Advanced Computer Architecture
31-01-2020

A) A processor embeds two cores that have private L1 and L2 caches; the L3 cache is shared. The caches obey the MESI protocol and have the following structure: 64KB, 4-way L1 I-cache and D-cache, each 32-byte block; 2048 KB, 8-way associative, 32-byte block L2 cache; 8MB, 8-way associative, 64-byte block L3 cache. The latencies (disregarding virtual memory TLB) expressed in clock cycles are: 2 in L1, 4 in L2, 8 in L3. Addresses are 48-bit long. Store operations are carried out with a write-back allocate policy. Assuming initially empty and invalidated cache lines throughout the hierarchy, consider the following memory accesses:

core 1) LD F1, 0000FFFFFFA0hex;

core 2) ST F2, 0000FFFFFFB0hex;

core 1) ST F3, 0000FFFFFFA0hex;

core 2) ST F3, 0000FFFFFFB0hex;

a1) show the blocks involved in each cache, and the associated MESI state after each instruction.

L1: disp 5; index=log2(64KB/32/4)=9; TAG=48-9-5=34

L2: disp=5; index=log2(2MB/32/8)=13; TAG=48-13-5=30

L3: disp=6; index=log2(8MB/64/8)=14; TAG=48-14-6=28

0000FFFFFFA0 -> 0000|0000|0000|0000|1111|1111|1111|1111|1111|1111|1010|0000

L1 0000000000000000111111111111111111|111111101|00000

L2 000000000000000011111111111111|1111111111101|00000

L3 0000000000000000111111111111|11111111111110|100000

0000FFFFFFB0 -> 0000|0000|0000|0000|1111|1111|1111|1111|1111|1111|1011|0000

L1 0000000000000000111111111111111111|111111101|10000

L2 000000000000000011111111111111|1111111111101|10000

L3 0000000000000000111111111111|11111111111110|110000

Same L3 block, which is shared.

Core1 Core2

L1 L1

L2 L2

Shared
L3

core 1) LD F1, 0000FFFFFFA0hex; miss

Core1 Core2

L1 E L1 -

L2 E L2 -

Shared
L3 E

core 2) ST F2, 0000FFFFFFB0hex; miss L1, miss L2, hit L3

Core1 Core2

L1 I L1 M

L2 I L2 I

Shared
L3 I

Core1 private L2 and L1 are invalidated because L3 has been invalidated

core 1) ST F3, 0000FFFFFFA0hex; miss L1, miss L2, miss L3 (because I)

Core1 Core2

L1 M L1 I

L2 I L2 I

Shared
L3 I

L3 invalid block is fetched from core 2 L1/L2, sent to memory, sent to core 1, which
updates L1 (M) and invalidates the just loaded L2 and L3; core 2 is invalidated in L1 and L2

core 2) ST F3, 0000FFFFFFB0hex;
same sequence as just executed by core 1), final situation

Core1 Core2

L1 I L1 M

L2 I L2 I

Shared
L3 I

a2) considering the situation generated by the instructions above, show a set of instructions that cause a block replacement in L3.

There is one invalid block in L3 in set 11111111111110, way number 0, TAG 0000000000000000111111111111.
The set is filled up after 7 more instructions (LD or ST) on the same set 11111111111110 with 7 different tags, which use the remaining 7 ways.
An 8th instruction on the same set 11111111111110 causes a replacement.

a3) the processor is attached to a memory subsystem through a 128-bit bus, and memory is built using DDR4 chip modules described in the attached table; choose system configuration (bus clock, memory chips) to minimize the cost of a cache miss by taking into account the following constraints: i) all modules support a memory word of 8 bytes; ii) DDR4 modules up to 1866 included can be configured in bank mode with a maximum interleaving of 2, while other modules do not allow for interleaving; iii) memory addressing requires 2 bus clock cycles; iv) memory activation requires 4 bus clock cycles if interleaving is active, otherwise 1 bus clock cycle. The processor bus interface can sustain a maximum transfer rate of 16GB/s.

The processor bus interface limits the possible DDR4 choices to DDR4-1600 and DDR41866, others would not be used properly, because their superior transfer rate would never be used.

Let’s use DDR4-1866, which is the fastest.

Maximum interleaving = 2

Nact= 64/8\*interleaving
Trans= cacheline/BytesTranferredperActivation \* TimeSingletransfer

1. No interleaving: Nact=8; Tact=1; Tadd=2; Ttransf=64/8\*0,5=4 ToT=2+8\*1+4=14
2. Interleaving by 2: Nact=4; Tact=4; Tadd=2; Ttransf=64/16\*0,5=2; TOT=2+4\*4+2=20

No interleaving is better.

B) A 512x512 image of 3-channel pixels is stored as an array V[] containg in each element a struct POINT representing the 3-channel pixel composed by double X,Y,Z (see the following fragment of C-Language code).

#include <stdio.h>

#define LEN = 262144

struct point {

 double x; // 8 bytes

 double y; // 8 bytes

 double z; // 8 bytes

};

void main(int argc, int \*\*argv)

{

 int i = 0;

 // definition

 struct point v[LEN];

 // phase A: inizialization

 for (i = 0; i < LEN; i++)

 {

 v[i].x = rand();

 v[i].y = rand();

 v[i].z = rand();

 }

 // phase B: computation

 for (i = LEN-1; i >=0; i—-)

 v[i].x = (v[i].x + v[i].y + v[i].z) / 3.0;

 }

}

Assuming that V[] is allocated in memory at increasing addresses in big-endian mode starting with V[0].x at address 0000FFFFA000hex ,

B1) compute the number of misses in L3 only due to phase A

Each struct (array element) is 24-byte wide. There are 512x512 such objects stored one after the other, starting at a 64-byte aligned address 0000FFFFA000hex. The overall memory space is 3\*23 \* 218 = 3\*2MB. The 8-way 8MB L3 cache can host the whole array: the first 1MB way hosts ⎣(220/(3\*23))⎦ = 43690 array elements, the subsequents ones are stored in the other ways (up to the sixth way).

In Phase A, the iterations scan the 6MB memory space and move all the 64-byte memory blocks to the L3 cache; these blocks are 6MB/64B=96K=98304. This is the number of L3 misses.

B2) compute the number of misses in L3 only due to phase B.

Since after phase A the whole array is in L3 cache, phase B always hits in L3, so no misses.

The above assumptions do not take into account the allocation of variable i and of constant LEN, which also are loaded from memory and therefore take up cache blocks. It is likely that this consumes at most 2 L3 blocks, which does not alter much of the computation (2 extra misses).

C) Each core of the processor described in A) is organized as a superscalar, 2-way pipeline, that fetches, decodes issues and retires (commits) bundles containing each 2 instructions. The front-end in-order section (fetch and decode) consists of 2 stages. The issue logic takes 1 clock cycle, if the instructions in the bundle are independent, otherwise it takes 2 clock cycles. The architecture supports dynamic speculative execution, and control dependencies from branches are solved when the branch evaluates the condition, even if it is not at commit. The execution model obeys the attached state transition diagram. There is a functional units (FUs) Int1 for integer arithmetic (arithmetic and local instructions, branches and jumps, no multiplication), one FU FAdd1 for floating point addition/subtraction, one FUs FMolt1 for floating point multiplication, and one FU for division, FDiv1.

There are 12 integer (R0-R11) and 12 floating point (F0-F11) registers. Speculation is handled through a 8-entry ROB, a pool of 4 Reservation Stations (RS) Rs1-4 shared among all FUs, 2 load buffers Load1-Load2, 1 store buffer Store1 (see the attached execution model): an instruction bundle is first placed in the ROB (if two entries are available), then up to 2 instructions are dispatched to the shared RS (if available) when they are ready for execution and then executed in the proper FU. FUs are *pipelined* and have the latencies quoted in the following table:

|  |  |
| --- | --- |
| Int - 2 | Fadd – 3 |
| Fmolt – 4 | Fdiv – 6 |

Further assumption

* The code is assumed to be already in the I-cache; data caches are described in point A) and are assumed empty and invalidated; the cost of a miss is conventionally set to 50.

c1) assuming write-back, allocate protocol for cache management, show state transitions for the instructions of the loop up to the moment when PC04 in the second iteration is issued:

PC01 ADDI R1,R0, 0000FFFFFFA0hex

PC02 ADDI R8,R0,1024

PC03 LD F1,0(R1)

PC04 LD F2,8(R1)

PC05 MULTF F3,F2,F1

PC06 ST F2,0(R1)

PC07 ADDF F4,F3,F1

PC08 ST F4,8(R1)

PC09 ADDI R1,R1,16

PC10 SUBI R8,R8,1

PC11 BNEZ R8,PC03

c2) show ROB, RS and buffer status at the issue of PC04 in the second iteration;

Dynamic speculative execution

Decoupled ROB RS execution model

|  |  |  |  |
| --- | --- | --- | --- |
| **ISTRUCTION** |  |  | **INSTRUCTION STATE**  |
|  | **n.ite** | **ROBpos** | **WO** | **RE** | **DI** | **EX** | **WB** | **RR** | **CO** |
| PC01 ADDI R1,R0, 0000FFFFFFA0hex |  | **0** |  | **1** | **2** | **3-4** | **5** | **6-7** | **8** |
| PC02 ADDI R8,R0,1024  |  | **1** |  | **1** | **2-3** | **4-5** | **6** | **7** | **8** |
| PC03 LD F1,0(R1) |  | **2** | **2-4** | **5** | **6** | **7-70** | **71** | **72-73** | **74** |
| PC04 LD F2,8(R1)  |  | **3** | **3-4** | **5** | **6** | **7-71** | **72** | **73** | **74** |
| PC05 MULTF F3,F2,F1 |  | **4** | **4-71** | **72** | **73** | **74-77** | **78** | **79** | **80** |
| PC06 ST F2,0(R1) |  | **5** | **5-71** | **72** | **73** | **74-75** | **-** | **76-79** | **80** |
| PC07 ADDF F4,F3,F1 |  | **6** | **6-77** | **78** | **79** | **80-82** | **83** | **84-87** | **88** |
| PC08 ST F4,8(R1) |  | **7** | **7-82** | **83** | **84** | **85-86** | **-** | **87** | **88** |
| PC09 ADDI R1,R1,16 |  | **0** |  | **9** | **10** | **11-12** | **13** | **14-88** | **89** |
| PC10 SUBI R8,R8,1 |  | **1** |  | **9** | **10-11** | **12-13** | **14** | **15-88** | **89** |
| PC11 BNEZ R8,PC03  |  | **2** |  | **75** | **76** | **77-78** | **-** | **79-89** | **90** |
| PC12 ISTRA |  | **3** | **75?** | **75?** | **Cancelled at 79** |
| PC03 LD F1,0(R1) |  | **4** |  | **83** | **84** | **85-86** | **87** | **88-90** | **91** |
| PC04 LD F2,8(R1)  |  | **5** |  | **84** | **85** | **86-87** | **88** | **89-90** | **91** |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

PC11 decides on the branch in 78, send the new PC to the NextPC logic at 79, the fetch with hit il L1 lasts 2 clocks 80-81, the decode is in 82, so the bundle is issued at 83.

|  |  |
| --- | --- |
|  | Reservation station and load/store buffers |
| Busy | Op | Vj | Vk | ROBj | ROBk | ROB pos | Address |
| Rs1 |  |  |  |  |  |  |  |  |
| Rs2 |  |  |  |  |  |  |  |  |
| Rs3 |  |  |  |  |  |  |  |  |
| Rs4 |  |  |  |  |  |  |  |  |
| Load1 |  | PC03 |  |  |  |  | 4 | [ROB0] |
| Load2 |  |  |  |  |  |  |  |  |
| Store1 |  | PC08 | [ROB6] |  |  |  | 7 | [R1]+8 |
|  |  |  |  |  |  |  |  |  |

ROBj ROBk: sources not yet available

ROB pos: ROB entry number where instruction is located

|  |  |
| --- | --- |
|  | Result Register status |
| Integer | R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9 | R10 | R11 |  |  |
| ROB pos |  | 0 |  |  |  |  |  |  | 1 |  |  |  |  |  |
| state |  | B |  |  |  |  |  |  | B |  |  |  |  |  |
| Float. | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10 | F11 |  |  |
| ROB pos |  | 4 | 5 |  | 6 |  |  |  |  |  |  |  |  |  |
| state |  | B | B |  | B |  |  |  |  |  |  |  |  |  |

|  |
| --- |
| Reorder Buffer (ROB) |
|  ROB Entry#  | Busy | Op | Status | Destination | Value |
| 0 |  | PC09 | RR | R1 | [R1]+16 |
| 1 |  | PC10 | RR | R8 | [R8]+1 |
| 2 |  | PC11 | RR | NEXTPC | [PC03] |
| 3 |  | NULLIFIED | - | - | - |
| 4 |  | PC03 | DI | F1 | Mem([ROB0]) |
| 5 |  | PC04 | RE | F2 | Mem([ROB0]+8) |
| 6 | H&T | PC07 | RR | F4 | [ROB4]+[F1] |
| 7 |  | PC08 | EX | Mem([R1]+8) | [ROB6] |

 Clock 84

c3) assuming the code is executed in a standard, “Von Neumann” processor, no pipelining, memory accesses through the cache hierarchy of point A with L1 cache only (no L2, no L3), what would be the PCI of the code ?

We assumes a Von-Neuman CPU, that executes instructions without pipelining, one after the other, in functional units whose lantencies are quoted above. Memory access instructions (LD and ST) act on a L1 cache described in point A), that is a 64KB, 4-way, 32-byte cache.

PCI= total number of clock cycles (A) / total number of executed instructions (B)

(B)=2+1024\*9=9218

(A)

The code consists of 2 INT type instructions in the preamble (PC01, PC02), that is 2\*2 clock cycles, then 1024 iterations that contain:

3 INT type (PC09,PC10,PC11) -> 3\*2 clock cycles
1 FMULT type (PC05) -> 4 clock cycles
1 FADD type (PC07) -> 3 clock cycles

Each couple of iterations (4 LD and 4 ST) uses a whole L1 block (32 bytes), so that the 512\*32 bytes (16KB) are well contained in the 32 KB cache; there are 512 L1 misses and 512 \* 7 L1 hits, with a miss lasting 2+50 clock cycles (assuming a miss cost of 50) and a hit costing 2 clock cycles.

So each couple of iterations access memory in: 2+50+7\*2=66 clock cycles.

(A)=4+1024 \* (3\*2+4+3) + 512 \* (2+50+7\*2)= 47108 clock cycles

PCI=(A)/(B)=47108/9218=5.11

c4) would it be better to introduce a L2 cache as in point A, or to reduce L1 latency from 2 to 1 clock cycle?

Give a properly argumented, quantitative answer.

Introducing a L2 cache produces no benefit, if the L2 cache block is as wide as the L1 block, because the number of misses is the same as with no L2. On the contrary, is increases the cost of each 512 miss from 2+50 to 2+4+50.

The decrese of L1 latency instead reduces the clock cycles of memory bound instructions to 512\*(1+50+7), so that

(A)= 4+1024 \* (3\*2+4+3) + 512 \* (1+50+7)= 43012 clock cycles

DI

EX

WB

RR

RE

CO

wo

QUEUE

**Decoupled execution model for bundled (paired) instructions**

The state diagram depicts the model for a dynamically scheduled, speculative execution microarchitecture equipped with a Reorder Buffer (ROB) and a set of Reservation Stations (RS). The ROB and RSs are allocated during the ISSUE phase, denoted as RAT (Register Alias Allocation Table) in INTEL microarchitectures, as follows: a bundle (2 instructions) if fetched from the QUEUE of decoded instructions and ISSUED if there is a free pair of consecutive entries in the ROB ( head and tail of the ROB queue do not match); a maximum of two instructions are moved into the RS (if available) when all of their operands are available. Access memory instructions are allocated in the ROB and then moved to a load/store buffer (if available) when operands (address and data, if proper) are available .

**States** are labelled as follows:

WO: Waiting for Operands (at least one of the operands is not available)

RE: Ready for Execution (all operands are available)

DI: Dispatched (posted to a free RS or load/store buffer)

EX: Execution (moved to a load/store buffer or to a matching and free UF)

WB: Write Back (result is ready and is returned to the Rob by using in exclusive mode the Common Data Bus CDB)

RR: Ready to Retire (result available or STORE has completed)

CO: Commit (result is copied to the final ISA register)

**State transitions** happen at the following events:

 *from* QUEUE *to* WO: ROB entry available, operand missing

*from* QUEUE *to* RE: ROB entry available, all operands available

*loop at* WO: waiting for operand(s)

*from* WO *to* RE: all operands available

*loop at* RE: waiting for a free RS or load/store buffer

*from* RE *to* DI: RS or load/store buffer available

*loop on* DI: waiting for a free UF

*from* DI *to* EX: UF available

*loop at* EX: multi-cycle execution in a UF, or waiting for CDB

*from* EX *to* WB: result written to the ROB with exclusive use of CDB

*from* EX *to* RR: STORE completed, branch evaluted

*loop at* RR: instruction completed, not at the head of the ROB, or bundled with a not RR instruction

*from* RR *to* CO: bundle of RR instructions at the head of the ROB, no exception raised

**Resources***Register-to-Register* instructions hold resources as follows:

ROB: from state WO (or RE) up to CO, inclusive;

RS: state DI

UF: EX and WB

*Load/Store* instructions hold resources as follows:

ROB: from state WO (or RE) up to CO, inclusive;

Load buffer: from state DI up to WB

Store buffer: from state DI up to EX (do not use WB)

**Forwarding**: a write on the CDB (WB) makes the operand available to the consumer in the same clock cycle. If the consumer is doing a state transition from QUEUE to WO or RE, that operand is made available; if the consumer is in WO, it goes to RE in the same clock cycle of WB for the producer.

**Branches**: they compute Next-PC and the branch condition in EX and optionally forward Next-PC to the “in-order” section of the pipeline (Fetch states) in the next clock cycle. They do not enter WB and go to RR instead.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Standardname** | **Memoryclock (MHz)** | **I/O busclock (MHz)** | **Datarate (**[**MT/s**](https://en.wikipedia.org/wiki/Transfer_%28computing%29)**)** | **Modulename** | **Peak transferrate (MB/s)** | **Timings,CL-tRCD-tRP** | **CAS latency(ns)** |
| DDR4-1600J\*DDR4-1600KDDR4-1600L | 200 | 800 | 1600 | PC4-12800 | 12800 | 10-10-1011-11-1112-12-12 | 12.513.7515 |
| DDR4-1866L\*DDR4-1866MDDR4-1866N | 233.33 | 933.33 | 1866.67 | PC4-14900 | 14933.33 | 12-12-1213-13-1314-14-14 | 12.85713.92915 |
| DDR4-2133N\*DDR4-2133PDDR4-2133R | 266.67 | 1066.67 | 2133.33 | PC4-17000 | 17066.67 | 14-14-1415-15-1516-16-16 | 13.12514.06315 |
| DDR4-2400P\*DDR4-2400RDDR4-2400U | 300 | 1200 | 2400 | PC4-19200 | 19200 | 15-15-1516-16-1618-18-18 | 12.513.3315 |