MO401

IC/Unicamp

Prof Mario Côrtes

Capítulo 3 – parte B (3.8 - 3.15):

Instruction-Level Parallelism and Its Exploitation

Tópicos - estrutura

- Parte A
  - Basic compiler ILP
  - Advanced branch prediction
  - Dynamic scheduling
  - Hardware based speculation
  - Multiple issue and static scheduling
- Parte B
  - Instruction delivery and speculation
  - Limitations of ILP
  - ILP and memory issues
  - Multithreading

3.8 Dynamic Scheduling, Multiple Issue, and Speculation

- Até agora, vistos separadamente
  - Dynamic scheduling, multiple issue, speculation
- Modern microarchitectures:
  - Dynamic scheduling + multiple issue + speculation
- Hipótese simplificadora: 2 issues / ciclo
- Extensão do alg. Tomasulo: multiple issue supersacalar pipeline, separate integer, LD/ST, FP units (add, mult)
  - FUs: initiate operation every clock
- Issue to RS in-order. Any two operations (every cycle)

Overview of Design

New:
issue and completion logic must support 2 instructions / clock cycle

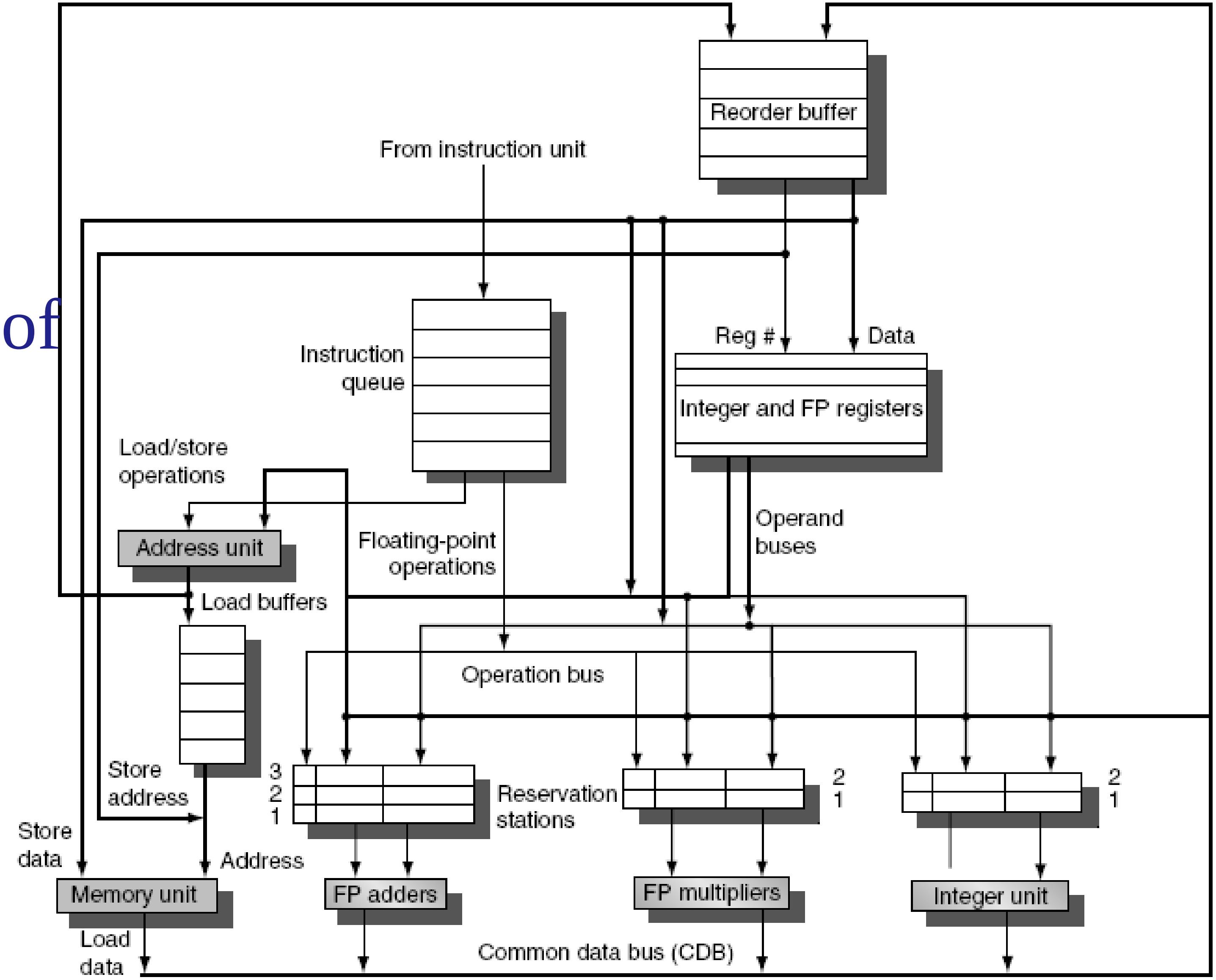


Figure 3.17 The basic organization of a multiple issue processor with speculation. In this case, the organization could allow a FP multiply, FP add, integer, and load/store to all issues simultaneously (assuming one issue per clock per functional unit). Note that several datapaths must be widened to support multiple issues: the CDB, the operand buses, and, critically, the instruction issue logic, which is not shown in this figure. The last is a difficult problem, as we discuss in the text.

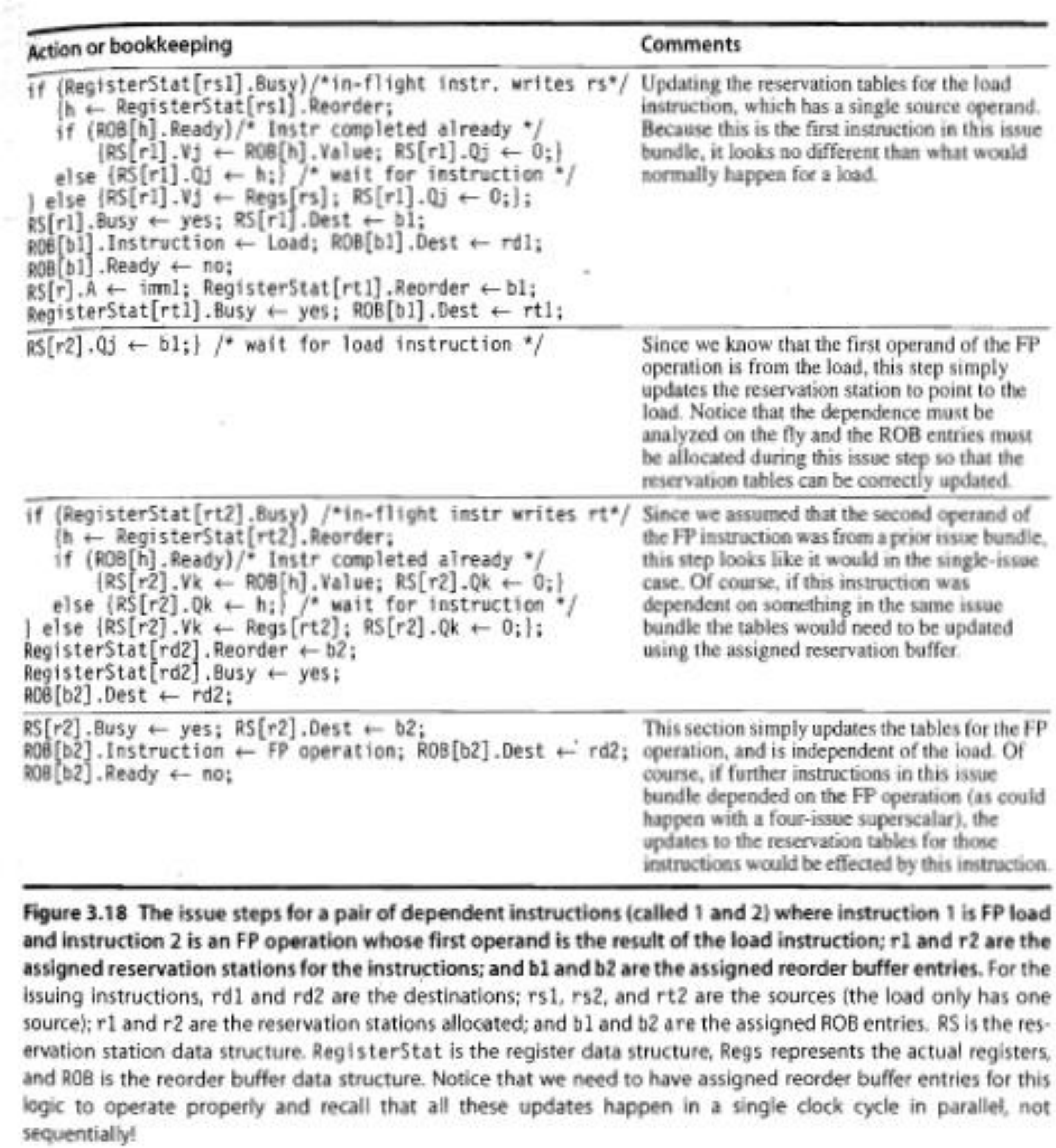
Extended Tomasulo

- Multiple issue / cycle: muito complicado.
  - ex: as duas operações podem ter dependência e tabelas tem que ser atualizadas em paralelo (no mesmo clk)
- Two approaches:
  - Assign reservation stations and update pipeline control table in half clock cycles
    - Only supports 2 instructions/clock
  - Design logic to handle any possible dependencies between the instructions
  - Hybrid approaches
- Modern superscalar processors (4+ issues) use both:
  - Issue logic: wide and pipelined
- Issue logic can become bottleneck
  - (ver Fig 3.18, para apenas um caso)

Complexidade: apenas uma dependência

ins1 = LD

ins2 = op FP com operando fornecido pelo LD



Action or bookkeeping	Comments

if (RegisterStat[rs1].Busy)/*in-flight instr. writes rs*/
{h ← RegisterStat[rs1].Reorder;
if (ROB[h].Ready)/* Instr completed already */
{RS[r1].Vj ← ROB[h].Value; RS[r1].Qj ← 0;}
else {RS[r1].Qj ← h;} /* wait for instruction */
} else {RS[r1].Vj ← Regs[rs]; RS[r1].Qj ← 0;};
RS[r1].Busy ← yes; RS[r1].Dest ← b1;
ROB[b1].Instruction ← Load; ROB[b1].Dest ← rd1;
ROB[b1].Ready ← no;
RS[r].A ← imm1; RegisterStat[rt1].Reorder ← b1;
RegisterStat[rt1].Busy ← yes; ROB[b1].Dest ← rt1;
	Updating the reservation tables for the load instruction, which has a single source operand. Because this is the first instruction in this issue bundle, it looks no different than what would normally happen for a load.

RS[r2].Qj ← b1;} /* wait for load instruction */	Since we know that the first operand of the FP operation is from the load, this step simply updates the reservation station to point to the load. Notice that the dependence must be analyzed on the fly and the ROB entries must be allocated during this issue step so that the reservation tables can be correctly updated.

if (RegisterStat[rt2].Busy) /*in-flight instr writes rt*/
{h ← RegisterStat[rt2].Reorder;
if (ROB[h].Ready)/* Instr completed already */
{RS[r2].Vk ← ROB[h].Value; RS[r2].Qk ← 0;}
else {RS[r2].Qk ← h;} /* wait for instruction */
} else {RS[r2].Vk ← Regs[rt2]; RS[r2].Qk ← 0;};
RegisterStat[rd2].Reorder ← b2;
RegisterStat[rd2].Busy ← yes;
ROB[b2].Dest ← rd2;
	Since we assumed that the second operand of the FP instruction was from a prior issue bundle, this step looks like it would in the single-issue case. Of course, if this instruction was dependent on something in the same issue bundle the tables would need to be updated using the assigned reservation buffer.

RS[r2].Busy ← yes; RS[r2].Dest ← b2;
ROB[b2].Instruction ← FP operation; ROB[b2].Dest ← rd2;
ROB[b2].Ready ← no;
	This section simply updates the tables for the FP operation, and is independent of the load. Of course, if further instructions in this issue bundle depended on the FP operation (as could happen with a four-issue superscalar), the updates to the reservation tables for those instructions would be effected by this instruction.

Figure 3.18 The issue steps for a pair of dependent instructions (called 1 and 2) where instruction 1 is FP load and instruction 2 is an FP operation whose first operand is the result of the load instruction; r1 and r2 are the assigned reservation stations for the instructions; and b1 and b2 are the assigned reorder buffer entries. For the issuing instructions, rd1 and rd2 are the destinations; rs1, rs2, and rt2 are the sources (the load only has one source); r1 and r2 are the reservation stations allocated; and b1 and b2 are the assigned ROB entries. RS is the reservation station data structure. RegisterStat is the register data structure, Regs represents the actual registers, and ROB is the reorder buffer data structure. Notice that we need to have assigned reorder buffer entries for this logic to operate properly and recall that all these updates happen in a single clock cycle in parallel, not sequentially!

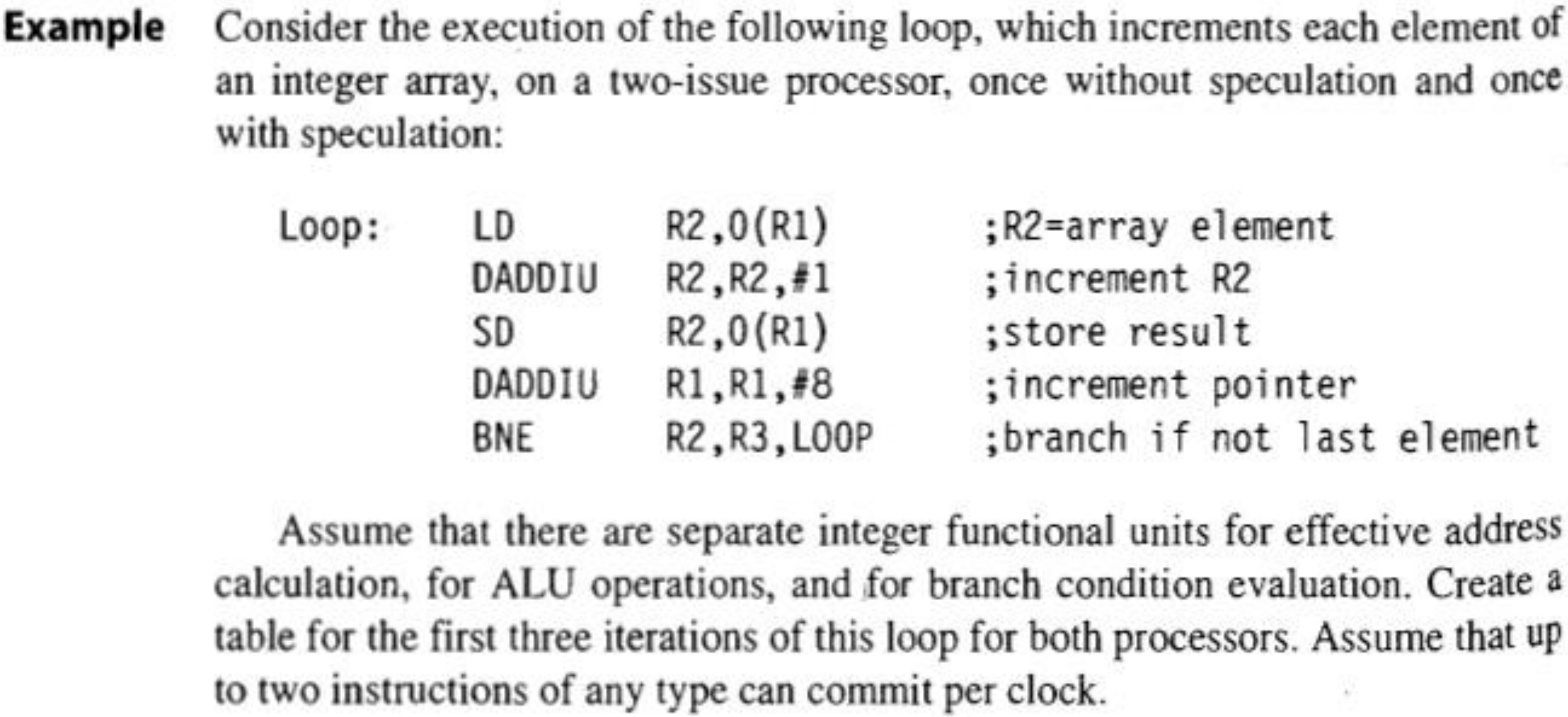
Multiple Issue

- 1- Pre-assign a RS and ROB entry. Limit the number of instructions of a given class that can be issued in a “bundle”
  - I.e. on FP, one integer, one load, one store
- 2- Examine all the dependencies among the instructions in the bundle
- 3- If dependencies exist in bundle, encode them in reservation stations and ROB

- All above: a single clock cycle
- At pipeline backend: need multiple completion/commit
  - Easier, because dependences have already been dealt with

- Intel i7 usa este esquema

Exmpl p 200: multiple issue with and without speculation

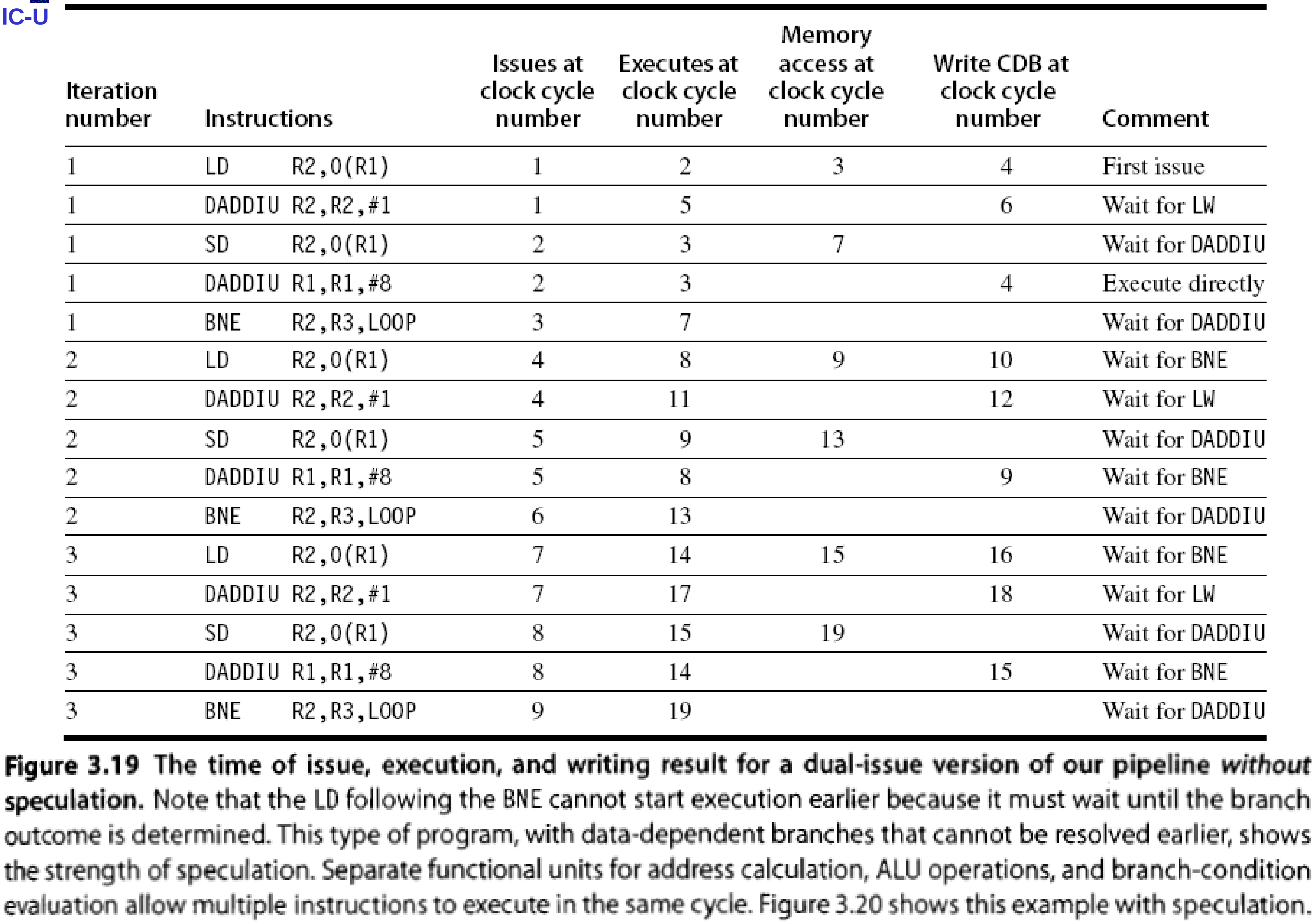


Example Consider the execution of the following loop, which increments each element of an integer array, on a two-issue processor, once without speculation and once with speculation:

```
Loop:     LD        R2,0(R1)          ;R2=array element
          DADDIU    R2,R2,#1          ;increment R2
          SD        R2,0(R1)          ;store result
          DADDIU    R1,R1,#8          ;increment pointer
          BNE       R2,R3,LOOP        ;branch if not last element
```

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table for the first three iterations of this loop for both processors. Assume that up to two instructions of any type can commit per clock.

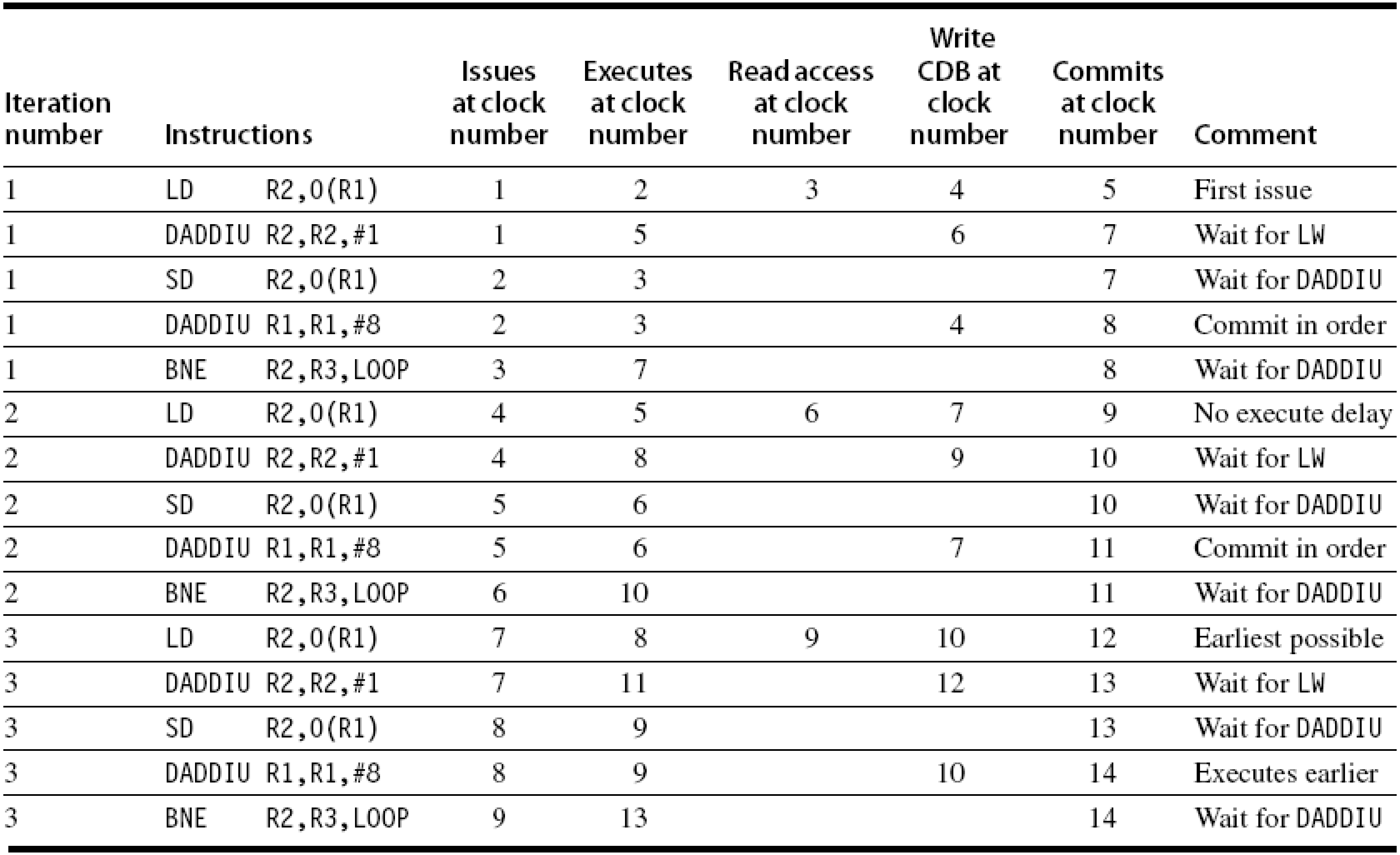
No speculation



Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#8	8	14		15	Wait for BNE
3	BNE R2,R3,LOOP	9	19			Wait for DADDIU

Figure 3.19 The time of issue, execution, and writing result for a dual-issue version of our pipeline *without* speculation. Note that the LD following the BNE cannot start execution earlier because it must wait until the branch outcome is determined. This type of program, with data-dependent branches that cannot be resolved earlier, shows the strength of speculation. Separate functional units for address calculation, ALU operations, and branch-condition evaluation allow multiple instructions to execute in the same cycle. Figure 3.20 shows this example with speculation.

With speculation



Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#8	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

Figure 3.20 **The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation.** Note that the LD following the BNE can start execution early because it is speculative.

3.9 Advanced Techniques

- Objetivo: possibilitar alta taxa de execução de instruções por ciclo
  - Increasing instruction delivery bandwidth
  - Advanced speculation techniques
  - Value prediction

Animações e simulações

- Ver site
  - http://www.williamstallings.com/COA/Animation/Links.html

- Contém várias simulações:
  - Branch prediction
  - Branch Target Buffer
  - Loop unrolling
  - Pipeline with static vs. dynamic scheduling
  - Reorder Buffer Simulator
  - Scoreboarding technique for dynamic scheduling:
  - Tomasulo's Algorithm:

Increasing instruction fetch bandwidth

- Need high instruction bandwidth (from Instr. Cache to Issue Unit)
  - problema: como saber antes da decodificação se instrução é desvio e qual é o próximo PC?
- Branch-Target buffers
  - Next PC prediction buffer, indexed by current PC
- Diferenças com o branch prediction buffer já visto
  - branch prediction buffer:
    - após decodificação; só branches são tratados; index pode apontar para outro branch
  - no Branch-Target buffer
    - antes da decodificação; todas as instruções são tratadas; “tag” do buffer identifica univocamente somente branches; somente “taken branches” são armazenados → demais instruções seguem o fetch normalmente

Branch-Target Buffer

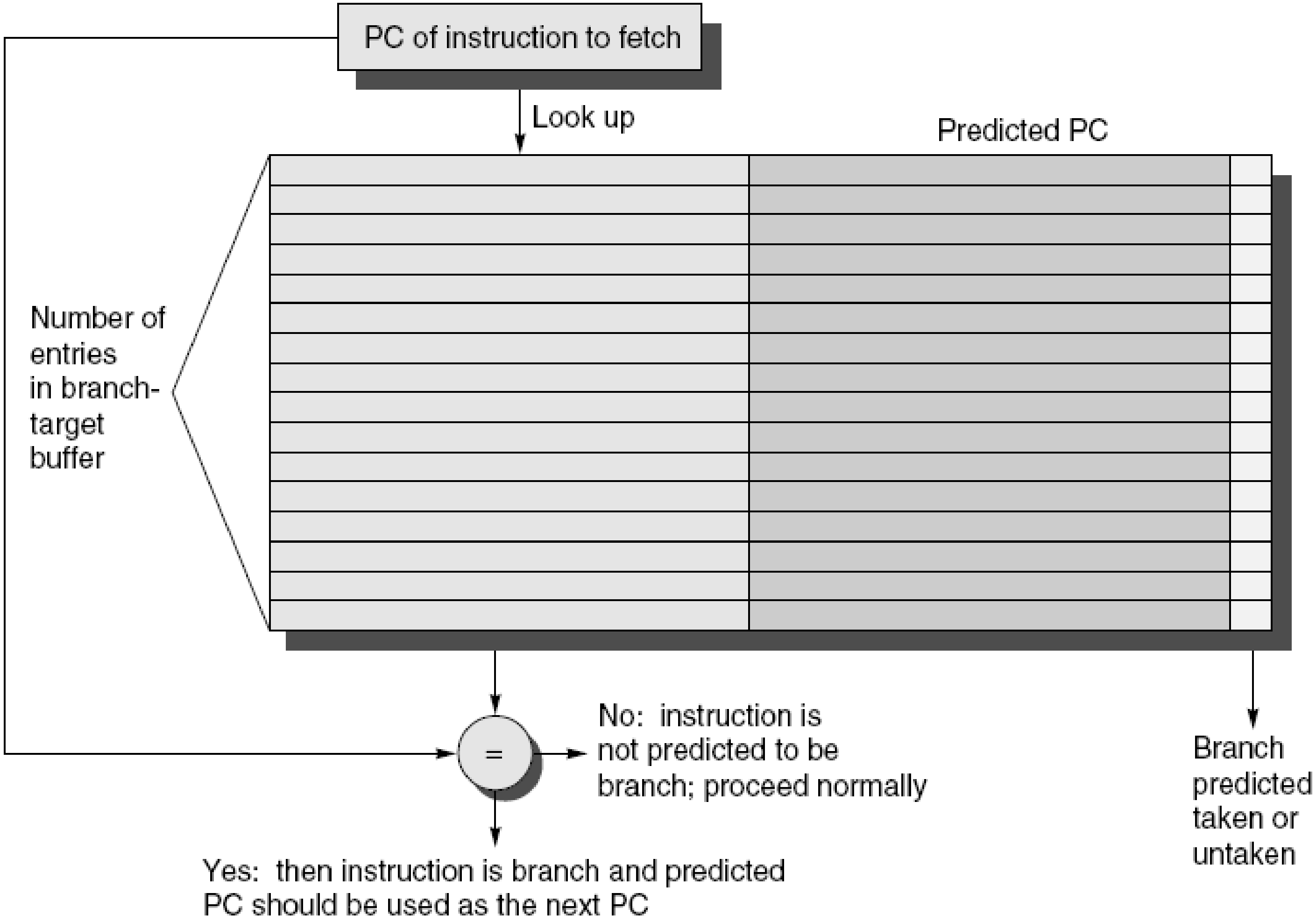

PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

Figure 3.21 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field, which is optional, may be used for extra prediction state bits.

Branch-Target Buffer: steps

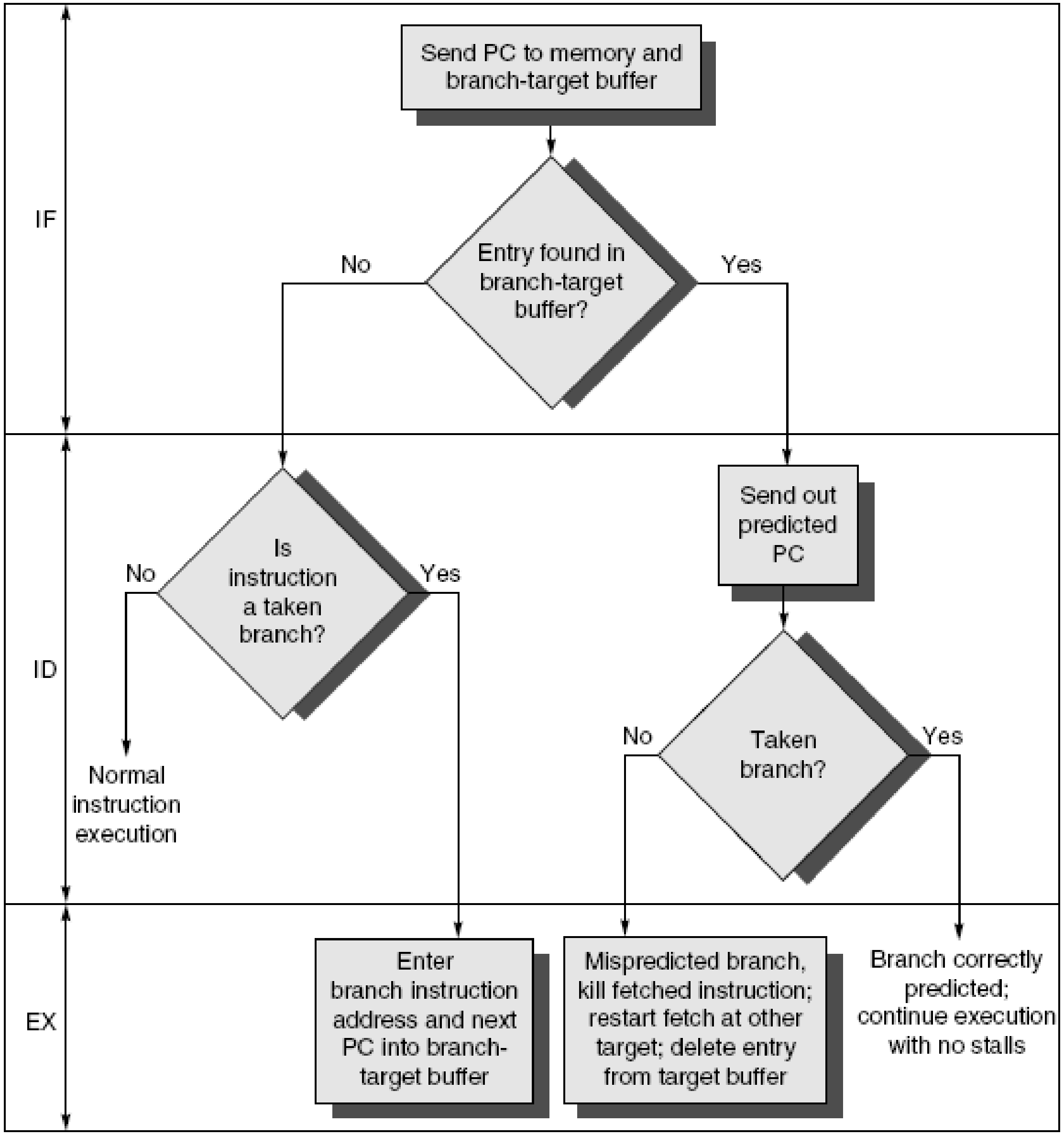
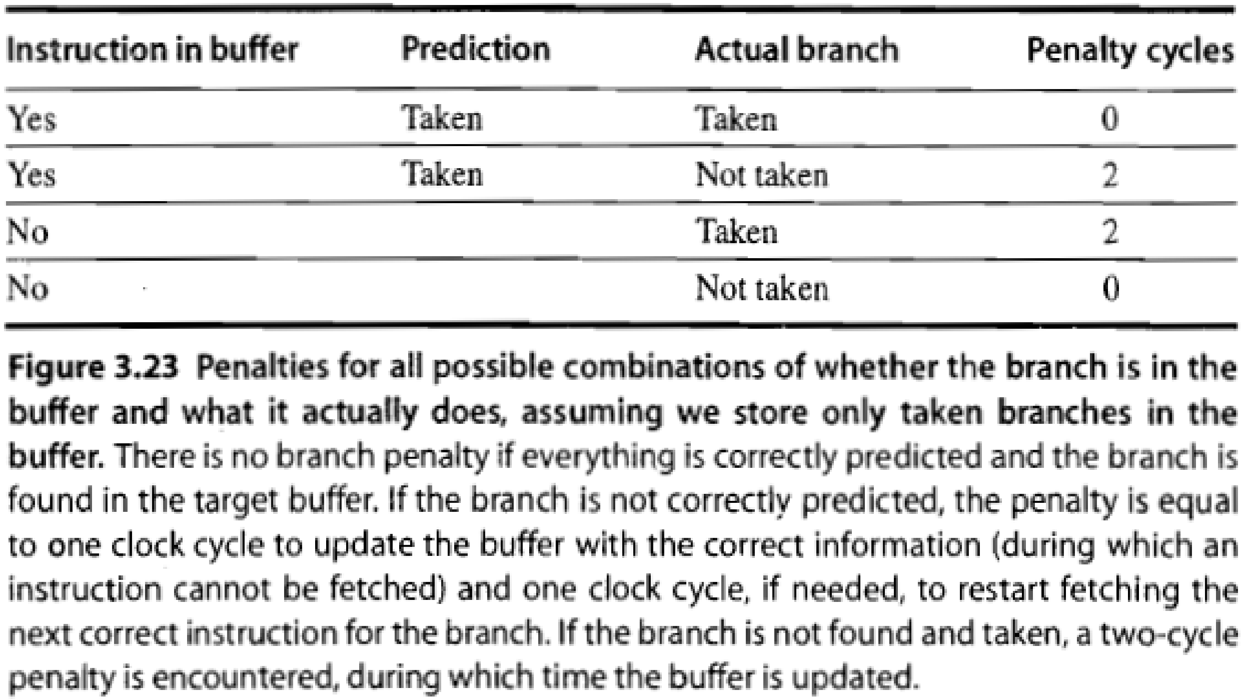

IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

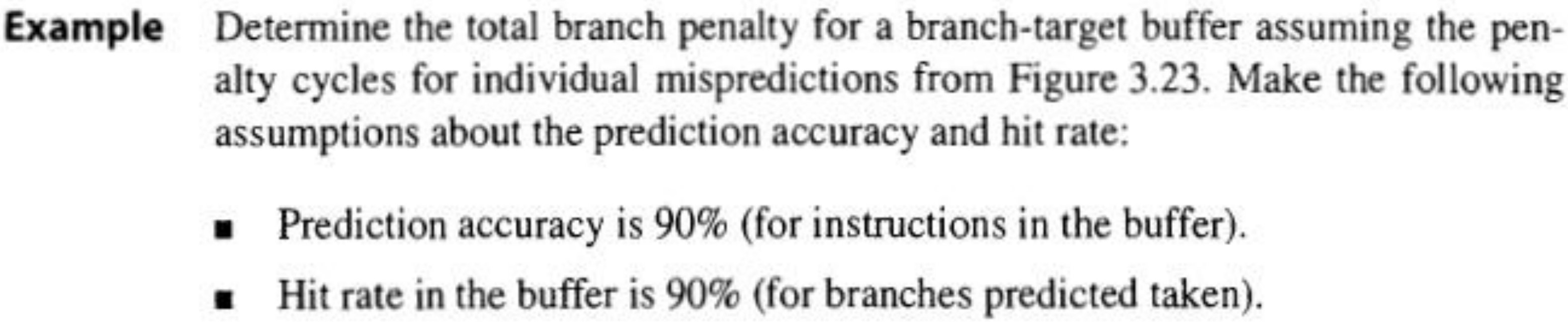
Figure 3.22 The steps involved in handling an instruction with a branch-target buffer.

Exmpl p205: penalidade



Instruction in buffer	Prediction	Actual branch	Penalty cycles
Yes	Taken	Taken	0
Yes	Taken	Not taken	2
No		Taken	2
No		Not taken	0

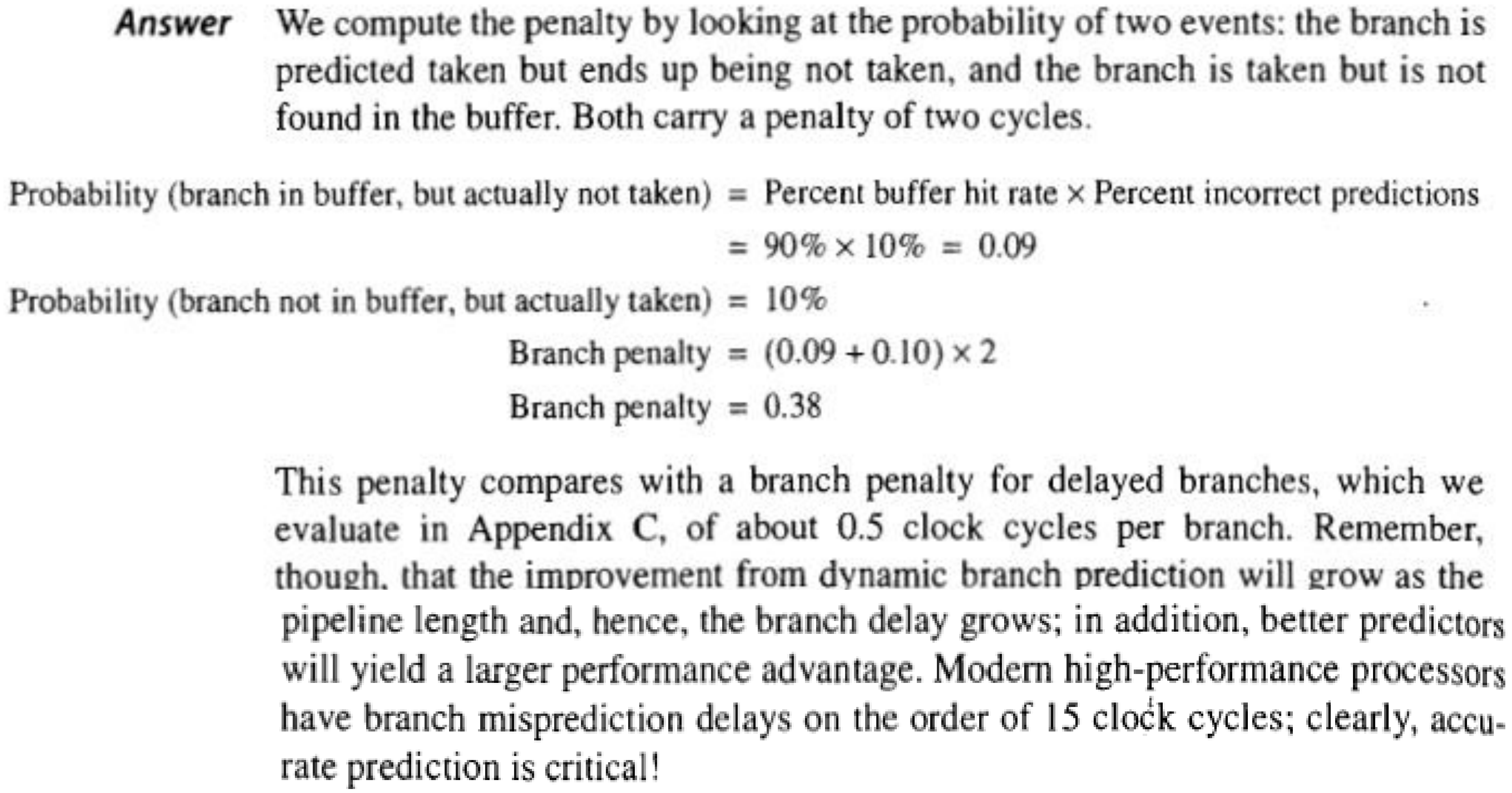
Figure 3.23 Penalties for all possible combinations of whether the branch is in the buffer and what it actually does, assuming we store only taken branches in the buffer. There is no branch penalty if everything is correctly predicted and the branch is found in the target buffer. If the branch is not correctly predicted, the penalty is equal to one clock cycle to update the buffer with the correct information (during which an instruction cannot be fetched) and one clock cycle, if needed, to restart fetching the next correct instruction for the branch. If the branch is not found and taken, a two-cycle penalty is encountered, during which time the buffer is updated.



Example Determine the total branch penalty for a branch-target buffer assuming the penalty cycles for individual mispredictions from Figure 3.23. Make the following assumptions about the prediction accuracy and hit rate:

- Prediction accuracy is 90% (for instructions in the buffer).
- Hit rate in the buffer is 90% (for branches predicted taken).

Exmpl p205: penalidade



Answer We compute the penalty by looking at the probability of two events: the branch is predicted taken but ends up being not taken, and the branch is taken but is not found in the buffer. Both carry a penalty of two cycles.

$$\begin{aligned}
\text{Probability (branch in buffer, but actually not taken)} &= \text{Percent buffer hit rate} \times \text{Percent incorrect predictions} \\
&= 90\% \times 10\% = 0.09 \\
\text{Probability (branch not in buffer, but actually taken)} &= 10\% \\
\text{Branch penalty} &= (0.09 + 0.10) \times 2 \\
\text{Branch penalty} &= 0.38
\end{aligned}$$

This penalty compares with a branch penalty for delayed branches, which we evaluate in Appendix C, of about 0.5 clock cycles per branch. Remember, though, that the improvement from dynamic branch prediction will grow as the pipeline length and, hence, the branch delay grows; in addition, better predictors will yield a larger performance advantage. Modern high-performance processors have branch misprediction delays on the order of 15 clock cycles; clearly, accurate prediction is critical!

Branch Folding

- Optimization:
  - Larger branch-target buffer
  - Add target instruction into buffer to deal with longer decoding time required by larger buffer
  - Allows “Branch folding”
- Branch folding
  - With unconditional branch: o hardware permite “pular” o jump (cuja única função é mudar o PC)
  - In some cases, also with conditional branch

Return Address Predictor

- Most unconditional branches come from function returns
  - Indirect jump: JR (target muda em tempo de execução)
  - SPEC95: retorno de procedimento = 15% de todos os branches e aproximadamente 100% dos desvios incondicionais
- The same procedure can be called from multiple sites
  - Causes the buffer to potentially forget about the return address from previous calls (changes at runtime)
  - SPEC CPU95: retorno de procedimento → misprediction = 40%
- Create return address buffer organized as a stack
  - melhora consideravelmente o desempenho (fig 3.24)
- (usado pelo Intel Core e AMD Phenom)

Desempenho do Return Address Predictor

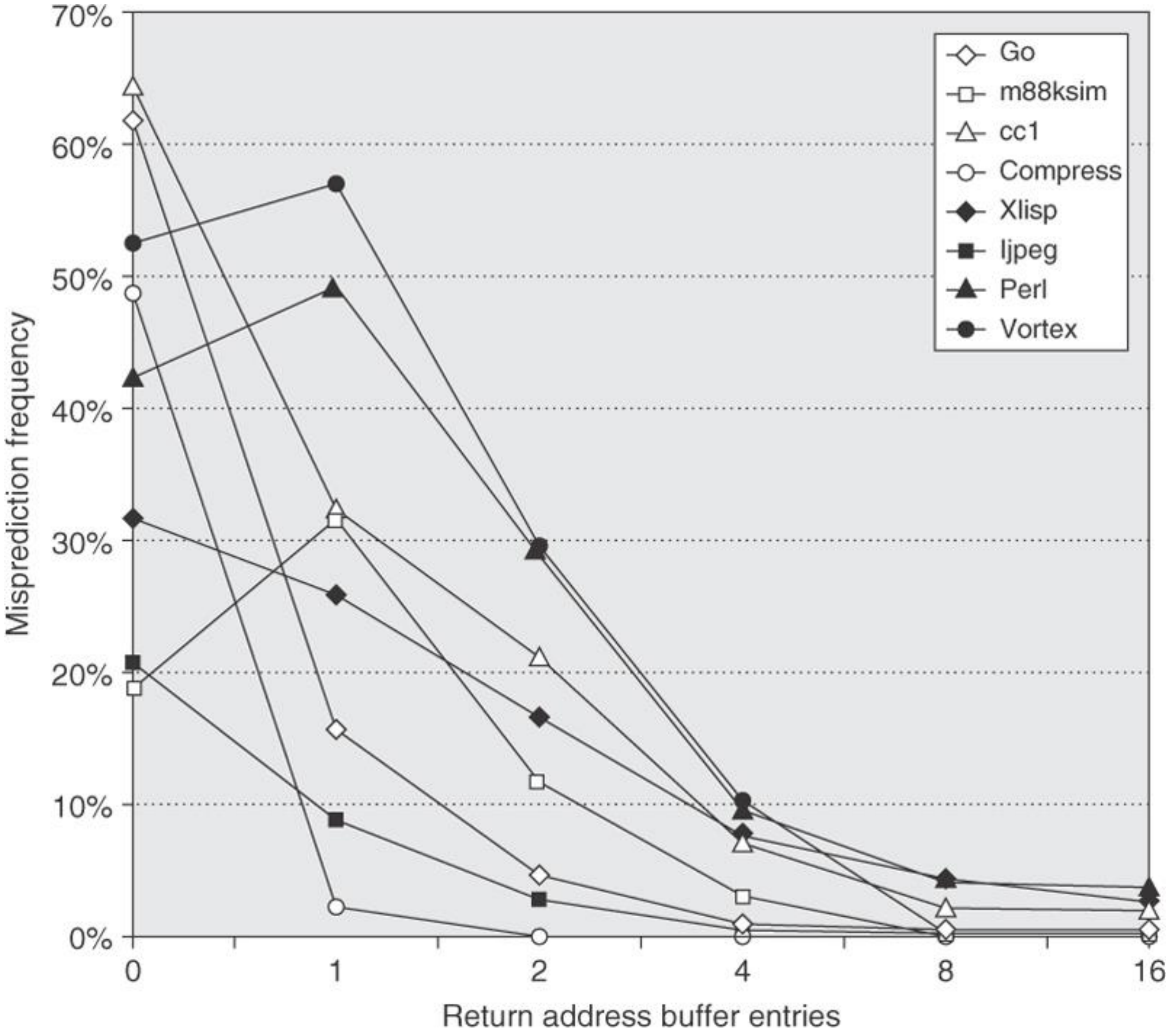


Figure 3.24 Prediction accuracy for a return address buffer operated as a stack on a number of SPEC CPU95 benchmarks. The accuracy is the fraction of return addresses predicted correctly. A buffer of 0 entries implies that the standard branch prediction is used. Since call depths are typically not large, with some exceptions, a modest buffer works well. These data come from Skadron et al. [1999] and use a fix-up mechanism to prevent corruption of the cached return addresses.

Integrated Instruction Fetch Unit

- Design monolithic unit that performs:
  - Integrated branch prediction:
    - parte da instruction fetch
  - Instruction prefetch
    - Fetch ahead
  - Instruction memory access and buffering
    - Accessing multiple cache lines
    - Deal (hide) with crossing cache lines
- (used by all high-end processors)

Register Renaming

- Register renaming vs. reorder buffers
  - Instead of virtual registers from reservation stations and reorder buffer, create a single register pool
    - Contains visible registers and virtual registers
  - Use hardware-based map to rename registers during issue
  - WAW and WAR hazards are avoided
  - Speculation recovery occurs by copying during commit
  - Still need a ROB-like queue to update table in order
  - Simplifies commit:
    - Record that mapping between architectural register and physical register is no longer speculative
    - Free up physical register used to hold older value
    - In other words: SWAP physical registers on commit
  - Physical register de-allocation is more difficult

Integrated Issue and Renaming

- Combining instruction issue with register renaming:
  - Issue logic pre-reserves enough physical registers for the bundle (ex: 4 registers for a 4 instruction bundle, 1 reg / result)
  - Issue logic finds dependencies within bundle, maps registers as necessary
  - Issue logic finds dependencies between current bundle and already in-flight bundles, maps registers as necessary

- Como no ROB, o hardware deve determinar as dependências e atualizar as tabelas de renaming em um único clock
  - quanto maior o número de instruções emitidas por clock, mais complicado

How Much?

- How much to speculate
  - Mis-speculation degrades performance and power relative to no speculation
    - May cause additional misses (cache, TLB)
  - Prevent speculative code from causing higher costing misses (e.g. L2)

- Speculating through multiple branches
  - Poderia ser útil em
    - very high branch frequency
    - branch clustering
    - long delay in FUs
  - Complicates speculation recovery (mas o resto seria simples)
  - Até 2011, esquema não utilizado comercialmente
    - No processor can resolve multiple branches per cycle

Energy Efficiency

- Custo energético da especulação errada
  - Trabalho inútil que deve ser descartado
  - Custo adicional da recuperação
- Speculation and energy efficiency
  - Note: speculation is only energy efficient when it significantly improves performance
- Se um número grande de instruções desnecessárias estão sendo executadas, é provável que, além do custo de energia, também o desempenho está piorando
  - fig 3.25 → resultado ruim para inteiros → provável que cause baixa eficiência energética

Fração de instruções desnecessárias

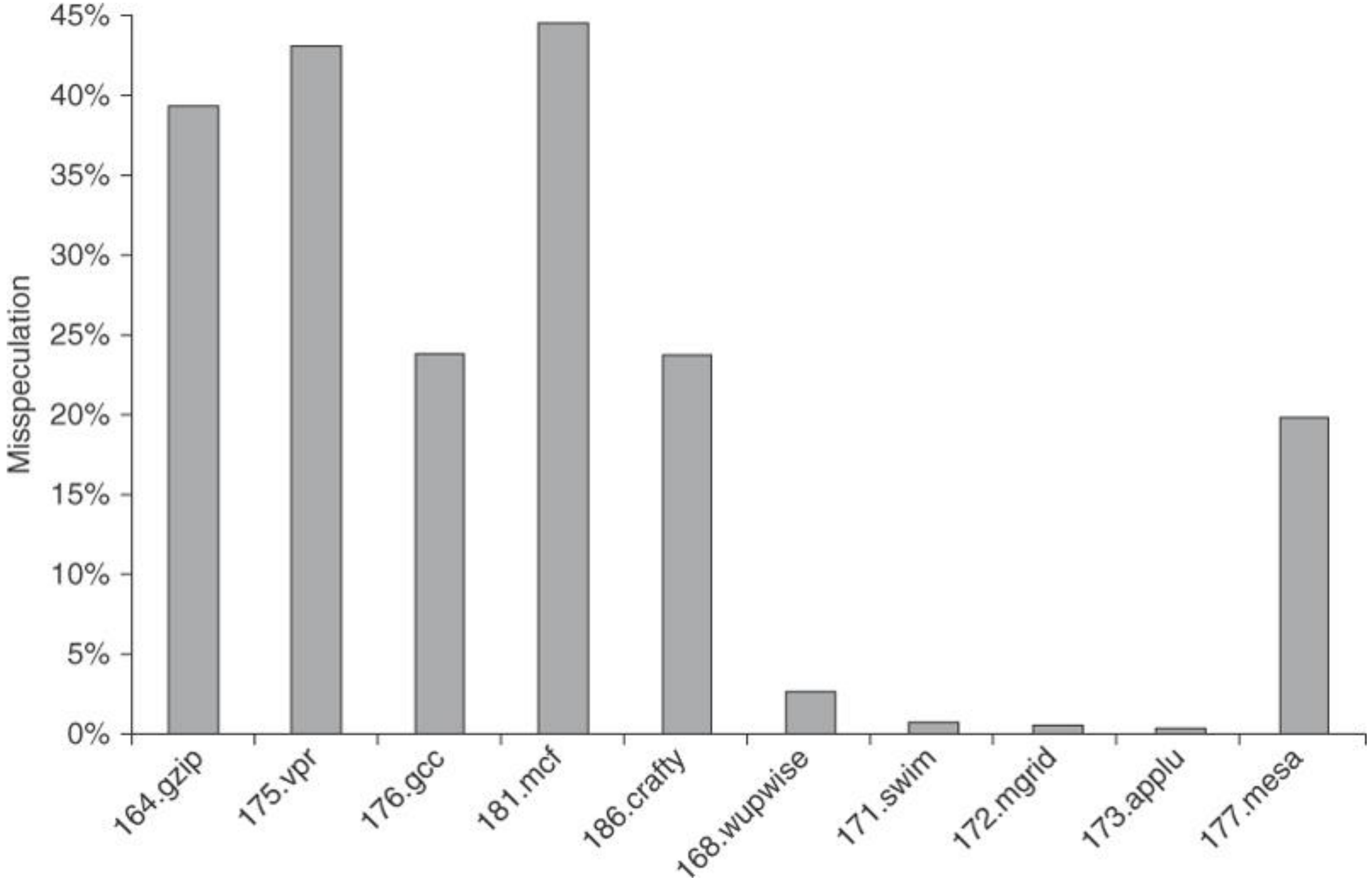


Figure 3.25 The fraction of instructions that are executed as a result of misspeculation is typically much higher for integer Programs (the first five) versus FP programs (the last five).

Value prediction

- Tenta predizer o resultado das instruções
  - Em geral, difícil
- Casos de aplicabilidade:
  - Loads that load from a pool of constants (or values that change unfrequently)
  - Instruction that produces a value from a small set of values (possível prever de comportamentos observados anteriormente)
- Not been incorporated into modern processors
- Similar idea – address aliasing prediction – is used on some processors
  - para prever se dois ST ou um LD e um ST apontam para o mesmo endereço
  - caso negativo, instruções podem ser reordenadas
  - em uso limitado ainda hoje

3.10 Limitações do ILP

- ILP: pipelined processors (60´s), key to perfomance improvements (80´s 90´s)
- Estudos atuais → limitações
  - especulação muito agressivas → alto custo (área, power)
  - mesmo os principais defensores → mudança de idéia (2005)

- (artigo importante: Wall 1993)

Modelo de HW para estudo

- Modelo de hardware para estudos: computador ideal, onde o único limite ao ILP é imposto pelo data flow do programa
  - 1. Infinite register renaming
  - 2. Perfect branch prediction
  - 3. Perfect jump prediction (including indirect jump register)
  - 4. Perfect memory address aliasing analysis: todos os endereços efetivos são conhecidos (possível reordenar LD/ST)
  - 5. Perfect caches: acessos uniformes com 1 ciclo
- Hipóteses 2 e 3 eliminam control dependencies; 1 e 4 todas as outras dependências exceto true data dependences
- Prefetching infinito, capacidade de múltiplo (infinito) issue
- FUs tem latência de 1 ciclo
- Esta máquina ideal é irrealizável hoje
  - Power 7 (mais avançado superescalar): issue 6 instructions / clock, SMT, large set of renaming registers (allowing 100´s instructions to be in flight)

ILP em um processador perfeito

- Set of benchmarks → program trace → schedule as early as possible (perfect branch prediction)
- Measure: average instruction issue rate

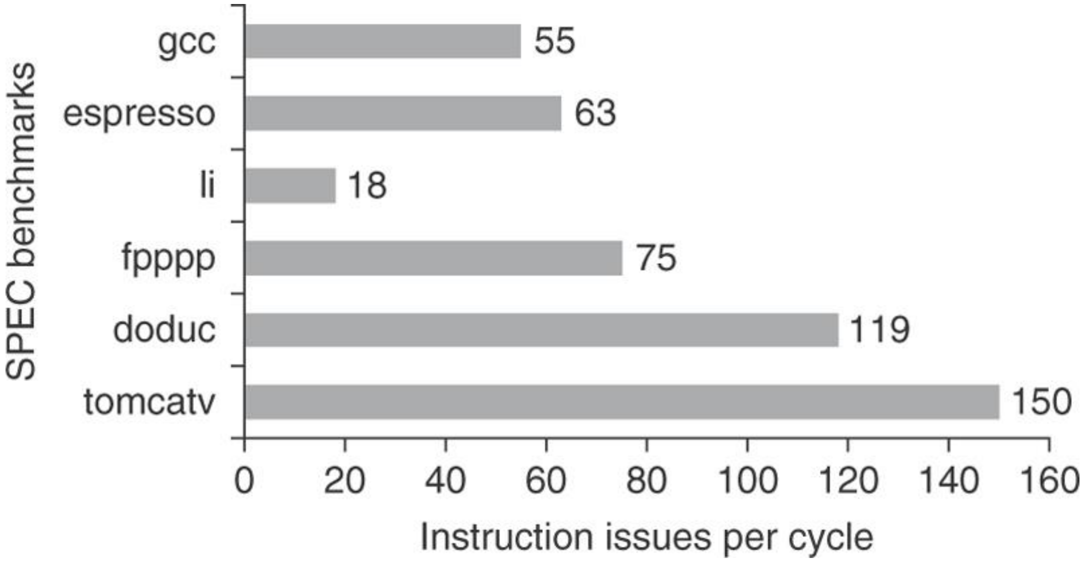


Figure 3.26 ILP available in a perfect processor for six of the SPEC92 benchmarks. The first three programs are integer programs, and the last three are floating-point programs. The floating-point programs are loop intensive and have large amounts of loop-level parallelism.

ILP para processadores realizáveis (hoje)

- Até 64 instruction issues / clock (10x valor disponível hoje)
- Tournament predictor com 1K linhas e resultado (predictor) de 16 linhas
- Perfect desambiguation of memory references, on the fly
- Very large register renaming set

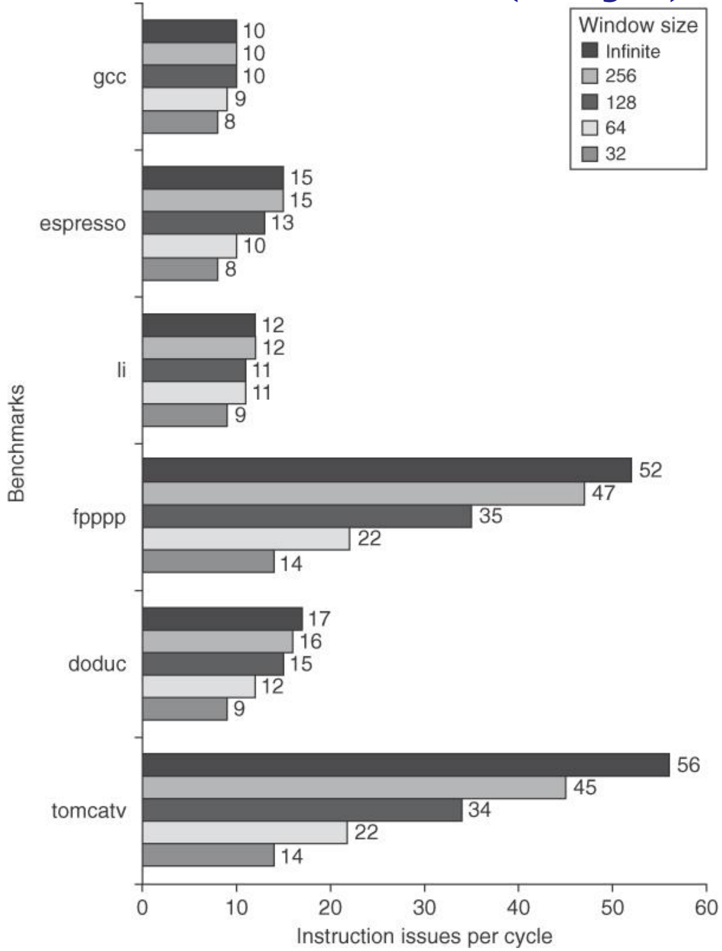
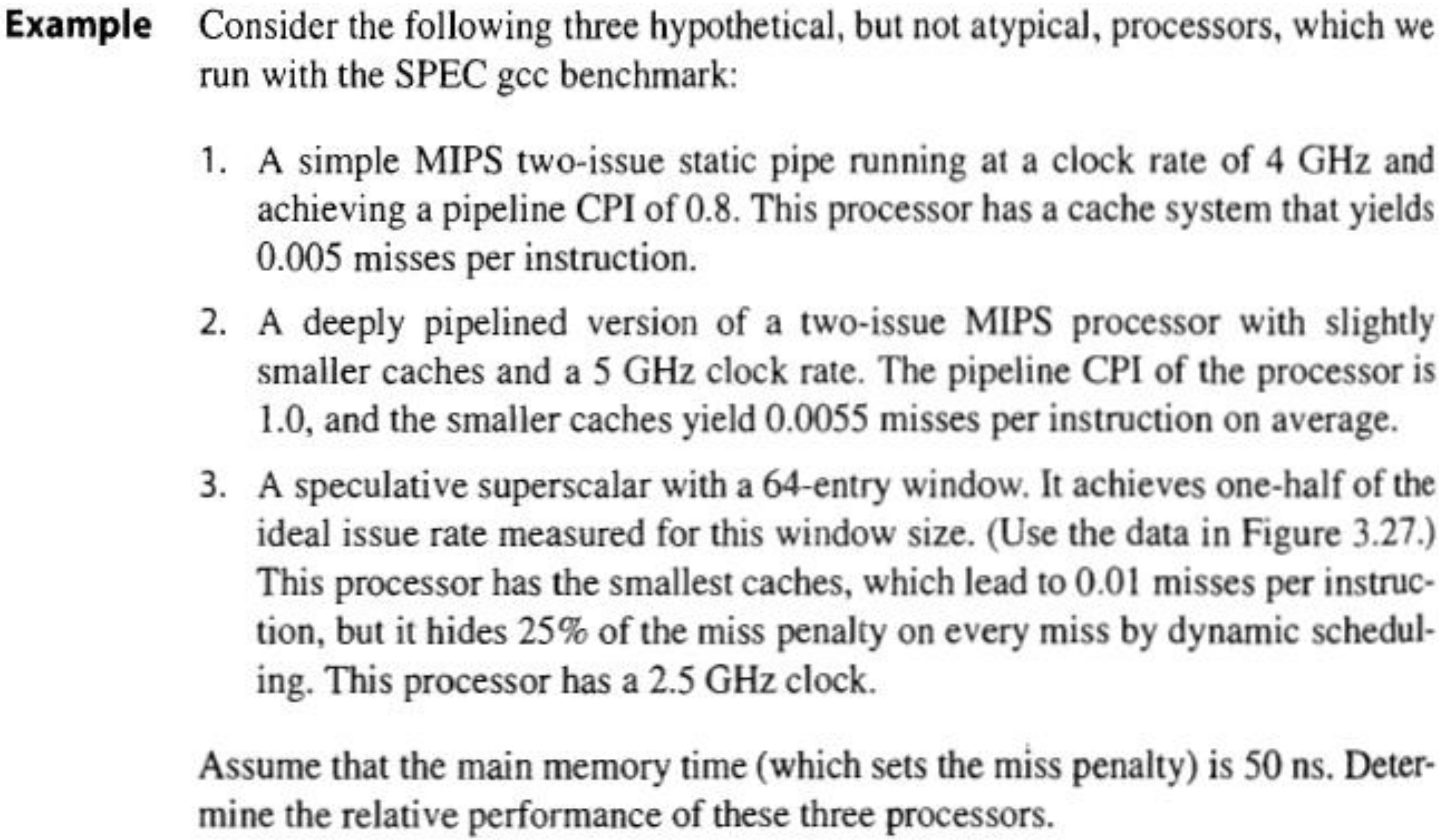


Figure 3.27 The amount of parallelism available versus the window size for a variety of integer and floating-point programs with up to 64 arbitrary instruction issues per clock. Although there are fewer renaming registers than the window size, the fact that all operations have one-cycle latency and the number of renaming registers equals the issue width allows the processor to exploit parallelism within the entire window. In a real implementation, the window size and the number of renaming registers must be balanced to prevent one of these factors from overly constraining the issue rate.

Exmpl p 218: comparação desempenho

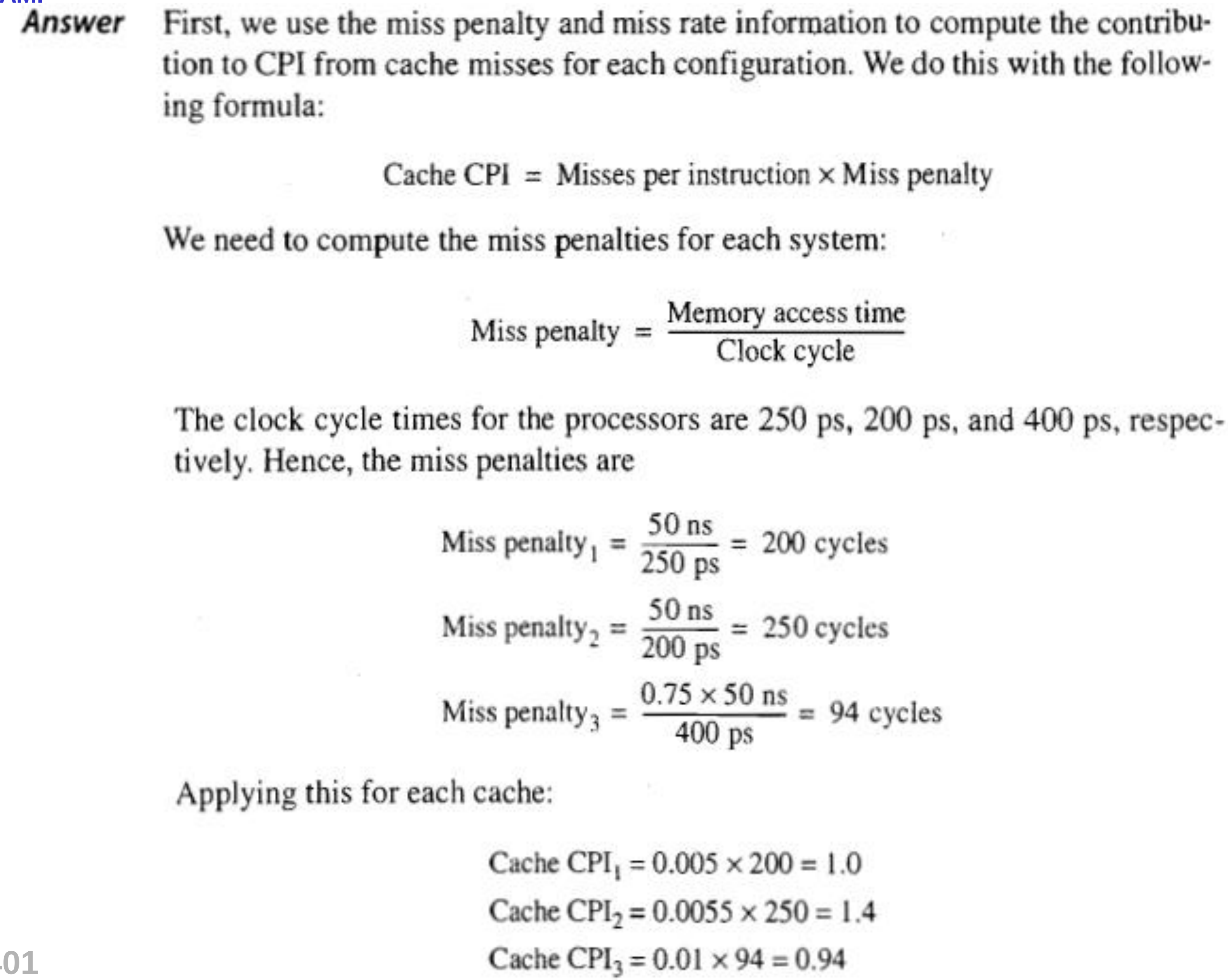


Example Consider the following three hypothetical, but not atypical, processors, which we run with the SPEC gcc benchmark:

1. A simple MIPS two-issue static pipe running at a clock rate of 4 GHz and achieving a pipeline CPI of 0.8. This processor has a cache system that yields 0.005 misses per instruction.
2. A deeply pipelined version of a two-issue MIPS processor with slightly smaller caches and a 5 GHz clock rate. The pipeline CPI of the processor is 1.0, and the smaller caches yield 0.0055 misses per instruction on average.
3. A speculative superscalar with a 64-entry window. It achieves one-half of the ideal issue rate measured for this window size. (Use the data in Figure 3.27.) This processor has the smallest caches, which lead to 0.01 misses per instruction, but it hides 25% of the miss penalty on every miss by dynamic scheduling. This processor has a 2.5 GHz clock.

Assume that the main memory time (which sets the miss penalty) is 50 ns. Determine the relative performance of these three processors.

Exmpl p 218: comparação desempenho (2)



Answer First, we use the miss penalty and miss rate information to compute the contribution to CPI from cache misses for each configuration. We do this with the following formula:

$$\text{Cache CPI} = \text{Misses per instruction} \times \text{Miss penalty}$$

We need to compute the miss penalties for each system:

$$\text{Miss penalty} = \frac{\text{Memory access time}}{\text{Clock cycle}}$$

The clock cycle times for the processors are 250 ps, 200 ps, and 400 ps, respectively. Hence, the miss penalties are

$$\text{Miss penalty}_1 = \frac{50\text{ ns}}{250\text{ ps}} = 200\text{ cycles}$$

$$\text{Miss penalty}_2 = \frac{50\text{ ns}}{200\text{ ps}} = 250\text{ cycles}$$

$$\text{Miss penalty}_3 = \frac{0.75 \times 50\text{ ns}}{400\text{ ps}} = 94\text{ cycles}$$

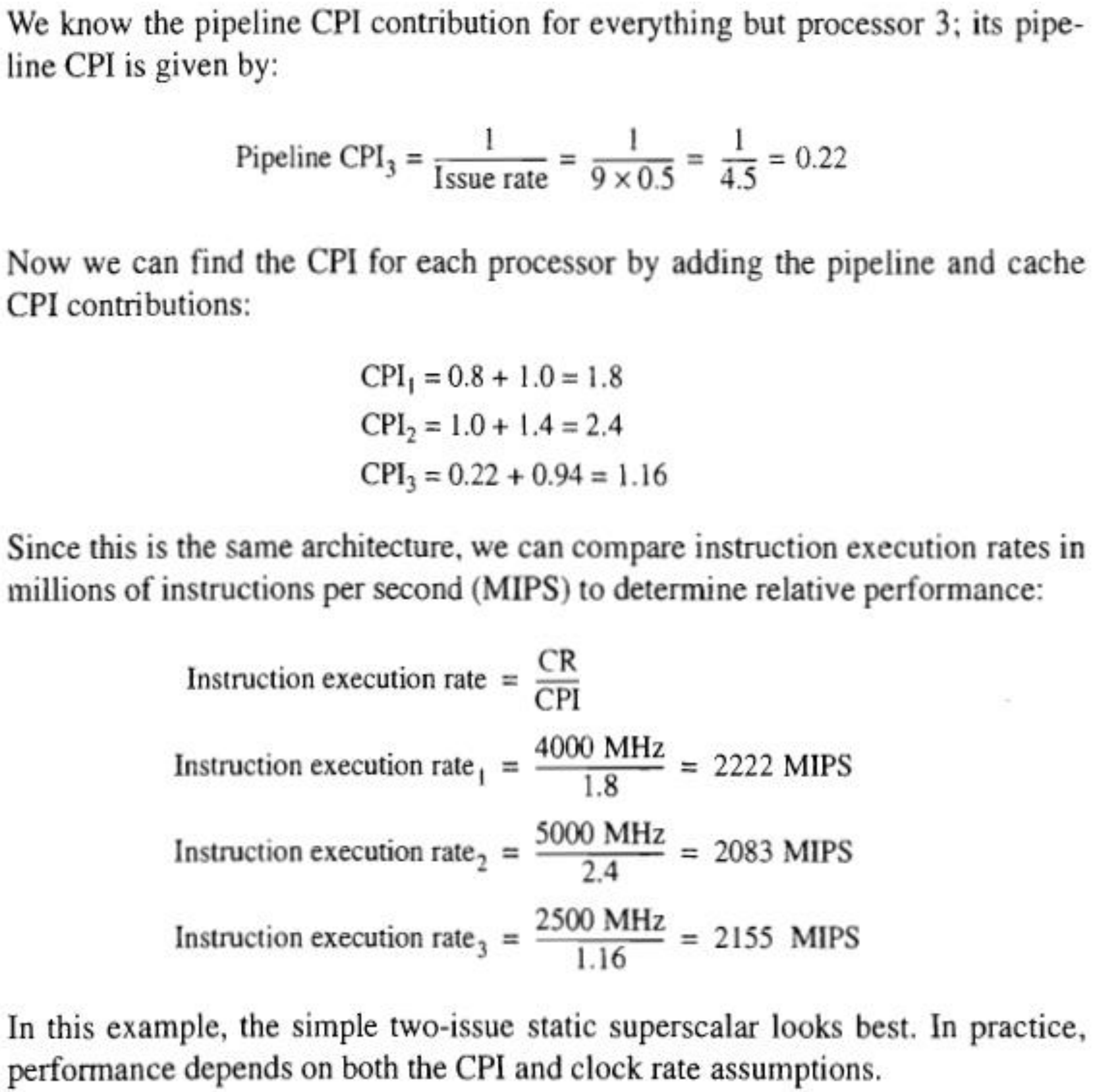
Applying this for each cache:

$$\text{Cache CPI}_1 = 0.005 \times 200 = 1.0$$

$$\text{Cache CPI}_2 = 0.0055 \times 250 = 1.4$$

$$\text{Cache CPI}_3 = 0.01 \times 94 = 0.94$$

Exmpl p 218: (3)



We know the pipeline CPI contribution for everything but processor 3; its pipeline CPI is given by:

$$\text{Pipeline CPI}_3 = \frac{1}{\text{Issue rate}} = \frac{1}{9 \times 0.5} = \frac{1}{4.5} = 0.22$$

Now we can find the CPI for each processor by adding the pipeline and cache CPI contributions:

$$\text{CPI}_1 = 0.8 + 1.0 = 1.8$$
$$\text{CPI}_2 = 1.0 + 1.4 = 2.4$$
$$\text{CPI}_3 = 0.22 + 0.94 = 1.16$$

Since this is the same architecture, we can compare instruction execution rates in millions of instructions per second (MIPS) to determine relative performance:

$$\text{Instruction execution rate} = \frac{\text{CR}}{\text{CPI}}$$

$$\text{Instruction execution rate}_1 = \frac{4000 \text{ MHz}}{1.8} = 2222 \text{ MIPS}$$

$$\text{Instruction execution rate}_2 = \frac{5000 \text{ MHz}}{2.4} = 2083 \text{ MIPS}$$

$$\text{Instruction execution rate}_3 = \frac{2500 \text{ MHz}}{1.16} = 2155 \text{ MIPS}$$

In this example, the simple two-issue static superscalar looks best. In practice, performance depends on both the CPI and clock rate assumptions.

Conclusões

- Limitations of this study
  - WAW e WAR through memory: hipóteses simplificadores subestimaram o efeito desses hazards
  - Dependência desnecessária: algumas dependências reais (RAW) poderiam ser eliminadas (por ex, por loop unrolling)
  - Value prediction não foi considerado (poderia melhorar ILP)

- Limites observados de ILP são intrínsecos, e não podem ser superados por avanços tecnológicos por exemplo
  - Dificuldades para melhorar são imensas
  - ILP wall

3.11 ILP and the memory system

- Hardware versus Software Speculation – trade offs
  - Memory disambiguation → enable extensive speculation; Difficult to do at compile time → hardware based and dynamic disambiguation
  - HW based speculation better when control flow unpredictable
  - HW based better for precise exception
  - HW based does not require additional compensation or bookkeeping code
  - Compiler based benefit: it can “see” ahead in code (statically) → better code scheduling
  - HW based does not require different code to different implementations of an architecture → Vantagem extremamente relevante
  - HW based → complex implementation
  - Some designers try hybrid approaches
  - Most ambitious design with compiler based speculation → Itanium → did not deliver the expected performance

ILP and the memory system (2)

- Speculative execution and the memory system
  - Especulação pode gerar endereços inválidos (que não apareceriam sem especulação) → (false) exception overhead → memória deve identificar a especulação e desprezar a exceção
  - Especulação pode gerar cache miss → importante o uso de non blocking caches
    - penalidade em L2 é tão grande que normalmente compiladores somente especulam em L1

3.12 Multithreading (in uniprocessor)

- Crosscutting issue
  - pipeline, uniprocessor (ch 3)
  - graphics processing units (ch 4)
  - multiprocessors (ch5)
- Explorando paralelismo em uniprocessadores
  - Uso de ILP: limites
    - principalmente, em altas taxas de issue/clock → difícil esconder cache misses
  - Em On-line Transaction Processing → paralelismo natural (multiprogramação)
  - Em programação científica → paralelismo natural, se explorarmos threads independentes
    - também em aplicações desktop (muitas tarefas em paralelo)
- Paralelismo em multiprocessador: replicated processor
- Multithread in uniprocessor: replicated PC and private state

Multithreading: aspectos gerais

- Per-thread state
  - separate: PC, register file, page table
  - memory: ok to share via virtual memory (como em multiprogramação)
- HW deve permitir mudança de thread rapidamente
  - thread switch should be much faster than process switch
- Threads devem estar identificadas no código
  - pelo compilador ou pelo programador
- Granularidade do Multithreading
  - Fine Grain: thread switch in each clock. Round-robin interleaving (skip stalled). Advantadge: hides short/long stalls. Disadvante: slows down individual thread (latency). Trade-off **throughput** x latency. Used by Sun Niagara and NVidia GPU
  - Coarse Grain: thread switch on costly stalls. Trade-off throughput x **latency,** Disadvantage: throughput losses, specially in short stalls. Pipeline start-up costs. Not used today

Multithreading Approaches

- Four different approaches (in Fig 3.28)
  - A superscalar with no multithreading support
  - A superscalar with coarse-grained multithreading
  - A superscalar with fine-grained multithreading
  - A superscalar with simultaneous multithreading
    - Fine Grain MT on top of a multiple-issue, dynamically scheduled processor
      - hides long latency events

Multithreading Approaches

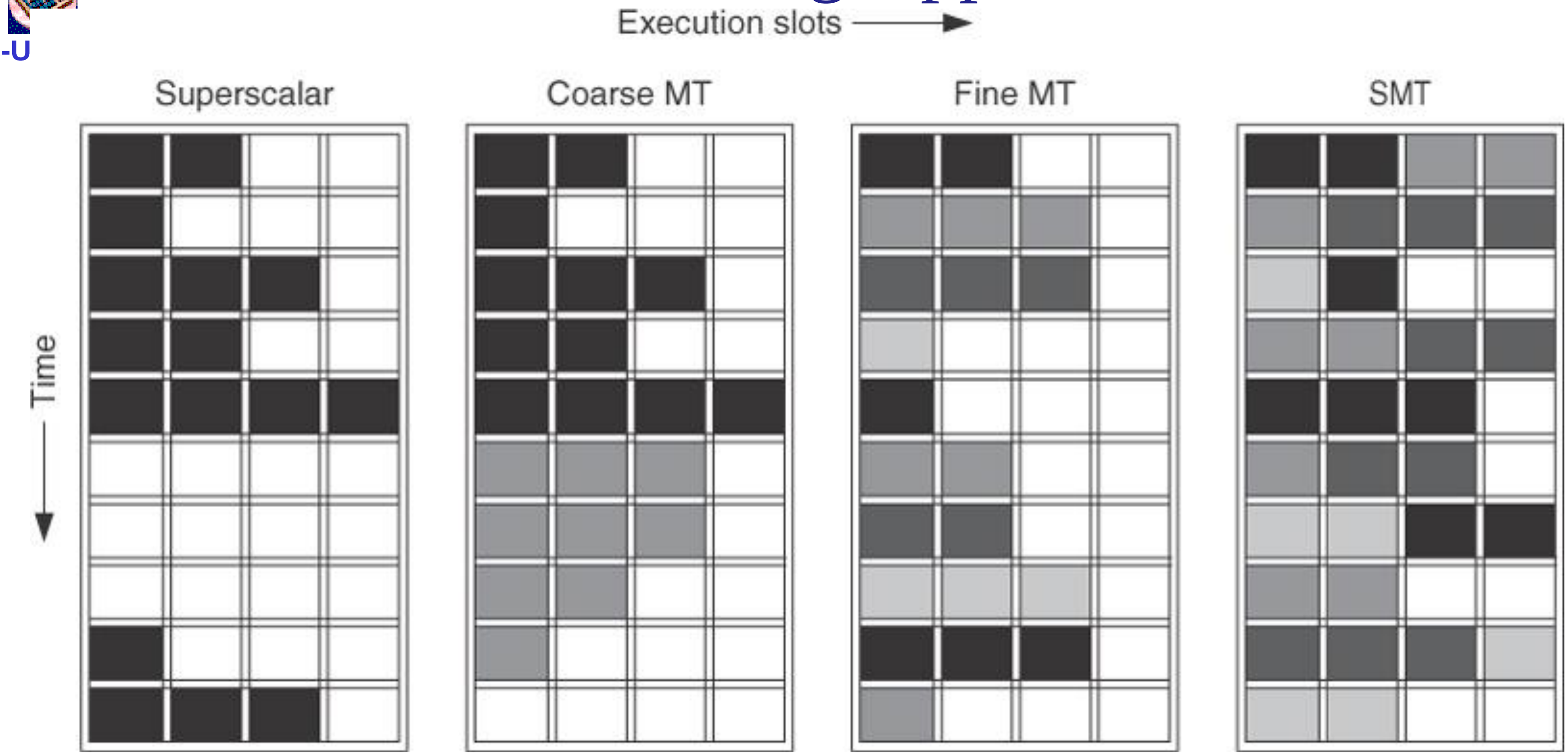
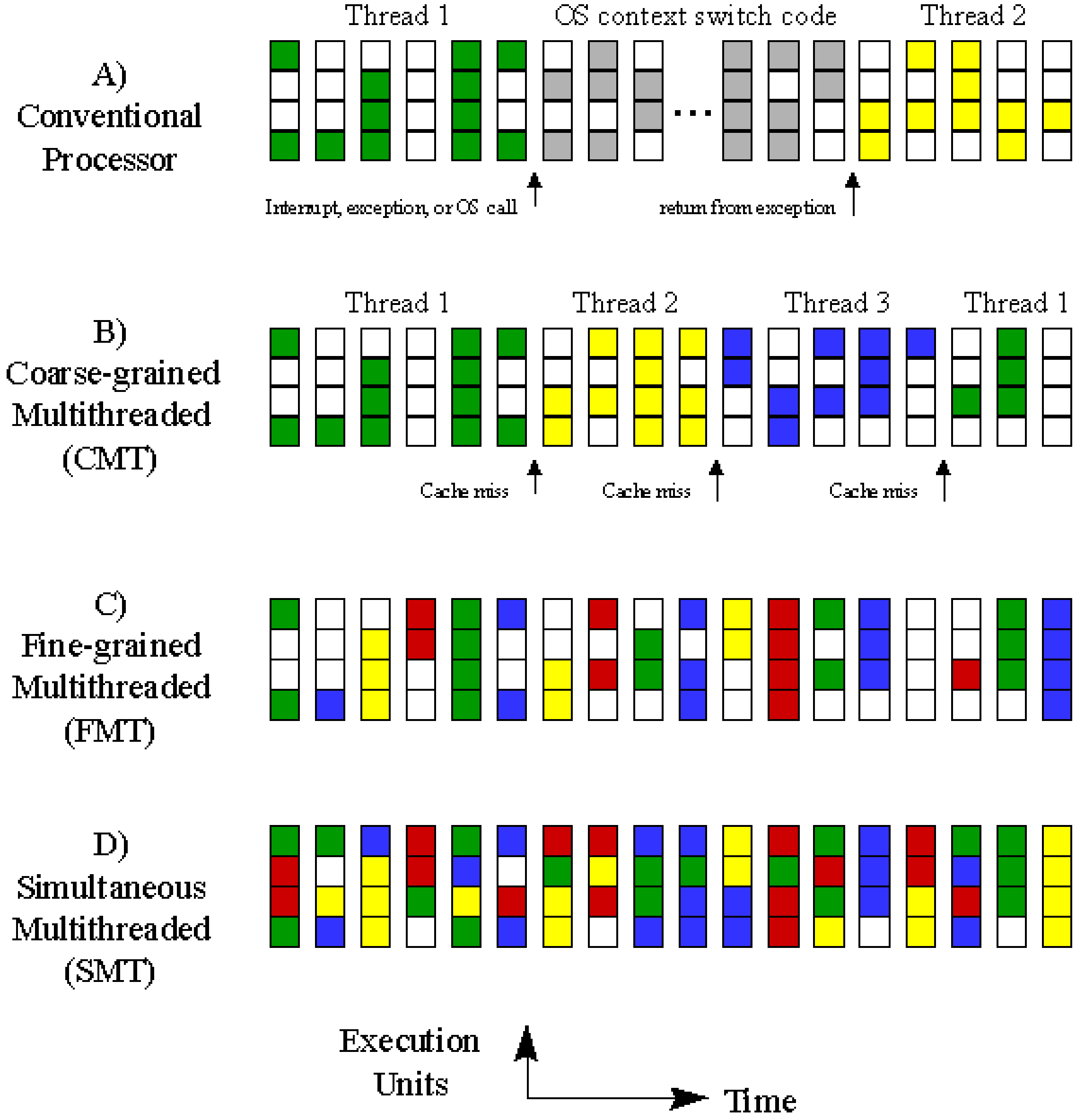


Figure 3.28 How four different approaches use the functional unit execution slots of a superscalar processor. The horizontal dimension represents the instruction execution capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding execution slot is unused in that clock cycle. The shades of gray and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support. The Sun T1 and T2 (aka Niagara) processors are fine-grained multithreaded processors, while the Intel Core i7 and IBM Power7 processors use SMT. The T2 has eight threads, the Power7 has four, and the Intel i7 has two. In all existing SMTs, instructions issue from only one thread at a time. The difference in SMT is that the subsequent decision to execute an instruction is decoupled and could execute the operations coming from several different instructions in the same clock cycle.

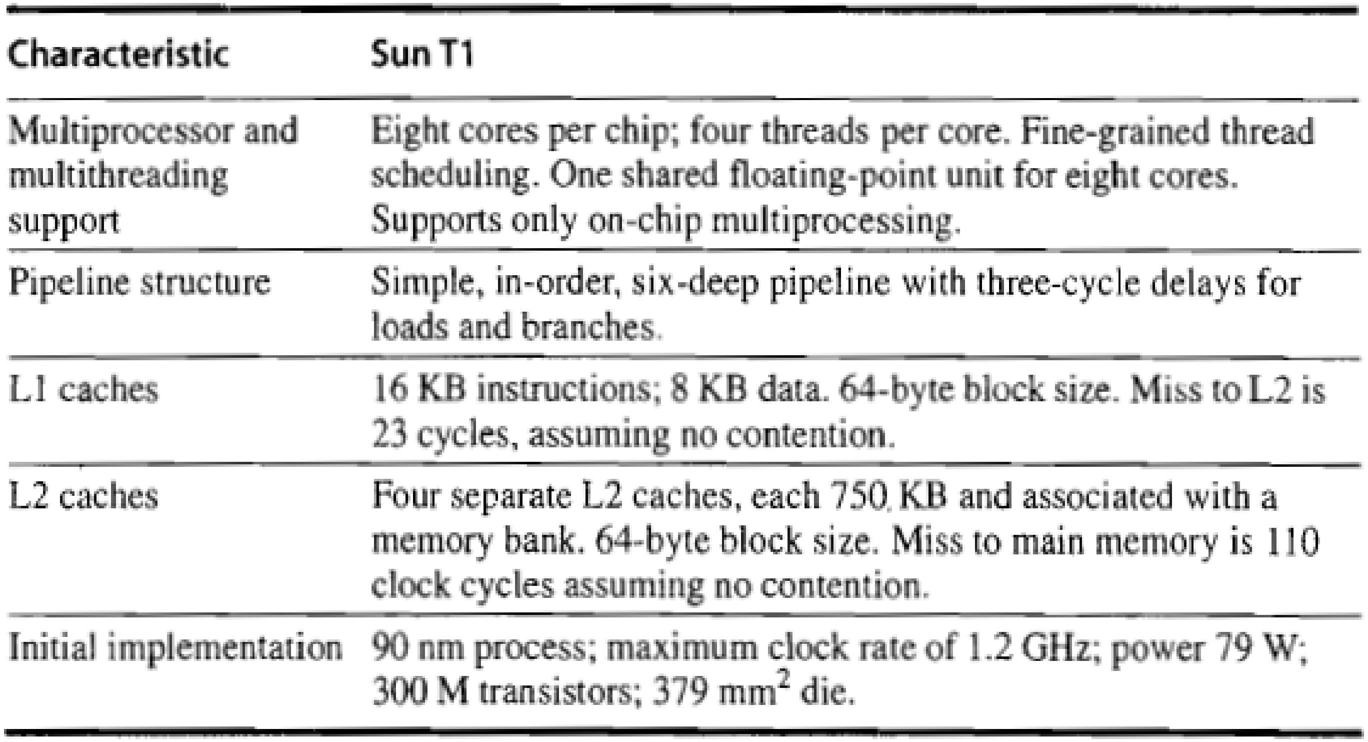
Multithreading: outra figura



http://www.realworldtech.com/alpha-ev8-smt/

FG Multithreading na SUN T1

- Foco: explorar paralelismo via TLP (e não ILP). (2005)
- FGMT 1 thread / cycle
- Core: single-issue, six-stage pipeline (5 estágios do MIPS clássico + 1 estágio para thread switch)
- Loads/branches → 3 cycle latency → hidden by other threads



Characteristic	Sun T1
Multiprocessor and multithreading support	Eight cores per chip; four threads per core. Fine-grained thread scheduling. One shared floating-point unit for eight cores. Supports only on-chip multiprocessing.
Pipeline structure	Simple, in-order, six-deep pipeline with three-cycle delays for loads and branches.
L1 caches	16 KB instructions; 8 KB data. 64-byte block size. Miss to L2 is 23 cycles, assuming no contention.
L2 caches	Four separate L2 caches, each 750 KB and associated with a memory bank. 64-byte block size. Miss to main memory is 110 clock cycles assuming no contention.
Initial implementation	90 nm process; maximum clock rate of 1.2 GHz; power 79 W; 300 M transistors; 379 mm^2 die.

Figure 3.29 A summary of the T1 processor.

Effect of FGMT on T1 cache performance

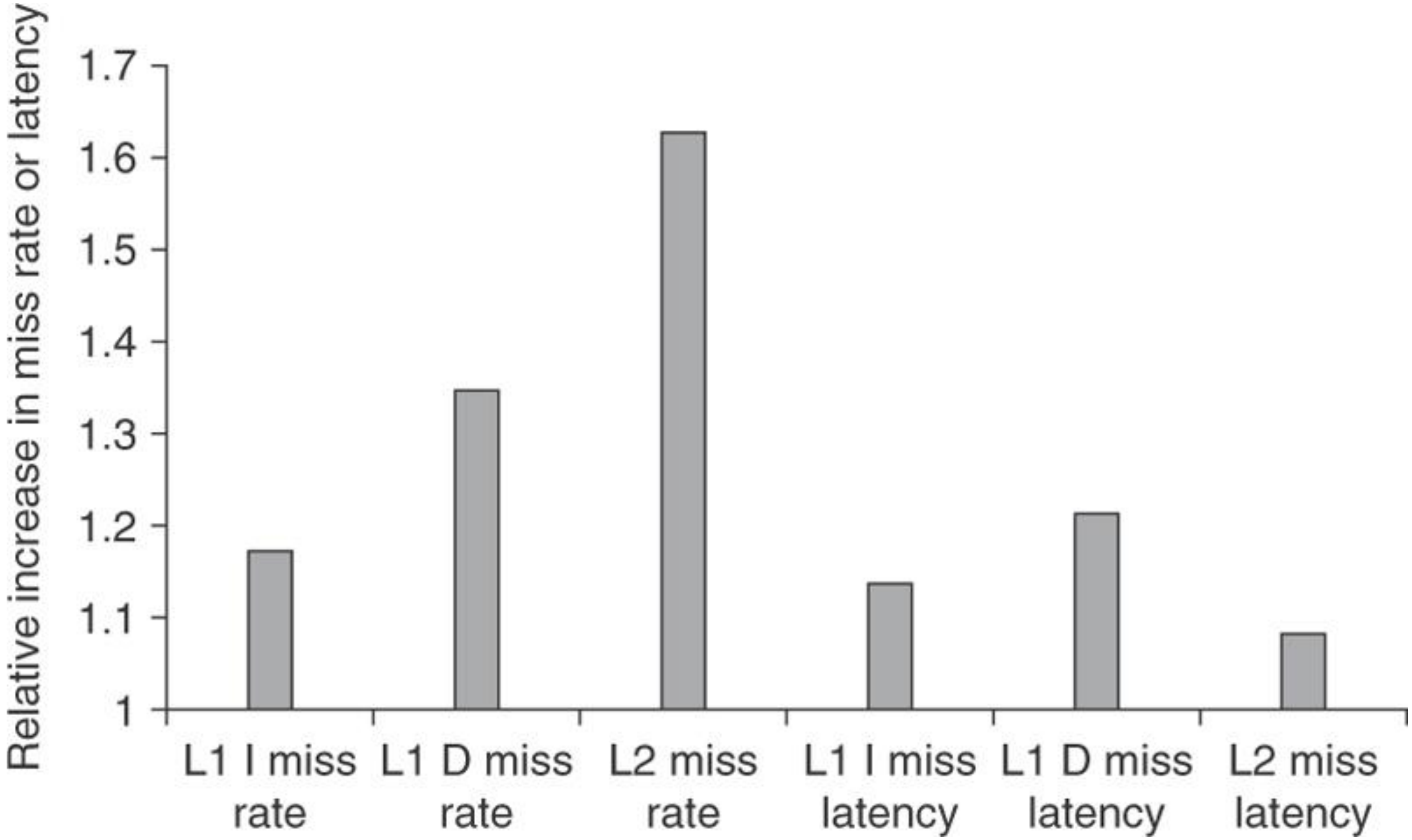


Figure 3.30 The relative change in the miss rates and miss latencies when executing with one thread per core versus four threads per core on the TPC-C benchmark. The latencies are the actual time to return the requested data after a miss. In the four-thread case, the execution of other threads could potentially hide much of this latency.

Effect of FGMT on T1 cache performance

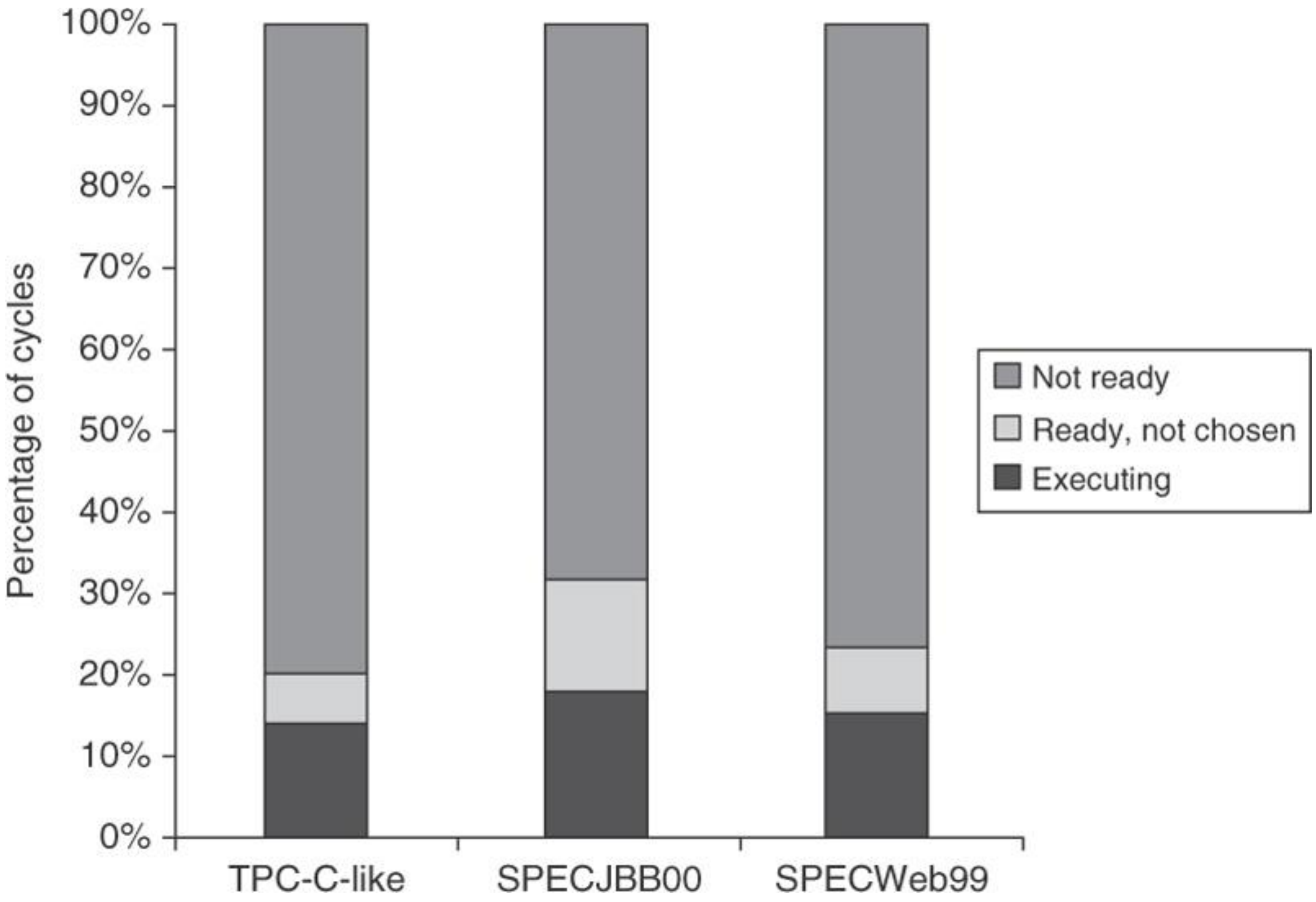
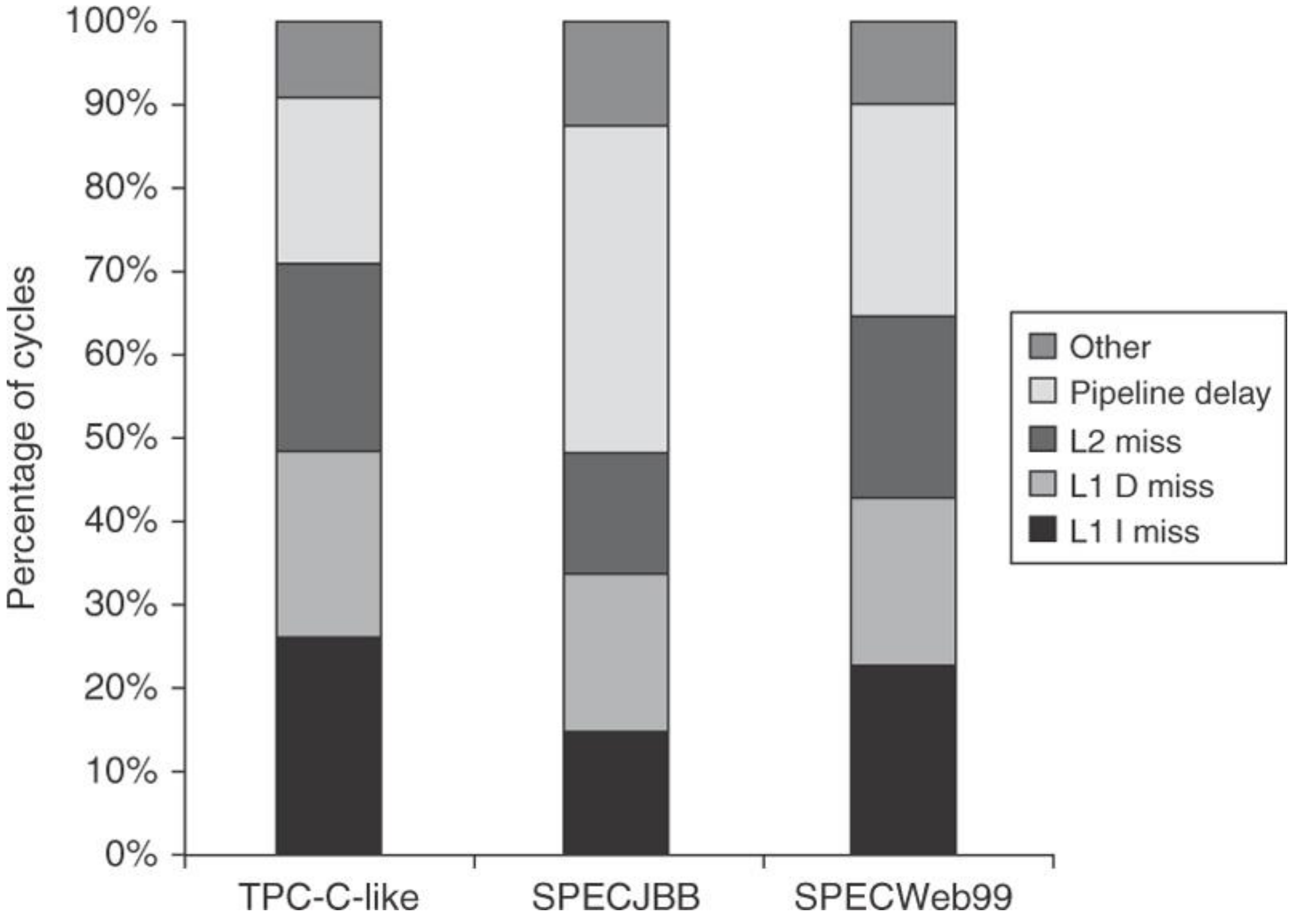


Figure 3.31 Breakdown of the status on an average thread. “Executing” indicates the thread issues an instruction in that cycle. “Ready but not chosen” means it could issue but another thread has been chosen, and “not ready” indicates that the thread is awaiting the completion of an event (a pipeline delay or cache miss, for example).

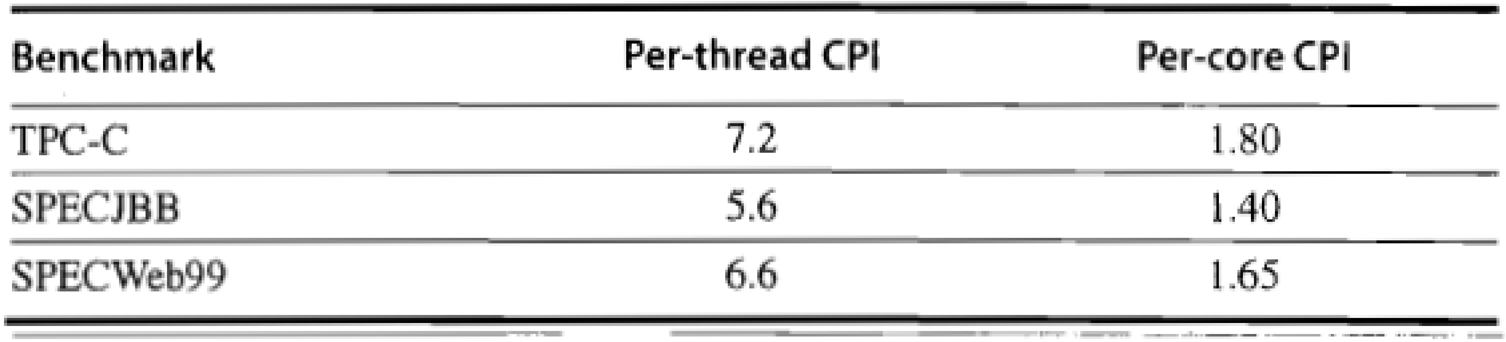
Thread not ready



} cache misses: 50-75%

Figure 3.32 The breakdown of causes for a thread being not ready. The contribution to the "other" category varies. In PC-C, store buffer full is the largest contributor; in SPEC-JBB, atomic instructions are the largest contributor; and in SPECWeb99, both factors contribute.

CPI



Benchmark	Per-thread CPI	Per-core CPI
TPC-C	7.2	1.80
SPECJBB	5.6	1.40
SPECWeb99	6.6	1.65

Figure 3.33 The per-thread CPI, the per-core CPI, the effective eight-core CPI, and the effective IPC (inverse of CPI) for the eight-core T1 processor.

- CPI / thread ideal = 4
  - cada thread consome 1 ciclo em 4
- CPI / core ideal = 1
- Resultados do T1 em 2005, parecidos com processadores muito maiores e complexos, com ILP agressivo
  - 8 cores (T1) vs 2-4 outros processadores
- 2005: T1 melhor desempenho para inteiros

Effectiveness of SMT on Superscalar

- Estudos feitos em 2000-2001 → ganhos modestos
  - H & P: condições dos experimentos tem problemas
  - Na época, grandes expectativas com ILP agressivo
- Experimentos em 2011
  - Desempenho e energy efficiency (tempo tarefa /consumo) no Intel i7 e i5 (Fig 3.35). Benchmarks usados (Fig 3.34)
  - Experimentos: um único core do i7 (ou i5), comparação entre 1 thread e SMT
- Resultados: SMT em um processador com especulação agressiva → aumento do desempenho de forma eficiente quanto ao consumo de energia
  - ILP não consegue o mesmo em 2011
- Hoje: melhor mais cores mais simples com SMT do que menos cores complexos
  - experimentos com o i5 e Atom → ainda melhores resultados

IC-UNICAMP

Speedup e Energia no i7, com e sem SMT

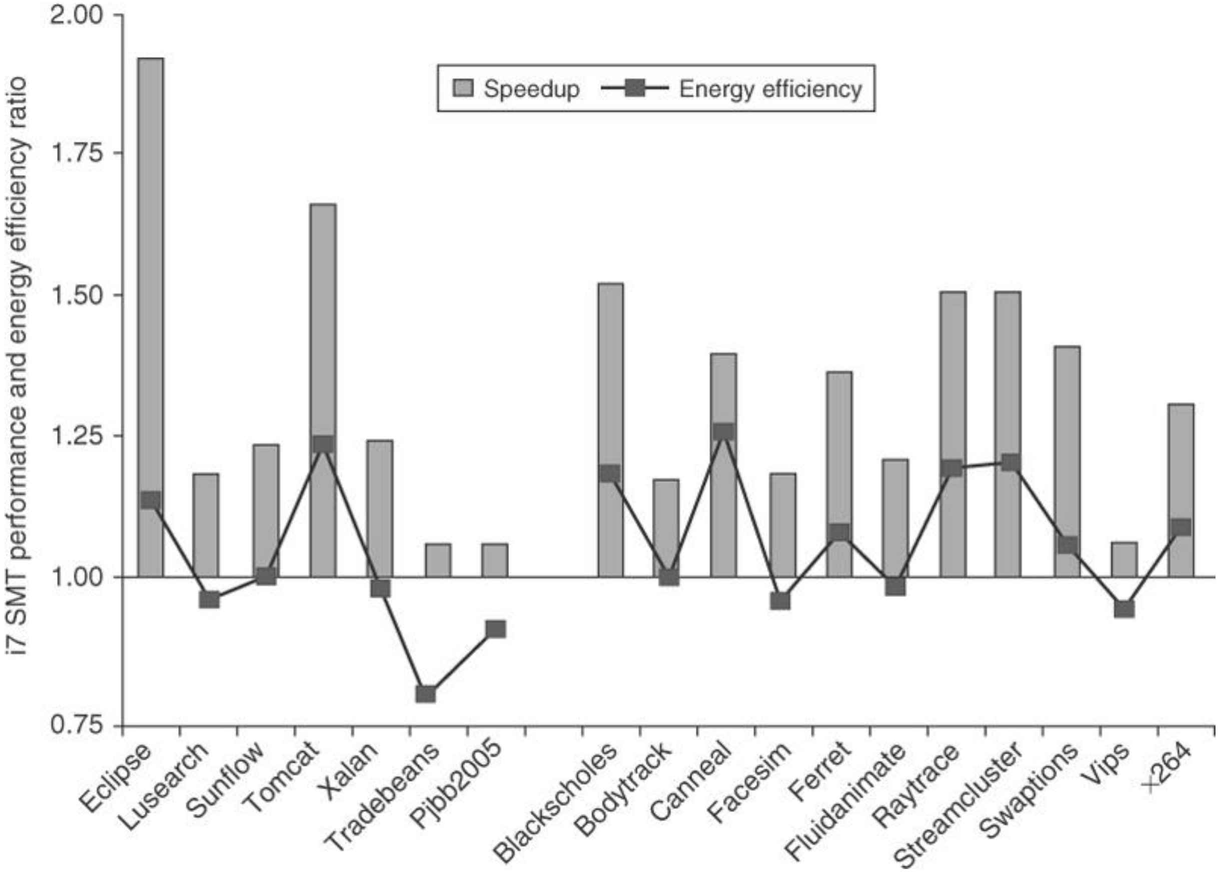


Figure 3.35 The speedup from using multithreading on one core on an i7 processor averages 1.28 for the Java benchmarks and 1.31 for the PARSEC benchmarks (using an unweighted harmonic mean, which implies a workload where the total time spent executing each benchmark in the single-threaded base set was the same). The energy efficiency averages 0.99 and 1.07, respectively (using the harmonic mean). Recall that anything above 1.0 for energy efficiency indicates that the feature reduces execution time by more than it increases average power. Two of the Java benchmarks experience little speedup and have significant negative energy efficiency because of this. Turbo Boost is off in all cases. These data were collected and analyzed by Esmaeilzadeh et al. [2011] using the Oracle (Sun) HotSpot build 16.3-b01 Java 1.6.0 Virtual Machine and the gcc v4.4.1 native compiler.

3.13 O ARM Cortex-A8 e o Intel Core i7

- Intel Core i7
  - High end, dynamically scheduled, speculative processor → high-end desktops and servers
- ARM Cortex-A8
  - Uso em smartphones e tablets
  - Dual issue, statically scheduled superscalar, dynamic issue detection (1-2 instructions/cycle)
  - Dynamic brach predictor, 512-entry 2-way set associative branch targe buffer, 4k-entry global history buffer

A8: pipeline structure

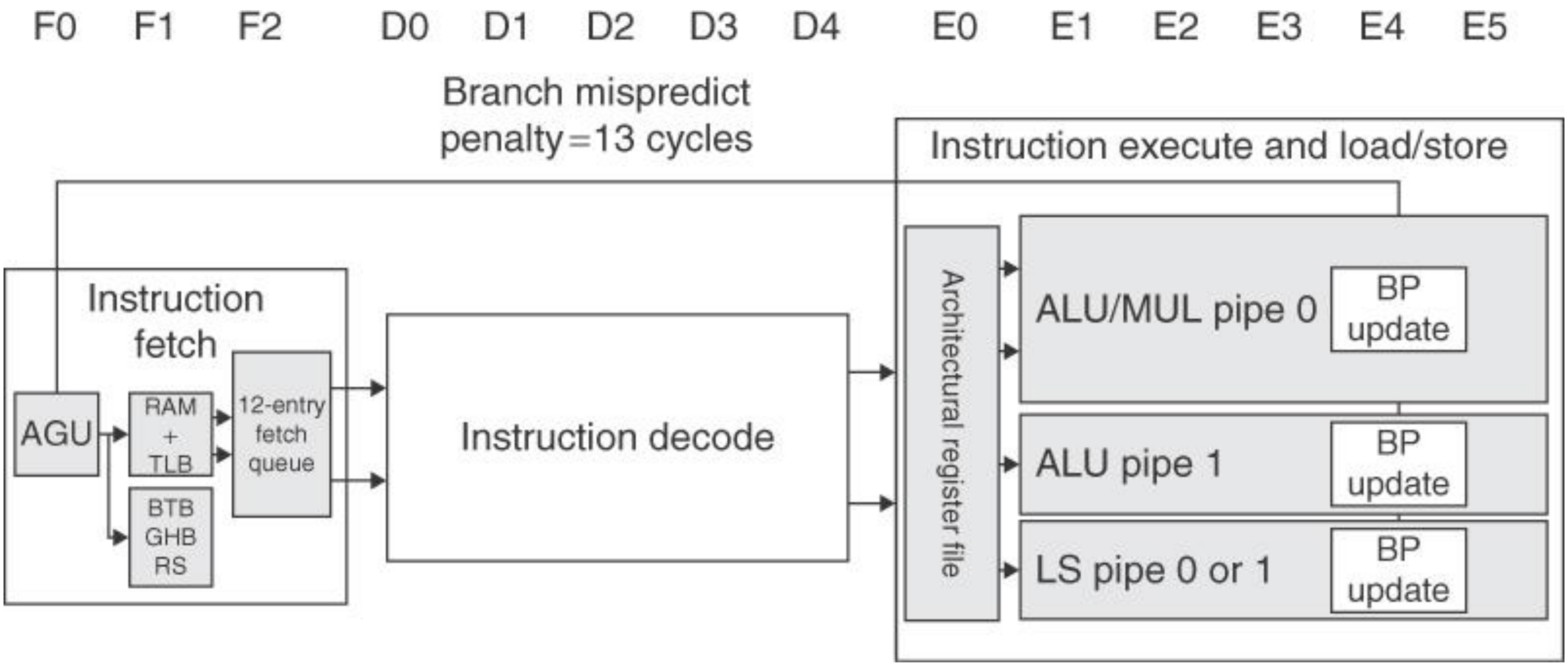


Figure 3.36 The basic structure of the A8 pipeline is 13 stages. Three cycles are used for instruction fetch and four for instruction decode, in addition to a five-cycle integer pipeline. This yields a 13-cycle branch misprediction penalty. The instruction fetch unit tries to keep the 12-entry instruction queue filled.

A8: Instruction Decode Pipeline

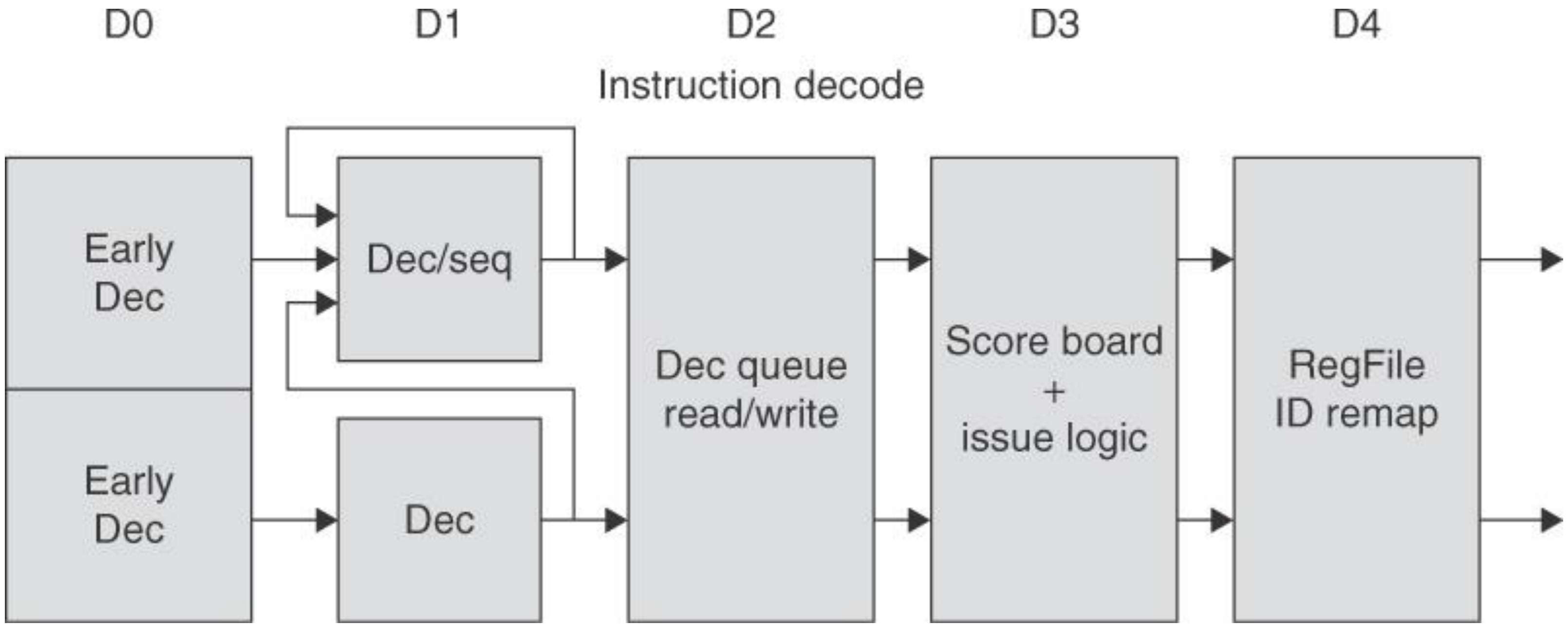


Figure 3.37 The five-stage instruction decode of the A8. In the first stage, a PC produced by the fetch unit (either from the branch target buffer or the PC incrementer) is used to retrieve an 8-byte block from the cache. Up to two instructions are decoded and placed into the decode queue; if neither instruction is a branch, the PC is incremented for the next fetch. Once in the decode queue, the scoreboard logic decides when the instructions can issue. In the issue, the register operands are read; recall that in a simple scoreboard, the operands always come from the registers. The register operands and opcode are sent to the instruction execution portion of the pipeline.

A8: Execution Pipeline

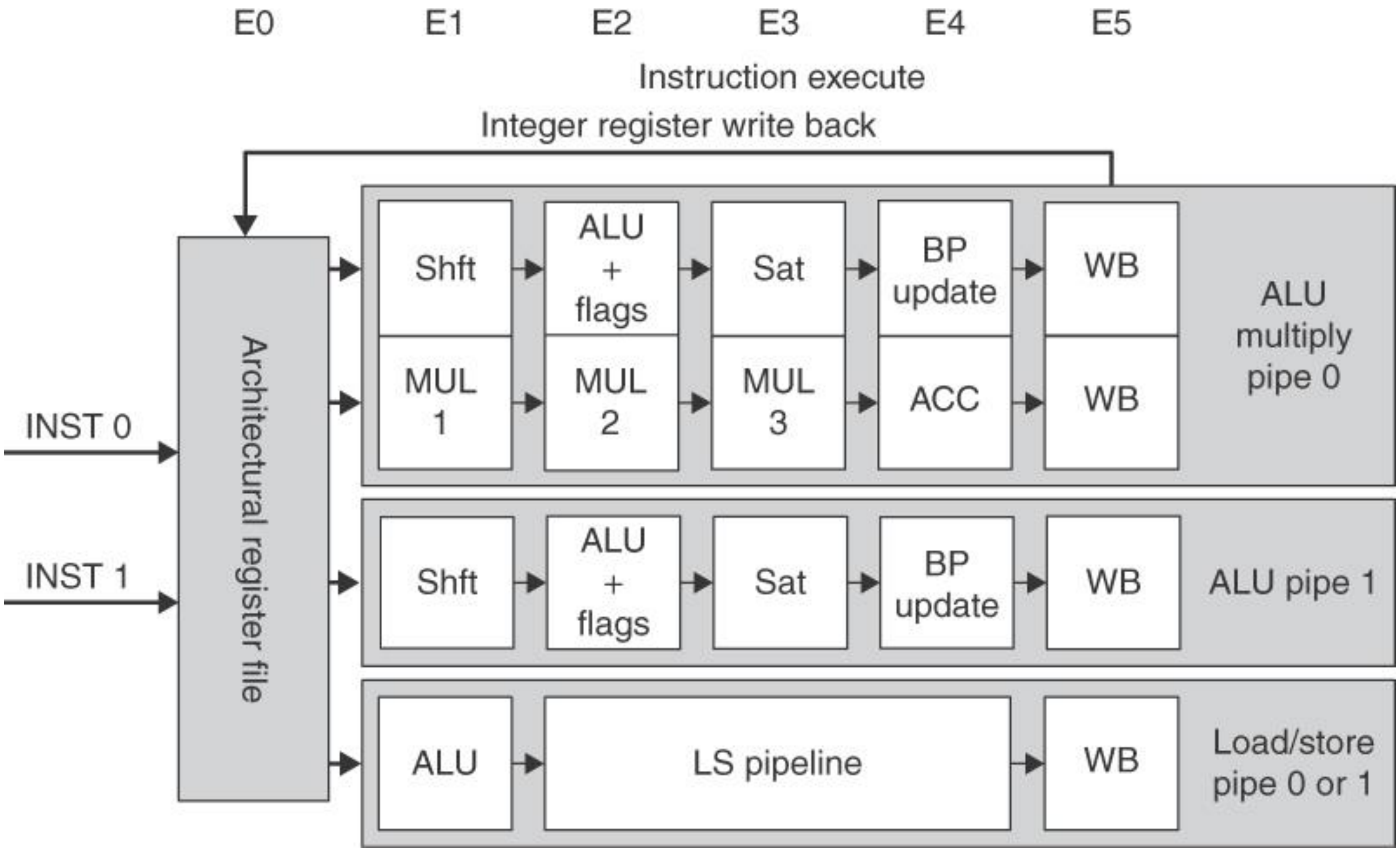


Figure 3.38 The six-stage execution pipeline of the A8.
Multiply operations are always performed in ALU pipeline 0.

A8: CPI composition

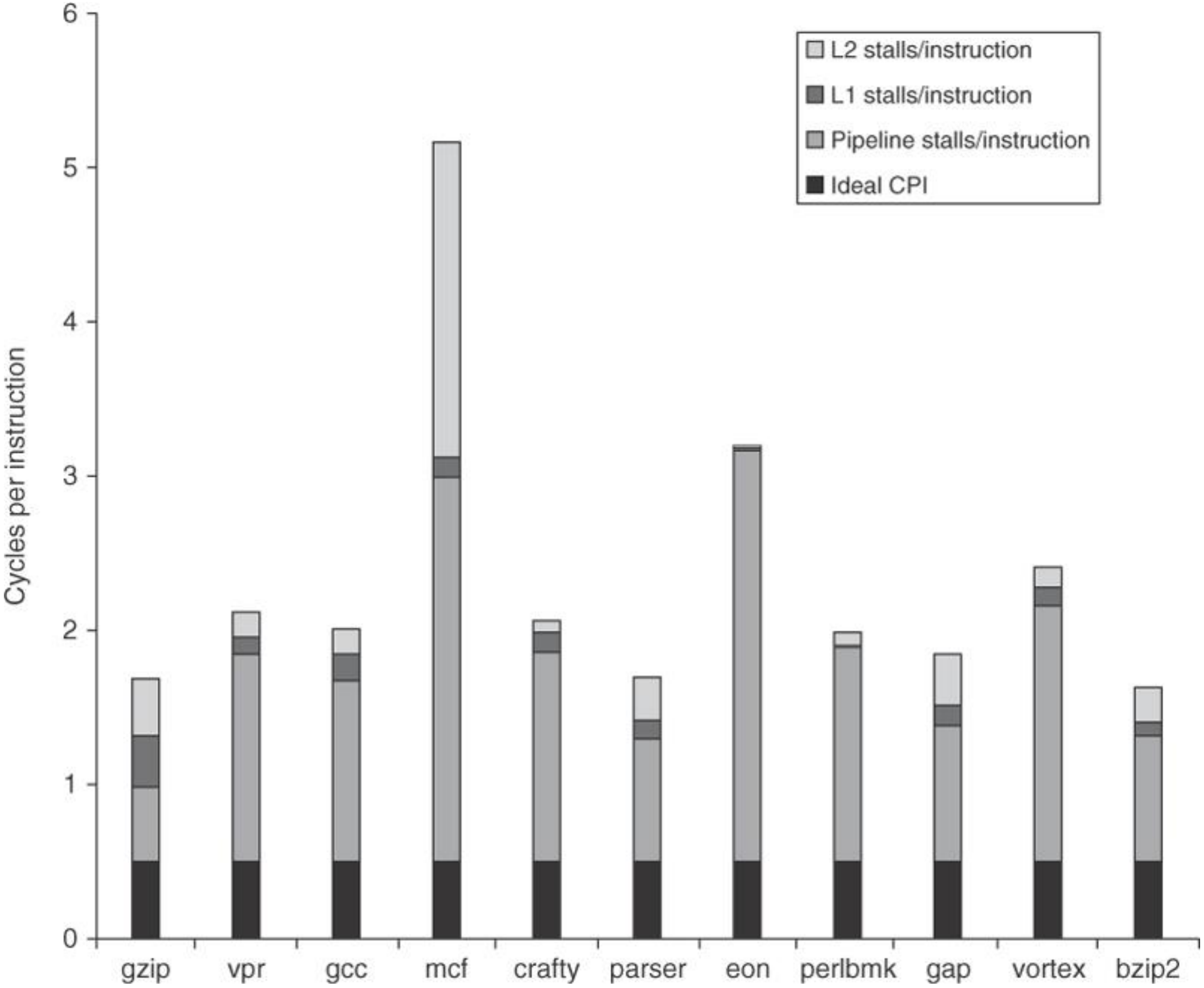


Figure 3.39 The estimated composition of the CPI on the ARM A8 shows that pipeline stalls are the primary addition to the base CPI. Benchmark eon deserves some special mention, as it does integer-based graphics calculations (ray tracing) and has very few cache misses. It is computationally intensive with heavy use of multiples, and the single multiply pipeline becomes a major bottleneck. This estimate is obtained by using the L1 and L2 miss rates and penalties to compute the L1 and L2 generated stalls per instruction. These are subtracted from the CPI measured by a detailed simulator to obtain the pipeline stalls. Pipeline stalls include all three hazards plus minor effects such as way misprediction.

A9 vs A8

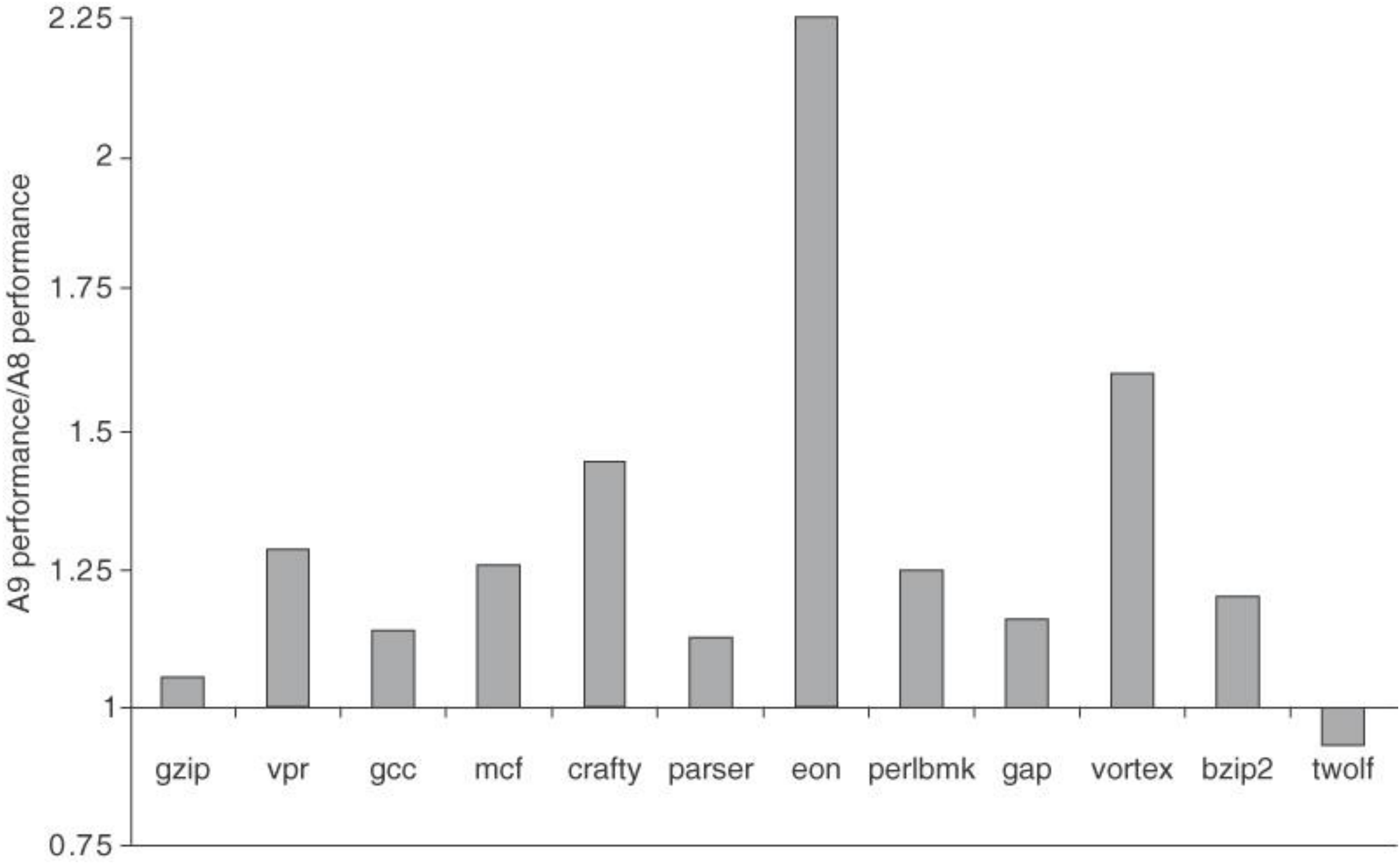


Figure 3.40 The performance ratio for the A9 compared to the A8, both using a 1 GHz clock and the same size caches for L1 and L2, shows that the A9 is about 1.28 times faster. Both runs use a 32 KB primary cache and a 1 MB secondary cache, which is 8-way set associative for the A8 and 16-way for the A9. The block sizes in the caches are 64 bytes for the A8 and 32 bytes for the A9. As mentioned in the caption of Figure 3.39, eon makes intensive use of integer multiply, and the combination of dynamic scheduling and a faster multiply pipeline significantly improves performance on the A9. twolf experiences a small slowdown, likely due to the fact that its cache behavior is worse with the smaller L1 block size of the A9.

Intel Core i7

- Aggressive out-of-order speculative microarchitecture; Deep pipelines. Multiple issue. High clock rates.
- Pipeline structure
  - IF: Multi level branch target buffer. Return address stack. Fetch 16B
  - 16B→ predecode instruction buffer. Micro-op fusion. x86 instructions
  - Micro-op decode: x86 instructions → micro-ops (simple MIPS-like instructions) → 28-entry micro-op buffer
  - Micro-op buffer: Loop stream detection (análise de loops curtos) and microfusion (fusão de instruções).
  - Basic Instruction Issue: Look up register tables. Renaming. Allocating ROB. Send to reservation stations
  - RS: 36-entry centralized RS shared by 6 FU. 6 micro-ops can be dispatched to FUs / cycle
  - Execution: Results → RS+register retirement unit. Instr complete.
  - ROB: Instructions at head → pending writes executed.

i7 pipeline structure

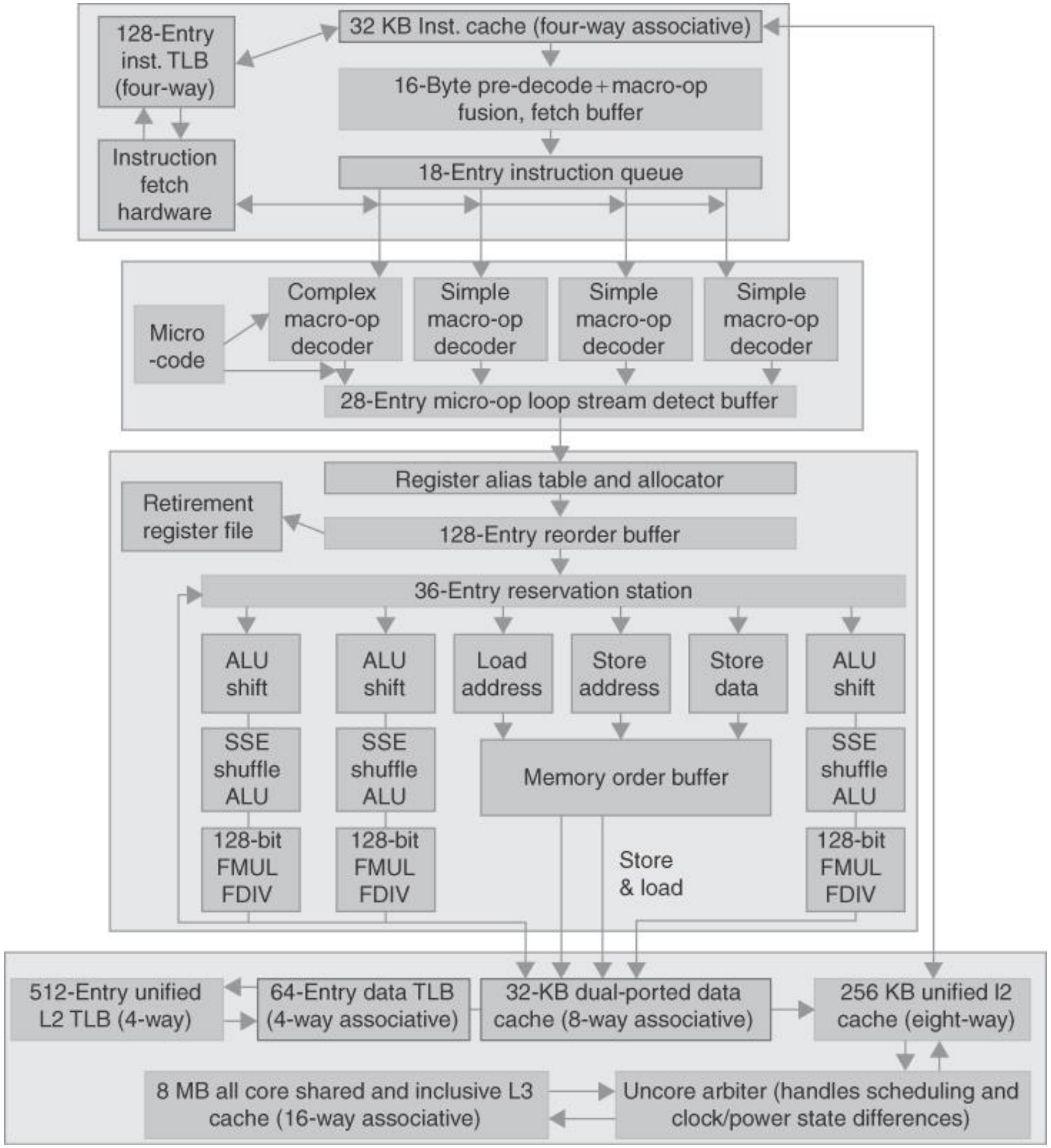

128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode+macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
28-Entry micro-op loop stream detect buffer
Register alias table and allocator
Retirement register file
128-Entry reorder buffer
36-Entry reservation station
ALU shift
ALU shift
Load address
Store address
Store data
ALU shift
SSE shuffle ALU
SSE shuffle ALU
Memory order buffer
SSE shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & load
128-bit FMUL FDIV
512-Entry unified L2 TLB (4-way)
64-Entry data TLB (4-way associative)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (eight-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

Figure 3.41 The Intel Core i7 pipeline structure shown with the memory system components. The total pipeline depth is 14 stages, with branch mispredictions costing 17 cycles. There are 48 load and 32 store buffers. The six independent functional units can each begin execution of a ready micro-op in the same cycle.

i7: % Wasted Work

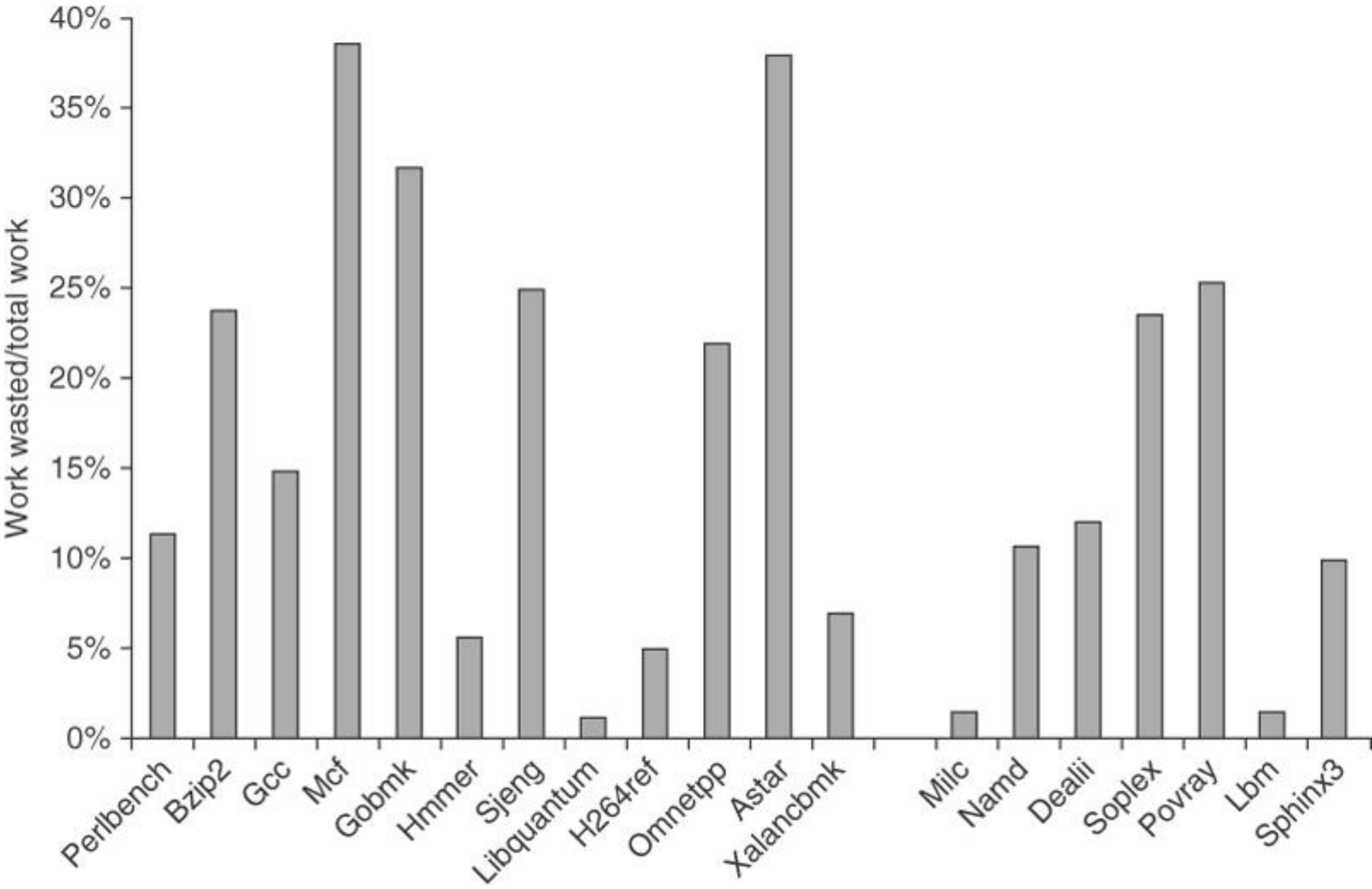


Figure 3.42 The amount of “wasted work” is plotted by taking the ratio of dispatched micro-ops that do not graduate to all dispatched micro-ops. For example, the ratio is 25% for sjeng, meaning that 25% of the dispatched and executed micro-ops are thrown away. The data in this section were collected by Professor Lu Peng and Ph.D. student Ying Zhang, both of Louisiana State University.

i7: CPI

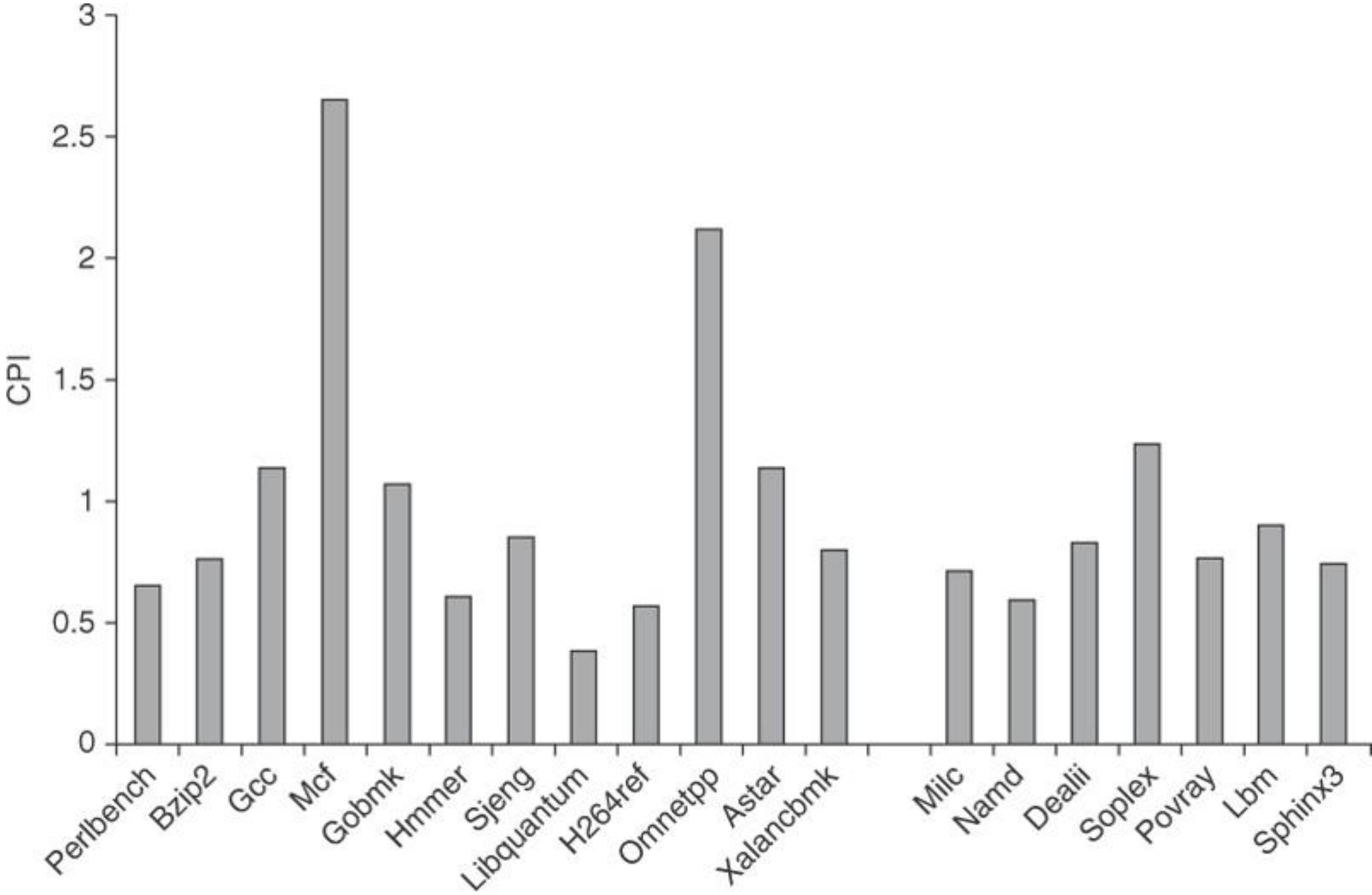
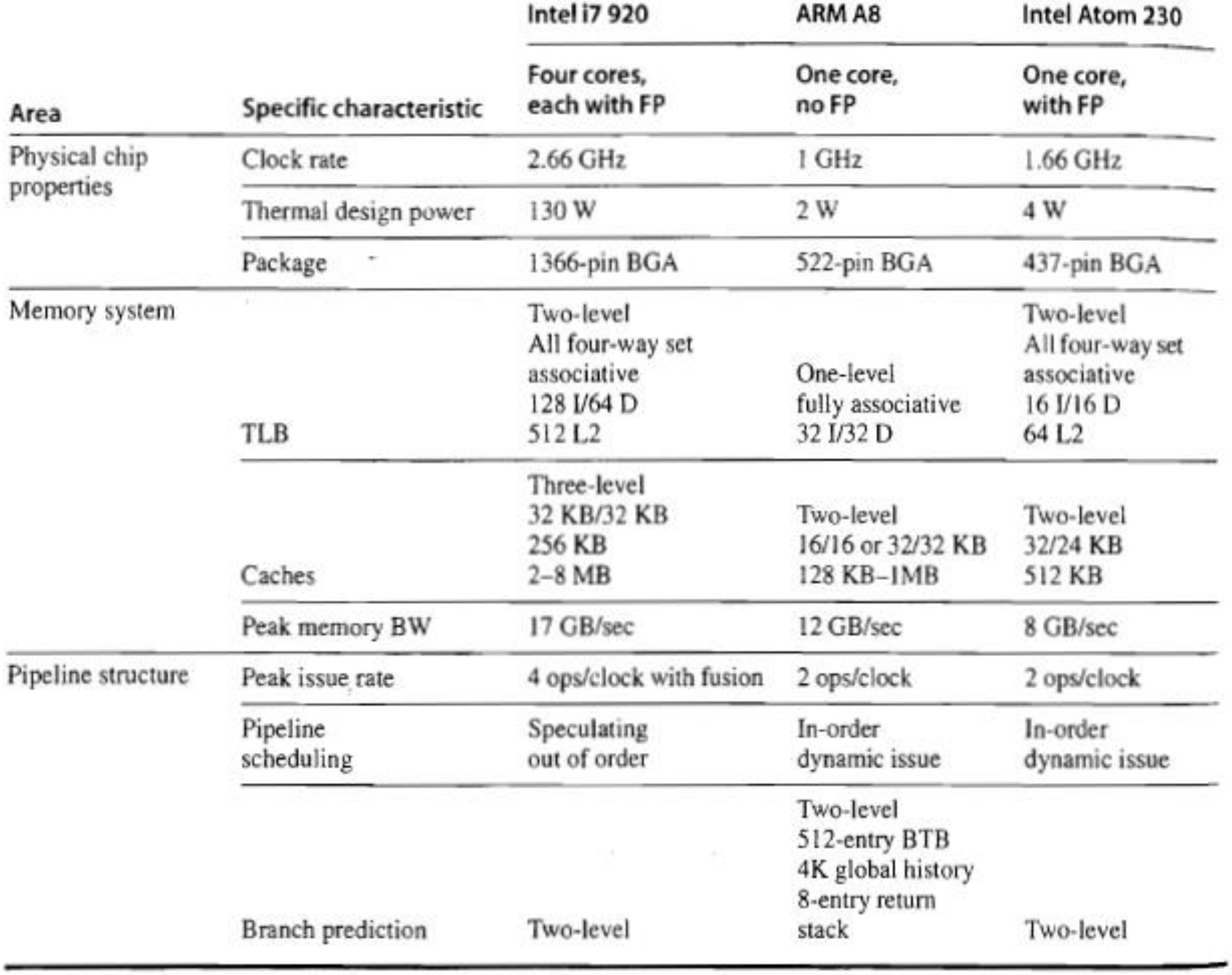


Figure 3.43 The CPI for the 19 SPECCPU2006 benchmarks shows an average CPI for 0.83 for both the FP and integer benchmarks, although the behavior is quite different. In the integer case, the CPI values range from 0.44 to 2.66 with a standard deviation of 0.77, while the variation in the FP case is from 0.62 to 1.38 with a standard deviation of 0.25. The data in this section were collected by Professor Lu Peng and Ph.D. student Ying Zhang, both of Louisiana State University.

3.14 Fallacies and Pitfalls

Comparing 2 versions of the same ISA with technology constant



		Intel i7 920	ARM A8	Intel Atom 230
Area	Specific characteristic	Four cores, each with FP	One core, no FP	One core, with FP
Physical chip properties	Clock rate	2.66 GHz	1 GHz	1.66 GHz
	Thermal design power	130 W	2 W	4 W
	Package	1366-pin BGA	522-pin BGA	437-pin BGA
Memory system	TLB	Two-level All four-way set associative 128 I/64 D 512 L2	One-level fully associative 32 I/32 D	Two-level All four-way set associative 16 I/16 D 64 L2
	Caches	Three-level 32 KB/32 KB 256 KB 2–8 MB	Two-level 16/16 or 32/32 KB 128 KB–1MB	Two-level 32/24 KB 512 KB
	Peak memory BW	17 GB/sec	12 GB/sec	8 GB/sec
Pipeline structure	Peak issue rate	4 ops/clock with fusion	2 ops/clock	2 ops/clock
	Pipeline scheduling	Speculating out of order	In-order dynamic issue	In-order dynamic issue
	Branch prediction	Two-level	Two-level 512-entry BTB 4K global history 8-entry return stack	Two-level

Figure 3.44 An overview of the four-core Intel i7 920, an example of a typical Arm A8 processor chip (with a 256 MB L2, 32K L1s, and no floating point), and the Intel ARM 230 clearly showing the difference in design philosophy between a processor intended for the PMD (in the case of ARM) or netbook space (in the case of Atom) and a processor for use in servers and high-end desktops. Remember, the i7 includes four cores, each of which is several times higher in performance than the one-core A8 or Atom. All these processors are implemented in a comparable 45 nm technology.

Comparison

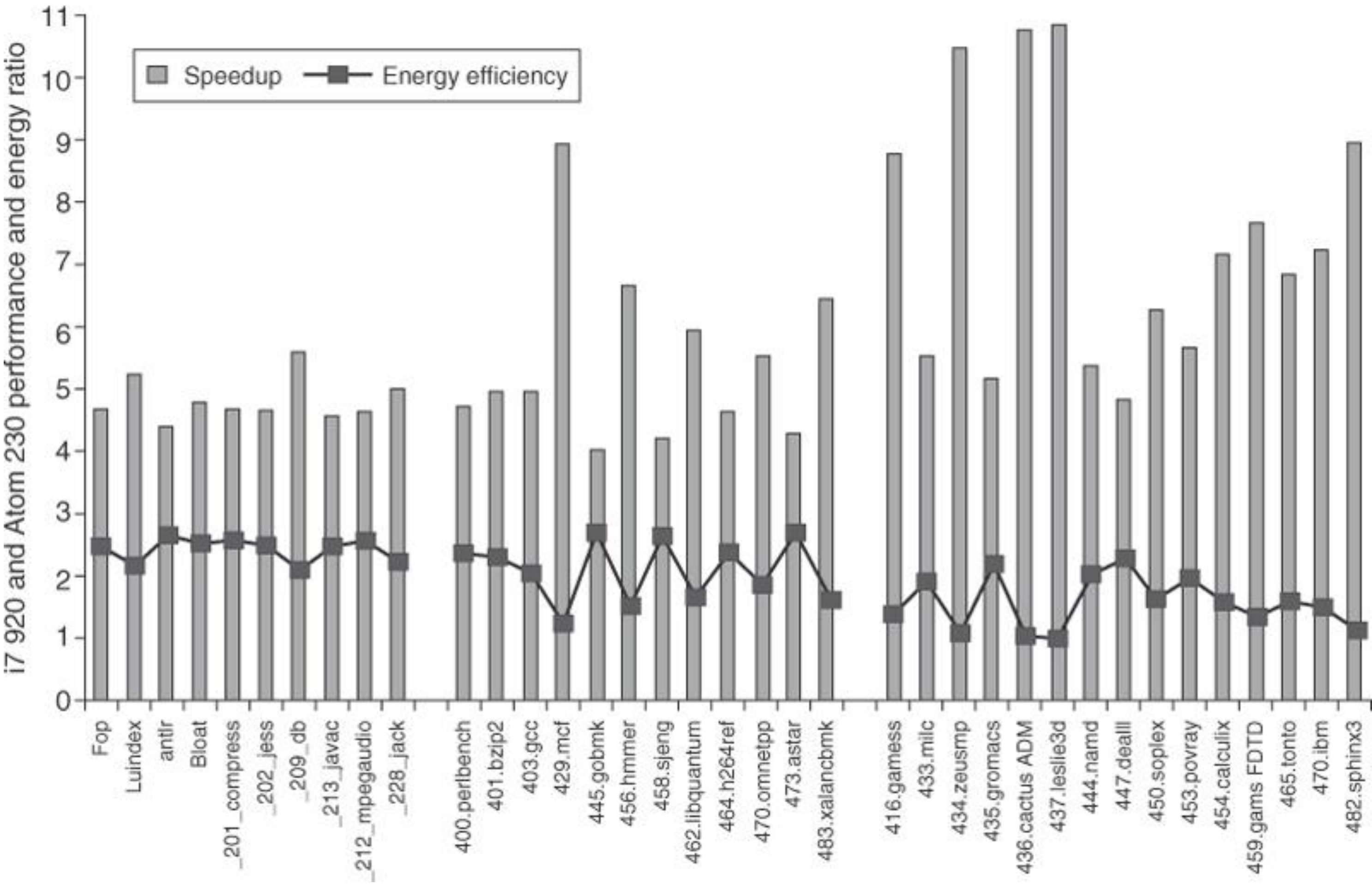
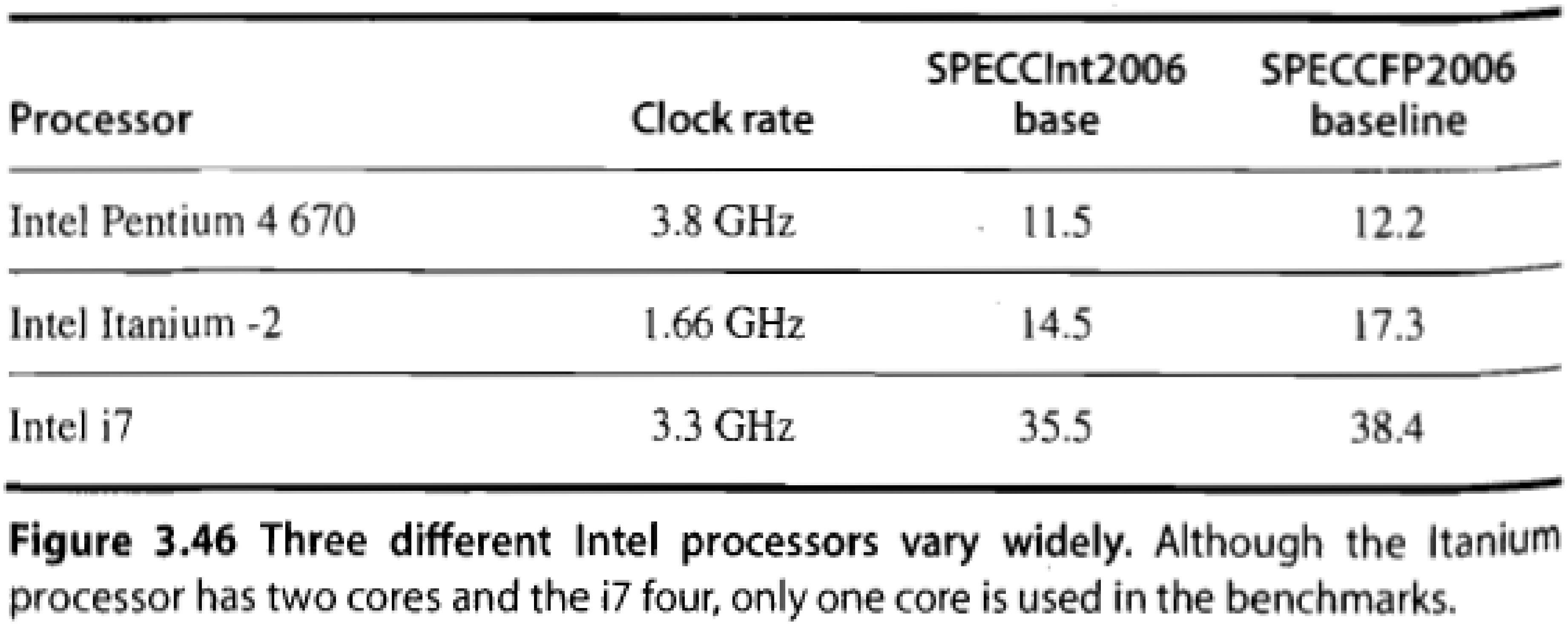


Figure 3.45 The relative performance and energy efficiency for a set of single-threaded benchmarks shows the i7 920 is 4 to over 10 times faster than the Atom 230 but that it is about 2 times *less* power efficient on average! Performance is shown in the columns as i7 relative to Atom, which is execution time (i7)/execution time (Atom). Energy is shown with the line as Energy (Atom)/Energy (i7). The i7 never beats the Atom in energy efficiency, although it is essentially as good on four benchmarks, three of which are floating point. The data shown here were collected by Esmaeilzadeh et al. [2011]. The SPEC benchmarks were compiled with optimization on using the standard Intel compiler, while the Java benchmarks use the Sun (Oracle) Hotspot Java VM. Only one core is active on the i7, and the rest are in deep power saving mode. Turbo Boost is used on the i7, which increases its performance advantage but slightly decreases its relative energy efficiency.

Fallacy

- Processors with lower CPIs will always be faster
- Processors with faster clock rates will always be faster



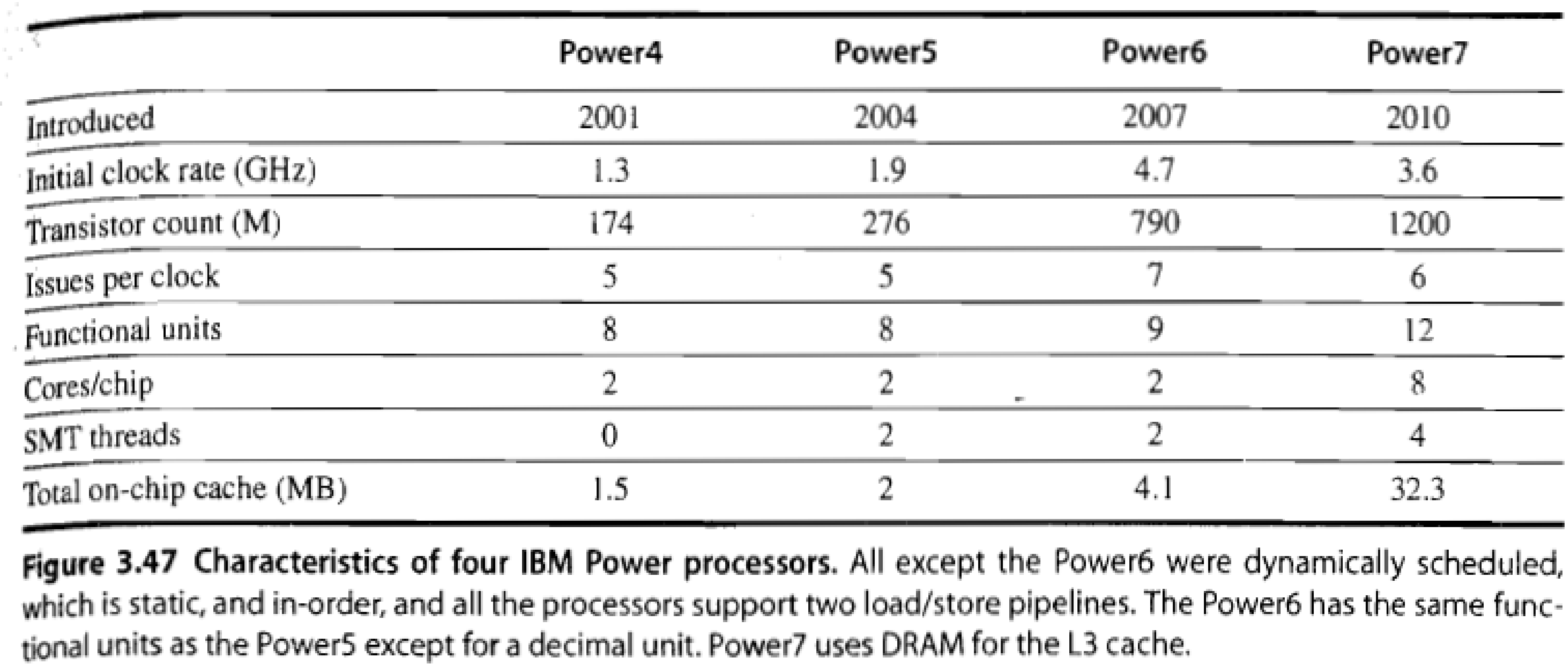
Processor	Clock rate	SPECCInt2006 base	SPECCFP2006 baseline
Intel Pentium 4 670	3.8 GHz	11.5	12.2
Intel Itanium -2	1.66 GHz	14.5	17.3
Intel i7	3.3 GHz	35.5	38.4

Figure 3.46 Three different Intel processors vary widely. Although the Itanium processor has two cores and the i7 four, only one core is used in the benchmarks.

3.15 What´s ahead

- 2000: ILP at peak
- 2005:
  - mudança de rumos → TLP e multi-core
  - data level parallelism (DLP)
- Unlikely: more increase in width of issue

Processadores da IBM: Evolução



	Power4	Power5	Power6	Power7
Introduced	2001	2004	2007	2010
Initial clock rate (GHz)	1.3	1.9	4.7	3.6
Transistor count (M)	174	276	790	1200
Issues per clock	5	5	7	6
Functional units	8	8	9	12
Cores/chip	2	2	2	8
SMT threads	0	2	2	4
Total on-chip cache (MB)	1.5	2	4.1	32.3

Figure 3.47 Characteristics of four IBM Power processors. All except the Power6 were dynamically scheduled, which is static, and in-order, and all the processors support two load/store pipelines. The Power6 has the same functional units as the Power5 except for a decimal unit. Power7 uses DRAM for the L3 cache.