CPEN 411: Computer Architecture

Slide Set #10: Loop Unrolling, Pentium 4, ILP Limits

Instructor: Mieszko Lis

Original slides: Prof. Tor Aamodt

Introduction to Slide Set 10

- In prior side sets we explored hardware only approaches to extracting more performance from software.
- In this slide set, we explore simple software approaches to extracting performance from a single thread. We will do this in the context of a single issue pipelined processor with multicycle floating-point operations (e.g., as in Slide Set 6)
- Then we will look at an example of a real processor that uses Tomasulo's algorithm, multiple instruction issue and a reorder buffer—the Pentium 4.
- Finally, we will consider the limits of instruction level parallelism found in software.

Learning Objectives

- After we finish this slide set in lectures you will be able to
  - List two software approaches to enhance instruction level parallelism (scheduling, loop unrolling)
  - Apply loop unrolling to a simple loop for a scalar pipeline.
  - Apply loop unrolling to a simple loop for a VLIW pipeline.
  - Explain why the software based approaches do not completely replace the need for hardware approaches.
  - List one or two details of Pentium 4 microarchitecture. Explain how they impact performance of Pentium 4.
  - Explain sources of limits to instruction level parallelism (ILP)

Running Example

- This code adds a scalar to a vector:

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

- Assume a single-issue, in order pipeline with the following latencies in all examples:

Instruction producing result	*Instruction using result*	*Execution in cycles*	*Latency in cycles*
FP ALU op	**Another FP ALU op**	**4**	**3**
FP ALU op	**Store double**	**3**	**2**
Load double	**FP ALU op**	**1**	**1**
Load double	**Store double**	**1**	**0**
Integer op	**Integer op**	**1**	**0**
Integer op	**Branch op**	**1**	**1**

FP Loop: Where are the Hazards?

- First translate into MIPS code:

```
Loop:   L.D      F0,0(R1)     ; Regs[F0]=vector element
        ADD.D    F4,F0,F2     ; add scalar from Regs[F2]
        S.D      F4,0(R1)     ; store result
        DADDI    R1,R1,#-8    ; decrement pointer 8B (DW)
        BNE      R1,R2,Loop   ; branch Regs[R1]!=Regs[R2]
```

1st Question we wish to answer: Where are the stalls?

2nd Question we wish to answer: How to reduce stalls by adjusting the software.

FP Loop: Showing Stalls
(in order, single-issue pipeline)

```
1 Loop:  L.D     F0,0(R1)   ; Regs[F0]=vector element
2        stall
3        ADD.D   F4,F0,F2   ; add scalar in Regs[F2]
4        stall
5        stall
6        S.D     F4,0(R1)   ; store result
7        DADDI   R1,R1,#-8  ; decrement pointer 8B (DW)
8        stall
9        BNE     R1,R2,Loop ; branch Regs[R1]!=Regs[R2]
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Integer op	Branch op	1

- 9 clocks: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

```
1 Loop: L.D    F0,0(R1)
2       DADDI   R1,R1,#-8
3       ADD.D  F4,F0,F2
4       stall
5       stall
6       S.D    F4,8(R1)
7       BNEZ   R1,Loop
```

Swap DADDUI and S.D

7 clocks, but just 3 for execution (L.D,ADD.D,S.D), 4 for loop overhead (DADDI, BNE, two stalls);

How to make faster?

Loop Unrolling

- We can reduce loop overhead by making multiple copies of loop body inside loop.
- This is known as loop unrolling.
- In the next few slides we will look at how to apply loop unrolling to the example on the last few slides.

- The steps we will apply are:
  1. Make copy of loop body and remove overhead instructions
  2. Rename registers in the assembly code to remove name dependencies (unlike Tomasulo’s algorithm, this renaming is done in software rather than in hardware)
  3. Schedule instructions to reduce stalls ensuring that new code computes same result as original code.

Unroll Loop Four Times
(straightforward way)

1 cycle stall

2 cycles stall

Rewrite loop to minimize stalls?

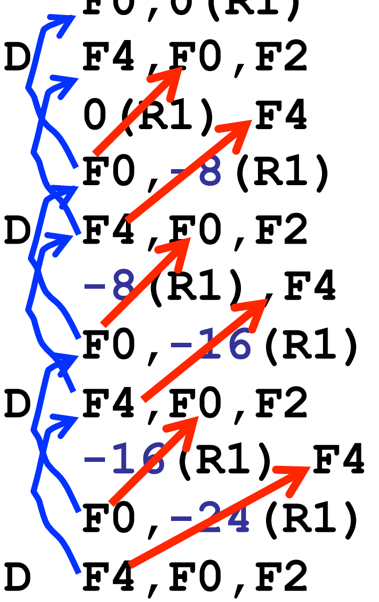
```
1  Loop: L.D     F0,0(R1)
2        ADD.D   F4,F0,F2
3        S.D     F4,0(R1)      ;drop DADDUI & BNE
4        L.D     F0,-8(R1)
5        ADD.D   F4,F0,F2
6        S.D     F4,-8(R1      ;drop DADDUI & BNE
7        L.D     F0,-16(R1)
8        ADD.D   F4,F0,F2
9        S.D     F4,-16(R1)    ;drop DADDUI & BNE
10       L.D     F0,-24(R1)
11       ADD.D   F4,F0,F2
12       S.D     F4,-24(R1)
13       DADDUI  R1,R1,#-32    ;alter to -4*8
14       BNE     R1,R2,LOOP
```

1 cycle stall

14 + 4 x (1+2) + 1 = 27 clock cycles, or 6.75 cycles per iteration

Assumes # of loop iterations is multiple of 4

Where are the name dependencies?



```
Loop:L.D     F0,0(R1)
     ADD.D   F4,F0,F2
     S.D     0(R1),F4        ;drop DADDUI & BNE
     L.D     F0,-8(R1)
     ADD.D   F4,F0,F2
     S.D     -8(R1),F4       ;drop DADDUI & BNE
     L.D     F0,-16(R1)
     ADD.D   F4,F0,F2
     S.D     -16(R1),F4      ;drop DADDUI & BNE
     L.D     F0,-24(R1)
     ADD.D   F4,F0,F2
     S.D     -24(R1),F4
     DADDUI  R1,R1,#-32      ;alter to 4*8
     BNE     R1,R2,LOOP
```

anti-dependencies on R1 omitted for clarity

How can we remove them?

Apply Renaming

```
Loop:L.D     F0,0(R1)
     ADD.D   F4,F0,F2
     S.D     0(R1),F4        ;drop DADDUI & BNE
     L.D     F6,-8(R1)
     ADD.D   F8,F6,F2
     S.D     -8(R1),F8       ;drop DADDUI & BNE
     L.D     F10,-16(R1)
     ADD.D   F12,F10,F2
     S.D     -16(R1),F12     ;drop DADDUI & BNE
     L.D     F14,-24(R1)
     ADD.D   F16,F14,F2
     S.D     -24(R1),F16
     DADDUI  R1,R1,#-32      ;alter to 4*8
     BNE     R1,R2,LOOP
```

Unrolled Loop That Minimizes Stalls

```
1  Loop:L.D     F0,0(R1)
2       L.D     F6,-8(R1)
3       L.D     F10,-16(R1)
4       L.D     F14,-24(R1)
5       ADD.D   F4,F0,F2
6       ADD.D   F8,F6,F2
7       ADD.D   F12,F10,F2
8       ADD.D   F16,F14,F2
9       S.D     0(R1),F4
10      S.D     -8(R1),F8
11      DADDUI  R1,R1,#-32
12      S.D     16(R1),F12; 16 = -16 + 32
13      S.D     8(R1),F16 ; 8  = -24 + 32
14      BNE     R1,R2, LOOP
```

- Assumptions we made:
  - OK to move store past DADDUI even though DADDUI changes register
  - OK to move loads before stores: Needed to check if we get the right data
  - In general, compiler needs to check for both register and memory dependencies

14 clock cycles, or 3.5 per iteration

Compiler Based Scheduling (Static Scheduling)

- Compiler analyzes dependencies in program
- Tries to schedule to avoid hazards that cause performance losses

- Easy to determine for registers (fixed names)

- Hard for memory ("memory disambiguation" problem):
  - Example:

    ```
    SD R1, 100(R4)
    LD R2, 20(R6)
    ```
  - Is there a data dependence?
  - Equivalent question: "Does 100(R4) = 20(R6)"?
  - Possible answers: definitely yes, maybe, definitely no
  - If "by inspection" we can see the answer is "definitely no", we can move LD before SD. Compilers answer this question using complex algorithms beyond the scope of this course

Compiler Perspectives on Code Movement

- Our example required compiler to know that if R1 doesn’t change then:

  `0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)`

  There were no dependencies between some loads and stores so they could be moved by each other.

Details Steps to Unroll

- Determine unrolling the loop would be useful by finding that the loop iterations were independent
- Eliminate extra test and branch instructions and adjust the loop termination and iteration code
- Rename registers to avoid name dependencies
- Determine loads and stores in unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent
  - Requires analyzing memory addresses and finding that they do not refer to the same address.
- Schedule the code, preserving any dependences needed to yield same result as the original code

Unknown Loop Bounds

- Do not usually know upper bound of loop
- Suppose it is “n”, and we would like to unroll the loop to make “k” copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
  - 1st executes (n mod k) times and has a body that is the original loop
  - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
  - For large values of n, most of the execution time will be spent in the unrolled loop

Example
(using C syntax for clarity, not fully optimized)

```
// Original Loop
for( i=0; i < n; i++ ) {
    A[i] = B[i];
}
```

```
// After applying loop unrolling:
// first loop : original loop body, iterates n mod k times
for( i=0; i < n%4; i++ ) {
    A[i]=B[i];
}
// second loop : unroll original loop 4 times, iterates n/k times
for( i=0; i < n/4; i++ ) {
    A[4*i + n%4 + 0]=B[4*i + n%4 + 0];
    A[4*i + n%4 + 1]=B[4*i + n%4 + 1];
    A[4*i + n%4 + 2]=B[4*i + n%4 + 2];
    A[4*i + n%4 + 3]=B[4*i + n%4 + 3];
}
```

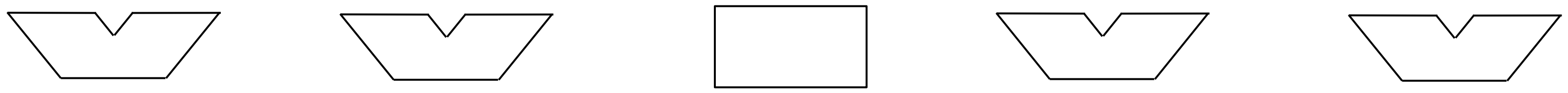
Very Long Instruction Word (VLIW)

VLIW: instructions that "encode" multiple operations.

The hardware executes the entire instruction "at once" on parallel function units in execute stage.

The "long instructions" encode the fact that the operations are independent. So, the hardware does not need to dynamically figure this out. This can save area and power and is now popular in embedded processors (e.g., Texas Instruments C6x series of DSPs).

Inst N:	ADD.D F1,F1,F4	**NOP**	L.D F7, 0(R1)	DSUBI R2,R2,#1	**NOP**
Inst N+1:	ADD.D F1,F1,F7	MUL.D F4,F5,F6	**NOP**	DSUBI R2,R2,#1	BEQZ R2, Loop



Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
L.D F0,0(R1)	L.D F6,-8(R1)				
L.D F10,-16(R1)	L.D F14,-24(R1)				
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		
		ADD.D F20,F18,F2	ADD.D F24,F22,F2		
S.D 0(R1),F4	S.D -8(R1),F8	ADD.D F28,F26,F2			
S.D -16(R1),F12	S.D -24(R1),F16				
S.D -32(R1),F20	S.D -40(R1),F24			DSUBUI R1,R1,#48	
S.D -0(R1),F28				BNEZ R1,LOOP	

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration (~2.7X)

23 ops / 9 cycles = 2.5 ops per clock, 2.5/5 = 50% efficiency

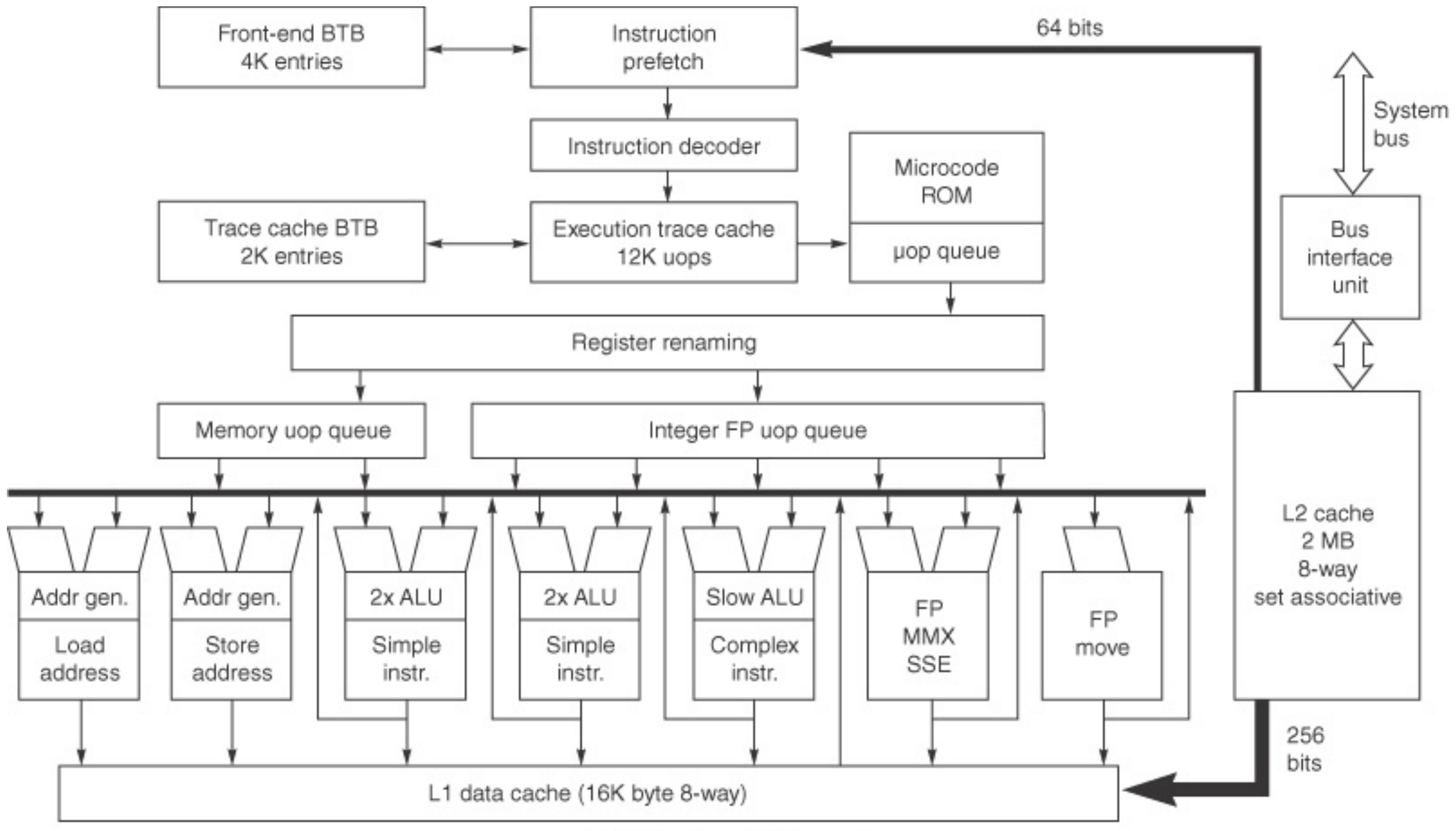
Note: Need more registers in VLIW (15 vs. 6)

Pentium 4 History

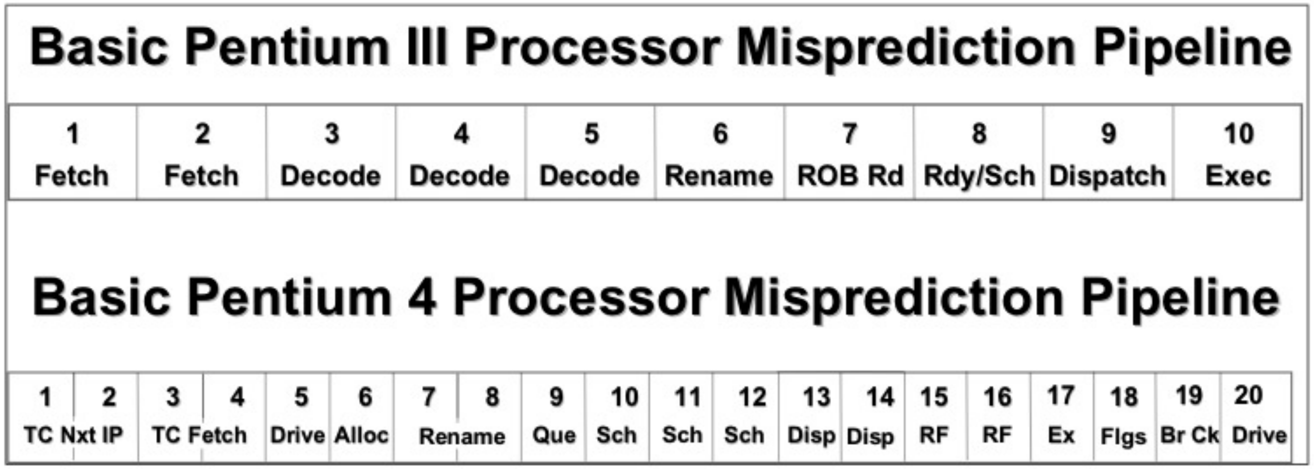
- Willamette (Nov 2000)
  - Process technology: 180 nm
  - Pipeline stages: 21
- Northwood (Jan 2002)
  - Process technology: 130 nm
  - Pipeline stages: 21
  - Hyperthreading support added (or finally debugged) for later versions
- Prescott (Feb 2004)
  - Process technology: 90 nm
  - Pipeline stages: 31
  - Hyperthreading
- Cedar Mill (early 2006)
  - Prescott, but in 65 nm
  - End of road for Pentium 4 Microarchitecture… consuming too much power

[source (for dates): Wikipedia]

Pentium 4 (“Prescott”) Block Diagram



Pentium 4 Pipeline ("Willamette")

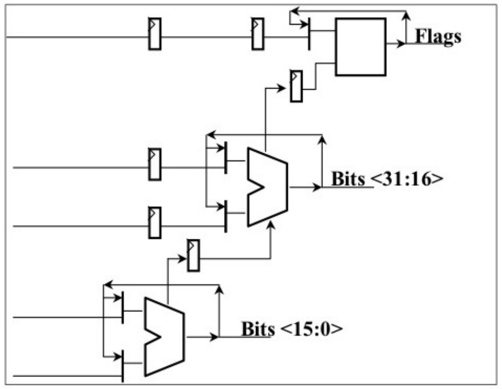


Basic Pentium III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

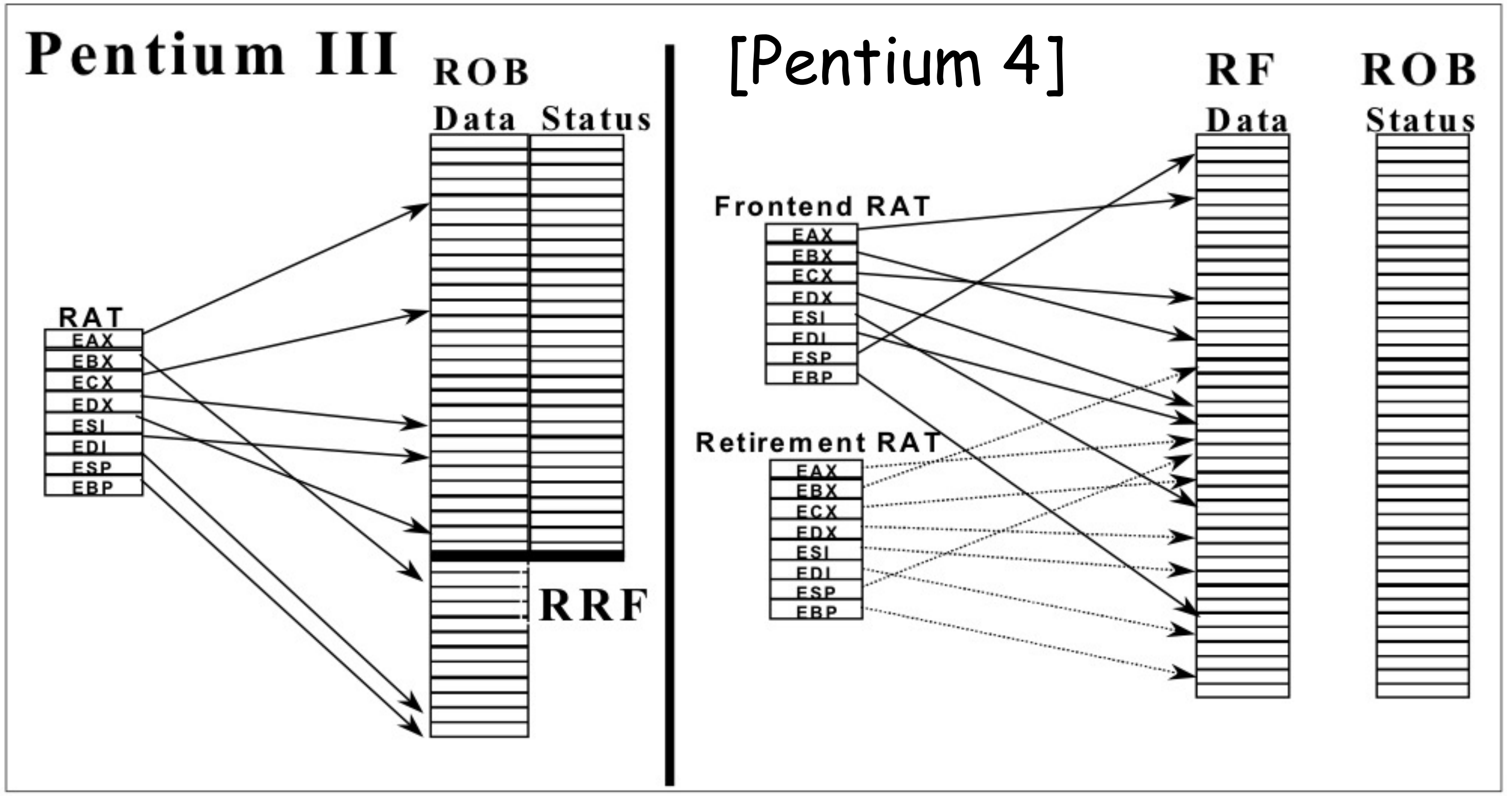
Basic Pentium 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive



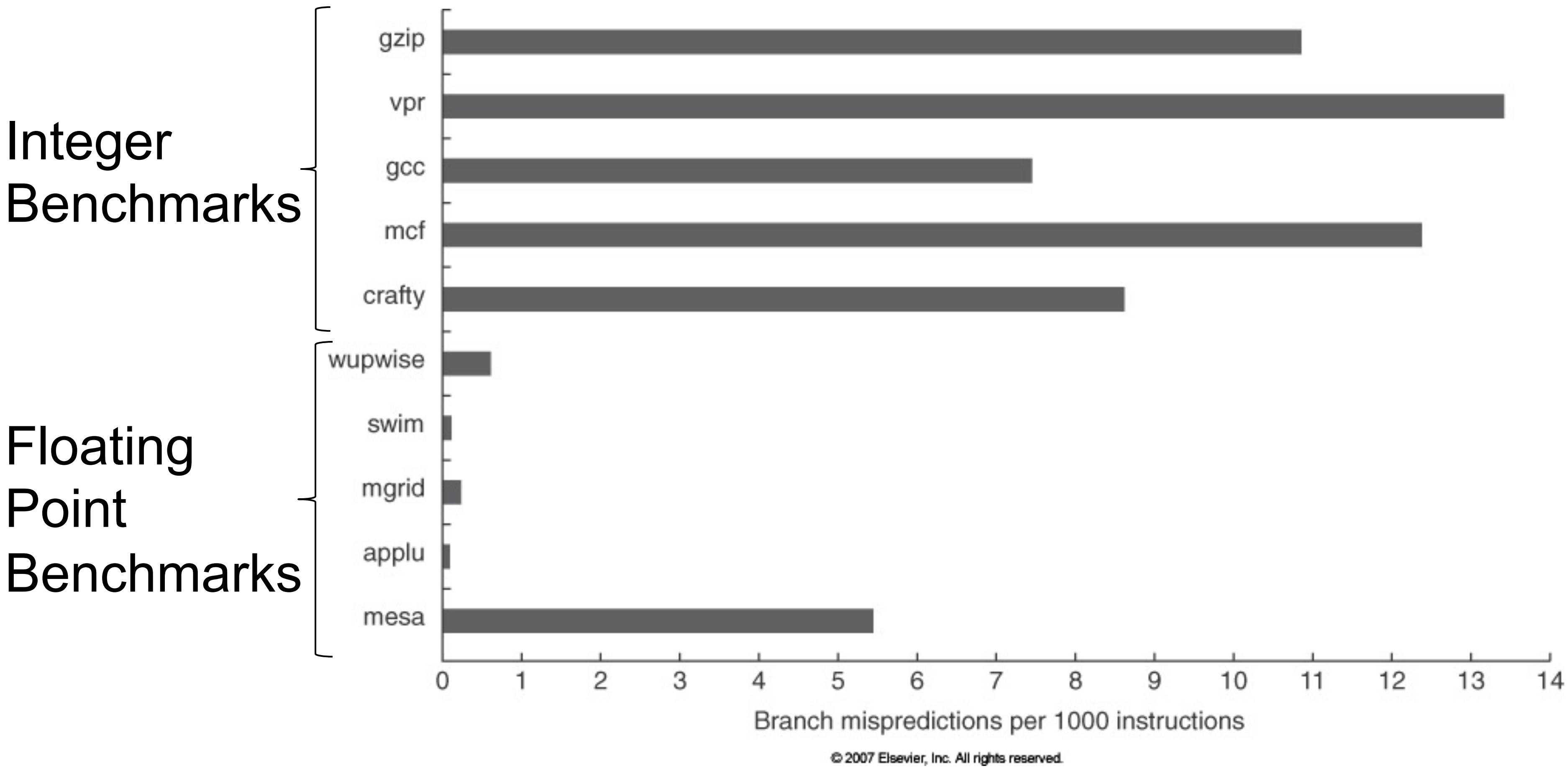
[Source: "The Microarchitecture of the Pentium® 4 Processor", Intel Technology Journal, Vol. 5 Issue 1 (February 2001) special issue on the Pentium® 4 Processor]

Register Renaming



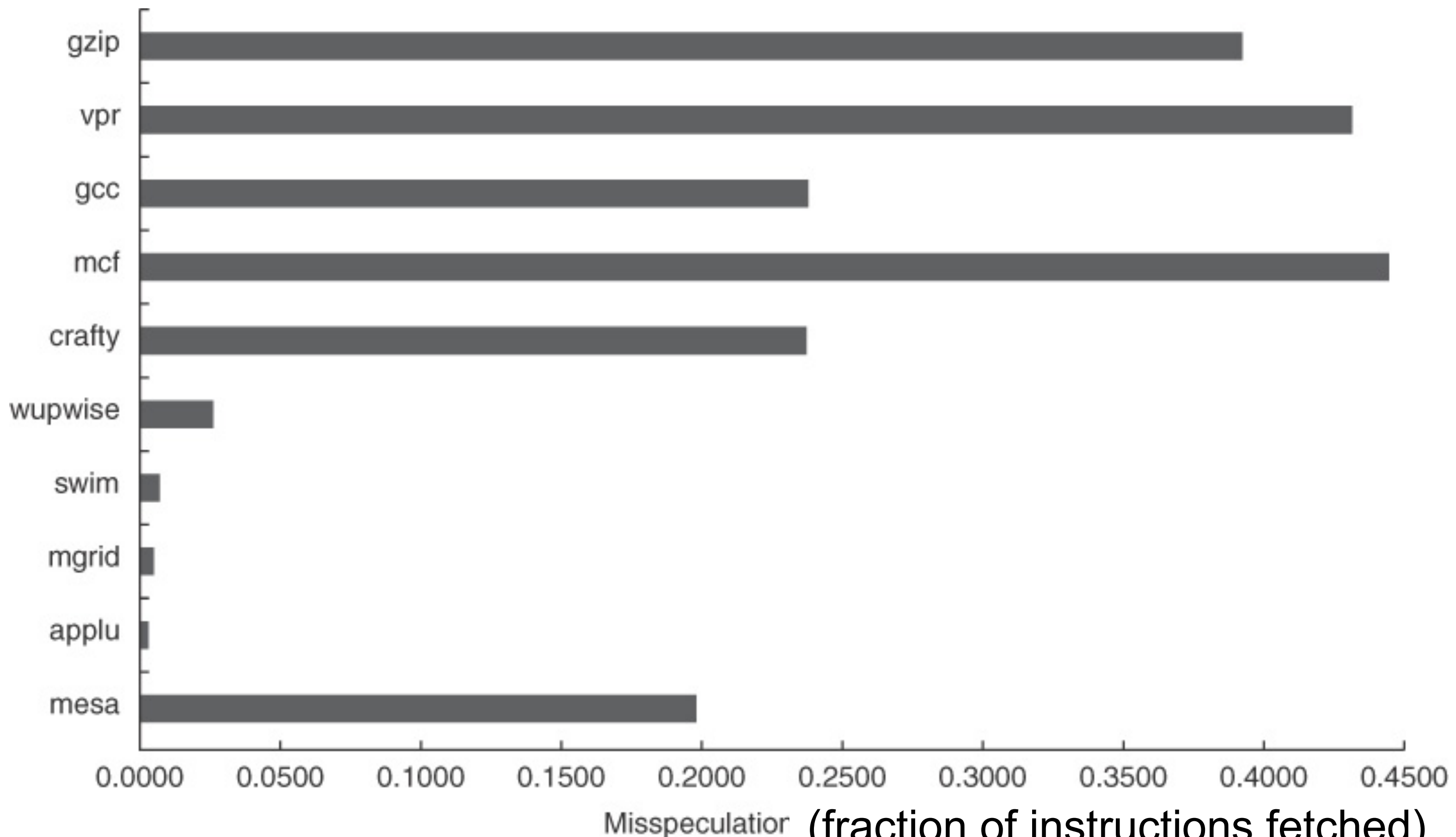
[Source: “The Microarchitecture of the Pentium® 4 Processor”, Intel Technology Journal, Vol. 5 Issue 1 (February 2001) special issue on the Pentium® 4 Processor]

Branch Mispredictions



Floating point benchmarks: Less branches, and branches are more predictable.

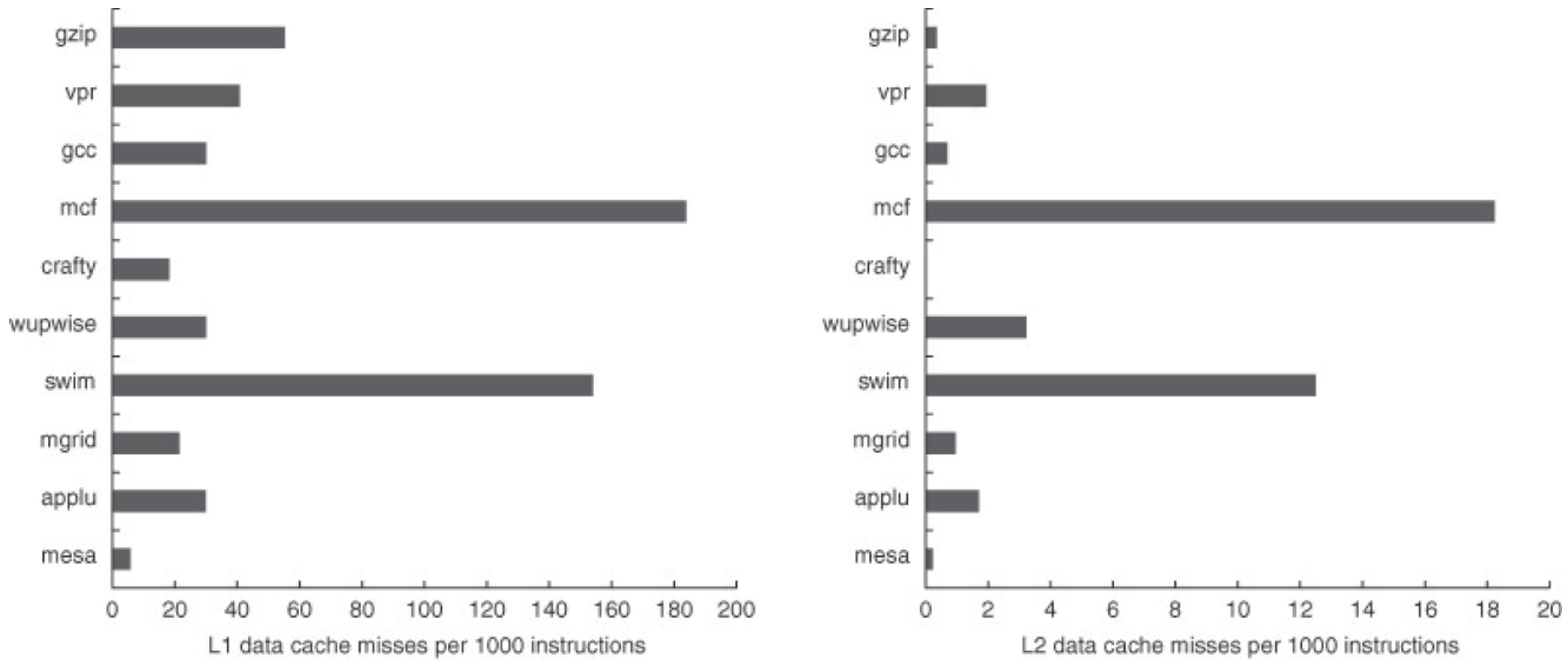
Instructions “thrown out” due to branch mispredictions (or jump target mispredictions)



(fraction of instructions fetched)

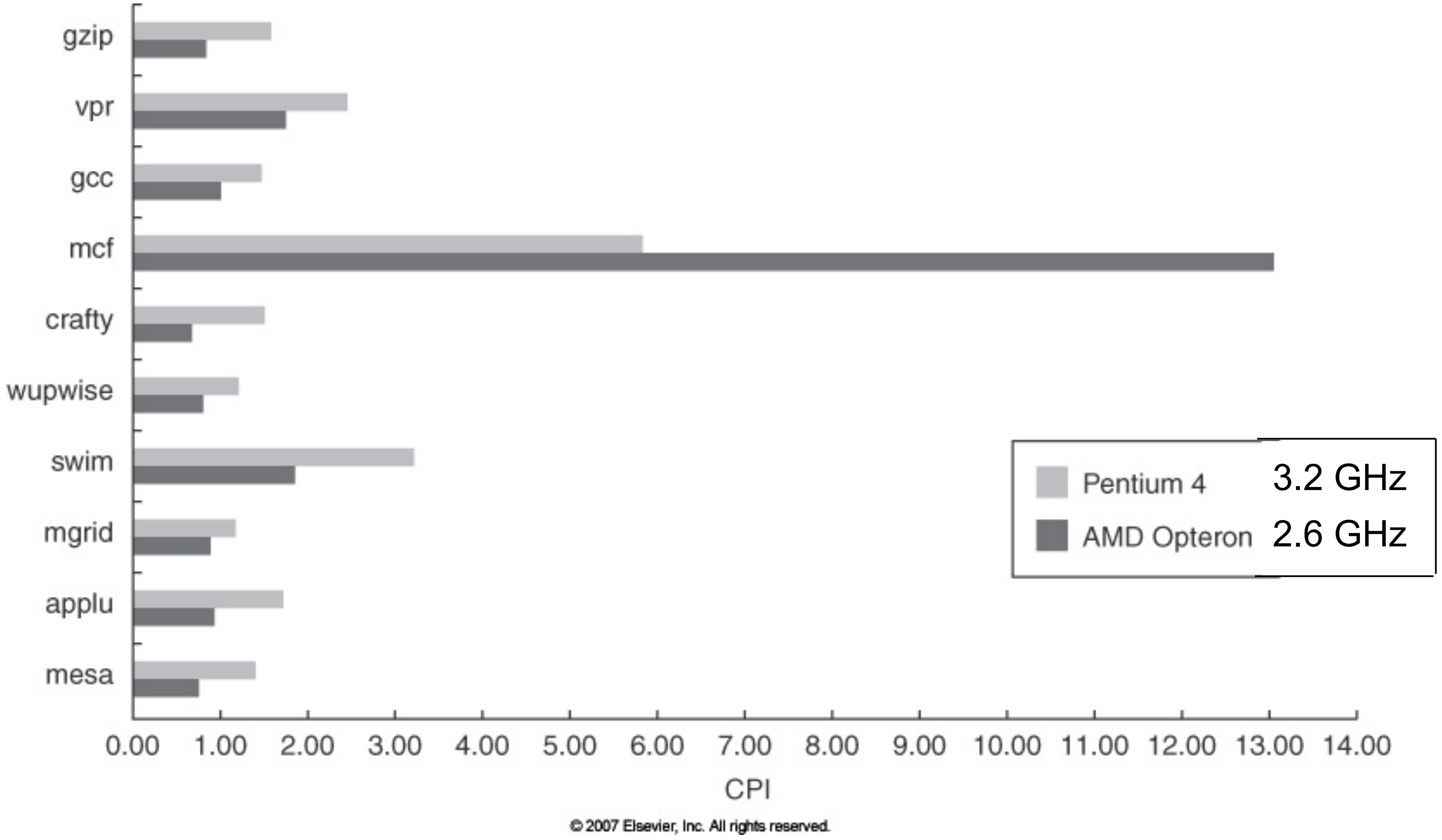
Note similarity to branch prediction accuracy

Cache Misses



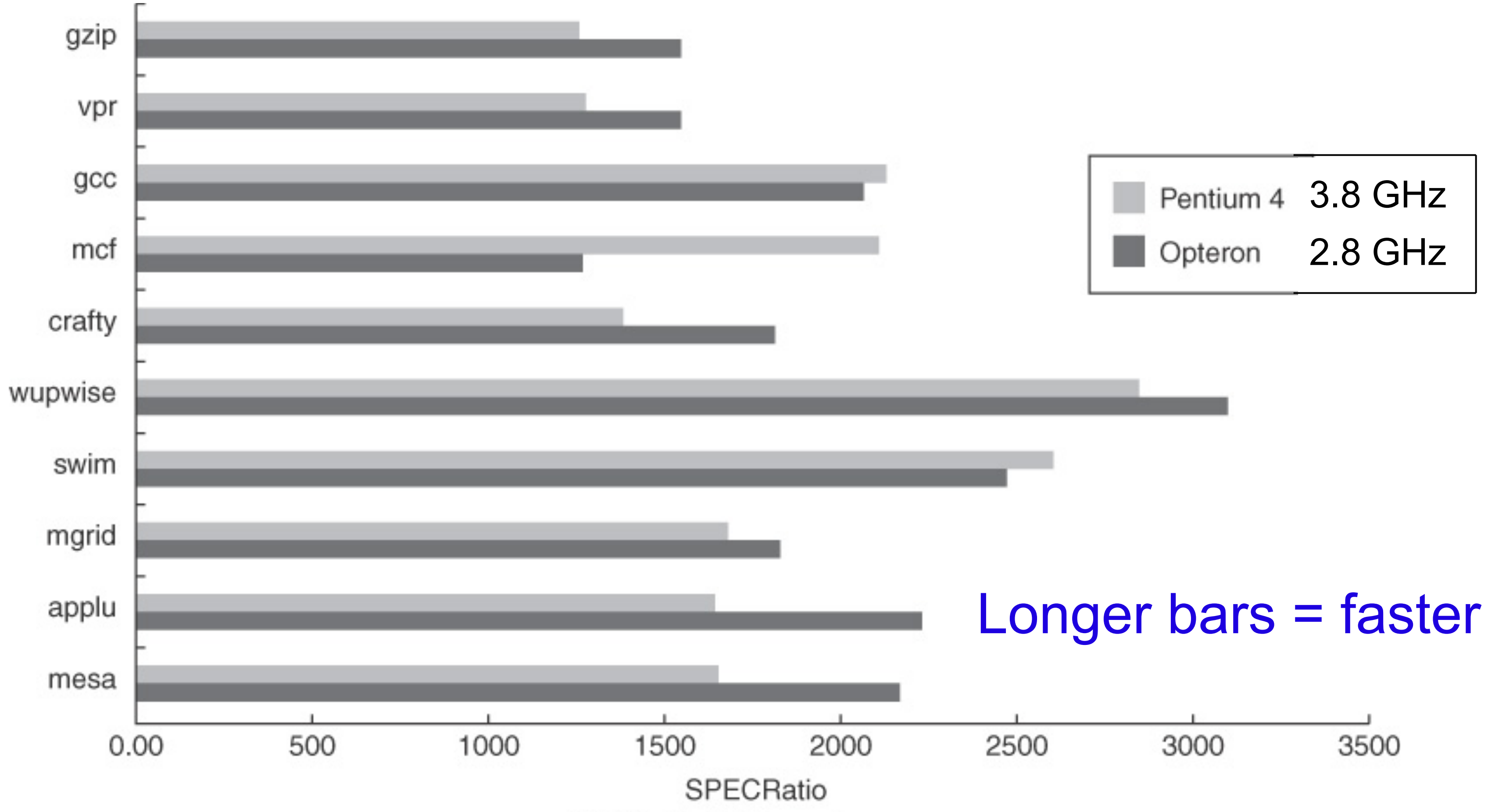
L2 cache misses more important for performance… why?

Pentium 4 CPI v.s AMD Opteron CPI



Opteron lower average CPI (factor of 1.27)

Pentium 4 Performance v.s AMD Opteron Performance



Opteron slightly faster overall: Higher clock frequency not enough to overcome reduction in CPI due to increased stalls

Limits to ILP (Instruction Level Parallelism)

- How much ILP is available using superscalar OoO mechanisms with increasing HW budgets?
- Do we need to invent new HW/SW mechanisms to keep on processor performance curve?
  - Intel MMX, SSE (Streaming SIMD Extensions): 64 bit ints
  - GPUs/Cell Processor?
  - Other?

Limits to ILP

Initial HW Model here; MIPS.

Assumptions for ideal/perfect machine to start:

1. *Register renaming* – infinite virtual registers => all register WAW & WAR hazards are avoided

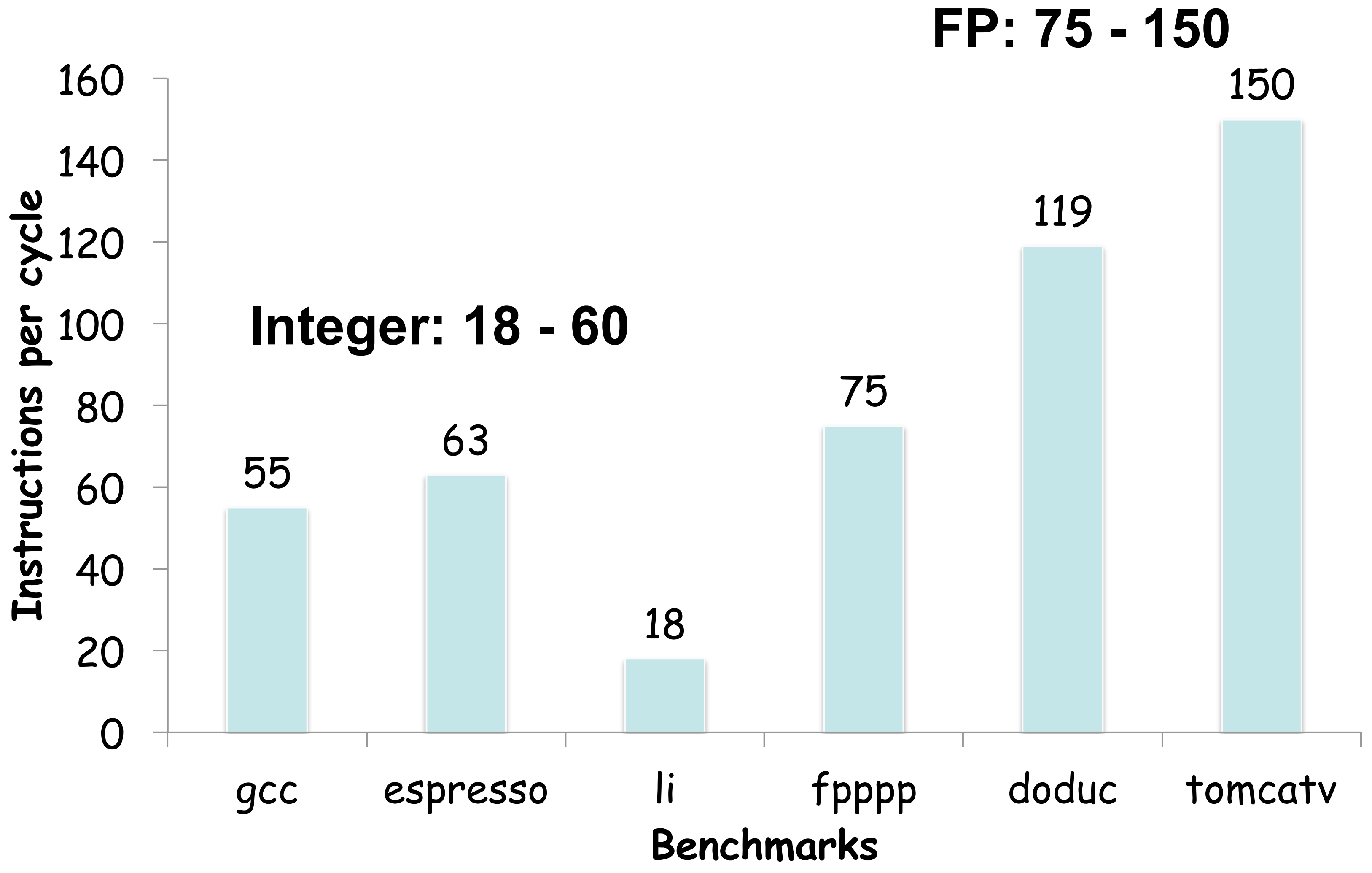
2. *Branch prediction* – perfect; no mispredictions

3. *Jump prediction* – all jumps perfectly predicted
   2 & 3 => machine with perfect speculation & an unbounded buffer of instructions available

4. *Memory-address alias analysis* – addresses are known & a store can be moved before a load provided addresses not equal

Also:
- unlimited number of instructions issued/clock cycle; perfect caches;
- 1 cycle latency for all instructions

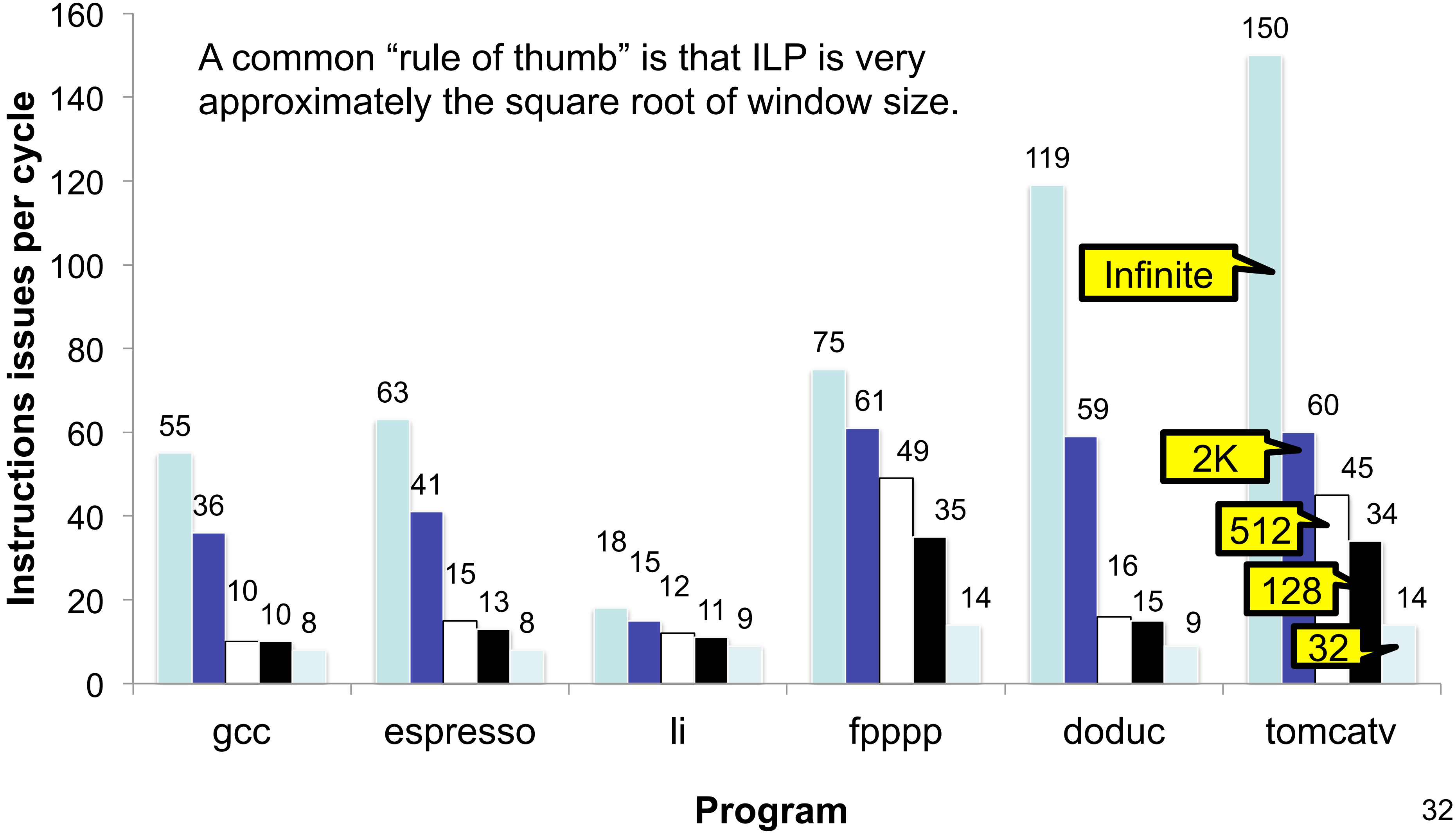
Upper Limit to Instruction Level Parallelism: Ideal Machine



Impact of Instruction Window Size

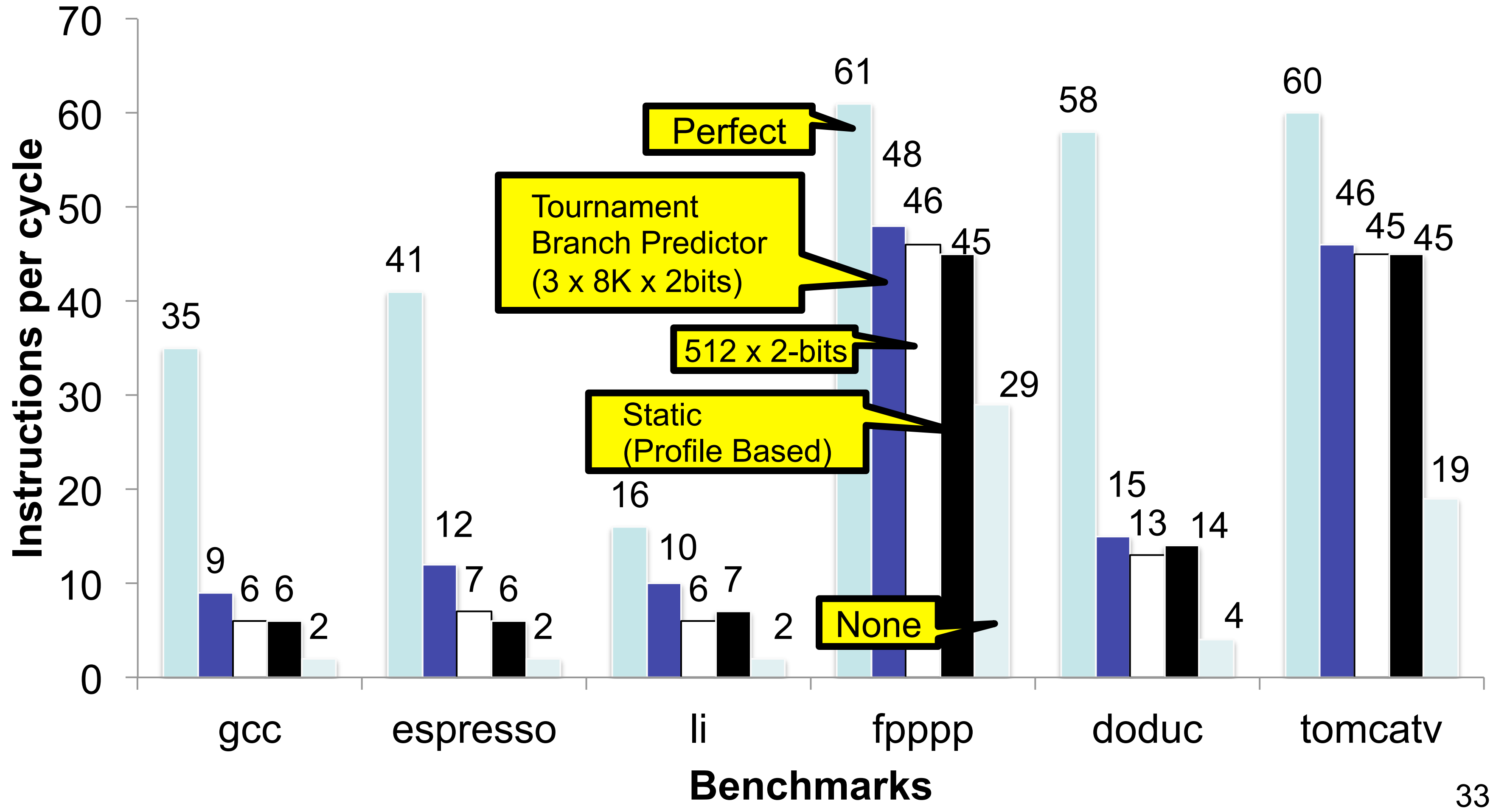
Instruction window size limits the number of instructions that we can search through while looking for parallelism among instructions.

A common "rule of thumb" is that ILP is very approximately the square root of window size.



More Realistic HW: Branch Impact

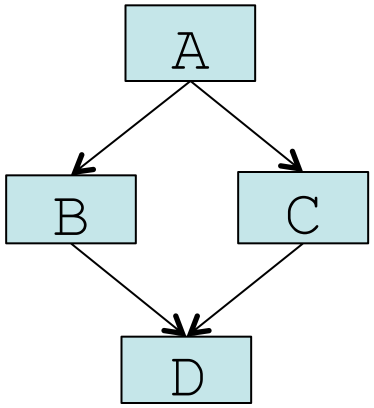
Change from Infinite window to examine to 2000 and maximum issue of 64 instructions per clock cycle



Beyond Branch Prediction

- Control flow very often “re-converges” shortly after a branch.

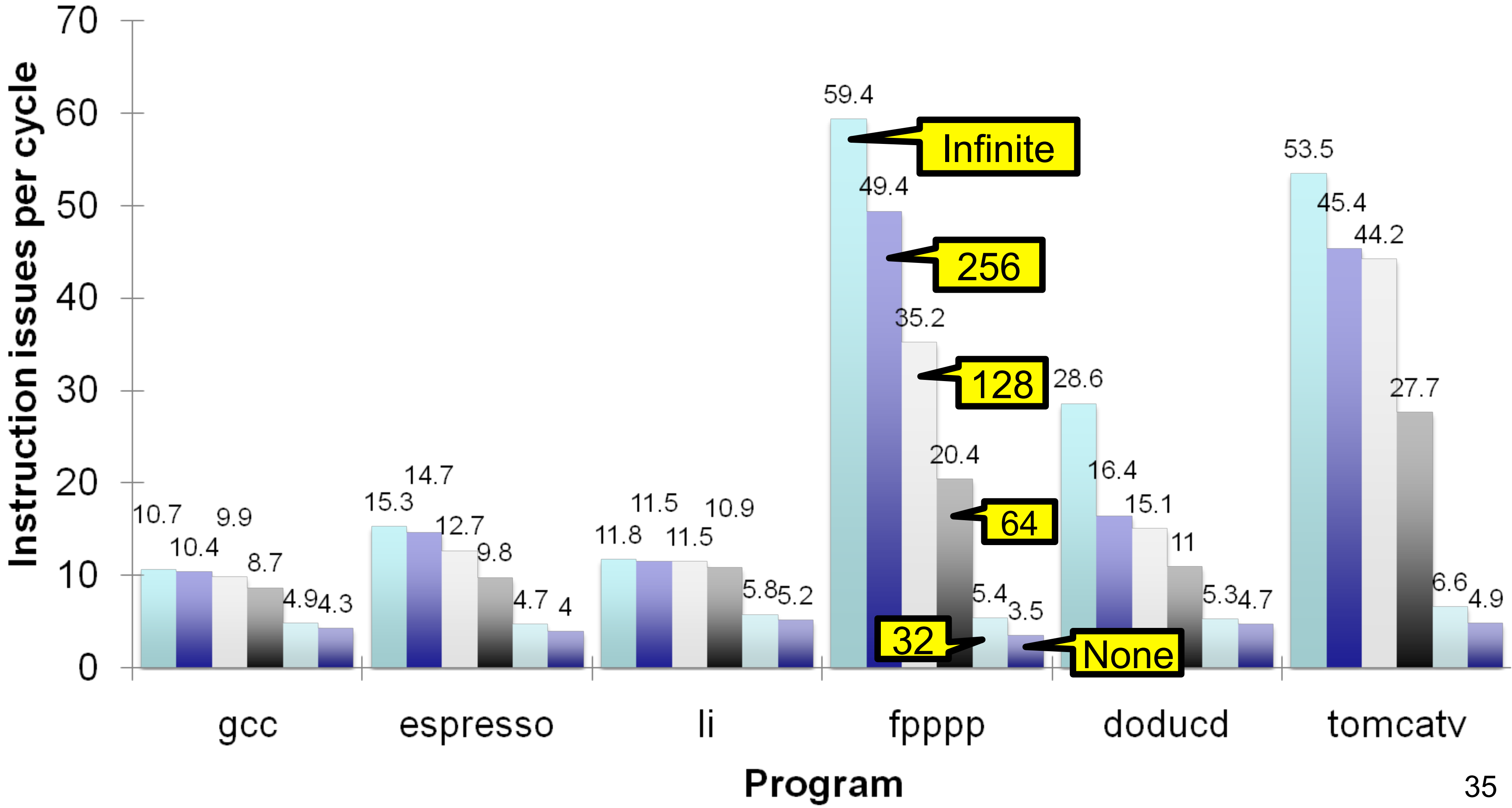
```
y = lookup(x); // A
if( y > 0 ) {
   B++;
} else {
   C++;
}
foo(); // D
```


$$\begin{tikzcd} & A \arrow[dl] \arrow[dr] & \\ B \arrow[dr] & & C \arrow[dl] \\ & D & \end{tikzcd}$$

- In 1995-2005 timeframe: much research on “speculative threads”. E.g., start short thread at D while executing at A. When original thread reaches D it combines state with speculative thread.
- Huge benefit: reduces sensitivity to branch mispredictions.
- Sun Microsystems “Rock” processor implemented a simple form of this and it seem likely Intel or AMD will eventually implement it since Amdahl’s Law suggests single thread performance still matters.

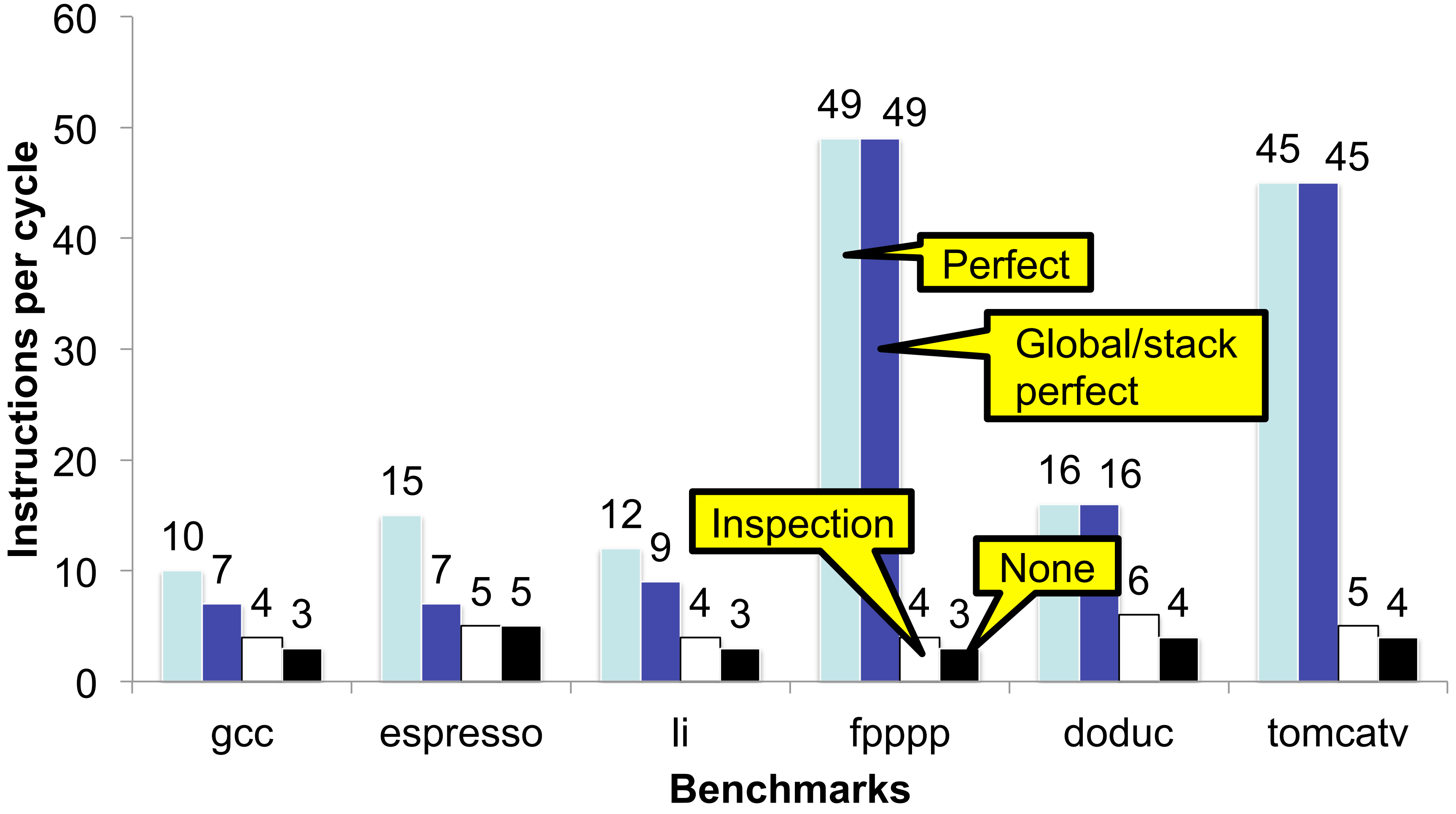
Impact of Number of Renaming Registers

2000 instruction window, maximum 64 inst/cycle, tournament predictor
Floating-point more sensitive to number of rename registers.

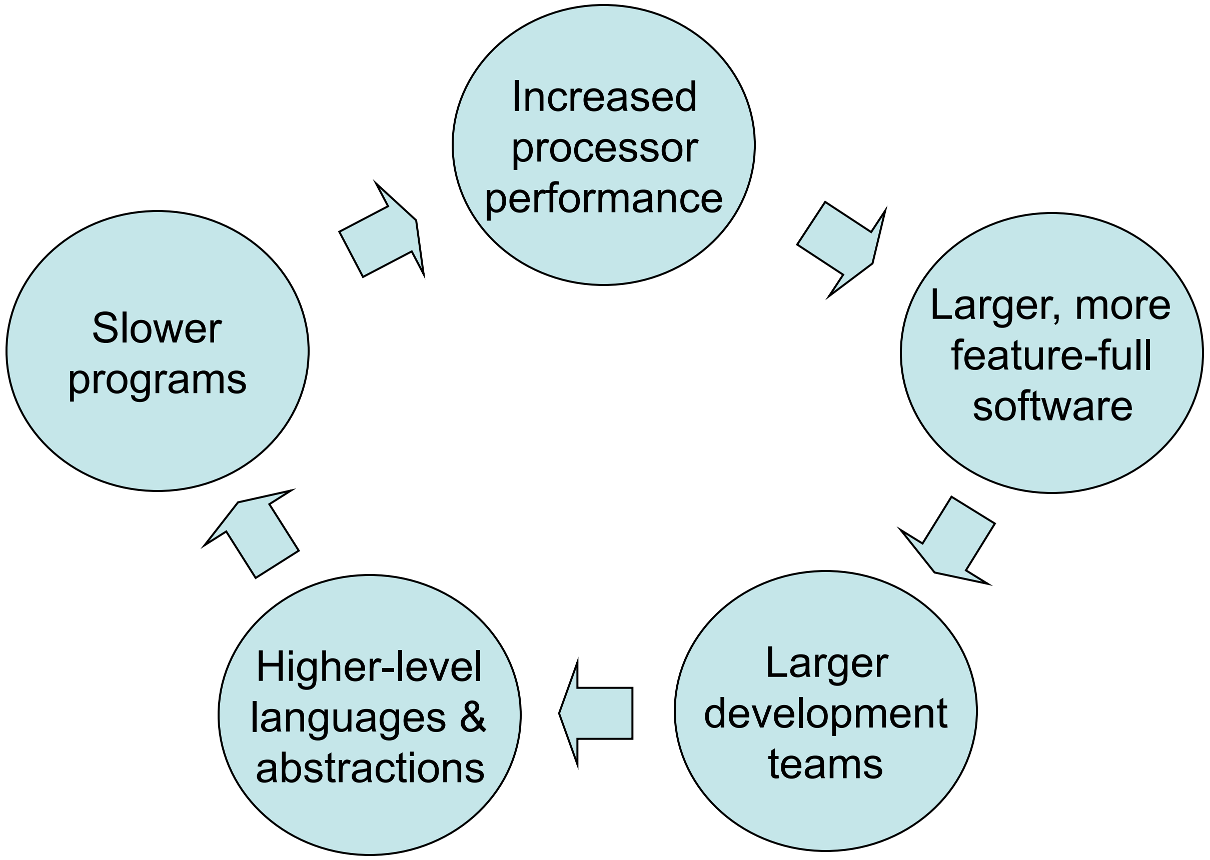


Impact of Memory Dependence Prediction

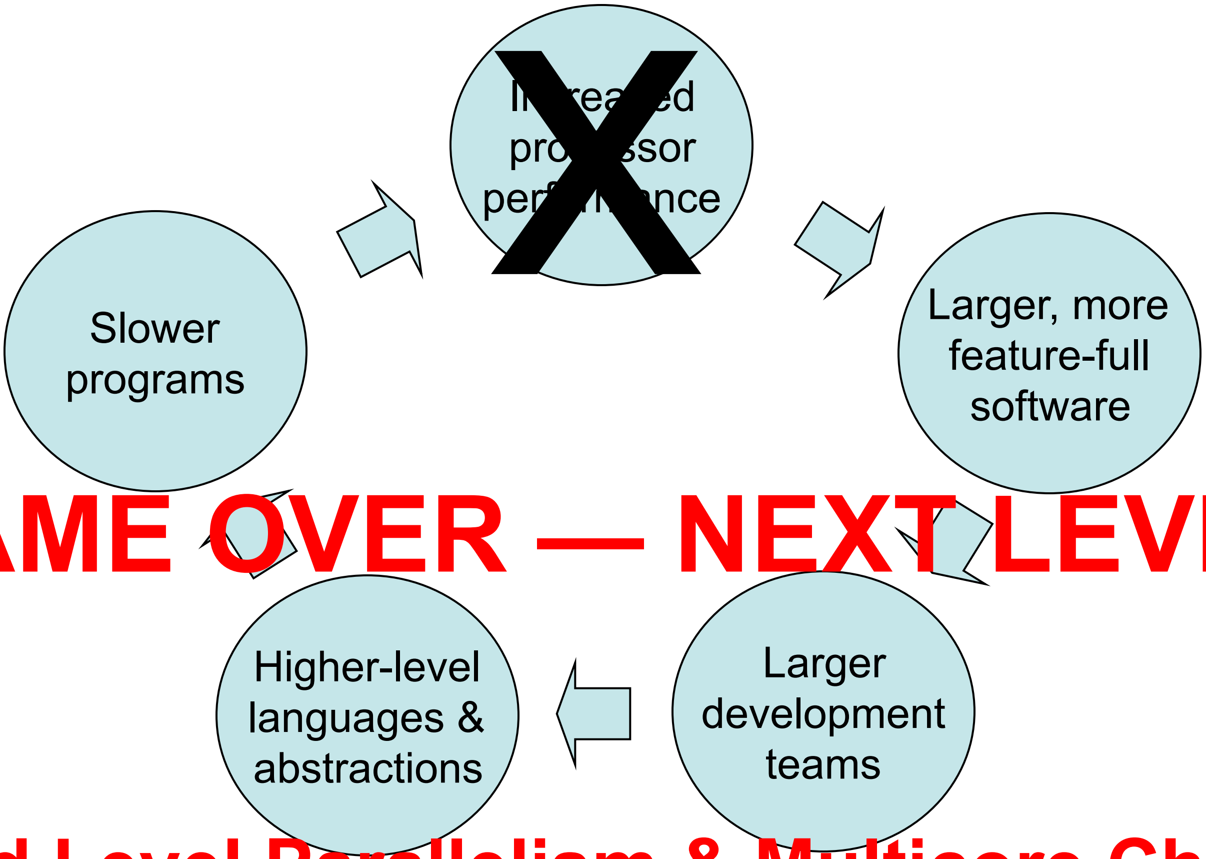
2000 instr window, 64 instr issue, 8K 2 level Prediction, 256 renaming registers
(potential impact of memory "dependence prediction")



Virtuous Cycle, circa 1950 – 2005
[Mark Hill, Jim Larus]


Increased processor performance
Larger, more feature-full software
Larger development teams
Higher-level languages & abstractions
Slower programs

Virtuous Cycle, 2005 – ???


Increased processor performance
X
Larger, more feature-full software
Larger development teams
Higher-level languages & abstractions
Slower programs

GAME OVER — NEXT LEVEL?

Thread Level Parallelism & Multicore Chips

Future Microprocessors: What must we do differently to effectively utilize multi-core and many-core chips?

…and what are the implications re: education?

Yale Patt

The University of Texas at Austin

World University Presidents' Symposium

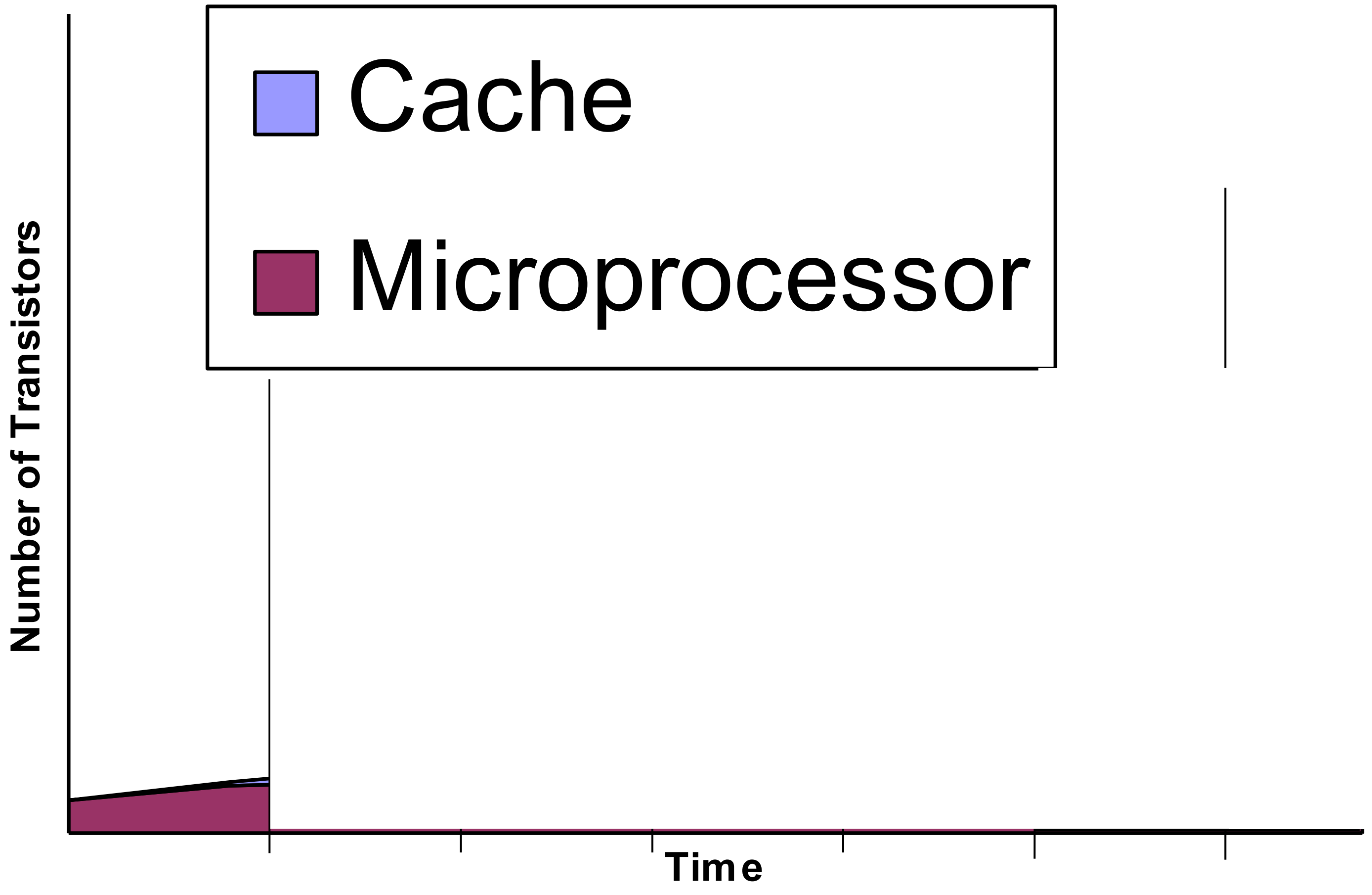
University of Belgrade

April 4, 2009

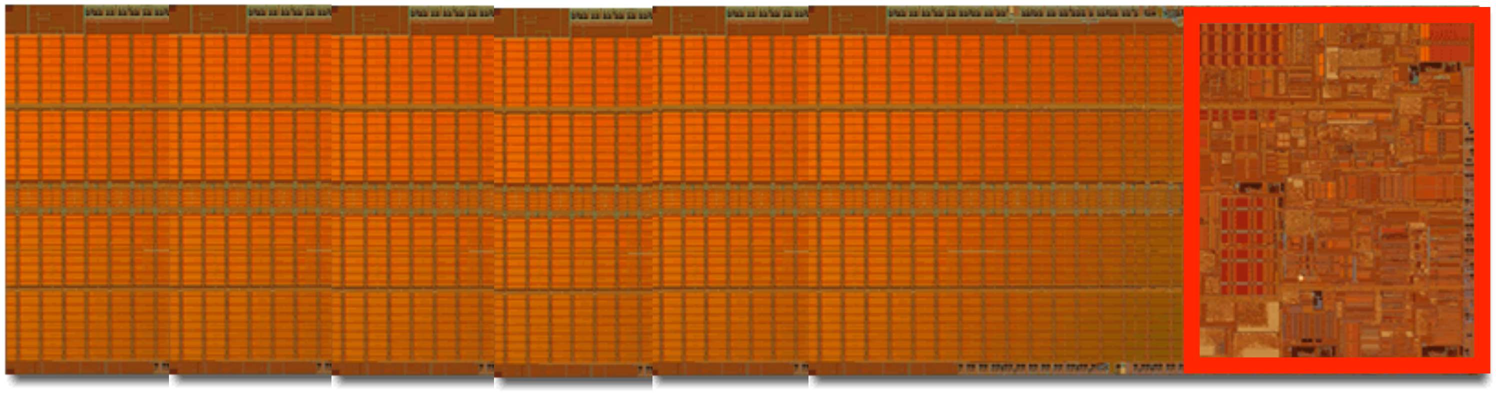
How will we use 50 billion transistors?

How have we used the transistors up to now?

How have we used the available transistors?

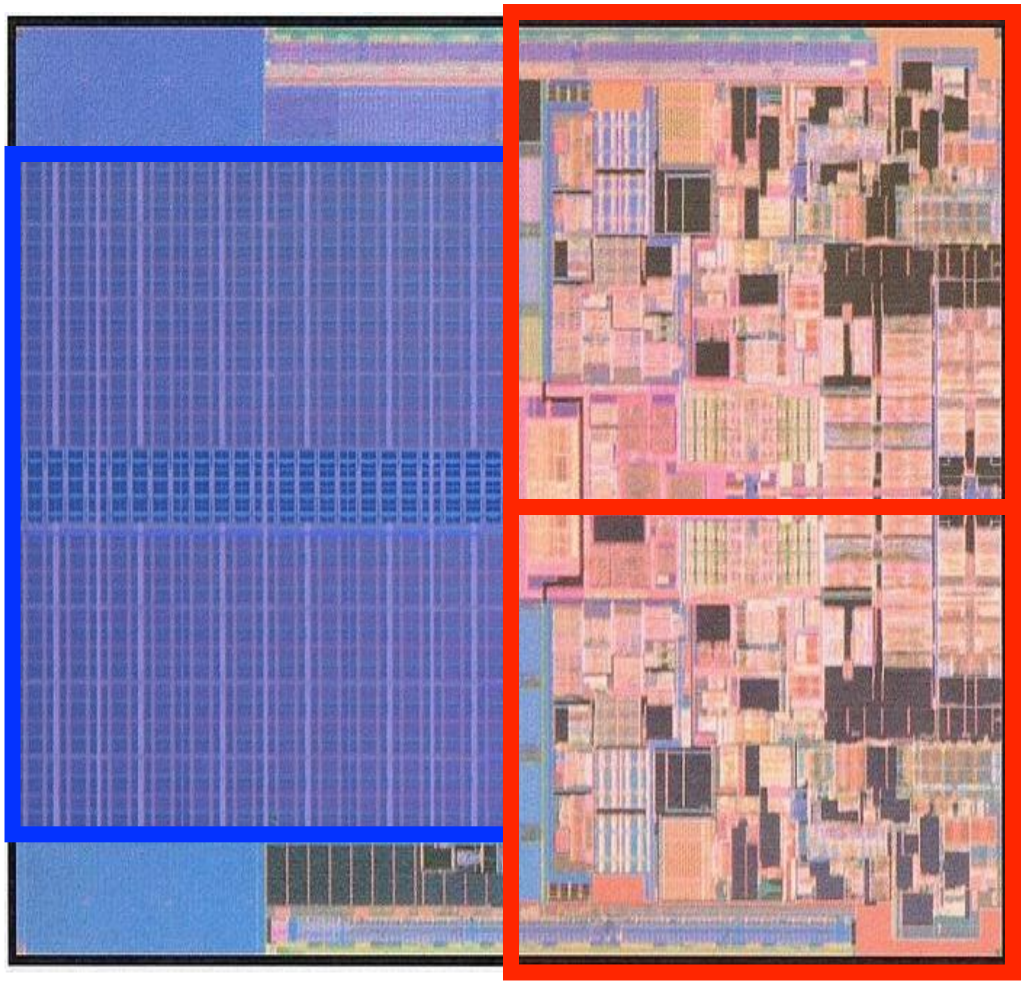


Intel Pentium M



Intel Core 2 Duo

- ***Penryn, 2007***
- ***45nm, 3MB L2***



Why Multi-core chips?

- ***It is easier than designing a much better uni-core***
- ***It is embarrassing to continue making L2 bigger***
- ***It is the next obvious step***
- ***It is not the Holy Grail***

that is…

- ***In the beginning: a better and better uniprocessor***
  - ***improving performance on the hard problems***
- ***More recently: a uniprocessor with a bigger L2 cache***
  - ***forsaking further improvement on the "hard" problems***
  - ***poorly utilizing the chip area***
  - ***and blaming the processor for not delivering performance***
- ***Today: dual core, quad core***
- ***Tomorrow: ???***

The Good News: Lots of cores on the chip

The Bad News: Not much benefit.

Evolution of Thread-Level Parallelism in Desktop Applications

Geoffrey Blake*, Ronald G. Dreslinski*, Trevor Mudge*, Krisztián Flautner†

University of Michigan – Ann Arbor*
ARM †

Introduction

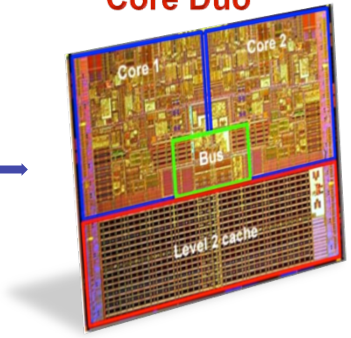
- 2000
  - Single core machines common
  - Clock speed steadily increasing – Intel predicts to reach **10**GHz by **2010**
- 2005
  - C
  - A
- 2010 and Future
  - Multi-core machines common
  - Core counts steadily increasing

"If you build it, they will come"

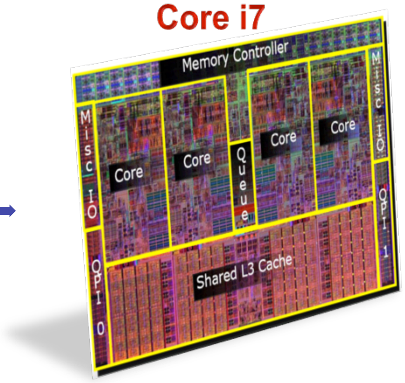
-- "Field of Dreams" 1989

Pentium 4

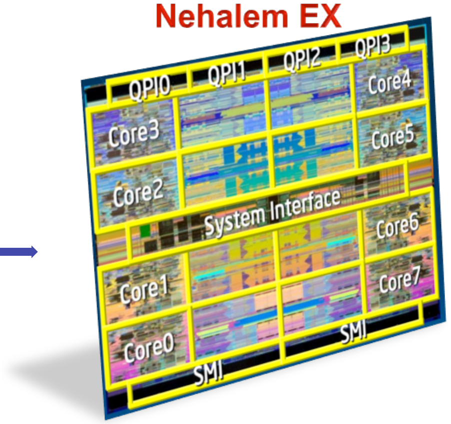
Core Duo



Core i7



Nehalem EX

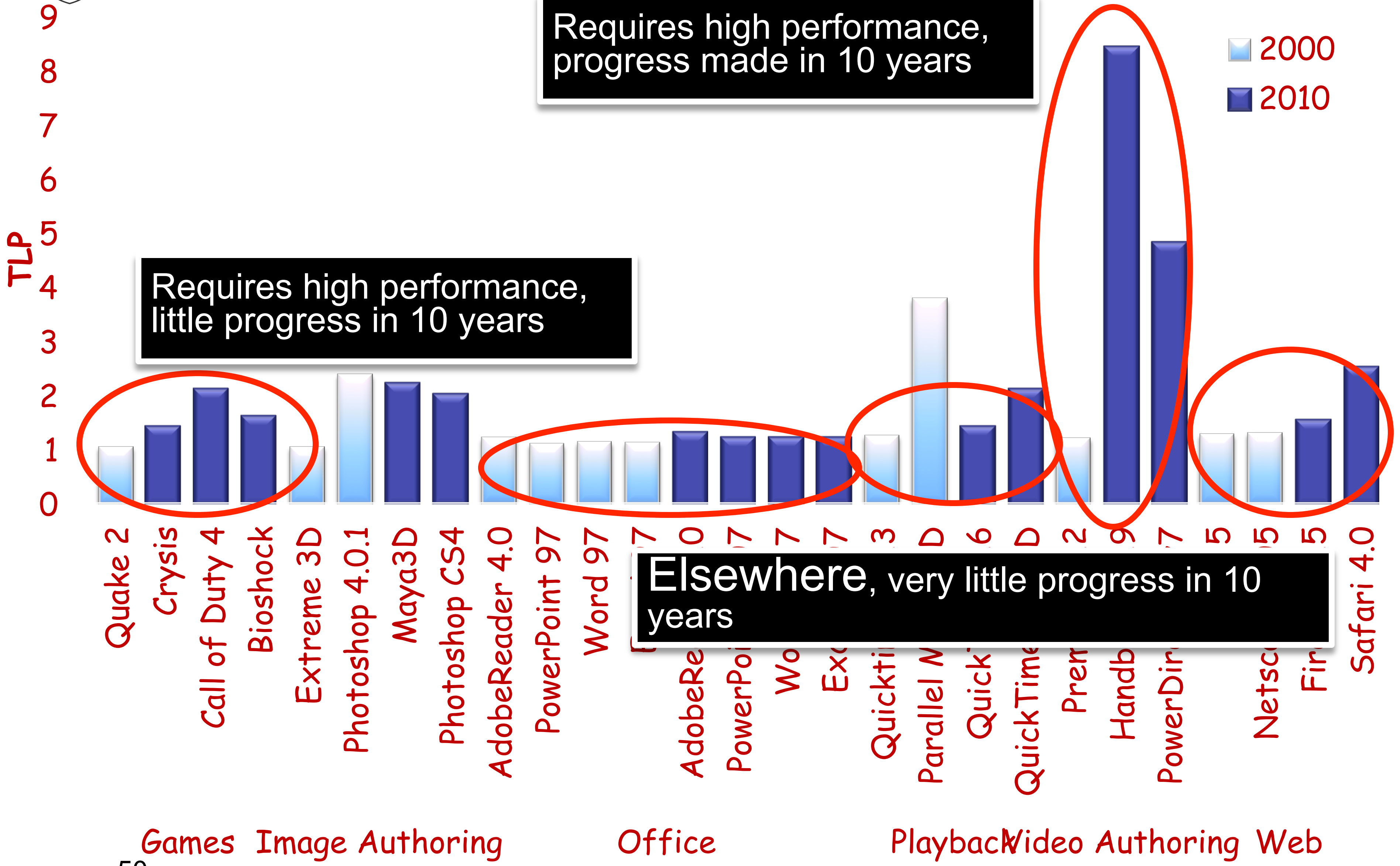


Are threads used?

Benchmark	Threads Created	Avg. Threads Alive
Handbrake 0.9	22511	24
Call of Duty 4	77	44
Photoshop CS4	82	75
Adobe Reader 9	239	24
Quicktime-HD	53	52
Firefox 3.5	522	38

- Many threads created
- Many threads alive and visible to the OS during runtime

10 Year Comparison



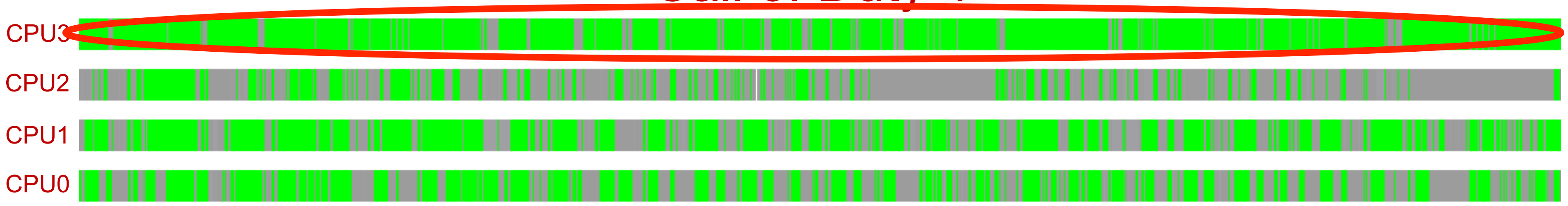
Requires high performance, progress made in 10 years

Requires high performance, little progress in 10 years

Elsewhere, very little progress in 10 years

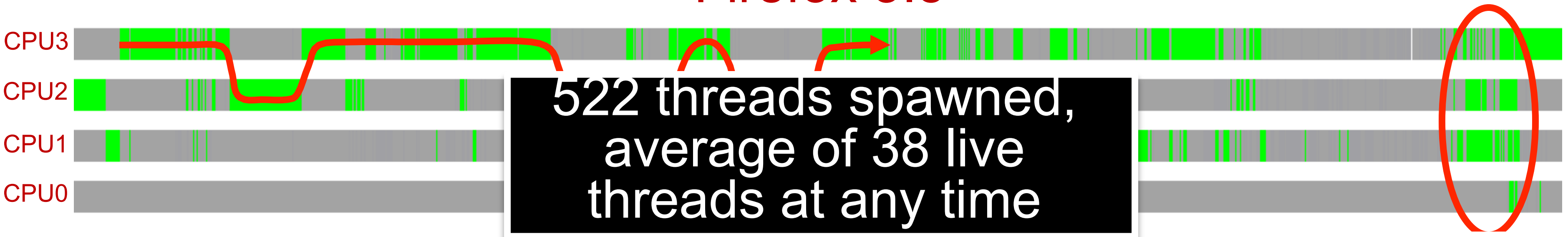
TLP vs Time – Atom Windows 7

Call of Duty 4



Time (3s)

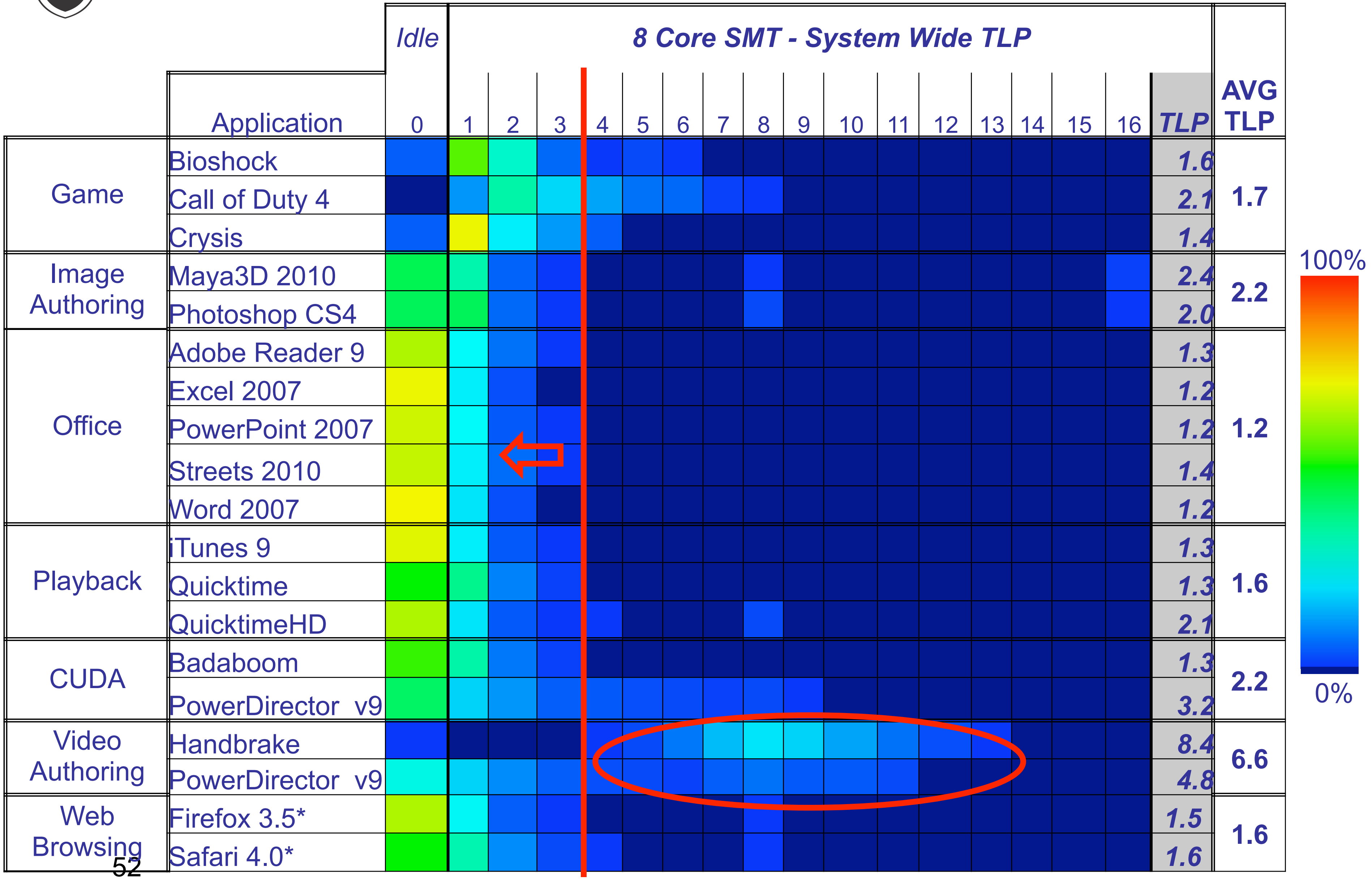
Firefox 3.5



522 threads spawned, average of 38 live threads at any time

Time (3s)

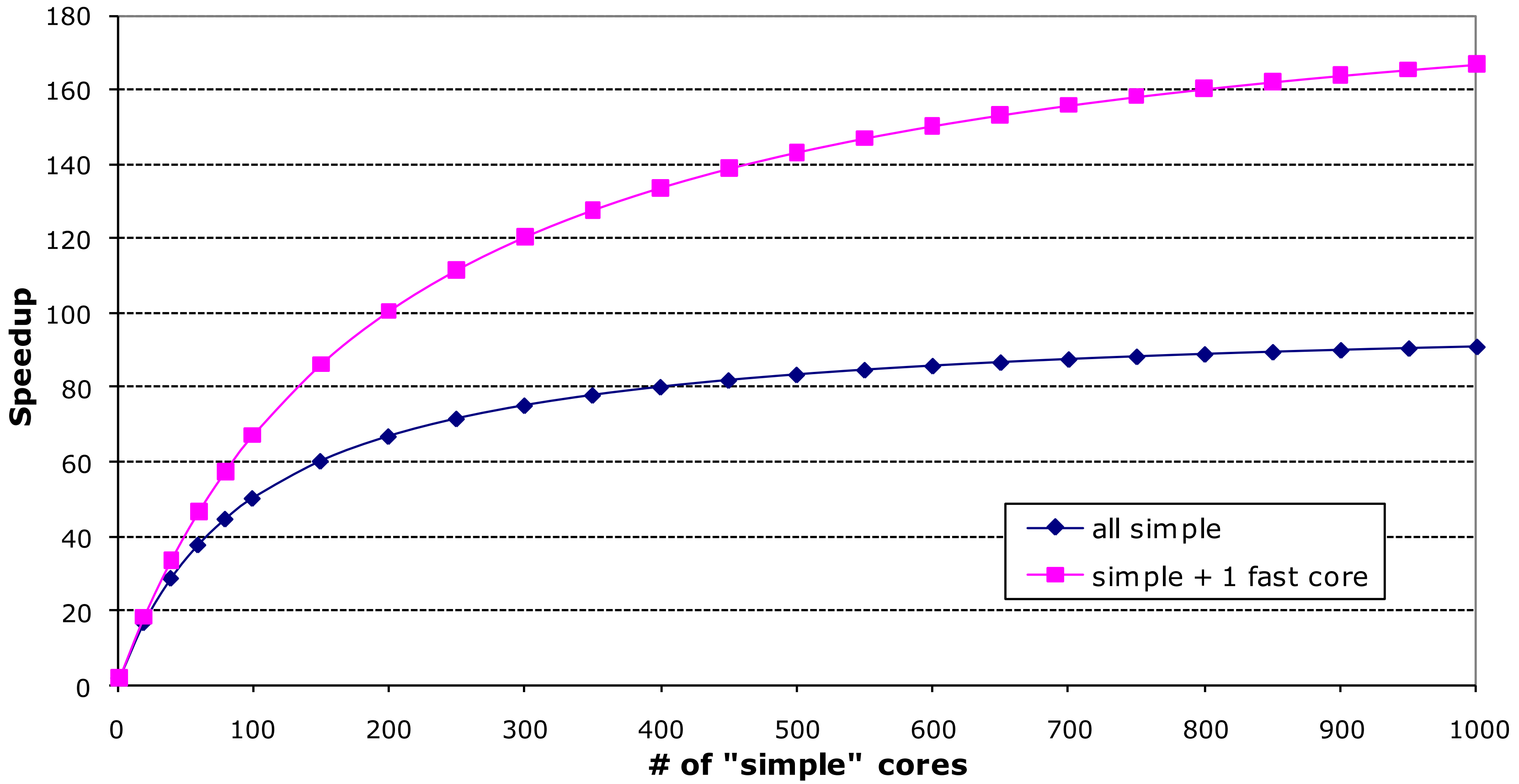
Overall TLP Results – Xeon Windows 7



	Application	TLP	AVG TLP
Game	Bioshock	1.6	1.7
	Call of Duty 4	2.1	
	Crysis	1.4	
Image Authoring	Maya3D 2010	2.4	2.2
	Photoshop CS4	2.0	
Office	Adobe Reader 9	1.3	1.2
	Excel 2007	1.2	
	PowerPoint 2007	1.2	
	Streets 2010	1.4	
	Word 2007	1.2	
Playback	iTunes 9	1.3	1.6
	Quicktime	1.3	
	QuicktimeHD	2.1	
CUDA	Badaboom	1.3	2.2
	PowerDirector v9	3.2	
Video Authoring	Handbrake	8.4	6.6
	PowerDirector v9	4.8	
Web Browsing	Firefox 3.5*	1.5	1.6
	Safari 4.0*	1.6	

Why Instruction Level Parallelism Still Matters (Even With Multicore)

Assuming application is 99% parallelizable, what is better: More simple cores or improve IPC for sequential fraction (e.g., using superscalar or other "cleverness")?


Speedup
# of "simple" cores
all simple
simple + 1 fast core