CPUs vs. Memory Hierarchies

eller
hur mår Moores lag

Erik Hagersten
Uppsala Universitet

CPU Improvements “Moore’s Law”
2x performance improv./18 mo
(Gordon Moore actually never said this...)

Relative Performance
[log scale]

Historical rate: 55% /year
How Fast is a Computer Today?

Outline

- Pipeline replay
- Problems with pipelines
- Memory system (caches)
- Future:
  - Are we hitting the wall hard
  - Multiprocessors
  - Multiple threads/CPU
Lifting the CPU hood (simplified...)

Instructions:

```
D
C
B
A
```

```
CPU
```

```
Mem
```

Pipeline

Instructions:

```
I
R
X
W
```

```
Regs
```

```
Mem
```
Pipeline

A

IRXW

Regs

Mem

Pipeline

A

IRXW

Regs

Mem
Pipeline:

- A
- I R X W
- Regs
- Mem

I = Instruction fetch
R = Read register
X = Execute
W = Write register
Register Operations:

R1:=R2 op R3

Load Operation:

LD R1, (cnst+R2)
Store Operation:

ST (cnst+R1), R2

Cycle 1

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
Cycle 1

- **IF** RegC < 100 **GOTO** A
- RegC := RegC + 1
- RegB := RegA + 1

LD RegA, (100 + RegC)

Cycle 2

- **IF** RegC < 100 **GOTO** A
- RegC := RegC + 1
- RegB := RegA + 1

LD RegA, (100 + RegC)
Cycle 3

LD RegA, (100 + RegC)
IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1

Cycle 4

LD RegA, (100 + RegC)
IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
Pipelining: a great idea!!

- More pipelines ⇒ faster clock ⇒ better throughput
- Why not have 100 pipeline stages?

Cycle 4

Control Dependence

```
PC  ID  IF RegC < 100 GOTO A
    RegC := RegC + 1
    RegB := RegA + 1
    LD RegA, (100 + RegC)
```

 ld RegA, (100 + RegC)
### Cycle 7

#### Control Dependence

PC $\rightarrow$

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF RegC &lt; 100 GOTO A</td>
</tr>
<tr>
<td>RegC := RegC + 1</td>
</tr>
<tr>
<td>RegB := RegA + 1</td>
</tr>
<tr>
<td>LD RegA, (100 + RegC)</td>
</tr>
</tbody>
</table>

**Regs**

Branch $\rightarrow$ Next PC

**Mem**

---

#### Data Dependence 😞

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF RegC &lt; 100 GOTO A</td>
</tr>
<tr>
<td>RegC := RegC + 1</td>
</tr>
<tr>
<td>RegB := RegA + 1</td>
</tr>
<tr>
<td>LD RegA, (100 + RegC)</td>
</tr>
</tbody>
</table>

**Regs**

Branch $\rightarrow$ Next PC

**Mem**
Compiler optimizations
Gotta’ find independent instructions

LD RegA, (100 + RegC)
IF RegC < 100 GOTO A
RegB := RegA + 1
RegC := RegC + 1

Mem

200 cycles to access memory

But: it is a lot worse!!

LD RegA, (100 + RegC)
IF RegC < 100 GOTO A
RegB := RegA + 1
RegC := RegC + 1

Mem

200 cycles to access memory
Pipeline stalls:
200 indep. instructions between LD and usage

IF RegC < 100 GOTO A
RegB := RegA + 1

"Stall x 199"

RegC := RegC + 1
LD RegA, (100 + RegC)

Fix ➔
- Small dynamic memories: Caches
- Prefetch data before it is needed (HW or SW)

The Memory Gap

Relative Performance
[log scale]

Historical rate: 55%/year
Historical rate: 10-20

CPUs
Mem
Caches and more caches
or
spam, spam, spam and spam

Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se

Cache knowledge useful when...

- Designing a new computer
- Writing an optimized program
  - or compiler
  - or operating system ...
- Implementing software caching
  - Web caches
  - Proxies
  - File systems
Memory/storage

- **sram**: 1ns, 1ns, 3ns, 10ns, 150ns, 5,000,000ns
- **dram**: 10ns, 4MB, 1GB, 10,000,000ns
- **disk**: 5,000,000ns, 1 TB

(1982: 200ns, 200ns, 200ns, 200ns, 10,000,000ns)

Address Book Cache

Looking for Tommy’s Telephone Number

- “Address Tag”
- “Data”

One entry per page => Direct-mapped caches with 28 entries

Indexing function
Address Book Cache
Looking for Tommy’s Number

OMMY 12345

EQ?

TOMMY

Index

Address Book Cache
Looking for Tomas’ Number

OMMY 12345

EQ?

TOMAS

Index

Miss!
Lookup Tomas’ number in the telephone directory
Address Book Cache
Looking for Tomas’ Number

Replace TOMMY’s data with TOMAS’ data. There is no other choice (direct mapped)

Cache

CPU

Memory

Cache

data (a word)

hit

address

data
Cache Organization

Cache Organization (really)
4kB, direct mapped

What is a good index function?

32 bit address identifying a byte in memory

Fast Memory

1k entries of 4 bytes each
Cache Organization
4kB, direct mapped

32 bit address
0010011000101001010011010100011

Identifies the byte within a word

1k entries of 4 bytes each

Valid
(1)
&
(1)
Hit?
(1)

Adresse
(20)

Data
(32)

Hit: Use the data provided from the cache
~Hit: Use data from memory and also store it in the cache
Cache performance parameters

- Cache “hit rate” [%]
- Cache “miss rate” [%] (= 1 - hit_rate)
- Hit time [CPU cycles]
- Miss time [CPU cycles]
- Hit bandwidth
- Miss bandwidth
- Write strategy
- ....

Simple performance equation:
(Cycles per instruction = CPI)

CPI = 1 - mem_ratio + 
     mem_ratio * (hit_rate * hit_time) + 
     mem_ratio * (1 - hit_rate) * miss_time

mem_ratio = 0.25
hit_rate = 0.85
hit_time = 3
miss_time = 100

CPI = 0.75 + 0.25 * 0.85 * 3 + 0.25 * 0.15 * 100 =
     0.75 + 0.64 + 3.75 = 5.14

CPU     HIT     MISS
What if …

\[ \text{CPI} = 1 - \text{mem\_ratio} + \text{mem\_ratio} \times (\text{hit\_rate} \times \text{hit\_time}) + \text{mem\_ratio} \times (1 - \text{hit\_rate}) \times \text{miss\_time} \]

<table>
<thead>
<tr>
<th>mem_ratio</th>
<th>hit_rate</th>
<th>hit_time</th>
<th>miss_time</th>
<th>CPU</th>
<th>HIT</th>
<th>MISS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>0.85</td>
<td>3</td>
<td>100</td>
<td>0.75 + 0.64 + 3.75 = 5.14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Twice as fast CPU  
  \[ 0.37 + 0.64 + 3.75 = 4.77 \]

• Faster memory (70c)  
  \[ 0.75 + 0.64 + 2.62 = 4.01 \]

• Improve hit\_rate (0.95)  
  \[ 0.75 + 0.71 + 1.25 = 2.71 \]

How to get more effective caches:

- Larger cache (more capacity)
- Cache block size (larger cache lines)
- More placement choice (more associativity)
- Innovative caches (victim, skewed, …)
- Cache hierarchies (L1, L2, L3, CMR)
- Latency-hiding (weaker memory models)
- Latency-avoiding (prefetching)
- Cache avoidance (cache bypass)
Why do you miss in a cache

- Mark Hill’s three “Cs”
  - Compulsory miss (touching data for the first time)
  - Capacity miss (the cache is too small)
  - Conflict misses (imperfect cache implementation)

Why do you hit in a cache?

- Temporal locality
  - Likely to access the same data again soon
- Spatial locality
  - Likely to access nearby data again soon

*Typical access pattern:*
(inner loop stepping through an array)
Avoiding Capacity Misses – a huge address book
Lots of pages. One entry per page.

New Indexing function

“Address Tag”
“Data”

One entry per page =>
Direct-mapped caches with 28 entries

Cache Organization
1MB, direct mapped

32 bit address
256k entries
Identifies the byte within a word
Pros/Cons Large Caches

++ The safest way to get improved hit rate
-- SRAMs are very expensive!!
-- Larger size ==> slower speed
  more load on “signals”
  longer distances
-- (power consumption)
-- (reliability)

Fetch more than a word:

cache blocks

1MB, direct mapped, CacheLine=16B

<table>
<thead>
<tr>
<th>Index</th>
<th>Select (12)</th>
<th>Hit? (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100110001010010011010100011</td>
<td>(1)</td>
<td>&amp;</td>
</tr>
</tbody>
</table>

identifies the word within a cache line

Identifies a byte within a word
Pros/Cons Large Cache Lines

++ Explores spatial locality
++ Fits well with modern DRAMs
  * first DRAM access slow
  * subsequent accesses fast ("page mode")
-- Poor usage of SRAM & BW for some patterns
-- Higher miss penalty (fix: critical word first)
-- (False sharing in multiprocessors)

![Graph showing Perf vs Cache line size](image)

Cache Conflicts

Typical access pattern:
(inner loop stepping through an array)
A, B, C, A+1, B, C, A+2, B, C, ...

What if B and C index to the same cache location
Conflict misses -- big time!
Potential performance loss 10-100x
Address Book Cache
Two names per page: First index -- then search.

Avoiding conflict: More associativity
1MB, 2-way, CL=4B
A combination thereof
1MB, 2-way, CL=16B

Identifies the word within a cache line
Identifies a byte within a word

Going all the way...
1MB, fully associative, CL=16B

Identifies the word within a cache line
Identifies a byte within a word

One “set”
Pros/Cons Associativity

++ Avoids conflict misses
-- Slower access time
-- More complex implementation
    comparators, muxes, ...
-- Requires more pins (for external SRAM...)

Who to replace?
Picking a “victim”

- Least-recently used
  - Considered the best algorithm
  - Only practical up to 4-way
- Not most recently used
  - Remember who used it last: 8-way -> 3 bits/CL
- Pseudo-LRU
  - Course Time stamps, used in the VM system
- Random replacement
  - Can’t continuously to have “bad luck..."
Replacing dirty cache lines

- **Write-back**
  - A “dirty bit” indicates an altered cache line
  - Write dirty data back to memory (next level) at replacement
- **Write-through**
  - Always write through to the next level (as well)
  - Never dirty data to write back

Cache Hierarchy

- L1 cache is fast but small
- VIPT caches adds constraints (later)
- Database application with huge datasets
- Latency 1:100 between on-chip SRAM and off-chip DRAM

=> Cache hierarchies
Cache Hierarchy

Memory

L3$ on-board

L2$ on-module

L1S on-chip

CPU

Future directions
“The future isn’t what it used to be”
[A.C. Clarke]

Questions:
- Are we hitting the wall?
- Is a technology shift happening?

It is a lot worse: Superscalars
Modern CPUs: “superscalars” with ~4 parallel pipelines

+ Higher throughput
- More complicated architecture
- Branch delay more expensive (more instr. missed)
- Harder to find “enough” independent instr. (need 8 instr. between write and use)
Modern CPUs: ~10-20 stages/pipe

+ Shorter cycle time (more MHz)
- Branch delay even more expensive
- Even harder to find “enough” independent instructions

Modern MEM: ~200 CPU cycles

Need to find lots of independent instructions

200 cycles 😊
Adding the memory hierarchy

Common speculations in CPUs
- Caches
- Address translation caches (TLBs)
- Prefetching (SW&HW, Data & Instr.)
- Branch prediction

More complications
- Out-of-order execution
- Value prediction
- Calculate both branches
- VLIW
**CPU Improvements**

Relative Performance  
[log scale]

- **Historical rate**: 55%/year

---

**55%/year**

- **CPU architectural improvements**
  - Pipelines, superscalar, O-O-O, smart caches...
    - --> ~15% improvement per year

- **Faster clock**
  - Shrinkage of the transistors, deeper pipelines...
    - --> ~30% improvement per year

- **More on-chip state**
  - Larger caches, reordering buffers, registers...
    - --> ~10% improvement per year
1 (3) slides about multiprocessors

But, we need more than Moore

- Banks
- WWW
- Airlines
- Science
- Simulation
- ...

→ Multiprocessors

2 (3) slides about multiprocessors

SMP:
Symmetric Multiprocessors

Gotta’ fix the cache coherence!! (often done in HW)
3 (3) slides about multiprocessors

The programmer’s view

Shared Memory

Thread
Thread
Thread
Thread
Thread
Thread
Thread

Technology “Improvements”

Wire delay 5mm trace (RC model)

Clock cycle time, SIA*)

Quantitative data and trends according to V. Agarwal et al., ISCA 2000

* SIA = (Semiconductor Industry Association) prediction, 1999
Technology “Improvements”

Quantitative data and trends according to V. Agarwal et al., ISCA 2000

Technology “Improvements”

Quantitative data and trends according to V. Agarwal et al., ISCA 2000
Technology “Improvements”
Span (Fraction of the chip area reachable in one cycle)

Quantitative data and trends according to V. Agarwal et al., ISCA 2000

Can we continue to make large and complicated CPU?...

- Diminishing return on..
  - pipeline stages
  - parallel pipelines
- Smaller reach ➔
  - Hard to add complexity, smaller caches
- Memory is often the bottleneck
**Technology “Improvements”**

Relative Performance

[log scale]

Quantitative data and trends according to V. Agarwal et al., ISCA 2000

- Historical rate: 55%/yr
- Business as usual (Same arch.): \(\sim 12\% \text{ / yr} \)
- \(6.0 - 7.4 \times\)
- \(1,720 \times\)

---

**CPU Future: there is hope**

- Remember the parallel threads?
- Run multiple “threads” per CPU chip
  - SMT (Simultaneous Multi Threading)
  - CMP (Chip Multi Processor)
SMT: Simultaneous Multithreading

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 cycles</td>
<td>2kB</td>
</tr>
<tr>
<td>10 cycles</td>
<td>64kB</td>
</tr>
<tr>
<td>30 cycles</td>
<td>2MB</td>
</tr>
<tr>
<td>150 cycles</td>
<td>1GB</td>
</tr>
</tbody>
</table>

Thread 1

Thread N

Issue logic

Regs

B M M W

B M M W

B M M W

B M M W

CMP: Chip Multiprocessor

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mem</td>
</tr>
</tbody>
</table>

Thread 1

Thread N

Issue logic

Regs

B M M W

B M M W

B M M W

B M M W

kr

€
**CMP: Chip Multiprocessor**

Chip Multiprocessor (CMP):

- External I/F
- Mem I/F
- L2$
- ST$ CPU
- ST$ CPU
- ST$ CPU
- ST$ CPU
- Simple fast CPU local caches
- 4 threads

**The Computer Shrunk Again**

Old Mainframes

Super Minis:

Microprocessor:

Chip Multiprocessor (CMP):
The future is here today!

- Intel: P4 XEON: SMT “hyperthreading”
- IBM: Power4: CMP
- Sun: Niagara (32-way), UltraSPARC4+
- Compaq: EV8 SMT
- Itanium: Rumors has it... 2006.
The Future

- CMP/SMT = SMP multiprocessor
- Multi CMP/SMT = cc-NUMA

Parallelism = no_chips * no_threads_per_chip
global_comm_cost > 10 * local_comm_cost
Application scalability??
OS scalability??
Gotta explore small caches, Gotta explore locality!

What’s in it for Sweden?

- Gotta write multithreaded programs
- Gotta get them right 😊
- Gotta build CPUs
  - Axis
  - Xelerated
  - Ericsson
  - ...
- Gotta build systems on a chip (SoC)
In a nutshell

- We’re going to miss the wall again!
- Multithreading will become key
- Multiprocessor technology is moving down the food chain
- Programmers better be prepared!
- SoC architects better be prepared!

- Oh yeah -- this assumes CMOS technology:
  Bio-, nano-, quantum-computing may eventually be around...