Multithreading Processors and Static Optimization Review

Adapted from Bhuyan, Patterson, Eggers, probably others

Class stuff

- CATME due today.
  - It's probably the most points per unit effort in the class. Do it.
- Quiz hopefully graded by tomorrow.
- HW5 out tomorrow, pushing due date to 4/23
- MS3 due Thursday 4/11
  - Meetings optional-ish.
- Lecture schedule:
  - 4/4: Multithreaded processors
  - 4/9: ISA
  - 4/11: Instruction scheduling (with a paper)
  - 4/16: No lecture (probably)
  - 4/18 Exam review
  - 4/23 Oral Presentations

Today

- Finish IA-64
- Review Complier stuff
- SMT

IA-64

- 128 64-bit registers
  - Use a register window similarish to SPARC
- 128 82 bit fp registers
- 64 1 bit predicate registers
- 8 64-bit branch target registers

Explicit Parallelism

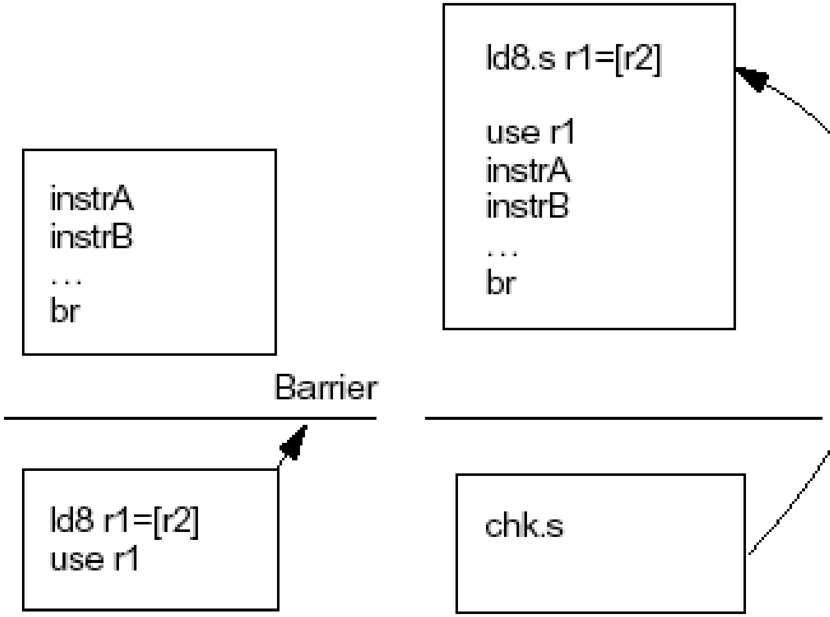
- Groups
  - Instructions which *could* be executed in parallel if hardware resources available.
- Bundle
  - Code format. 3 instructions fit into a 128-bit bundle.
  - 5 bits of template, 41*3 bits of instruction.
    - Template specifies what execution units each instruction requires.

Instructions

- 41 bits
    - 4 high order specify opcode (combined with template for bundle)
    - 6 low order bits specify predicate register number.
- Every instruction is predicated!
- Also NaT bits are used to handle speculated exceptions.

Speculative Load

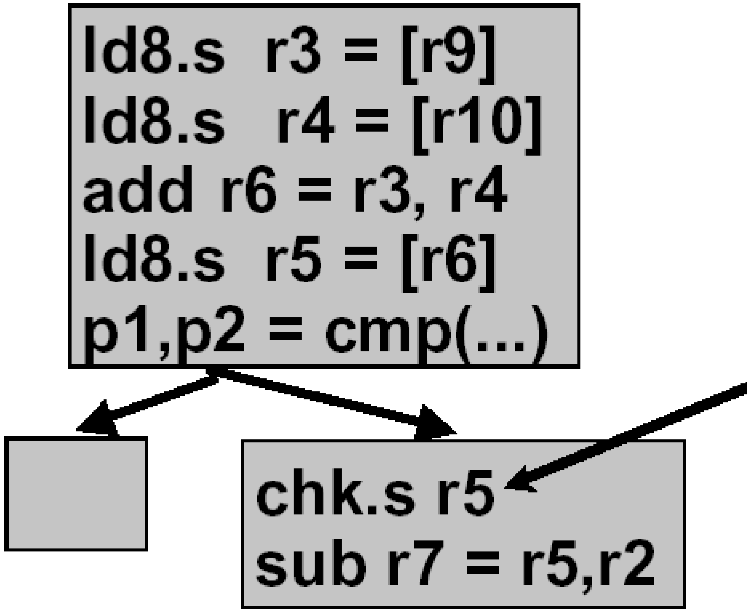
- Load instruction (ld.s) can be moved outside of a basic block even if branch target is not known
- Speculative loads does not produce exception - it sets the NaT
- Check instruction (chk.s) will jump to fix-up code if NaT is set



Traditional

IA-64

Propagation of NaT



```
ld8.s  r3 = [r9]
ld8.s   r4 = [r10]
add  r6 = r3, r4
ld8.s  r5 = [r6]
p1,p2 = cmp(...)
```

```
chk.s r5
sub r7 = r5,r2
```

Only single check required

NaT[reg] = NaT bit of reg

- IF (NaT[r3] || NaT[r4]) THEN set NaT[r6]
- IF (NaT[r6]) THEN set NaT[r5]
- Require check on NaT[r5] only since the NaT is inherited
- Reduce number of checks
- Fix-up will execute the entire chain

Advanced loads

- ld.a – Advanced load
  - Performs the load, puts it into the "ALAT"
    - If any following store writes to the same address, this is noted with a single bit.
    - When a ld.c is executed, if that bit is set, we refetch.
    - When chk.a is executed, if bit is set, fix up code is run. (Useful if load result already used.)
    - Both also cause any deferred exception to occur.

Software pipelining on IA-64

- Lots of tricks
  - – Rotating registers
  - – Special counters
- Often don’t need Prologue and Epilog.
  - – Special counters and prediction lets us only execute those instructions we need to.

Static optimization and IA64 review

- There are many important compiler techniques
  - We focused on hoisting loads.
  - But other include:
    - Register allocation (to reduce spills and fills)
    - Common sub-expression elimination
  - Wikipedia’s article on optimizing compliers provides a nice overview of standard optimizations.

How does static compare to dynamic?

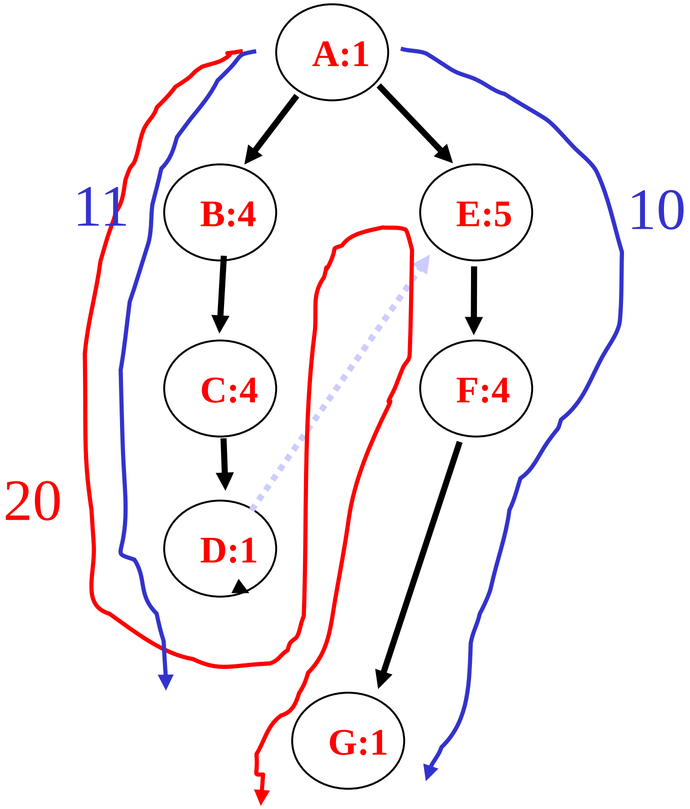
- Static
  - Has “a larger window” as it can see the whole program at once.
  - Can change the instructions executed.
- Dynamic
  - Has dynamic information
  - “Can be wrong”

That said problems are similar

- Static can reduce number of instructions, but for a given set of instructions, it is trying to optimize ILP just as dynamic does.
  - That will mean reordering instructions.
  - Suffers the same problems hardware does
    - Memory dependencies and branches.

Why can hoisting loads help?

- **add r15 = r2,r3** **//A**
- **mult r4 = r15,r2** **//B**
- **mult r4 = r4,r4** **//C**
- **st8 [r12] = r4** **//D**
- **ld8 r5 = [r15]** **//E**
- **div r6 = r5,r7** **//F**
- **add r5 = r6,r2** **//G**



Assume latencies are:

add, store: +0

mult, div: +3

ld: +4

IA 64 support

- Why is hoisting above a branch hard?
  - ______________________
  - IA64 solution?
    - Speculative load

- Why is hoisting above a store hard?
  - ____________________________
  - IA64 solution:
    - Advanced load

Other things we did

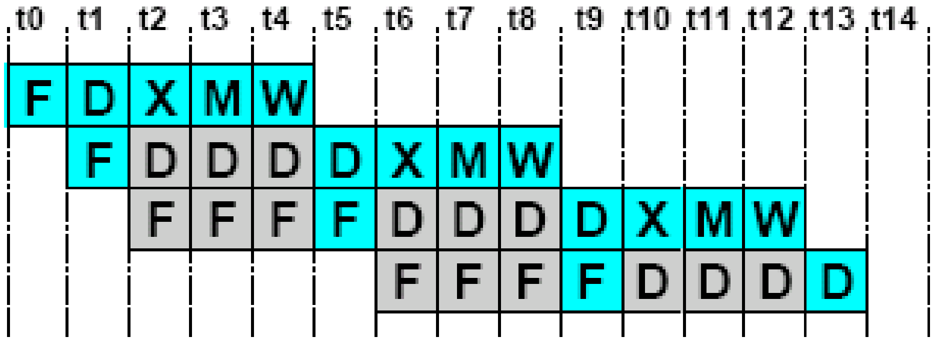
- Software pipelining
  - Idea, example
- Discussed register pressure
  - Idea, examples where optimization make it worse, IA64 “solution” (lots of registers…)
- Briefly discussed caching
  - Code size increase is a common side effect of complier optimizations
  - Optimizing for locality is good.

On to multi-threading

Pipeline Hazards

LW r1, 0(r2)
LW r5, 12(r1)
ADDI r5, r5, #12
SW 12(r1), r5

- Each instruction may depend on the next
  - Without forwarding, need stalls



	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14
LW r1, 0(r2)	F	D	X	M	W										
LW r5, 12(r1)		F	D	D	D	D	X	M	W						
ADDI r5, r5, #12			F	F	F	F	D	D	D	D	X	M	W		
SW 12(r1), r5							F	F	F	F	D	D	D	D	

- Bypassing/forwarding cannot completely eliminate interlocks or delay slots

Multithreading

- **How can we guarantee no dependencies between instructions in a pipeline?**
  - One way is to interleave execution of instructions from different program threads on same pipeline

Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe

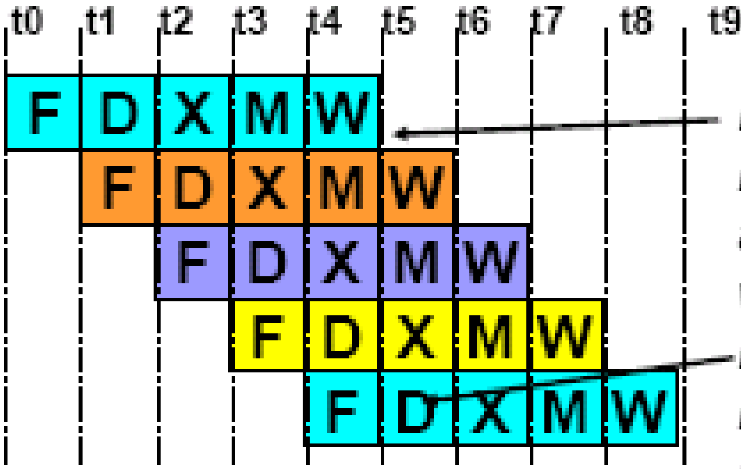
T1: LW r1, 0(r2)

T2: ADD r7, r1, r4

T3: XORI r5, r4, #12

T4: SW 0(r7), r5

T1: LW r5, 12(r1)

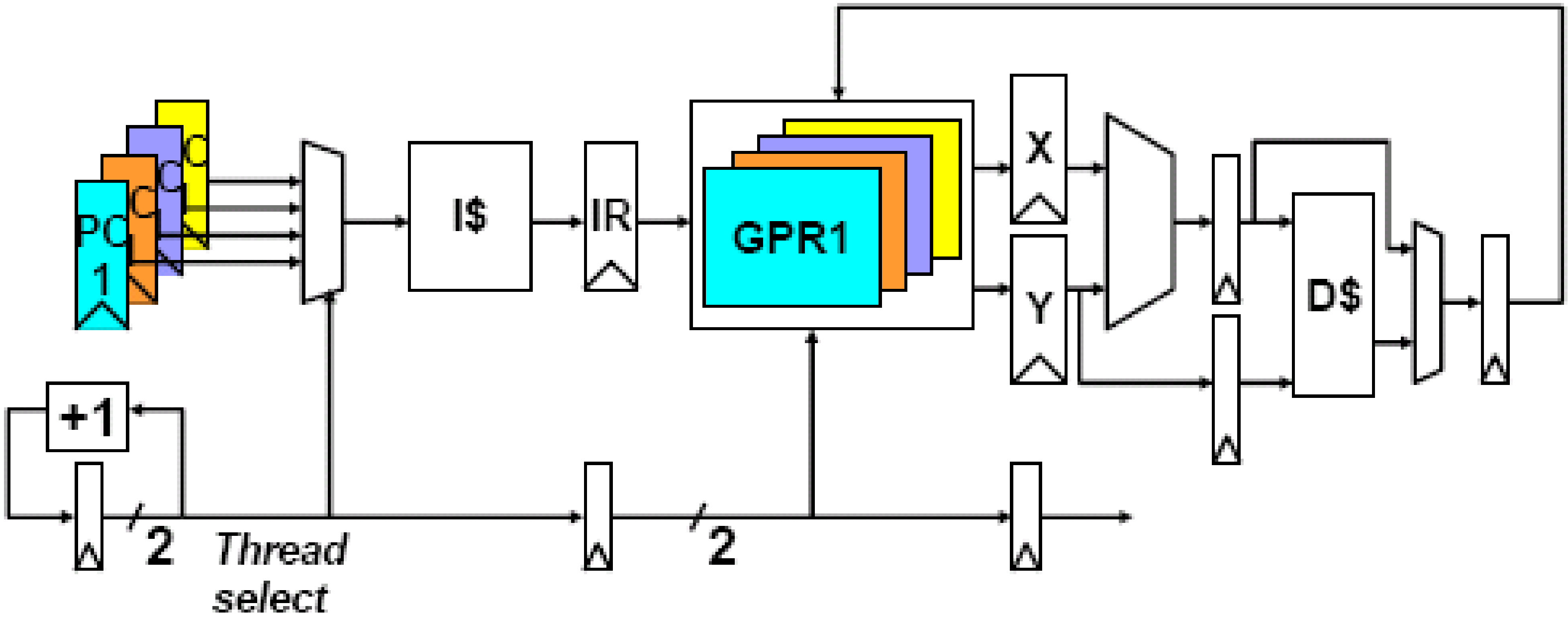


Last instruction in a thread always completes writeback before next instruction in same thread reads regfile

CDC 6600 Peripheral Processors (Cray, 1965)

- First multithreaded hardware
- 10 “virtual” I/O processors
- fixed interleave on simple pipeline
- pipeline has 100ns cycle time
- each processor executes one instruction every 1000ns
- accumulator-based instruction set to reduce processor state

Simple Multithreaded Pipeline



- **Have to carry thread select down pipeline to ensure correct state bits read/written at each pipe stage**

Multithreading Costs

- Appears to software (including OS) as multiple slower CPUs
- Each thread requires its own user state
  - GPRs
  - PC
- *Other costs?*

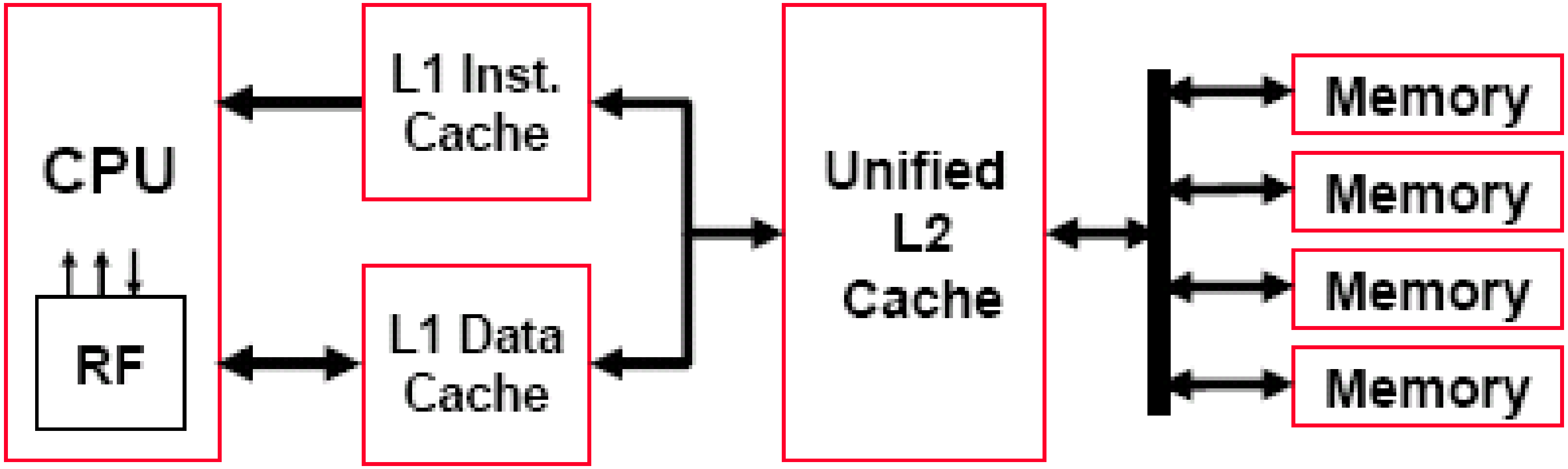
Thread Scheduling Policies

- **Fixed interleave *(CDC 6600 PPUs, 1965)***
  - each of N threads executes one instruction every N cycles
  - if thread not ready to go in its slot, insert pipeline bubble
- **Software-controlled interleave *(TI ASC PPUs, 1971)***
  - OS allocates S pipeline slots amongst N threads
  - hardware performs fixed interleave over S slots, executing whichever thread is in that slot
- **Hardware-controlled thread scheduling *(HEP, 1982)***
  - hardware keeps track of which threads are ready to go
  - picks next thread to execute based on hardware priority scheme

What “Grain” Multithreading?

- **So far assumed fine-grained multithreading**
  - CPU switches every cycle to a different thread
  - ***When does this make sense?***
- **Coarse-grained multithreading**
  - CPU switches every few cycles to a different thread
  - ***When does this make sense?***

Multithreading Design Choices



- **Context switch to another thread every cycle, or on hazard or L1 miss or L2 miss or network request**
- **Per-thread state and context-switch overhead**
- **Interactions between threads in memory hierarchy**

Denelcor HEP
(Burton Smith, 1982)

- **First commercial machine to use hardware threading in main CPU**
  - **120 threads per processor**
  - **10 MHz clock rate**
  - **Up to 8 processors**
  - **precursor to Tera MTA (Multithreaded Architecture)**

Tera MTA Overview

- **Up to 256 processors**
- **Up to 128 active threads per processor**
- **Processors and memory modules populate a 3D torus interconnection fabric**
- **Flat, shared main memory**
  - **No data cache**
  - **Sustains one main memory access per cycle per processor**
- **50W/processor @ 260MHz**

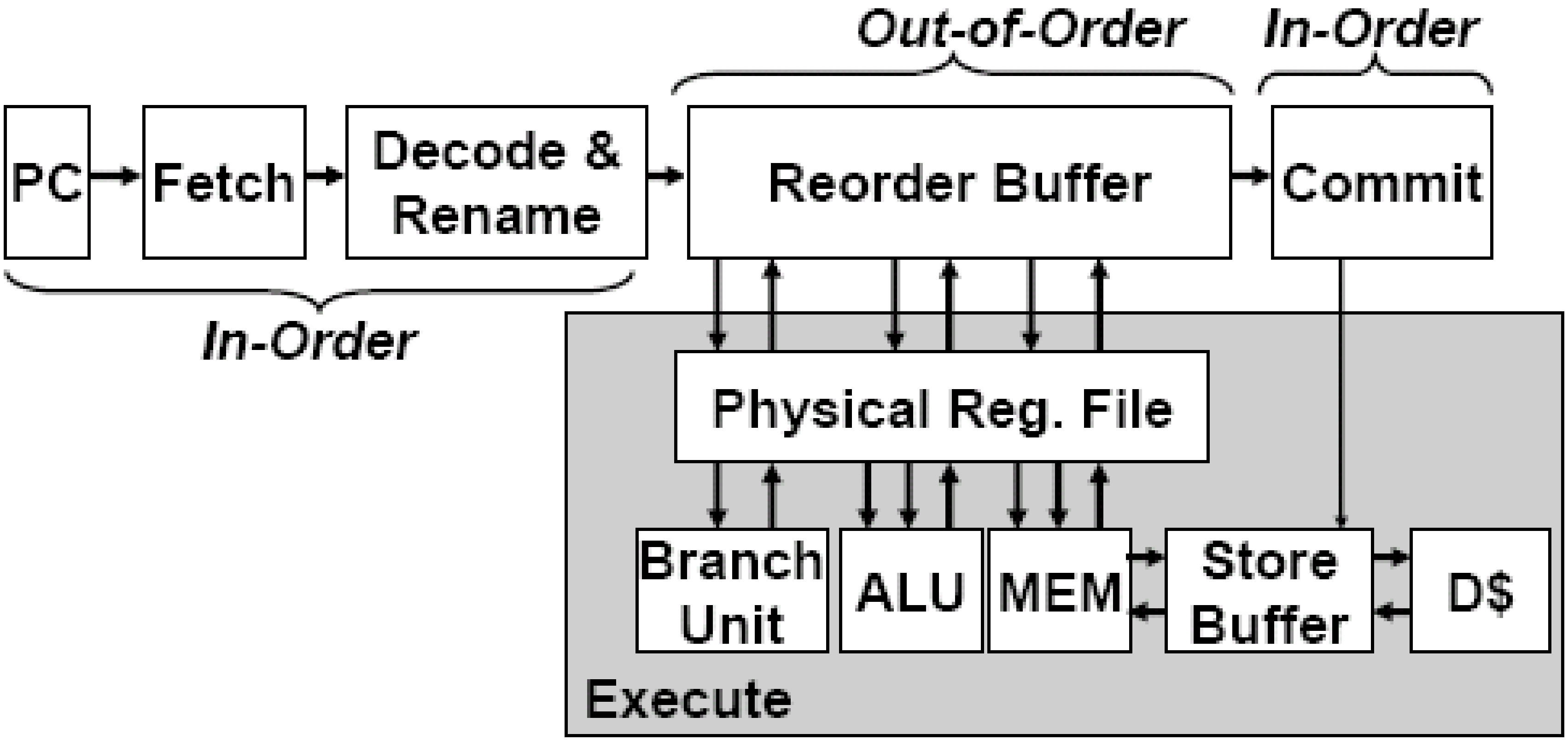
MTA Instruction Format

- Three operations packed into 64-bit instruction word (short VLIW)
- One memory operation, one arithmetic operation, plus one arithmetic or branch operation
- Memory operations incur ~150 cycles of latency
- Explicit 3-bit “lookahead” field in instruction gives number of subsequent instructions (0-7) that are independent of this one
  - c.f. Instruction grouping in VLIW
  - allows fewer threads to fill machine pipeline
  - used for variable- sized branch delay slots
- Thread creation and termination instructions

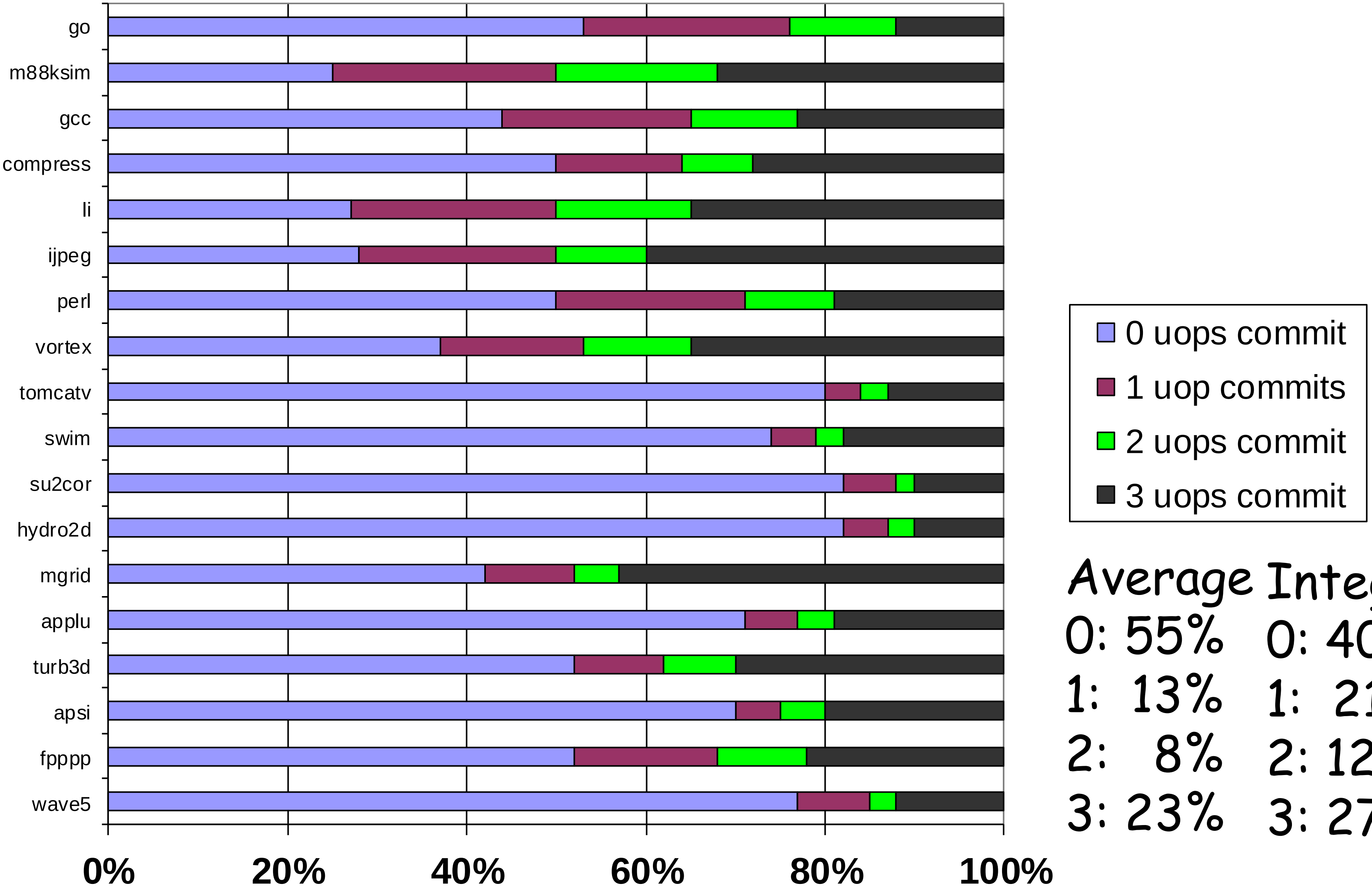
MTA Multithreading

- Each processor supports 128 active hardware threads
  - 128 SSWs, 1024 target registers, 4096 general-purpose registers
- Every cycle, one instruction from one active thread is launched into pipeline
- Instruction pipeline is 21 cycles long
- At best, a single thread can issue one instruction every 21 cycles
  - Clock rate is 260MHz, effective single thread issue rate is 260/21 = 12.4MHz

Speculative, Out-of-Order Superscalar Processor

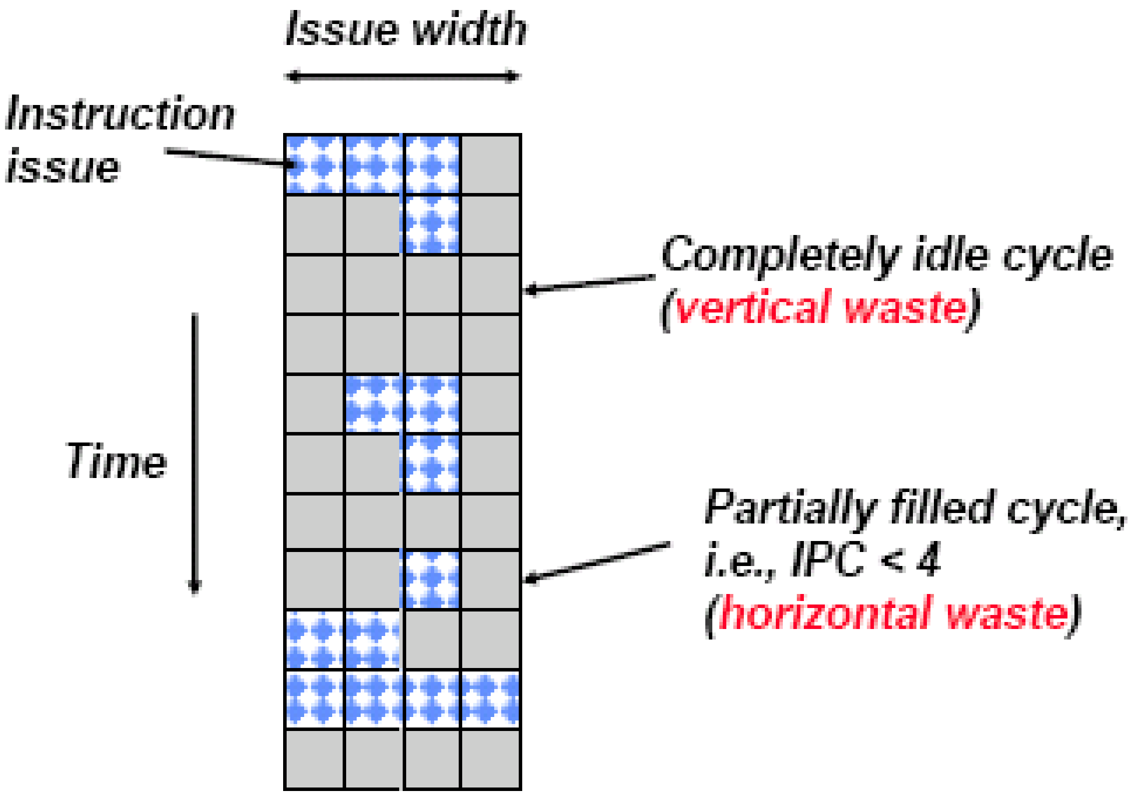


P6 Performance: uops commit/clock



Average	Integer
0: 55%	0: 40%
1: 13%	1: 21%
2: 8%	2: 12%
3: 23%	3: 27%

Superscalar Machine Efficiency



Issue width

Instruction issue

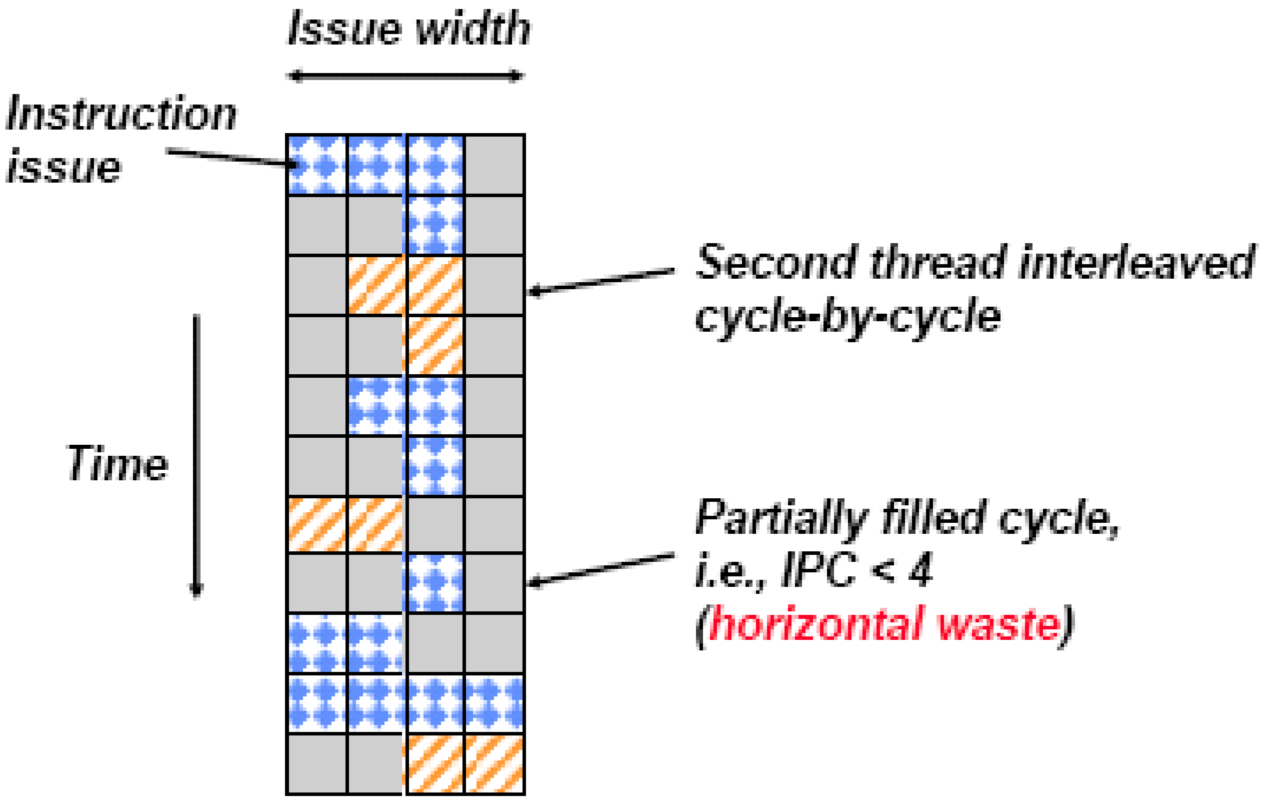
Time

Completely idle cycle (vertical waste)

Partially filled cycle, i.e., IPC < 4 (horizontal waste)

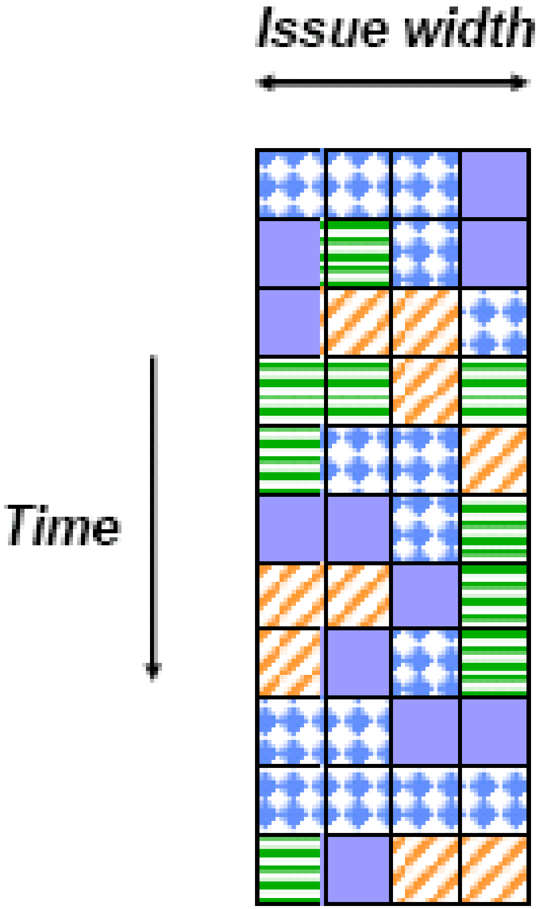
- ***Why horizontal waste?***
- ***Why vertical waste?***

Vertical Multithreading


Issue width
Instruction issue
Time
Second thread interleaved cycle-by-cycle
Partially filled cycle, i.e., IPC < 4 (horizontal waste)

- **Cycle-by-cycle interleaving of second thread removes vertical waste**

Ideal Multithreading for Superscalar



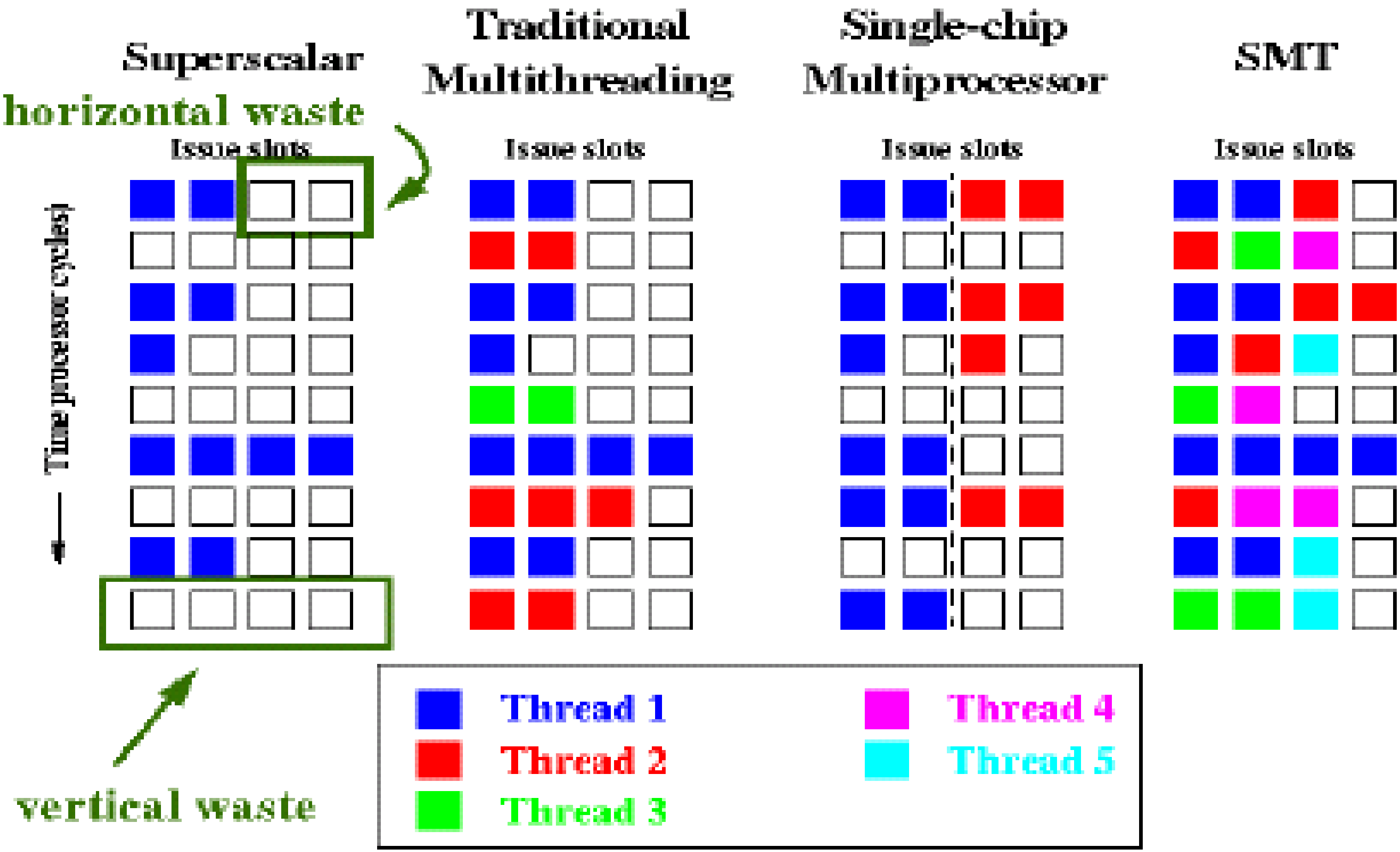
- **Interleave multiple threads to multiple issue slots with no restrictions**

Simultaneous Multithreading

- Add multiple contexts and fetch engines to wide out-of-order superscalar processor
  - [Tullsen, Eggers, Levy, UW, 1995]
- OOO instruction window already has most of the circuitry required to schedule from multiple threads
- Any single thread can utilize whole machine

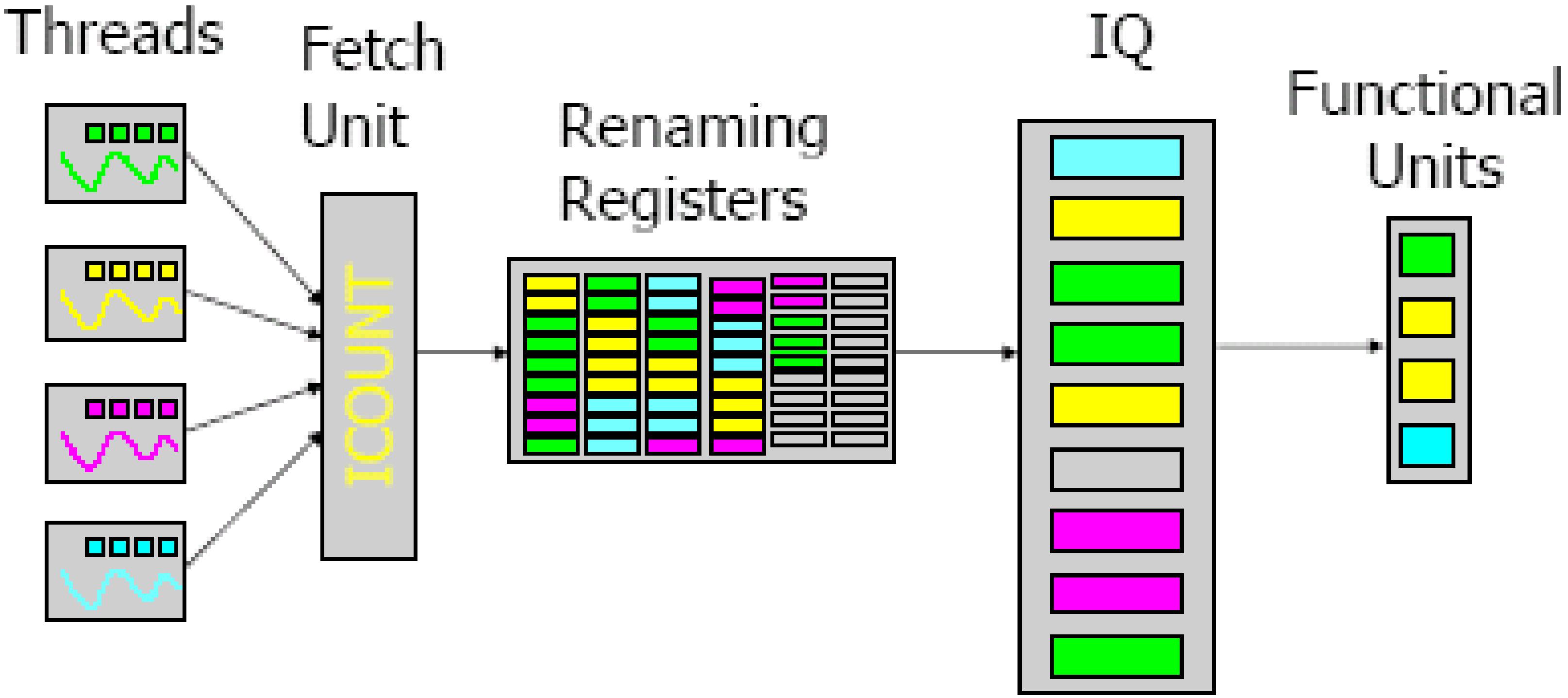
Comparison of Issue Capabilities

Courtesy of Susan Eggers



From Superscalar to SMT

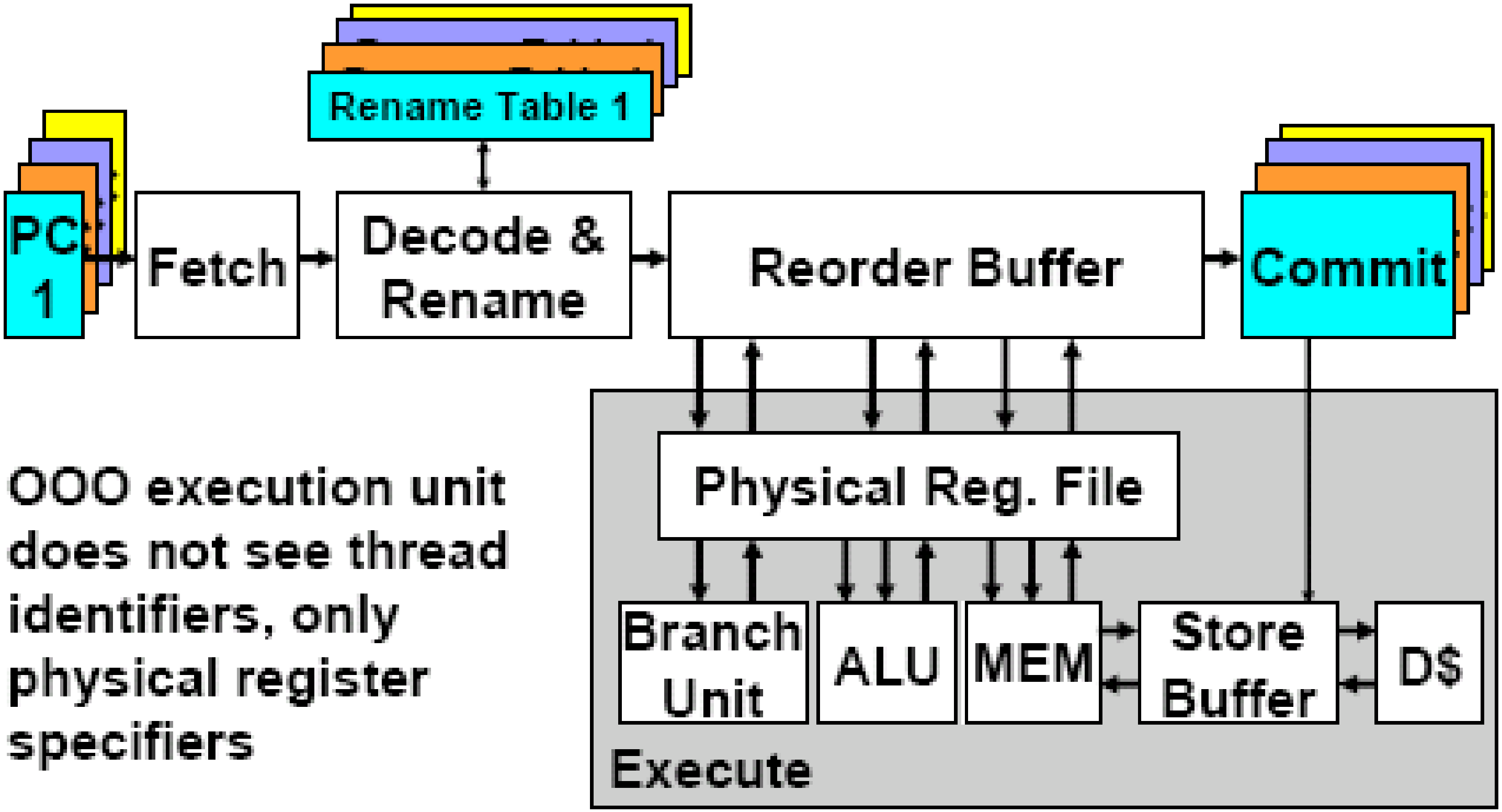
- **SMT is an out-of-order superscalar extended with hardware to support multiple executing threads**



From Superscalar to SMT

- Small items
  - per-thread program counters
  - per-thread return address stacks
  - per-thread bookkeeping for instruction retirement, trap & instruction dispatch queue flush
  - thread identifiers, e.g., with BTB & TLB entries

Simultaneous Multithreaded Processor



OOO execution unit does not see thread identifiers, only physical register specifiers

SMT Design Issues

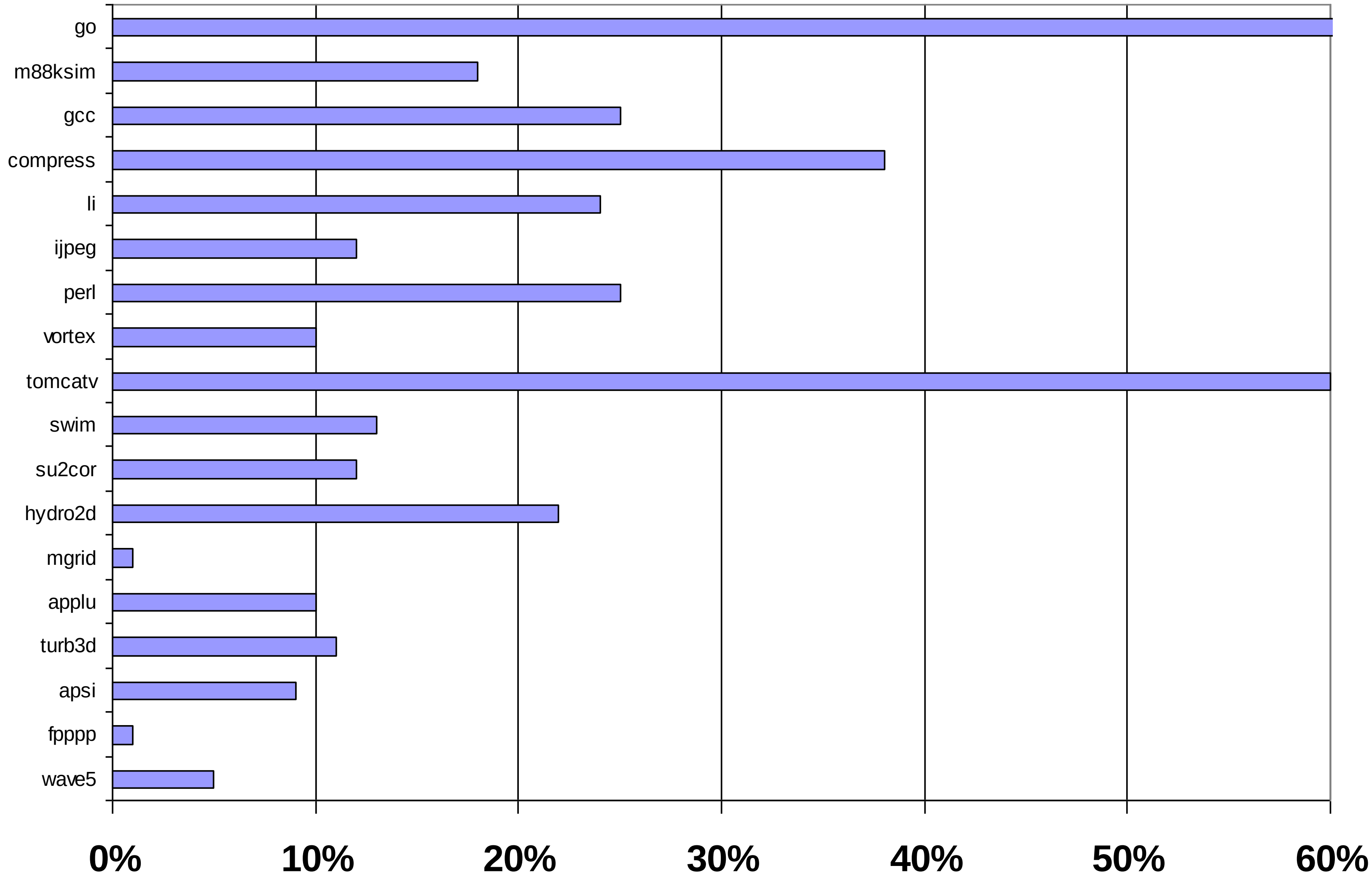
- **Which thread to fetch from next?**
  - **Don’t want to clog instruction window with thread with many stalls → try to fetch from thread that has fewest insts in window**
- **Locks**
  - **Virtual CPU spinning on lock executes many instructions but gets nowhere → add ISA support to lower priority of thread spinning on lock**

Intel Pentium-4 Xeon Processor

- **Hyperthreading == SMT**
- **Dual physical processors, each 2-way SMT**
- **Logical processors share nearly all resources of the physical processor**
  - **Caches, execution units, branch predictors**
- **Die area overhead of hyperthreading ~5 %**
- **When one logical processor is stalled, the other can make progress**
  - **No logical processor can use all entries in queues when two threads are active**
- **A processor running only one active software thread to run at the same speed with or without hyperthreading**
- **“Death by 1000 cuts”**

And things to think about: Wider in the front…

P6: (% instructions dispatched that do not commit)



1% to 60% instructions do not commit: 20% avg (30% integer)