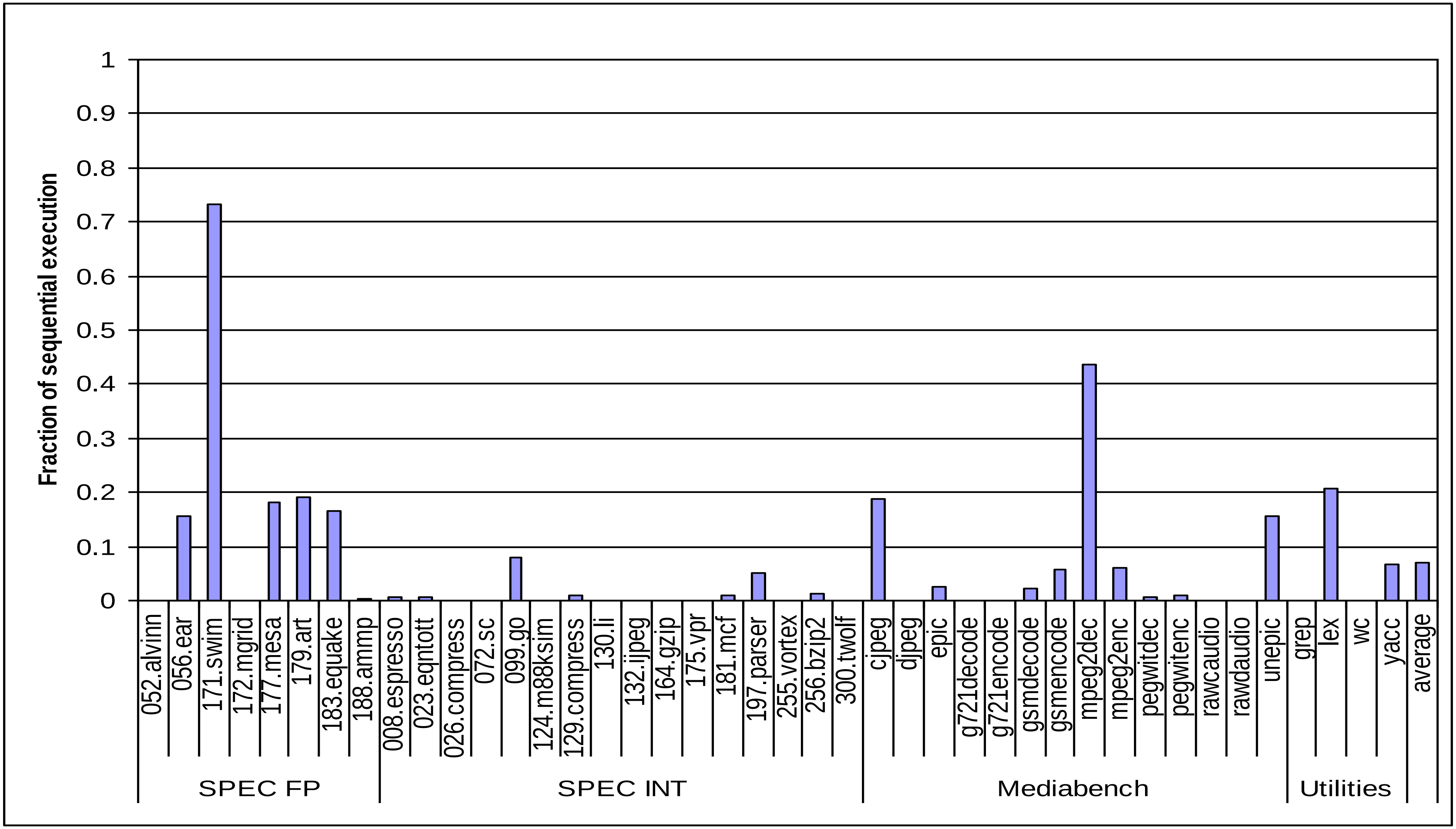
Loop Level Parallelization



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

– 1.3x speedup for SpecINT2000 on 4 cores

DOALL Loop Coverage



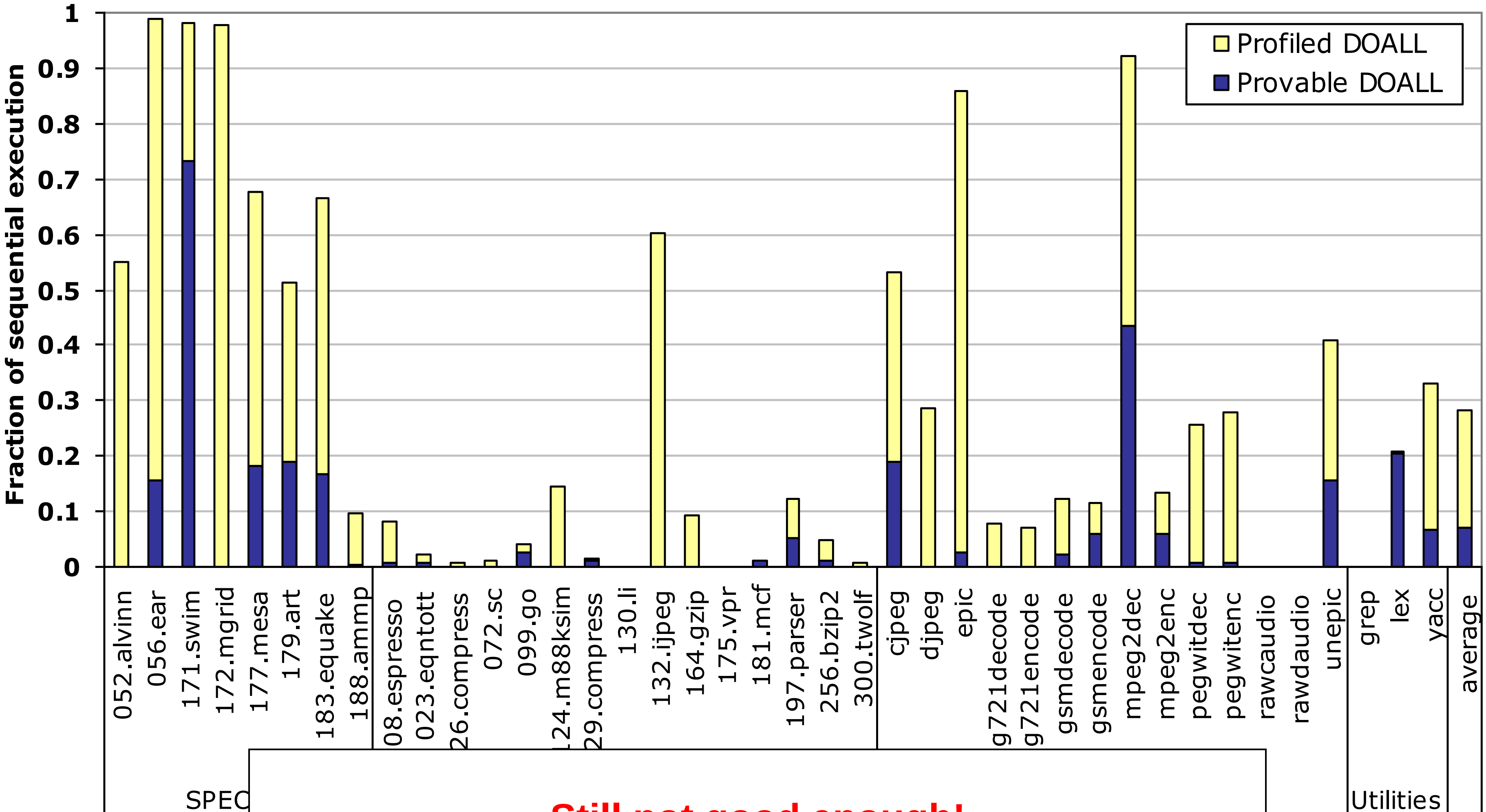
What's the Problem?

1. Memory dependence analysis

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

      . . . = *p;


      *q = . . .


}
```

Memory dependence profiling and speculative parallelization

DOALL Coverage – Provable and Profiled



Still not good enough!

What's the Next Problem?

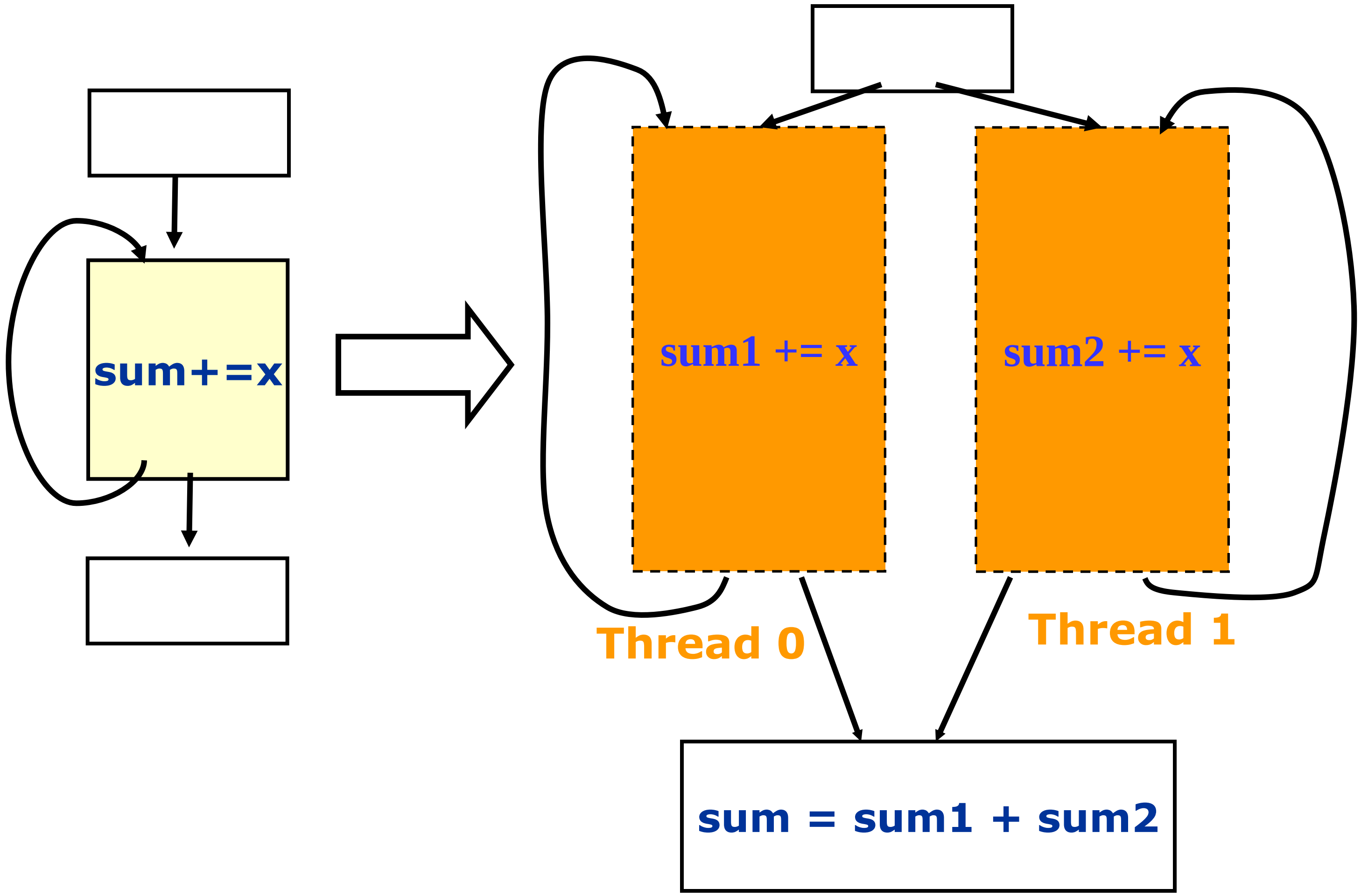
2. Data dependences

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

➡ Compiler transformations

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

Apply accumulator variable expansion!



Data Dependences Inhibit Parallelization

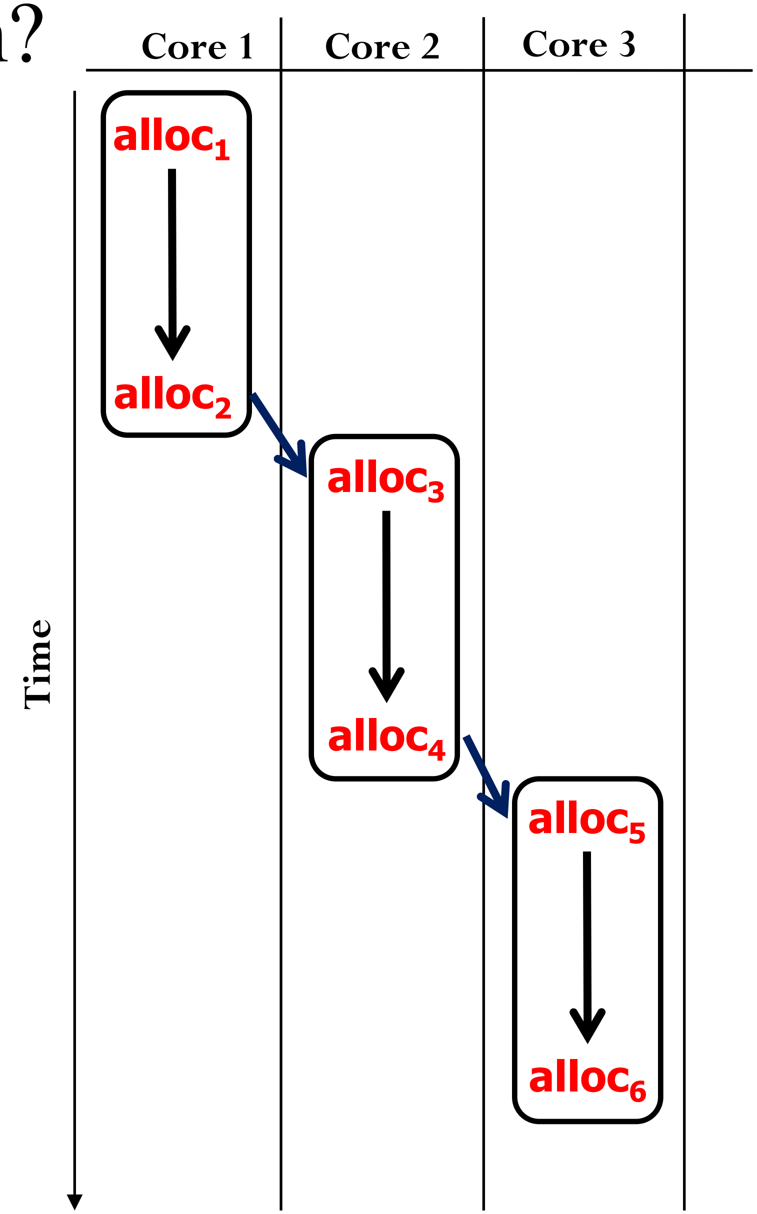
- Accumulator, induction, and min/max expansion only capture a small set of dependences
- 2 options
  - 1) Break more dependences – New transformations
  - 2) Parallelize in the presence of branches – more than DOALL parallelization
- We will talk about both
- For today, consider data dependences as a solved problem

What's the Next Problem?

3. C/C++ too restrictive

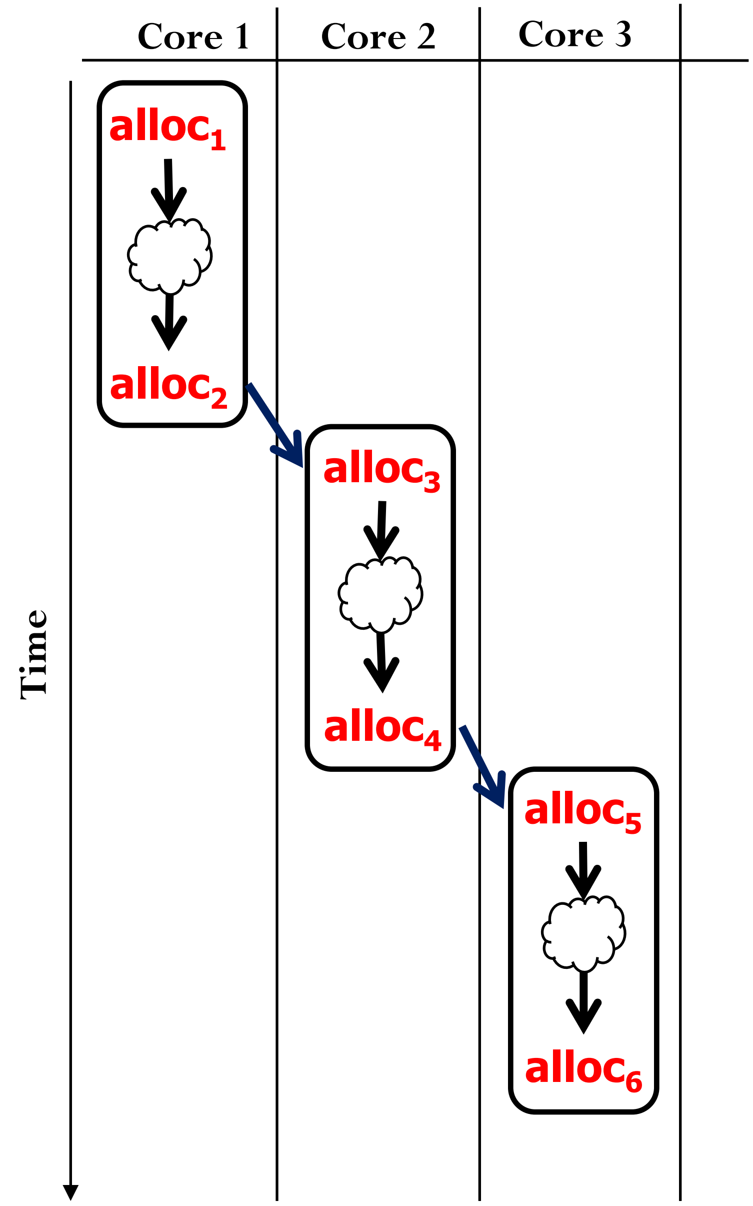
```
char *memory;

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;
}
```



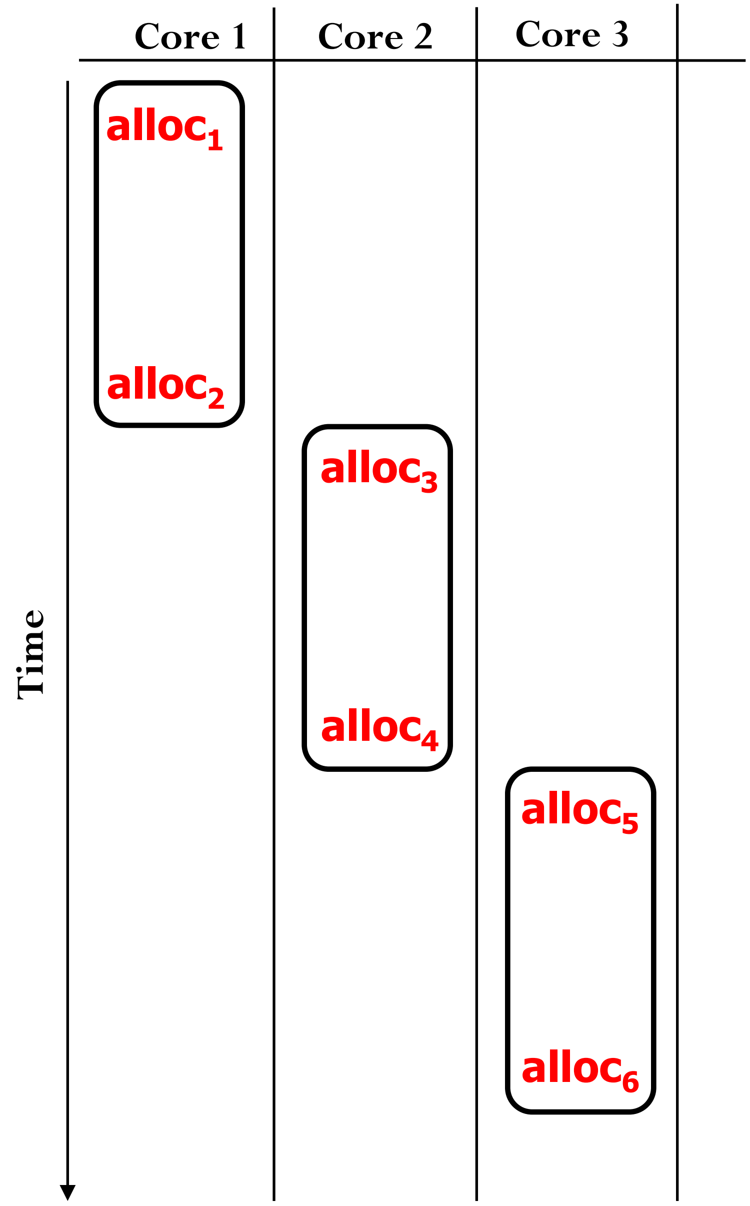
Loops cannot be parallelized even if computation is independent

Commutative Extension

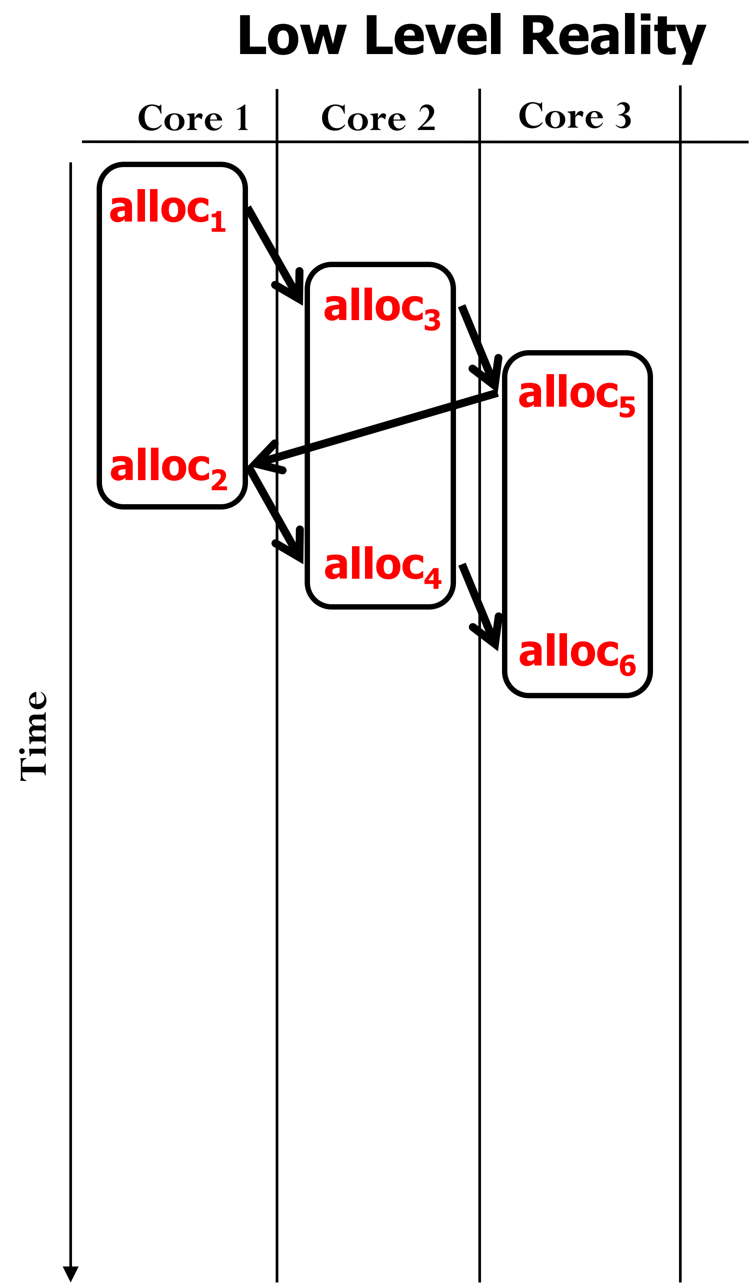
- Interchangeable call sites
  - Programmer doesn't care about the order that a particular function is called
  - Multiple different orders are all defined as correct
  - Impossible to express in C
- Prime example is memory allocation routine
  - Programmer does not care which address is returned on 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;
}
```

Low Level Reality



```
char *memory;

@Commutative
void * alloc(int size);

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

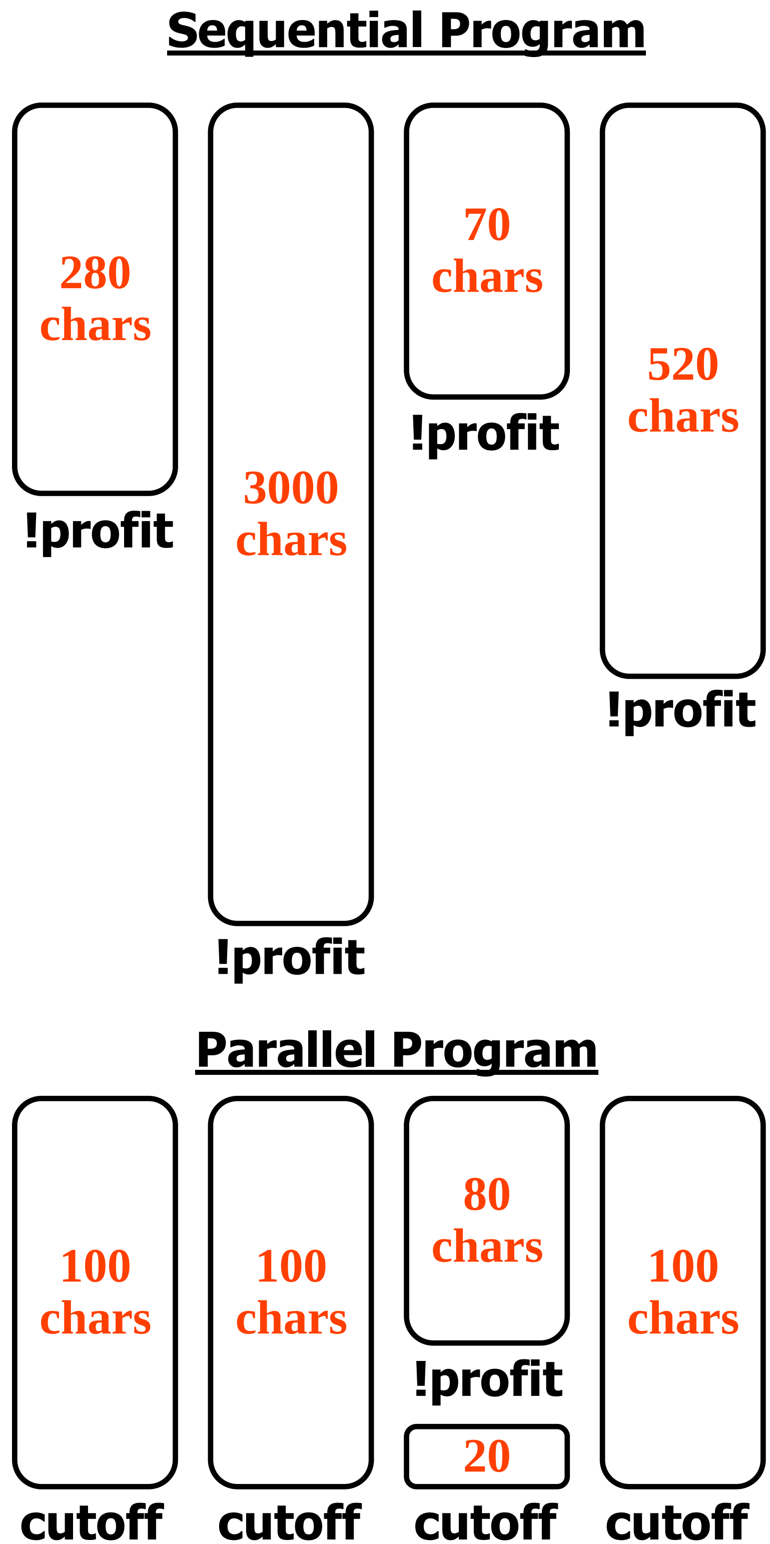


Implementation dependences should not cause serialization.

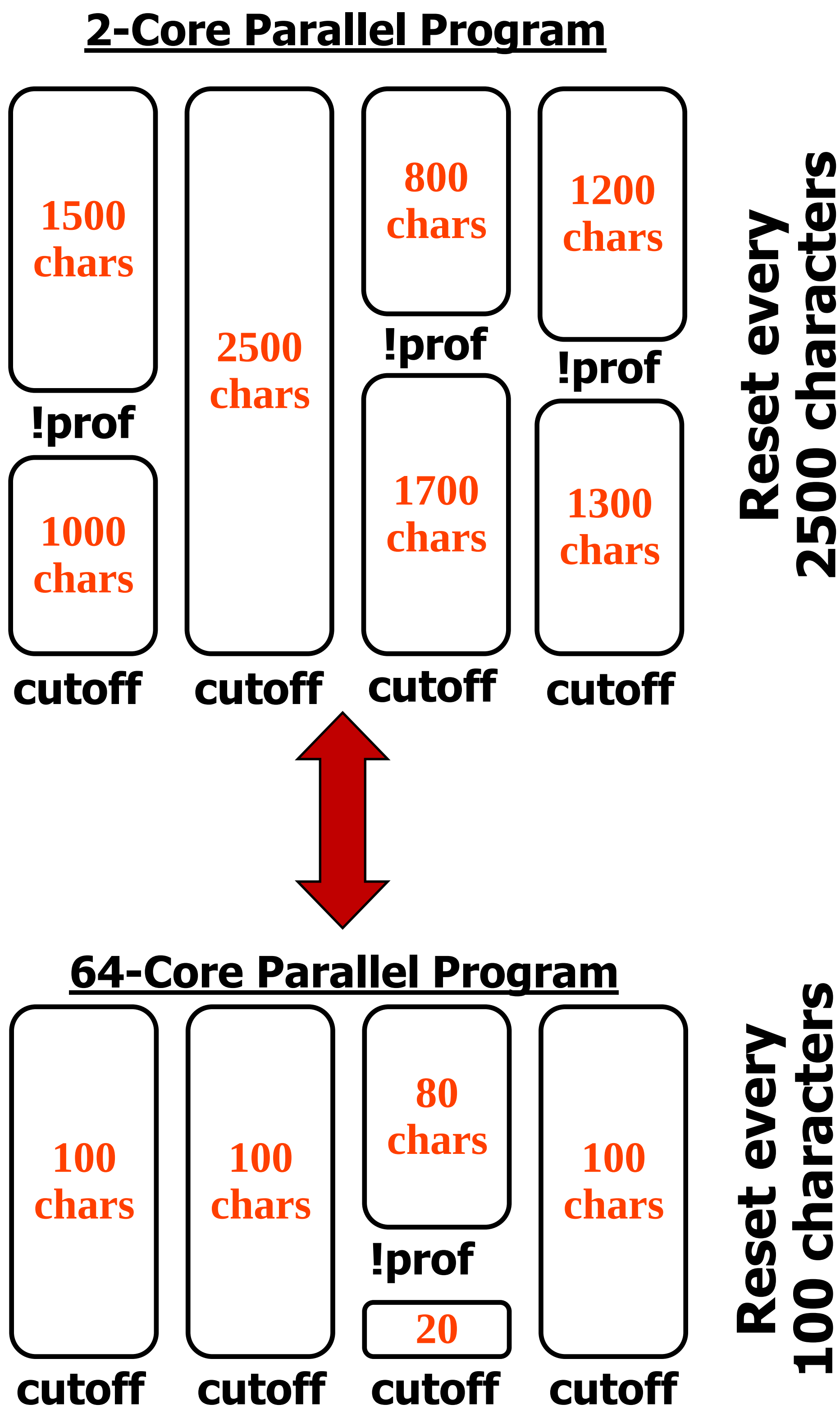
What is the Next Problem?

- 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();
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);
```



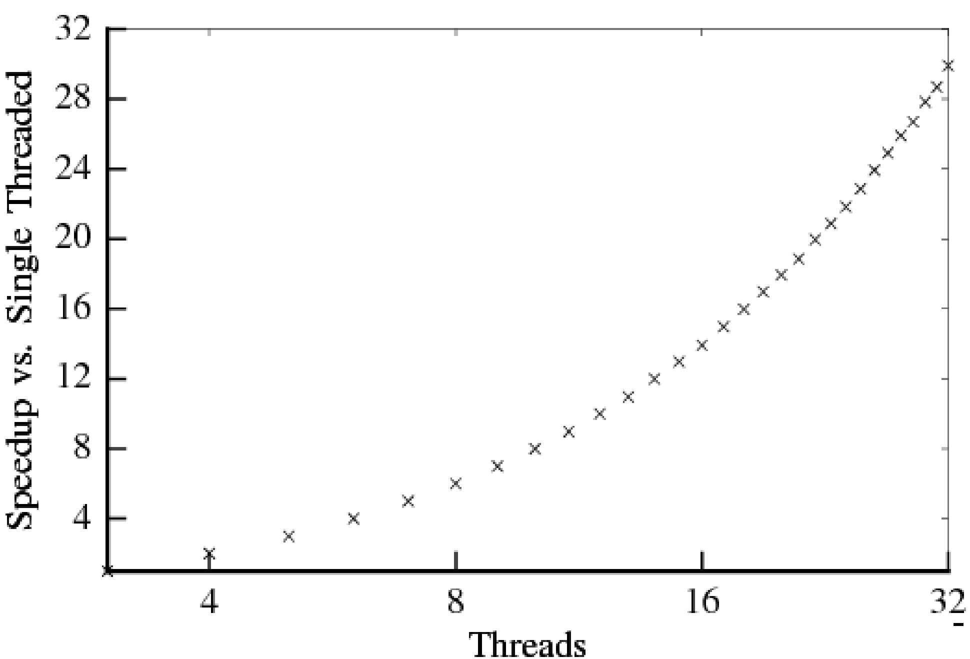
```
dict = create_dict();
while((char = read(1))) {
  profitable =
       compress(char, dict)

  @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);
```




```
#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.

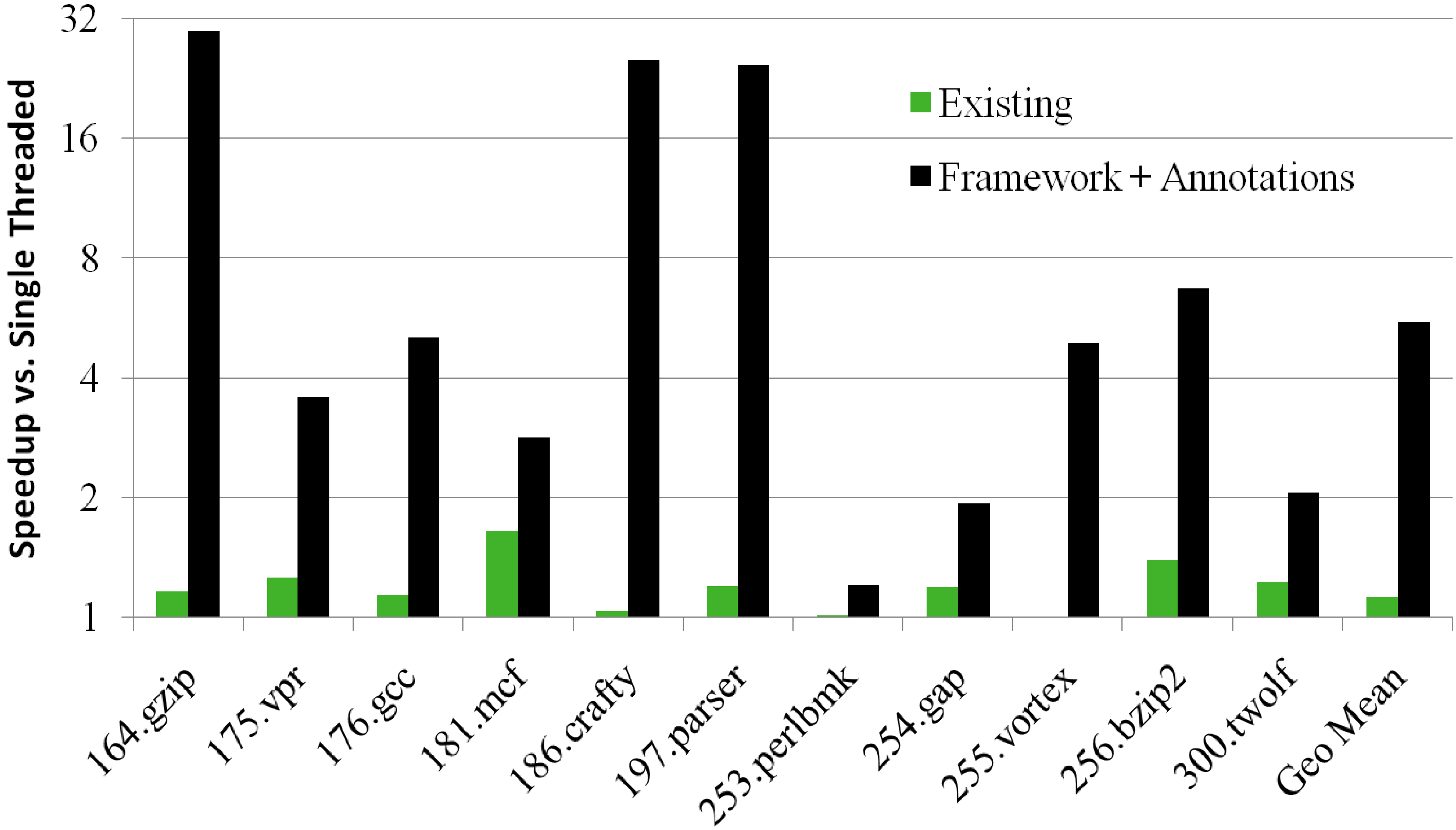
Low-Level Reality:

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

Performance Potential



What prevents the automatic extraction of parallelism?

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

~~**Sequential Programming Model**~~

Discussion Points

- Is implicit parallelism better than explicit?
  - » Is implicitly parallel code easier to write?
  - » What if the compiler cannot discover your parallelism?
  - » Would you use a tool that parallelized your code?
- What else is not expressable in C besides Y-branch and commutative?
  - » Or, what are other hurdles to parallelization?
  - » OpenMP already provides pragmas for parallel loops? Why are these not more popular?
- How do you write code that is more parallelizable?
  - » What about linked data structures?, recursion?, pointers?
  - » Should compilers speculate?