18-447
Computer Architecture
Lecture 28: Multiprocessors

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2014, 4/14/2013

Agenda Today

- Wrap up Prefetching
- Start Multiprocessing

Prefetching Buzzwords (Incomplete)

- What, when, where, how
- Hardware, software, execution based
- Accuracy, coverage, timeliness, bandwidth consumption, cache pollution
- Aggressiveness (prefetch degree, prefetch distance), throttling
- Prefetching for arbitrary access/address patterns

Execution-based Prefetchers (I)

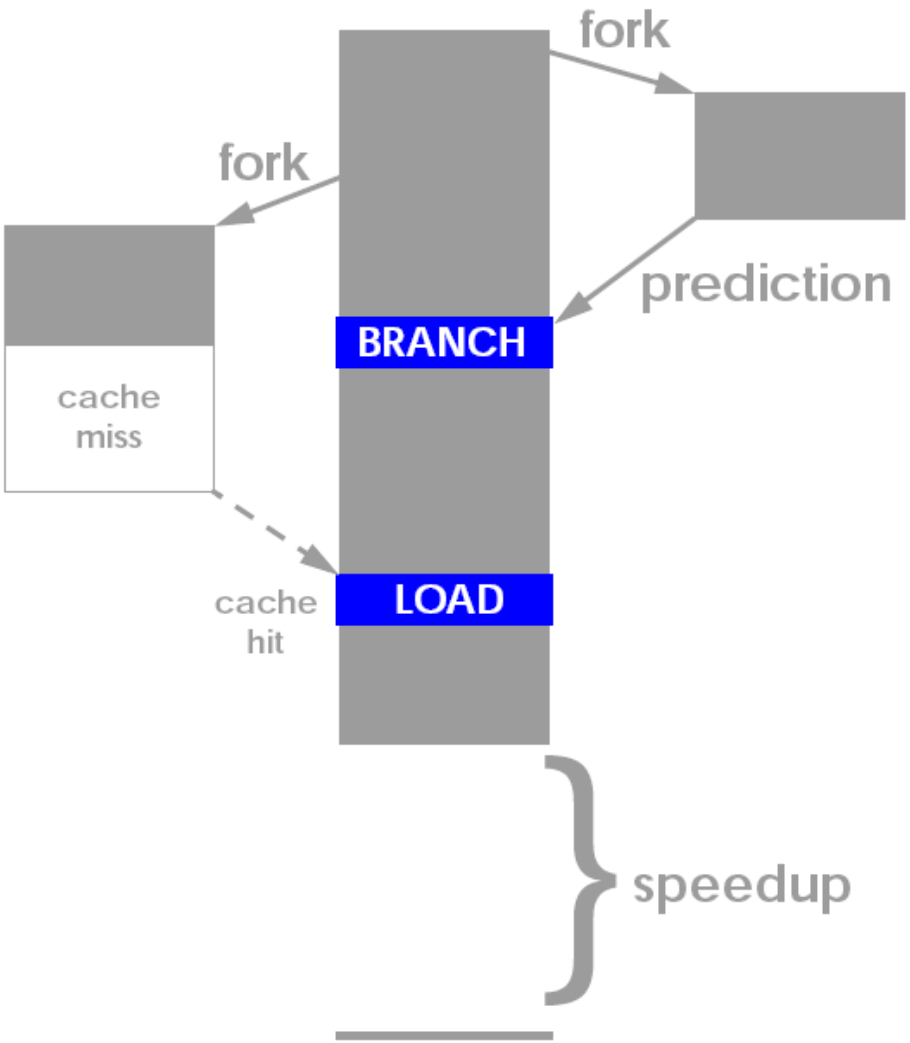
- Idea: Pre-execute a piece of the (pruned) program solely for prefetching data
  - Only need to distill pieces that lead to cache misses
  - Speculative thread: Pre-executed program piece can be considered a "thread"
  - Speculative thread can be executed
  - On a separate processor/core
  - On a separate hardware thread context (think fine-grained multithreading)
  - On the same thread context in idle cycles (during cache misses)

Execution-based Prefetchers (II)

- How to construct the speculative thread:
  - Software based pruning and "spawn" instructions
  - Hardware based pruning and "spawn" instructions
  - Use the original program (no construction), but
    - Execute it faster without stalling and correctness constraints

- Speculative thread
  - Needs to discover misses before the main program
    - Avoid waiting/stalling and/or compute less
  - To get ahead, uses
    - Perform only address generation computation, branch prediction, value prediction (to predict "unknown" values)
  - Purely speculative so there is no need for recovery of main program if the speculative thread is incorrect

Thread-Based Pre-Execution



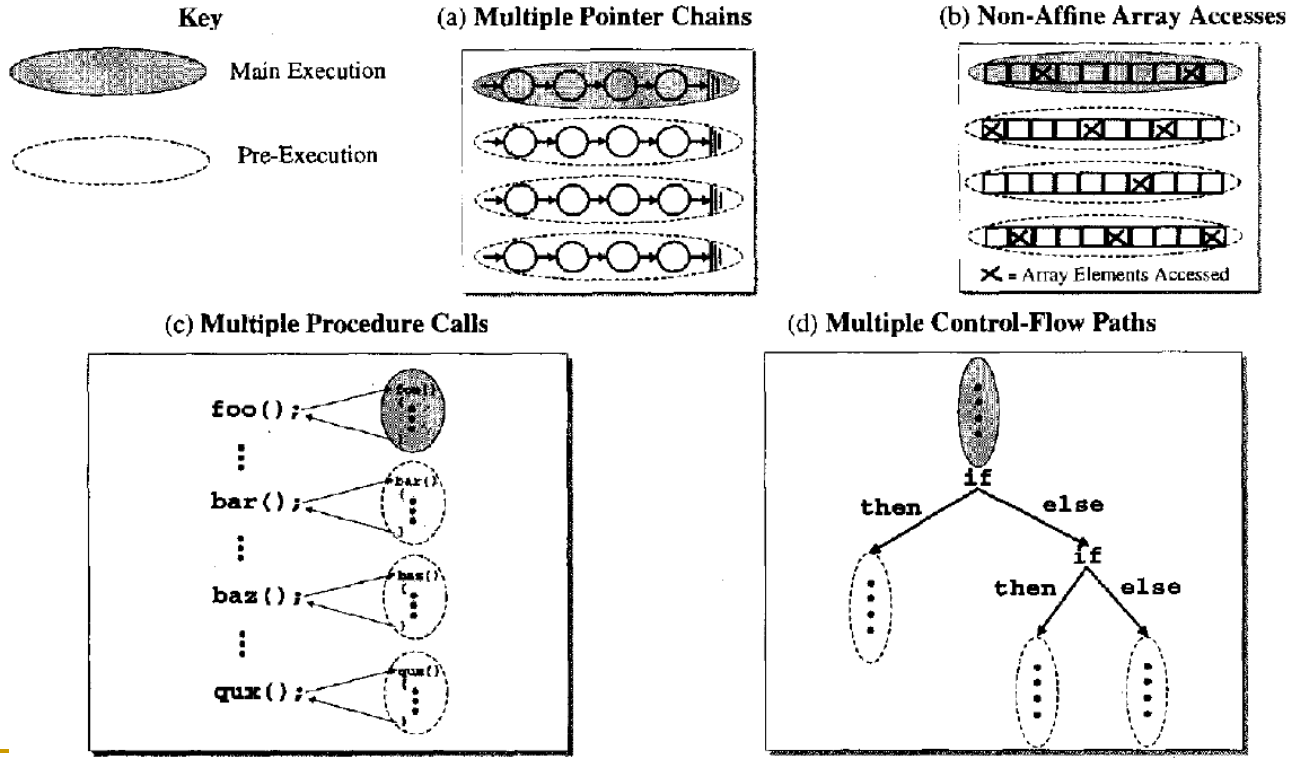
- Dubois and Song, "Assisted Execution," USC Tech Report 1998.
- Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," ISCA 1999.
- Zilles and Sohi, "Execution-based Prediction Using Speculative Slices", ISCA 2001.

Thread-Based Pre-Execution Issues

- Where to execute the precomputation thread?
  1. Separate core (least contention with main thread)
  2. Separate thread context on the same core (more contention)
  3. Same core, same context
     - When the main thread is stalled
- When to spawn the precomputation thread?
  1. Insert spawn instructions well before the "problem" load
     - How far ahead?
       - Too early: prefetch might not be needed
       - Too late: prefetch might not be timely
  2. When the main thread is stalled
- When to terminate the precomputation thread?
  1. With pre-inserted CANCEL instructions
  2. Based on effectiveness/contention feedback (recall throttling)

Thread-Based Pre-Execution Issues

- Read
  - Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," ISCA 2001.
  - Many issues in software-based pre-execution discussed



An Example

(a) Original Code

```
register int i;
register arc_t *arcout;
for(; i<trips; ){
   // loop over "trips" lists
   if (arcout[1].ident != FIXED) {
      ...
      first_of_sparse_list = arcout + 1;
   }
   ...
   arcin = (arc_t *)first_of_sparse_list
                  →tail→mark;
   // traverse the list starting with
   // the first node just assigned
   while (arcin) {
      tail = arcin→tail;
      ...
      arcin = (arc_t *)tail→mark;
   }
   i++, arcout+=3;
}
```

(b) Code with Pre-Execution

```
register int i;
register arc_t *arcout;
for(; i<trips; ){
   // loop over "trips" lists
   if (arcout[1].ident != FIXED) {
      ...
      first_of_sparse_list = arcout + 1;
   }
   ...
   // invoke a pre-execution starting
   // at END_FOR
   PreExecute_Start(END_FOR);
   arcin = (arc_t *)first_of_sparse_list
         →tail→mark;
   // traverse the list starting with
   // the first node just assigned
   while (arcin) {
      tail = arcin→tail;
      ...
      arcin = (arc_t *)tail→mark;
   }
   // terminate this pre-execution after
   // prefetching the entire list
   PreExecute_Stop();
END_FOR:
   // the target address of the pre-
   // execution
   i++, arcout+=3;
}
// terminate this pre-execution if we
// have passed the end of the for-loop
PreExecute_Stop();
```

Figure 2. Abstract versions of an important loop nest in the Spec2000 benchmark `mcf`. Loads that incur many cache misses are underlined.

The Spec2000 benchmark `mcf` spends roughly half of its execution time in a nested loop which traverses a set of linked lists. An abstract version of this loop is shown in Figure 2(a), in which the for-loop iterates over the lists and the while-loop visits the elements of each list. As we observe from the figure, the first node of each list is assigned by dereferencing the pointer `first_of_sparse_list`, whose value is in fact determined by `arcout`, an induction variable of the for-loop. Therefore, even when we are still working on the current list, the first and the remaining nodes on the next list can be loaded speculatively by pre-executing the next iteration of the for-loop.

Figure 2(b) shows a version of the program with pre-execution code inserted (shown in boldface). **END_FOR** is simply a label to denote the place where `arcout` gets updated. The new instruction **PreExecute_Start(END_FOR)** initiates a pre-execution thread, say T, starting at the PC represented by **END_FOR**. Right after the pre-execution begins, T's registers that hold the values of `i` and `arcout` will be updated. Then `i`'s value is compared against `trips` to see if we have reached the end of the for-loop. If so, thread T will exit the for-loop and encounters a **PreExecute_Stop()**, which will terminate the pre-execution and free up T for future use. Otherwise, T will continue pre-executing the body of the for-loop, and hence compute the first node of the next list automatically. Finally, after traversing the entire list through the while-loop, the pre-execution will be terminated by another **PreExecute_Stop()**. Notice that any **PreExecute_Start()** instructions encountered during pre-execution are simply ignored as we do not allow nested pre-execution in order to keep our design simple. Similarly, **PreExecute_Stop()** instructions cannot terminate the main thread either.

Example ISA Extensions

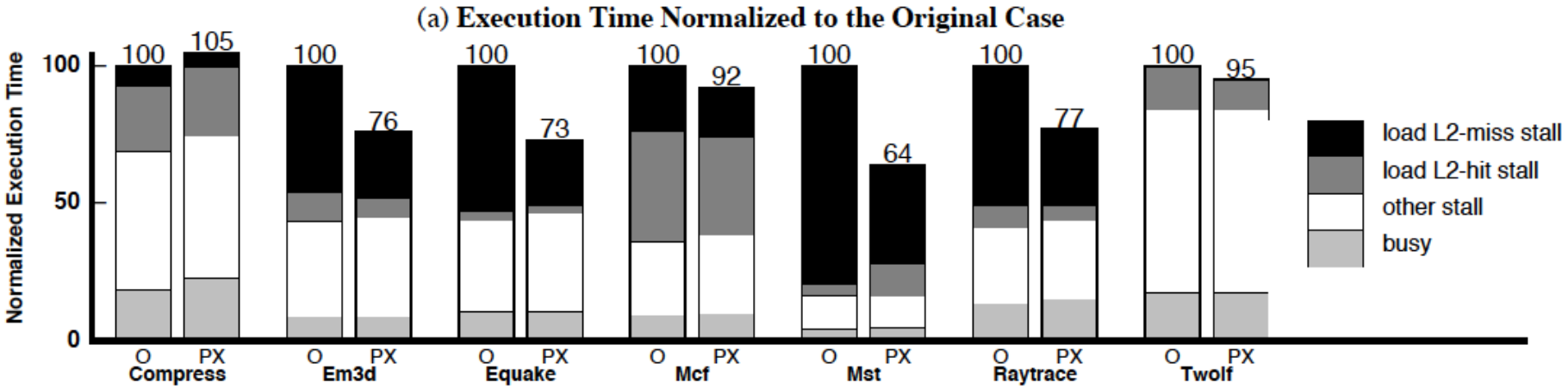
$Thread\_ID$ = **PreExecute_Start**($Start\_PC$, Max_Insts):
Request for an idle context to start pre-execution at $Start\_PC$ and stop when Max_Insts instructions have been executed; $Thread\_ID$ holds either the identity of the pre-execution thread or -1 if there is no idle context. This instruction has effect only if it is executed by the main thread.

PreExecute_Stop(): The thread that executes this instruction will be self terminated if it is a pre-execution thread; no effect otherwise.

PreExecute_Cancel($Thread\_ID$): Terminate the pre-execution thread with $Thread\_ID$. This instruction has effect only if it is executed by the main thread.

Figure 4. Proposed instruction set extensions to support pre-execution. (C syntax is used to improve readability.)

Results on a Multithreaded Processor



Problem Instructions

- Zilles and Sohi, "Execution-based Prediction Using Speculative Slices", ISCA 2001.
- Zilles and Sohi, "Understanding the backward slices of performance degrading instructions," ISCA 2000.

Figure 2. ***Example problem instructions from heap insertion routine in* vpr.**

```
struct s_heap **heap; // from [1..heap_size]
int heap_size; // # of slots in the heap
int heap_tail; // first unused slot in heap

  void add_to_heap (struct s_heap *hptr) {
     ...
1.   heap[heap_tail] = hptr;
2.   int ifrom = heap_tail;
3.   int ito = ifrom/2;
4.   heap_tail++;
5.   while ((ito >= 1) &&
6.        (heap[ifrom]->cost < heap[ito]->cost))
7.      struct s_heap *temp_ptr = heap[ito];
8.      heap[ito] = heap[ifrom];
9.      heap[ifrom] = temp_ptr;
10.     ifrom = ito;
11.     ito = ifrom/2;
     }
  }
```

branch misprediction

cache miss

Fork Point for Prefetching Thread

Figure 3. *The* **`node_to_heap`** *function, which serves as the fork point for the slice that covers* **`add_to_heap`**.

```
void node_to_heap (..., float cost, ...) {
    struct s_heap *hptr;   <-- fork point
    ...
    hptr = alloc_heap_data();
    hptr->cost = cost;
    ...
    add_to_heap (hptr);
}
```

Pre-execution Thread Construction

Figure 4. *Alpha assembly for the* ***add_to_heap*** *function. The instructions are annotated with the number of the line in Figure 2 to which they correspond. The problem instructions are in bold and the shaded instructions comprise the un-optimized slice.*

```
node_to_heap:
     ... /* skips ~40 instructions */
2    lda     s1, 252(gp)     # &heap_tail
2    ldl     t2, 0(s1)       # ifrom = heap_tail
1    ldq     t5, -76(s1)     # &heap[0]
3    cmplt   t2, 0, t4       # see note
4    addl    t2, 0x1, t6     # heap_tail ++
1    s8addq  t2, t5, t3      # &heap[heap_tail]
4    stl     t6, 0(s1)       # store heap_tail
1    stq     s0, 0(t3)       # heap[heap_tail]
3    addl    t2, t4, t4      # see note
3    sra     t4, 0x1, t4     # ito = ifrom/2
5    ble     t4, return      # (ito < 1)
loop:
6    s8addq  t2, t5, a0      # &heap[ifrom]
6    s8addq  t4, t5, t7      # &heap[ito]
11   cmplt   t4, 0, t9       # see note
10   move    t4, t2          # ifrom = ito
6    ldq     a2, 0(a0)       # heap[ifrom]
6    ldq     a4, 0(t7)       # heap[ito]
11   addl    t4, t9, t9      # see note
11   sra     t9, 0x1, t4     # ito = ifrom/2
6    lds     $f0, 4(a2)      # heap[ifrom]->cost
6    lds     $f1, 4(a4)      # heap[ito]->cost
6    cmptlt  $f0,$f1,$f0     # (heap[ifrom]->cost
6    fbeq    $f0, return     #  < heap[ito]->cost)
8    stq     a2, 0(t7)       # heap[ito]
9    stq     a4, 0(a0)       # heap[ifrom]
5    bgt     t4, loop        # (ito >= 1)
return:
     ... /* register restore code & return */
```

note: the divide by 2 operation is implemented by a 3 instruction sequence described in the strength reduction optimization.

Figure 5. *Slice constructed for example problem instructions. Much smaller than the original code, the slice contains a loop that mimics the loop in the original code.*

```
slice:
1    ldq     $6, 328(gp)     # &heap
2    ldl     $3, 252(gp)     # ito = heap_tail
slice_loop:
3,11 sra     $3, 0x1, $3     # ito /= 2
6    s8addq  $3, $6, $16     # &heap[ito]
6    ldq     $18, 0($16)     # heap[ito]
6    lds     $f1, 4($18)     # heap[ito]->cost
6    cmptle  $f1,$f17,$f31   # (heap[ito]->cost
                             #  < cost) PRED
     br      slice_loop

## Annotations
fork: on first instruction of node_to_heap
live-in: $f17<cost>, gp
max loop iterations: 4
```

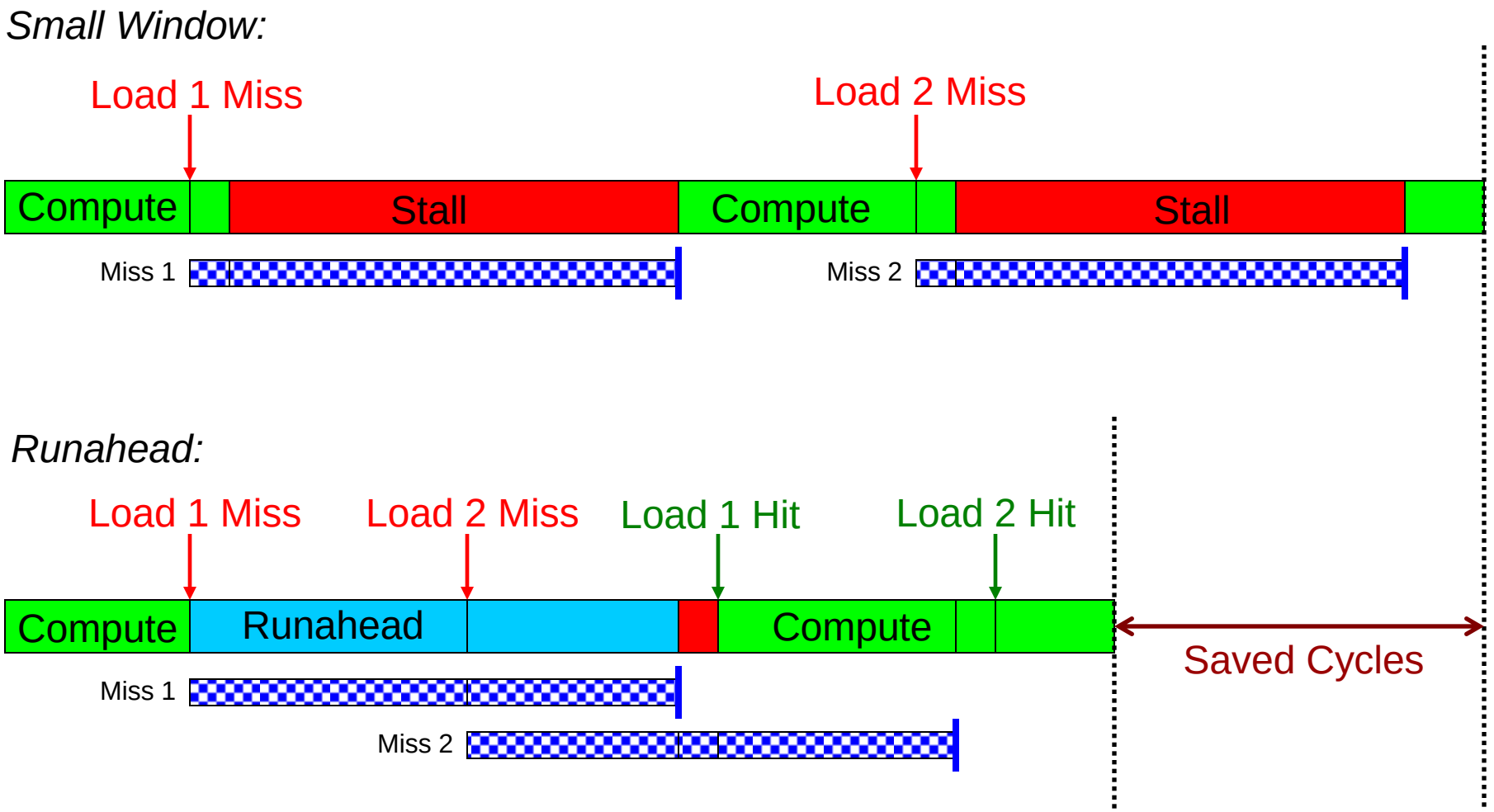
Review: Runahead Execution

- A simple pre-execution method for prefetching purposes

- When the oldest instruction is a long-latency cache miss:
  - Checkpoint architectural state and enter runahead mode
- In runahead mode:
  - Speculatively pre-execute instructions
  - The purpose of pre-execution is to generate prefetches
  - L2-miss dependent instructions are marked INV and dropped
- Runahead mode ends when the original miss returns
  - Checkpoint is restored and normal execution resumes

- Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003.

Review: Runahead Execution (Mutlu et al., HPCA 2003)



Runahead as an Execution-based Prefetcher

- Idea of an Execution-Based Prefetcher: Pre-execute a piece of the (pruned) program solely for prefetching data

- Idea of Runahead: Pre-execute the main program solely for prefetching data

Multiprocessors and Issues in Multiprocessing

Readings: Multiprocessing

- Required
  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.
  - Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979

- Recommended
  - Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966
  - Hill, Jouppi, Sohi, "Multiprocessors and Multicomputers," pp. 551-560 in Readings in Computer Architecture.
  - Hill, Jouppi, Sohi, "Dataflow and Multithreading," pp. 309-314 in Readings in Computer Architecture.

Readings: Cache Coherence

- Required
  - Culler and Singh, *Parallel Computer Architecture*
    - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
  - P&H, *Computer Organization and Design*
    - Chapter 5.8 (pp 534 – 538 in 4th and 4th revised eds.)

- Recommended:
  - Papamarcos and Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ISCA 1984.

Remember: Flynn's Taxonomy of Computers

- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966

- SISD: Single instruction operates on single data element
- SIMD: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- MISD: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- MIMD: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor

Why Parallel Computers?

- Parallelism: Doing multiple things at a time
- Things: instructions, operations, tasks

- Main Goal
  - Improve performance (Execution time or task throughput)
    - Execution time of a program governed by Amdahl's Law

- Other Goals
  - Reduce power consumption
    - (4N units at freq F/4) consume less power than (N units at freq F)
    - Why?
  - Improve cost efficiency and scalability, reduce complexity
    - Harder to design a single unit that performs as well as N simpler units
  - Improve dependability: Redundant execution in space

Types of Parallelism and How to Exploit Them

- Instruction Level Parallelism
  - Different instructions within a stream can be executed in parallel
  - Pipelining, out-of-order execution, speculative execution, VLIW
  - Dataflow

- Data Parallelism
  - Different pieces of data can be operated on in parallel
  - SIMD: Vector processing, array processing
  - Systolic arrays, streaming processors

- Task Level Parallelism
  - Different "tasks/threads" can be executed in parallel
  - Multithreading
  - Multiprocessing (multi-core)

Task-Level Parallelism: Creating Tasks

- Partition a single problem into multiple related tasks (threads)
  - Explicitly: Parallel programming
    - Easy when tasks are natural in the problem
      - Web/database queries
    - Difficult when natural task boundaries are unclear
  - Transparently/implicitly: Thread level speculation
    - Partition a single thread speculatively
- Run many independent tasks (processes) together
  - Easy when there are many processes
    - Batch simulations, different users, cloud computing workloads
  - Does not improve the performance of a single task

Multiprocessing Fundamentals

Multiprocessor Types

- Loosely coupled multiprocessors
  - No shared global memory address space
  - Multicomputer network
    - Network-based multiprocessors
  - Usually programmed via message passing
    - Explicit calls (send, receive) for communication

- Tightly coupled multiprocessors
  - Shared global memory address space
  - Traditional multiprocessing: symmetric multiprocessing (SMP)
    - Existing multi-core processors, multithreaded processors
  - Programming model similar to uniprocessors (i.e., multitasking uniprocessor) except
    - Operations on shared data require synchronization

Main Issues in Tightly-Coupled MP

- Shared memory synchronization
  - Locks, atomic operations
- Cache consistency
  - More commonly called cache coherence
- Ordering of memory operations
  - What should the programmer expect the hardware to provide?
- Resource sharing, contention, partitioning
- Communication: Interconnection networks
- Load imbalance

Aside: Hardware-based Multithreading

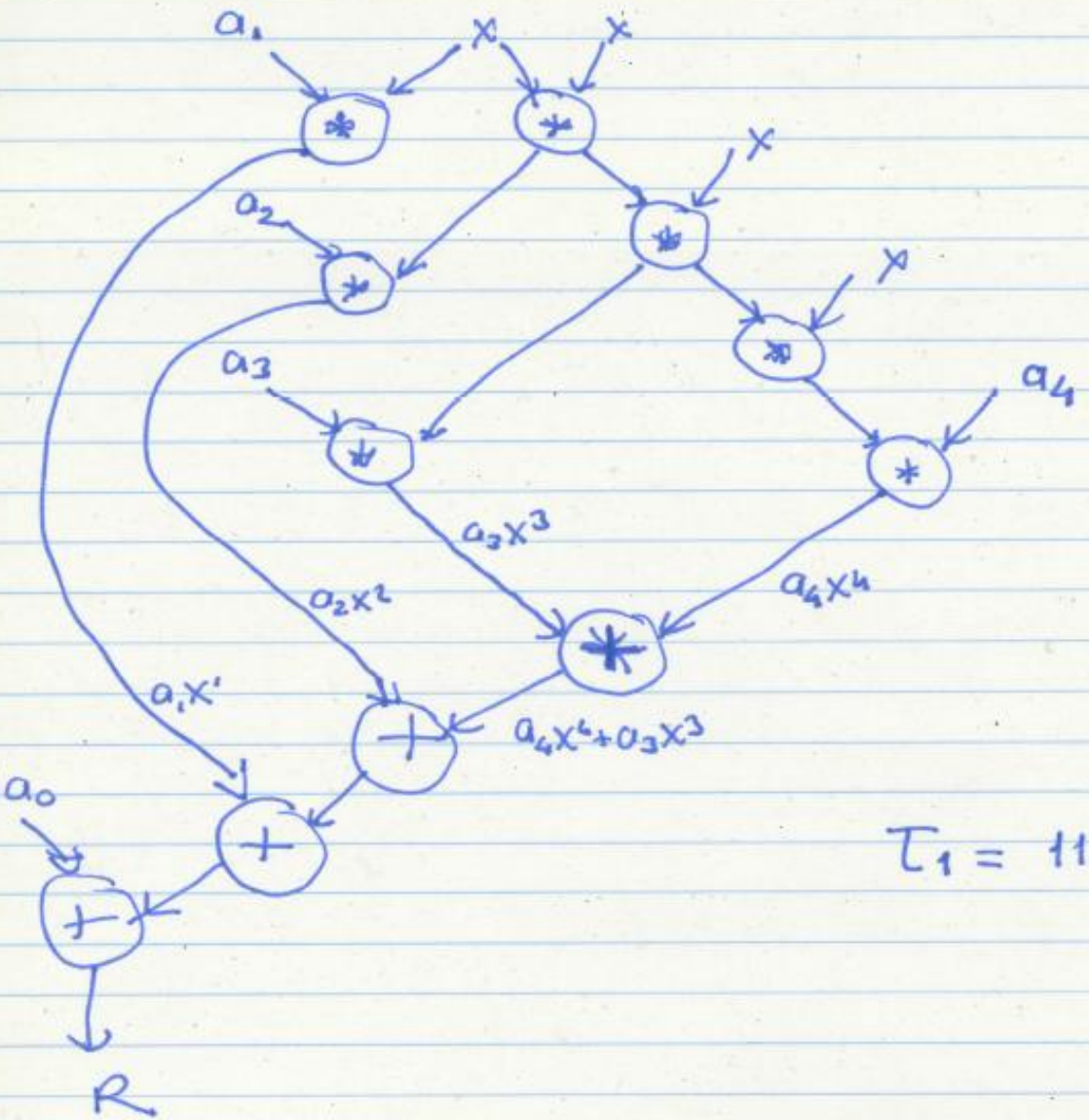
- Coarse grained
  - Quantum based
  - Event based (switch-on-event multithreading)
- Fine grained
  - Cycle by cycle
  - Thornton, "CDC 6600: Design of a Computer," 1970.
  - Burton Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.
- Simultaneous
  - Can dispatch instructions from multiple threads at the same time
  - Good for improving execution unit utilization

Parallel Speedup Example

- $a4x^4 + a3x^3 + a2x^2 + a1x + a0$
- Assume each operation 1 cycle, no communication cost, each op can be executed in a different processor
- How fast is this with a single processor?
  - Assume no pipelining or concurrent execution of instructions
- How fast is this with 3 processors?

$$R = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

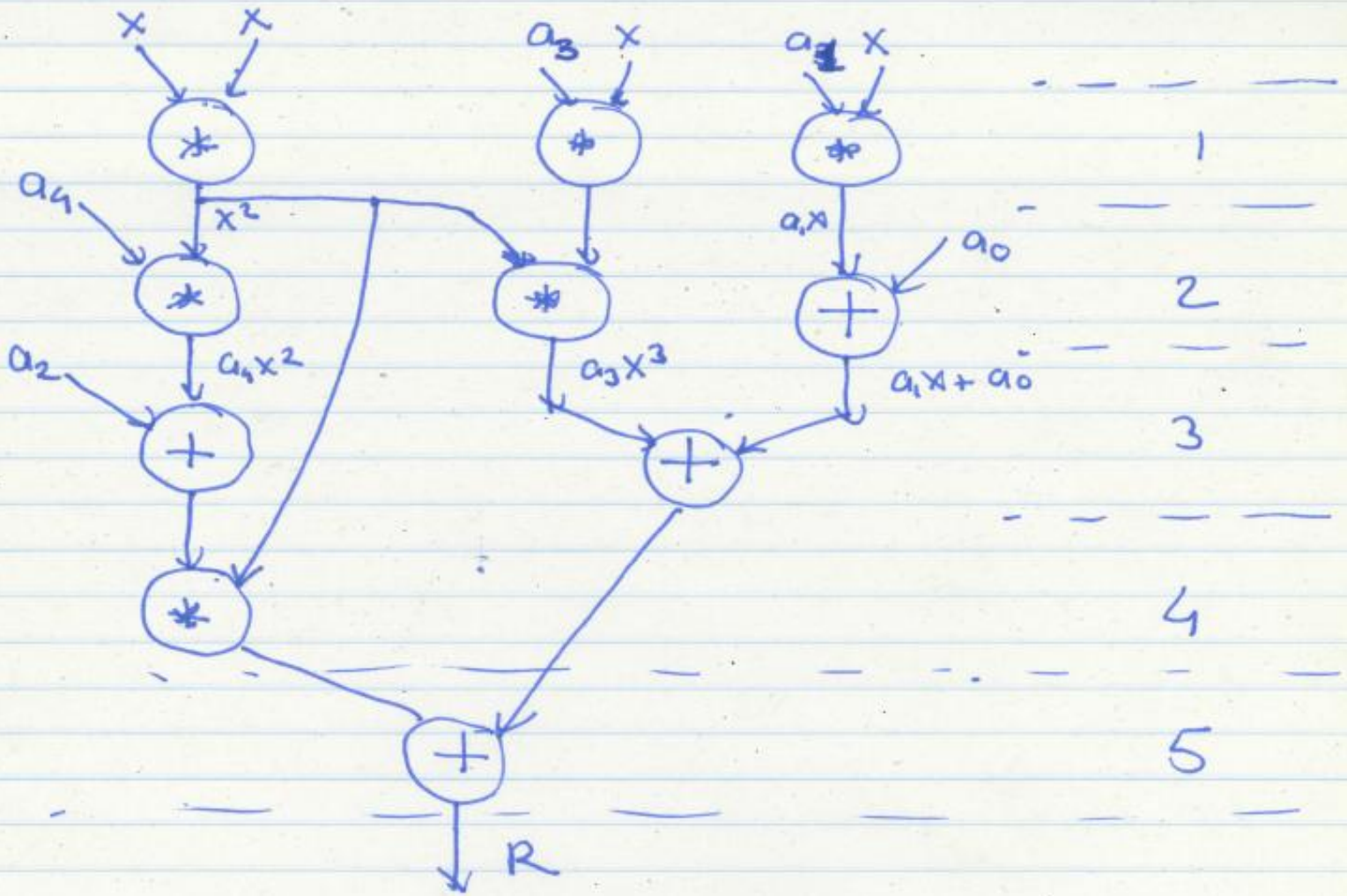
Single processor : 11 operations (draw the data flow graph)



$T_1 = 11$ cycles

$$R = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Three processors : T_3 (exec. time with 3 proc.)



$T_3 = 5$ cycles

Speedup with 3 Processors

$T_3 = 5$ cycles

Speedup with 3 processors $= \frac{11}{5} = 2.2$

$\left(\frac{T_1}{T_3}\right)$

Is this a fair comparison?

Revisiting the Single-Processor Algorithm

Revisit T_1

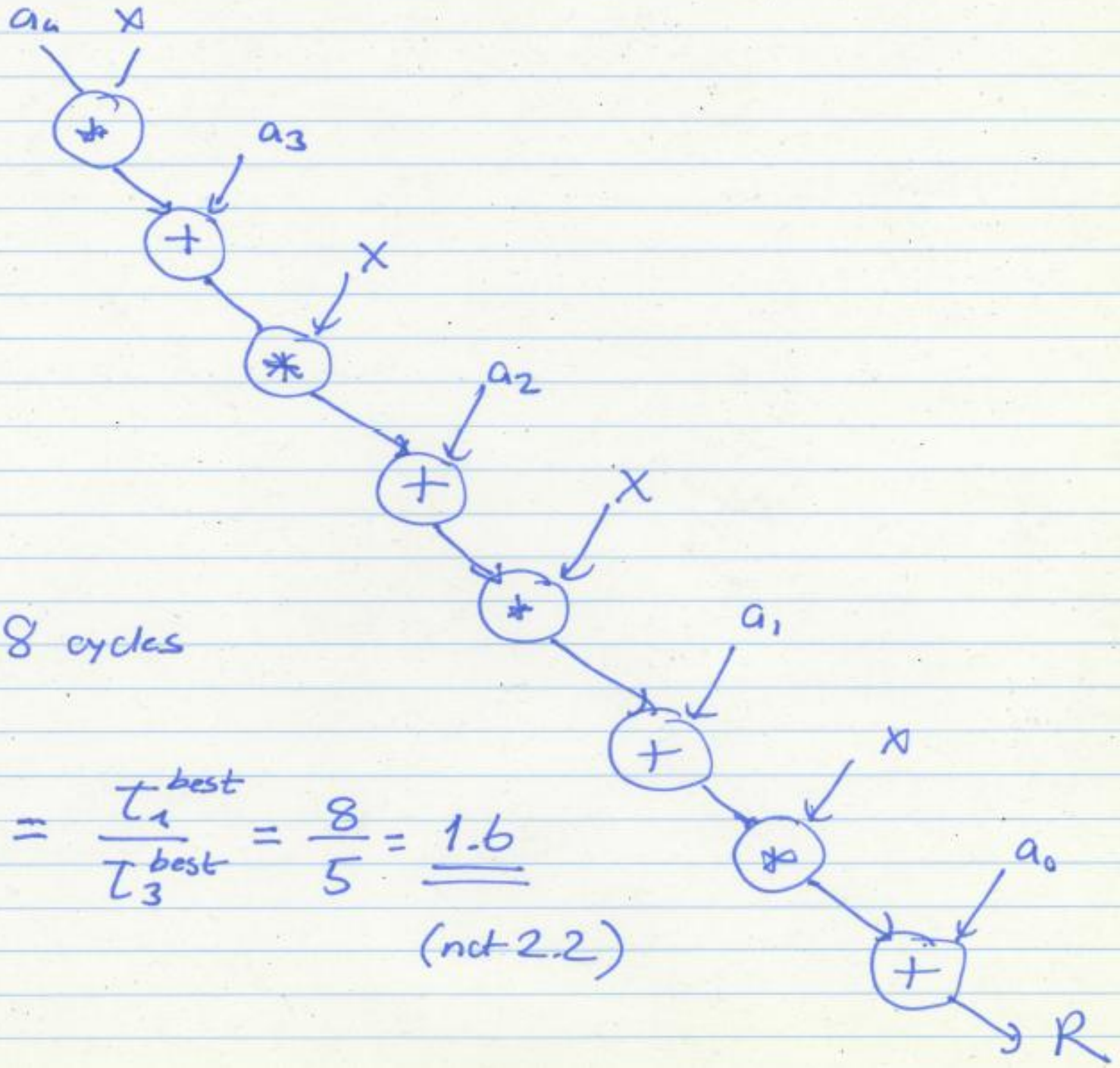
Better single-processor algorithm:

$$R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

$$R = (((a_4 x + a_3) x + a_2) x + a_1) x + a_0$$

(Horner's method)

Horner, "A new method of solving numerical equations of all orders, by continuous approximation," Philosophical Transactions of the Royal Society, 1819.



$T_1 = 8$ cycles

Speedup with 3 procs. $= \frac{T_1^{best}}{T_3^{best}} = \frac{8}{5} = \underline{\underline{1.6}}$

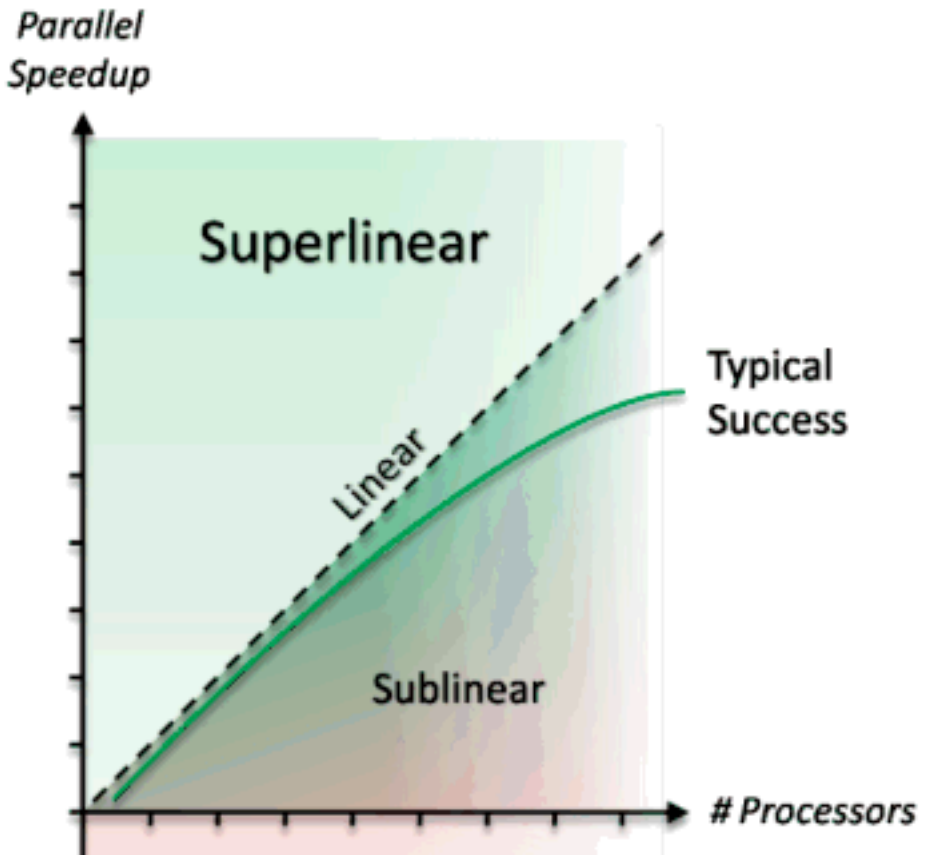
(not 2.2)

Superlinear Speedup

- Can speedup be greater than P with P processing elements?

- Cache effects
- Working set effects

- Happens in two ways:
  - Unfair comparisons
  - Memory effects



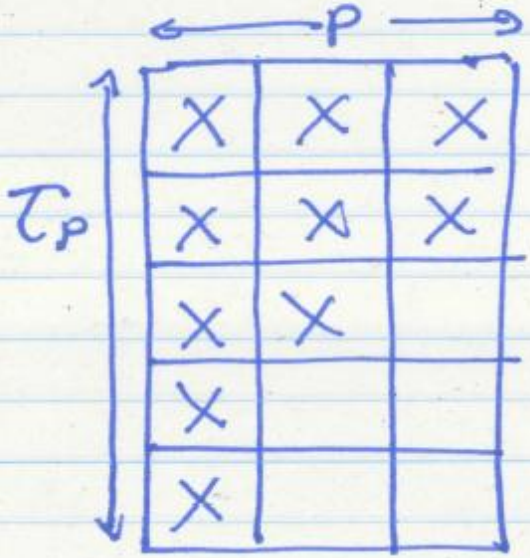
Utilization, Redundancy, Efficiency

- Traditional metrics
  - Assume all P processors are tied up for parallel computation
- Utilization: How much processing capability is used
  - U = (# Operations in parallel version) / (processors x Time)
- Redundancy: how much extra work is done with parallel processing
  - R = (# of operations in parallel version) / (# operations in best single processor algorithm version)
- Efficiency
  - E = (Time with 1 processor) / (processors x Time with P processors)
  - E = U/R

Utilization of a Multiprocessor

Multiprocessor metrics

Utilization : How much processing capability we use



$$U = \frac{\text{Ops with p proc.}}{p \times T_p}$$

$$U = \frac{\text{10 operations (in parallel version)}}{\text{3 processors} \times \text{5 time units}}$$

$$= \frac{10}{15}$$

Redundancy: How much extra work due to multiprocessing

$$R = \frac{\text{Ops with p proc.}^{best}}{\text{Ops with 1 proc.}^{best}} = \frac{10}{8}$$

R is always ≥ 1

Efficiency: How much resource we use compared to how much resource we can get away with

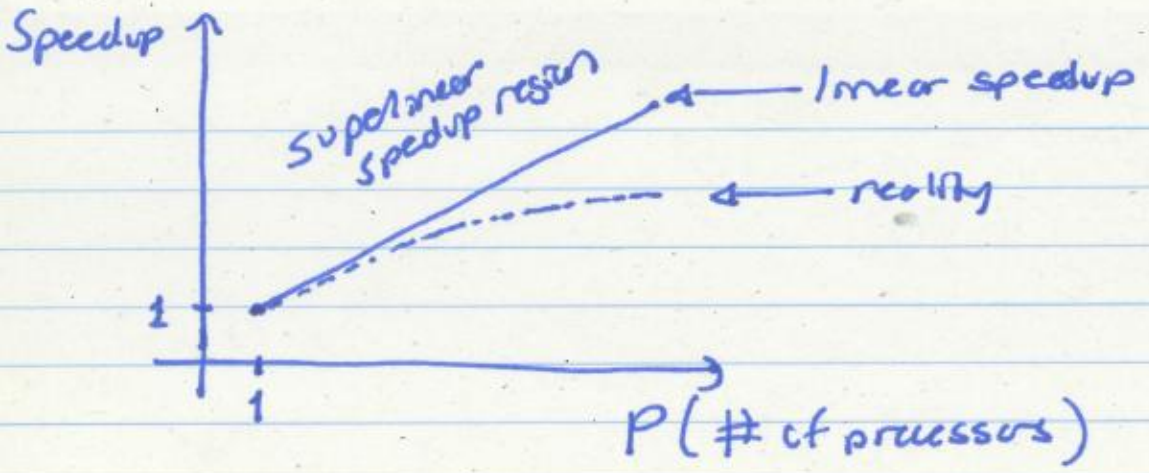
$$E = \frac{1 \cdot T_1^{best}}{p \cdot T_p^{best}}$$

(tying up 1 proc for T_p time units)

(tying up p proc. for T_p time units)

$$= \frac{8}{15} \qquad \left(E = \frac{U}{R} \right)$$

Caveats of Parallelism (I)



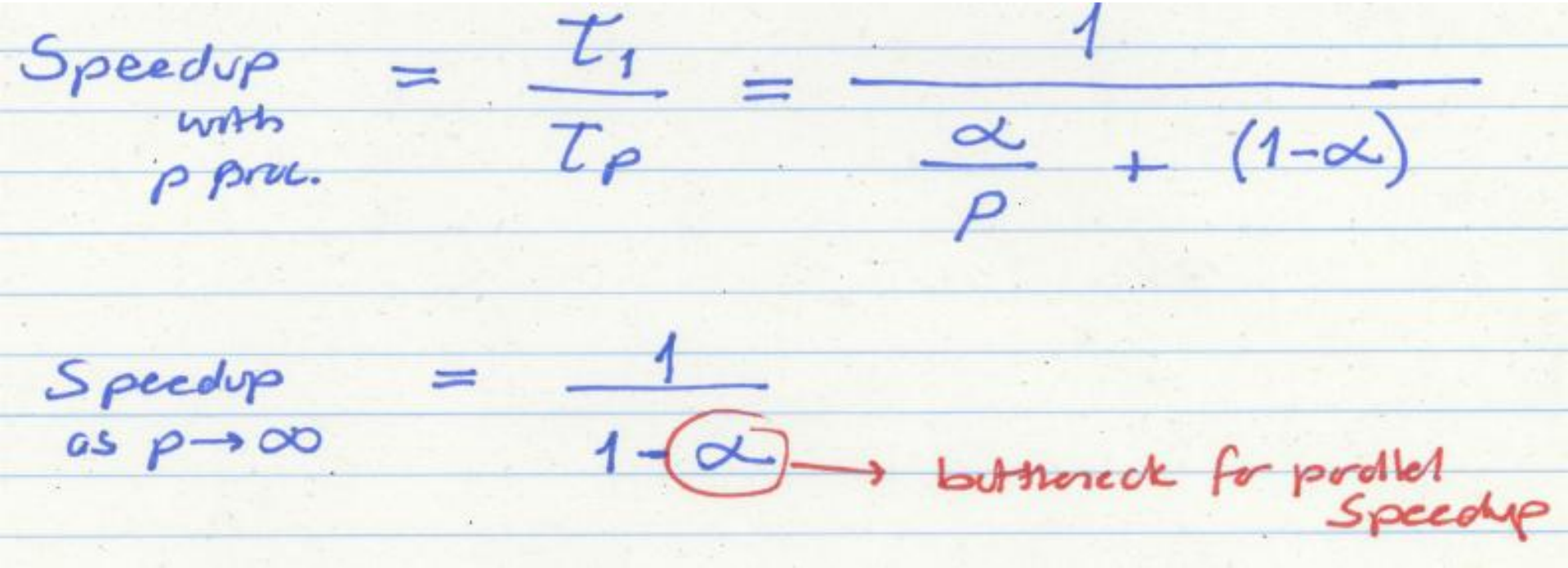
Why the reality? (diminishing returns)

$$T_p = \alpha \cdot \frac{T_1}{p} + (1-\alpha) \cdot T_1$$

α: parallelizable part/fraction of the single-processor program

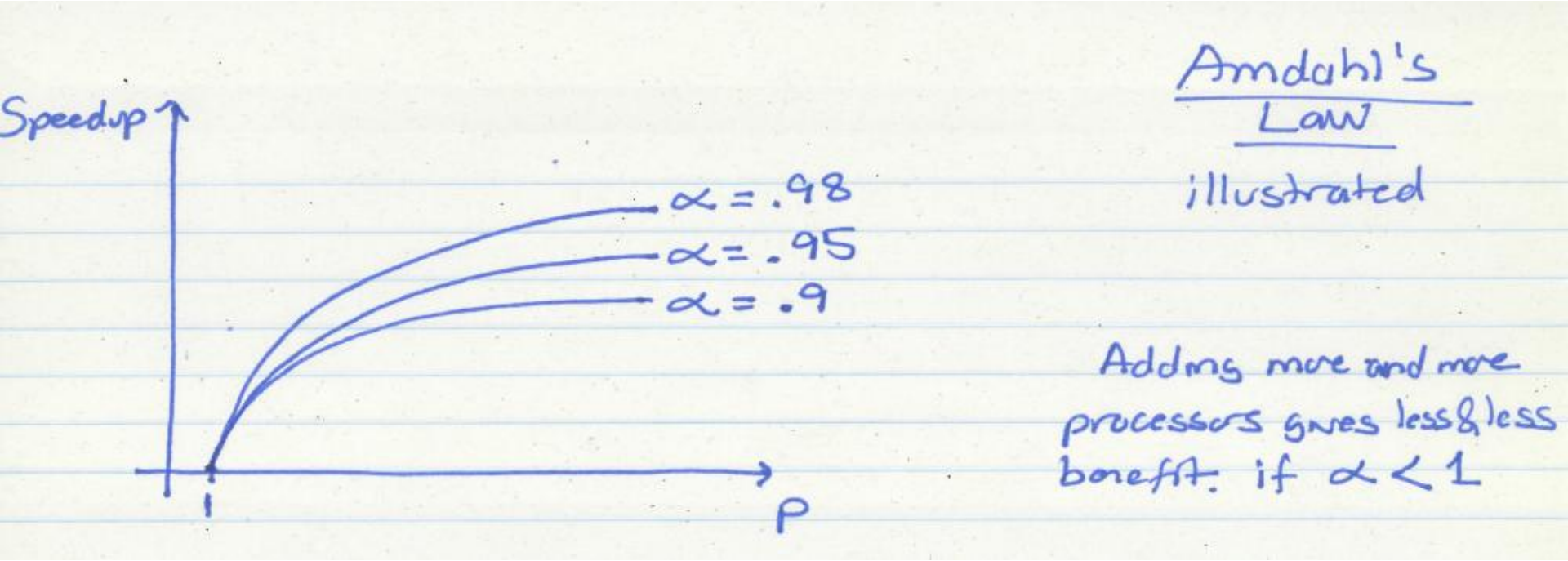
$(1-\alpha)$: non-parallelizable part

Amdahl's Law

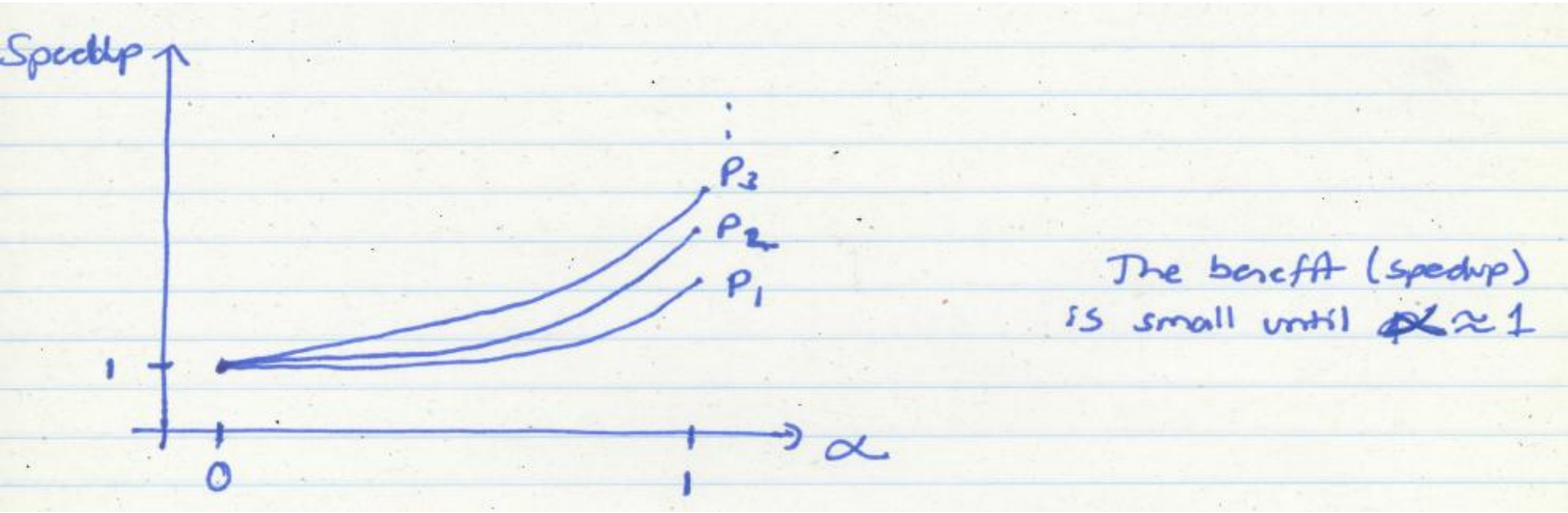


Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

Amdahl's Law Implication 1



Amdahl's Law Implication 2



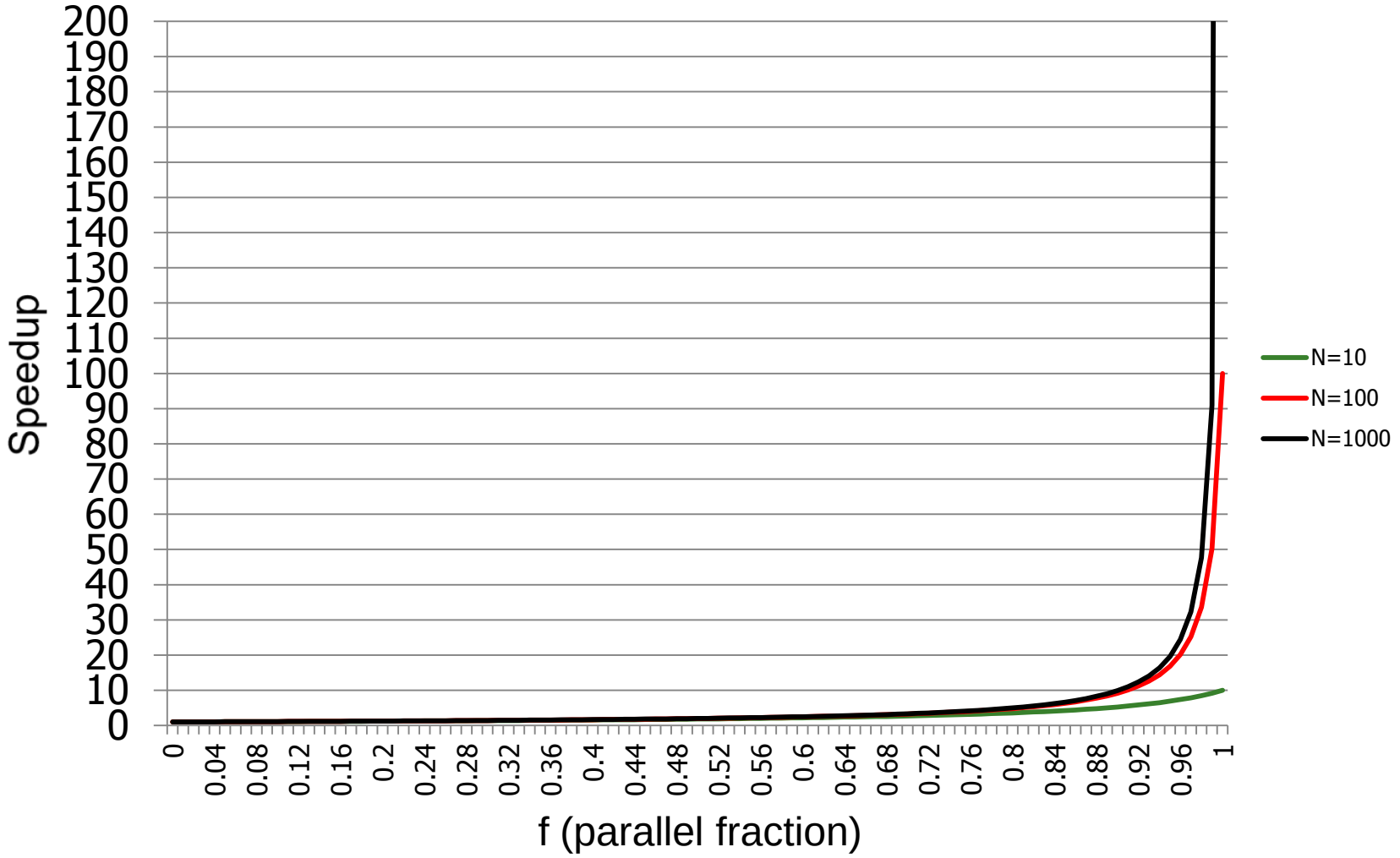
Caveats of Parallelism (II)

- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors

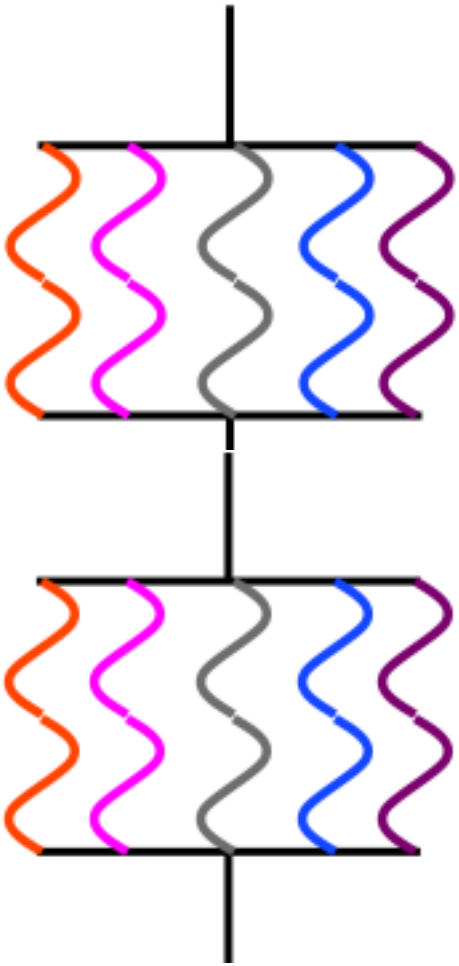
$$\text{Speedup} = \frac{1}{(1 - f) + \frac{f}{N}}$$

  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.
- Maximum speedup limited by serial portion: Serial bottleneck
- Parallel portion is usually not perfectly parallel
  - Synchronization overhead (e.g., updates to shared data)
  - Load imbalance overhead (imperfect parallelization)
  - Resource sharing overhead (contention among N processors)

Sequential Bottleneck



Why the Sequential Bottleneck?

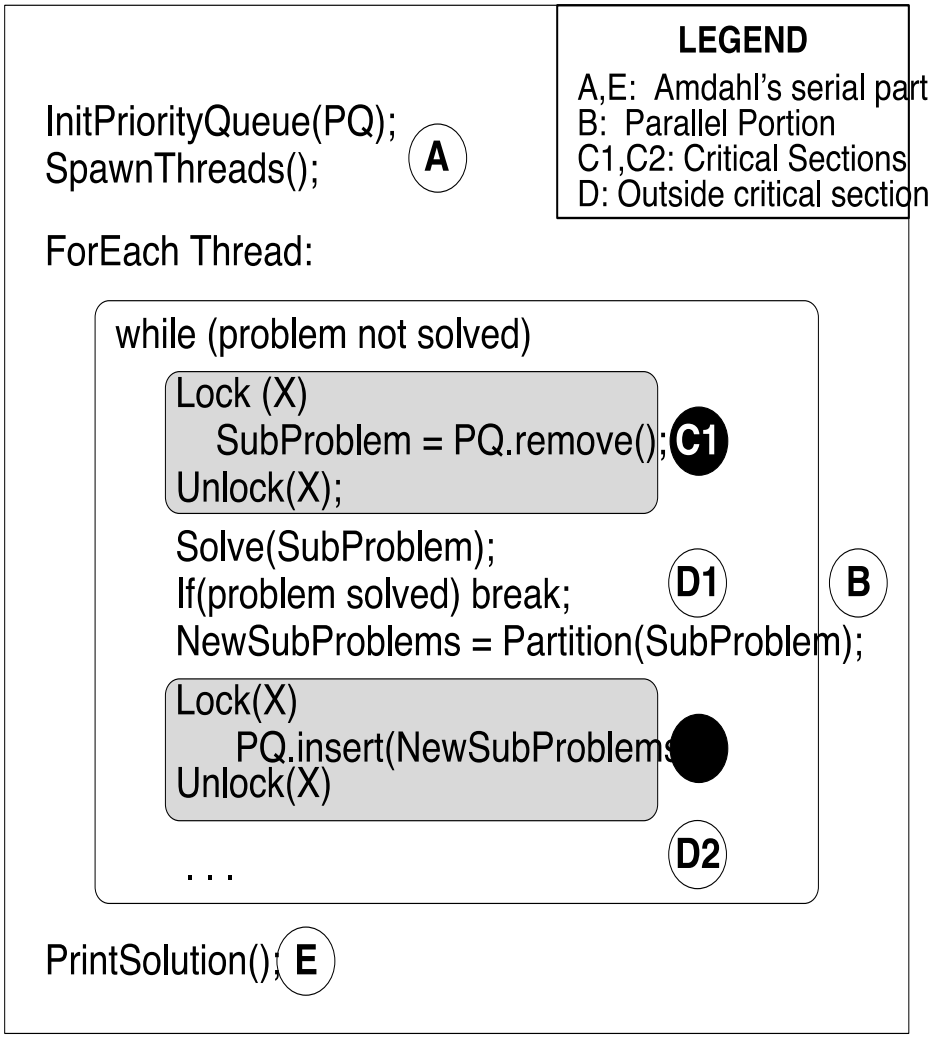


- Parallel machines have the sequential bottleneck
- Main cause: Non-parallelizable operations on data (e.g. non-parallelizable loops)

```
for ( i = 0 ; i < N; i++)
    A[i] = (A[i] + A[i-1]) / 2
```

- Single thread prepares data and spawns parallel tasks (usually sequential)

Another Example of Sequential Bottleneck



Bottlenecks in Parallel Portion

- Synchronization: Operations manipulating shared data cannot be parallelized
  - Locks, mutual exclusion, barrier synchronization
  - Communication: Tasks may need values from each other

  - Causes thread serialization when shared data is contended

- Load Imbalance: Parallel tasks may have different lengths
  - Due to imperfect parallelization or microarchitectural effects

  - Reduces speedup in parallel portion

- Resource Contention: Parallel tasks can share hardware resources, delaying each other
  - Replicating all resources (e.g., memory) expensive

  - Additional latency not present when each task runs alone

Difficulty in Parallel Programming

- Little difficulty if parallelism is natural
  - "Embarrassingly parallel" applications
  - Multimedia, physical simulation, graphics
  - Large web servers, databases?

- Difficulty is in
  - Getting parallel programs to work correctly
  - Optimizing performance in the presence of bottlenecks

- Much of **parallel computer architecture** is about
  - Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
  - Making programmer's job easier in writing correct and high-performance parallel programs