

Timing Peculiarities in Modern Computer Architectures

- The following example is taken from an exercise in “Systemprogrammierung”.
- It was not! constructed for challenging the timing predictability of modern computer architectures; the strange behavior was found by chance.
- A straightforward GCD algorithm was executed on an UltraSparc (Sun) architecture and timing was measured.
- ***Goal in this lecture:*** Determine the cause(s) for the strange timing behavior.

Program

- Only the relevant assembler program is shown (and the related C program); the calling *main* function just jumps to label *ggt* 1.000.000 times.

```
.text
.global ggt
.align 32
```

Here, we will introduces nop statements; there are NOT executed.

```
ggt:                                ! %o0:= x,%o1 := y
cmp    %o0, %o1
blu,a  ggt                          ! if (%o0 < %o1) {goto ggt;}
sub     %o1, %o0, %o1               ! %o1 = %o1 - %o0
bgu,a  ggt                          ! if (%o0 > %o1) {goto ggt;}
sub     %o0, %o1, %o0               ! %o0 = %o1 - %o0
retl
nop
```

```
int ggt_c (int x, int y) {
    while (x != y) {
        if (x < y) { y -= x; }
        else { x -= y; }
    }
    return (x);
}
```

Observation

- Depending on the number of nop statements before the **ggt** label, the execution time of **ggt(17, 17*97)** varies by a factor of almost 2. The execution time of **ggt(17*97, 17)** varies by a factor of more than 4.
- This behavior is periodic in the number of nop statements, i.e. it repeats after 8 nop statements.
- Measurements:

nop	time[s] ggt(17,17*97)	time[s] ggt(17*97,17)
0	0.36	0.62
1	0.35	2.78
2	0.36	0.64
3	0.35	2.79

nop	time[s] ggt(17,17*97)	time[s] ggt(17*97,17)
4	0.37	0.63
5	0.35	0.62
6	0.65	0.64
7	0.64	0.63

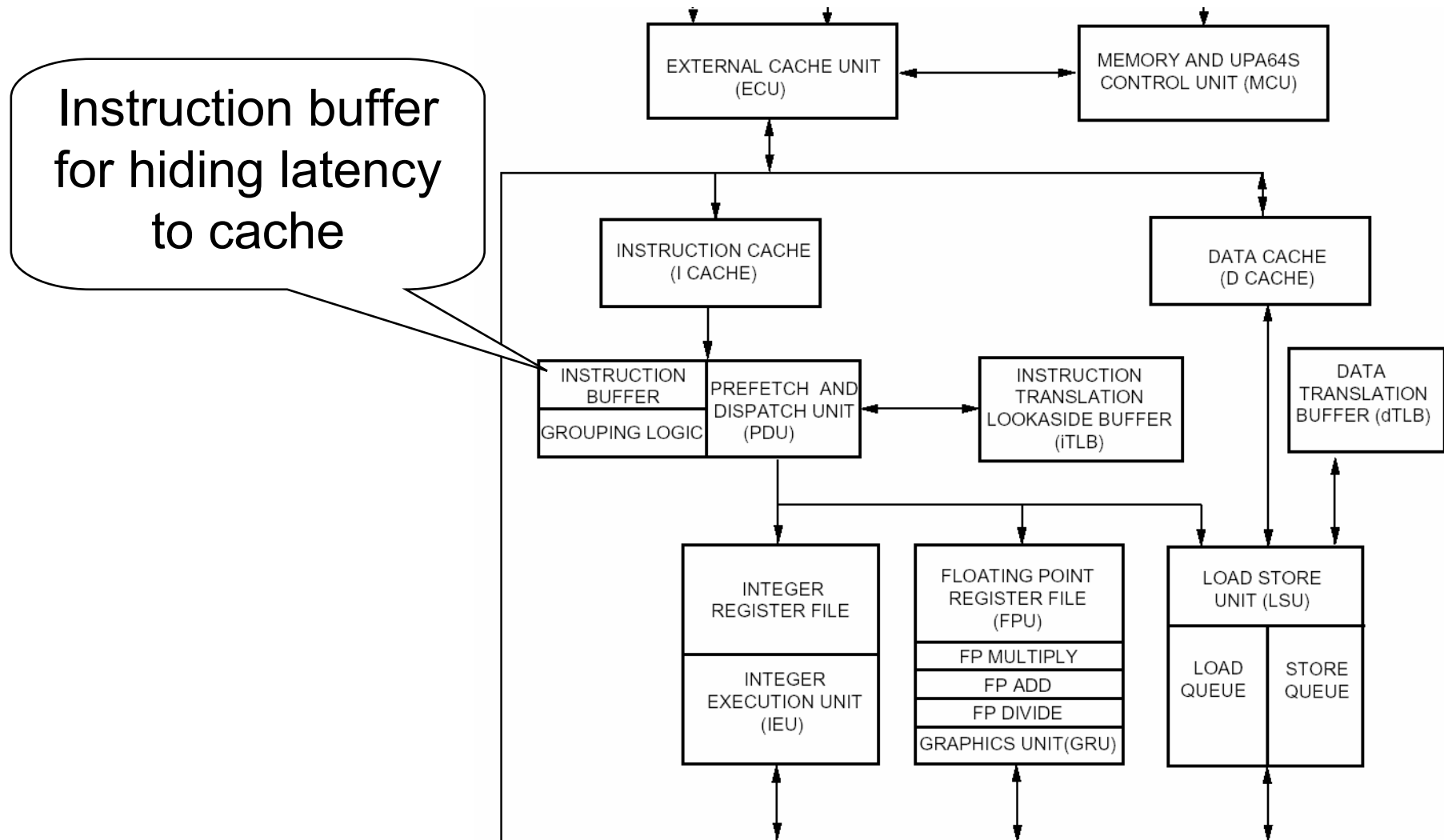
Simple Calculations

- The CPU is UltraSparc with 360 MHz clock rate.
- Problem 1 (ggt(17,17*97)):
 - Fast execution: $96 \cdot 3 \cdot 1.000.000 / 0.35 = 823$ MIPS and $0.35 \cdot 360 / 96 = 1.31$ cycles per iteration.
 - Slow execution: $96 \cdot 3 \cdot 1.000.000 / 0.65 = 443$ MIPS and $0.65 \cdot 360 / 96 = 2.44$ cycles per iteration.
 - Therefore, the difference is about 1 cycle per iteration.
- Problem 2 (ggt(17*97, 17)):
 - Fast execution: $96 \cdot 4 \cdot 1.000.000 / 0.63 = 609$ MIPS and $0.63 \cdot 360 / 96 = 2.36$ cycles per iteration.
 - Slow execution: $96 \cdot 4 \cdot 1.000.000 / 2.78 = 138$ MIPS and $2.78 \cdot 360 / 96 = 10.43$ cycles per iteration.
 - Therefore, the difference is about 8 cycles per iteration.

Explanations

- **Problem 1 ($ggt(17, 17 \cdot 97)$):**
 - The first three instructions (`cmp`, `blu`, `sub`) are called 96 times before *ggt* returns. The timing behavior depends on the location of the program in address space.
 - The reason is most probably the implementation of the 4 word instruction buffer between the instruction cache and the pipeline: The instruction buffer can not be filled by different cache lines in one cycle.
 - In the slow execution, one needs to fill the instruction buffer twice for each iteration. This needs at least two cycles (despite of any parallelism in the pipeline).

Block Diagram of UltraSparc



Instruction Availability

Instruction dispatch is limited to the number of instructions available in the instruction buffer. Several factors limit instruction availability. UltraSPARC-II*i* fetches up to four instructions per clock from an aligned group of eight instructions. When the fetch address (modulo 32) is equal to 20, 24, or 28, then three, two, or one instruction(s) respectively are added to the instruction buffer. The next cache line and set are predicted using a next field and set predictor for each aligned four instructions in the instruction cache. When a set or next field mispredict occurs, instructions are not added to the instruction buffer for two clocks.

Address Alignment

0 nop

Cache line:

cmp	blu	sub	...				
-----	-----	-----	-----	--	--	--	--

Instruction buffer:

cmp	blu	sub	...
-----	-----	-----	-----

5 nop

Cache line:

nop	nop	nop	nop	nop	cmp	blu	sub
-----	-----	-----	-----	-----	-----	-----	-----

Instruction buffer:

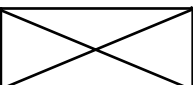
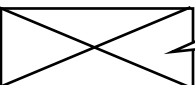
cmp	blu	sub	
-----	-----	-----	-------------------------------------------------------------------------------------

6 nop

Cache lines:

nop	nop	nop	nop	nop	nop	cmp	blu
sub

Instruction buffer:

cmp	blu		
-----	-----	--------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------

2 fetches are necessary
as sub is missing

Explanations

- **Problem 2 (ggt(17*97,17)):**
 - The loop is executed (cmp, blu, sub, bgu, sub) 96 times, where the first sub instruction is not executed (since blu is used with ',a' suffix, which means, that instruction in the delay slot is not executed if branch is not taken). Therefore, there are four instructions to be executed, but the loop has 5 instructions in total.
 - The main reason for this behavior is most probably due to the branch prediction scheme used in the architecture.
 - In particular, there is a prediction ***of the next block of 4 instructions*** to be fetched into the instruction buffer. This scheme is based on a two bit predictor and is also used to control the pipeline and to prevent stalls.
 - But there is a problem due to the optimization of the state information that is stored (prediction for blocks of instructions and single instructions):

User Manual (page 342 ...)

The following cases represent situations when the prediction bits and/or the next field do not operate optimally:

1. When the target of a branch is word 1 or word 3 of an I-cache line (FIGURE 21-2) and the fourth instruction to be fetched (instruction 4 and 6 respectively) is a branch, the branch prediction bits from the wrong pair of instructions are used.

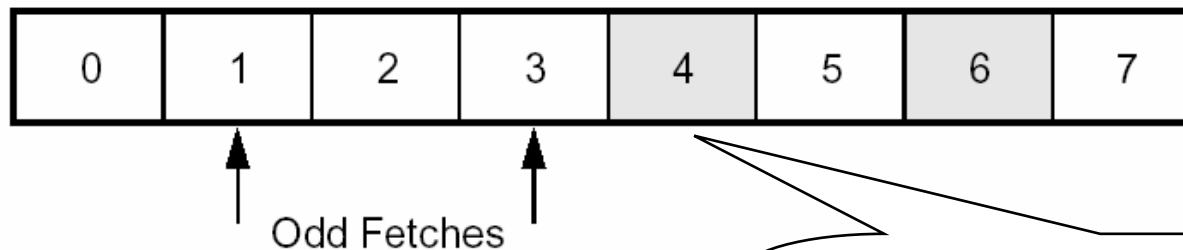


FIGURE 21-2 Odd Fetch to an I-cache Line

We exactly have this situation, if there are 1 or three nops statements inserted

Conclusions

- Innocent changes (just moving code in address space) can easily change the timing by a factor of 4.
- In our example, the timing oddities are caused by two different architectural features of modern superscalar processors:
 - branch prediction
 - instruction buffer
- It is hard to predict the timing of modern processors; this is bad in all situations, where timing is of importance (embedded systems, hard real-time systems).
- What is a proper approach to predictable system design ?