Welcome to AVDARK

Erik Hagersten
Uppsala University
AVDARK in a nutshell

**Literature**  Computer Architecture A Quantitative Approach (4th edition)
John Hennesey & David Patterson

**Lecturer**  
- **Erik Hagersten** gives most lectures and is responsible for the course.
- **Andreas Sandberg** is responsible for the labs and the hand-ins.
- **Jakob Carlström from Xelerated** will teach network processors.
- **Sverker Holmgren** will teach parallel programming.
- **David Black-Schaffer** will teach about graphics processors.

**Mandatory Assignment**  
There are four lab assignments that all participants have to complete before a hard deadline. Each can earn you a bonus point.

**Optional Assignment**  
There are four (optional) hand-in assignments. Each can earn you a bonus point.

**Examination**  
Written exam at the end of the course. No books are allowed.

**Bonus system**  
64p max/32p to pass. For each bonus point, there is a corresponding question 4p bonus question. Full bonus ➔ Pass.
AVDARK on the web

www.it.uu.se/edu/course/homepage/avdark/ht10

Menu:
Welcome!
News
FAQ
Schedule
Slides
New Papers
Assignments
Reading instr 4:ed
Exam
Schedule in a nutshell

1. Memory Systems (~Appendix C in 4th Ed)
   Caches, VM, DRAM, microbenchmarks, optimizing SW

2. Multiprocessors
   TLP: coherence, memory models, interconnects, scalability, clusters, ...

3. Scalable Multiprocessors
   Scalability, synchornization, clusters, ...

4. CPUs
   ILP: pipelines, scheduling, superscalars, VLIWs, Vecotor instructions...

5. Widening + Future (~Chapter 1 in 4th Ed)
   Technology impact, GPUs, Network processors, Multicores (!!)
# Lectures1: Memory Systems

<table>
<thead>
<tr>
<th>#</th>
<th>Day</th>
<th>Time</th>
<th>Room</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Thu 2 sept</td>
<td>08-10</td>
<td>1211</td>
<td>Welcome, intro and caches</td>
</tr>
<tr>
<td>2</td>
<td>Mo 6 sep</td>
<td>15-17</td>
<td>1211</td>
<td>Caches and virtual memory</td>
</tr>
<tr>
<td>3</td>
<td>Tue 7 sep</td>
<td>10-12</td>
<td>1211</td>
<td>Virtual memory and Microbenchmarks</td>
</tr>
<tr>
<td>4</td>
<td>Fri 10 sep</td>
<td>10-12</td>
<td>1211</td>
<td>Profiling and optimizing for the memory sys</td>
</tr>
<tr>
<td>5</td>
<td>Tue 14 sep</td>
<td>08-09</td>
<td>1211</td>
<td>Introduction to SIMICS and Lab1 intro</td>
</tr>
</tbody>
</table>

## Lab 1: Memory Systems

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Room</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tue 14 sep</td>
<td>9-12</td>
<td>1549D</td>
<td>Preparation slot *)</td>
</tr>
<tr>
<td>Wed 15 sep</td>
<td>8-12</td>
<td>1549D</td>
<td>Group A **)</td>
</tr>
<tr>
<td>Thu 16 sep</td>
<td>8-12</td>
<td>1549D</td>
<td>Group B **)</td>
</tr>
<tr>
<td>Fri 17 sep</td>
<td>8-12</td>
<td>1549D</td>
<td>Group C **)</td>
</tr>
</tbody>
</table>

**Hard deadline => solutions handed after deadline will be ignored**

• 2010-09-20 at 08:14: Lab 1 (or use the lab occasions).

• 2010-09-20 at 08:14: Handin 1 to AS (Leave them in AS’s Mail Box on the 4th floor, Building 1).
Exam and bonus

- 4 Mandatory labs
- 4 Hand-in (optional)
- Written Exam

How to get a bonus point:

- Complete extra bonus activity at lab occasion
- Complete optional bonus hand-in [with a reasonable accuracy] before a hard deadline

⇒ 32p/64p at the exam = PASS
Goal for this course

- Understand **how and why** modern computer systems are designed the way they are:
  - pipelines
  - memory organization
  - virtual/physical memory ...

- Understand **how and why** parallelism is created and
  - Instruction-level parallelism
  - Memory-level parallelism
  - Thread-level parallelism...

- Understand **how and why** multiprocessors are built
  - Cache coherence
  - Memory models
  - Synchronization...

- Understand **how and why** multiprocessors of combined SIMD/MIMD type are built
  - GPU
  - Vector processing...

- Understand **how** computer systems are adopted to different usage areas
  - General-purpose processors
  - Embedded/network processors...

- Understand the physical limitation of modern computers
  - Bandwidth
  - Energy
  - Cooling...
Introduction to Computer Architecture

Erik Hagersten
Uppsala University
What is computer architecture?

“Bridging the gap between programs and transistors”

“Finding the best model to execute the programs”

best={fast, cheap, energy-efficient, reliable, predictable, ...}

...
“Only” 20 years ago: APZ 212
“the AXE supercomputer”
APZ 212
marketing brochure quotes:

- "Very compact"
  - 6 times the performance
  - 1/6:th the size
  - 1/5 the power consumption
- "A breakthrough in computer science"
- "Why more CPU power?"
- "All the power needed for future development"
- "...800,000 BHCA, should that ever be needed"
- "SPC computer science at its most elegance"
- "Using 64 kbit memory chips"
- "1500W power consumption"
CPU Improvements

Relative Performance
[log scale]

Historical rate: 55%/year

Year
How do we get good performance?

Creating and exploring:

1) Locality
   a) Spatial locality
   b) Temporal locality
   c) Geographical locality

2) Parallelism
   a) Instruction level
   b) Thread level
Compiler Organization

Fortran Front-end → Intermediate Representation → High-level Optimization → Global & Local Optimization → Code Generation → "Machine Code"

C Front-end → Intermediate Representation → High-level Optimization → Global & Local Optimization → Code Generation → "Machine Code"

C++ Front-end → Intermediate Representation → High-level Optimization → Global & Local Optimization → Code Generation → "Machine Code"

Machine-independent Translation

Procedure in-lining
Loop transformation
Register Allocation
Common sub-expressions
Instruction selection
constant folding
Execution in a CPU

Memory

"Machine Code"

"Data"

CPU
Load/Store architecture (e.g., "RISC")

ALU ops: Reg --> Reg
Mem ops: Reg <--> Mem

Example: C = A + B

Compiler

Memory Accesses

Three Regs: Source1 Source2 Destination

Load R1, [A]
Load R3, [B]
Add R2, R1, R3
Store R2, [C]
Register-based machine

Example: C := A + B

Data:

A: 12
B: 14
C: 26

LD R1, [A]
LD R7, [B]
ADD R2, R1, R7
ST R2, [C]

Program counter (PC)

"Machine Code"

11 10 9 8 6 5 3
? ? ?

14 26 12
How “long” is a CPU cycle?

- 1982: 5MHz
  200ns $\rightarrow$ 60 m (in vacuum)

- 2002: 3GHz clock
  0.3ns $\rightarrow$ 10cm (in vacuum)
  0.3ns $\rightarrow$ 3mm (on silicon)
Lifting the CPU hood (simplified...)

Instructions:

1. D
2. C
3. B
4. A

CPU

Mem
Pipeline

Instructions:

D -> C -> B -> A

I  R  X  W

Regs

Mem
Pipeline
Pipeline
Pipeline

A

I R X W

Regs

Mem
Pipeline:

I = Instruction fetch
R = Read register
X = Execute
W = Write register
Pipeline system in the book

- Instruction fetch
- Instruction decode/register fetch
- Execute/address calculation
- Memory access
- Write back

Diagram showing the pipeline system with stages labeled as I, R, X, M, W.
Register Operations:

Add R1, R2, R3

Ifetch

OP: +

Mem

Regs

PC T W X R

A
Initially

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

LD RegA, (100 + RegC)

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
Cycle 2

LD RegA, (100 + RegC)

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

PC ➔
Cycle 3

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
Cycle 4

PC \rightarrow D

IF RegC < 100 GOTO A

RegC := RegC + 1

RegB := RegA + 1

LD RegA, (100 + RegC)
Data dependency 😞

![Diagram showing data dependency]

**Problem:** The new value of RegA is written too late into the register file in order to be seen by Instruction B.
Data dependency 😞

(Could also be solved by compiler optimizations. More about this when we study instruction scheduling and out-of-order execution)
It is actually a lot worse!

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. (In this example: need 8 instr. between reg write and usage)
Today: ~10-20 stages and 4-6 pipes

+ Shorter cycle time (many GHz)
+ Many instructions started each cycle
- Very hard to find “enough” independent instr = ILP
Modern MEM: ~200 CPU cycles

- Shorter cycle time (more GHz)
- Many instructions started each cycle
- Very hard to find "enough" independent instr.
- Slow memory access will dominate

200 cycles
Connecting to the Memory System

Data Memory System

Instr Memory System

I R X M W

Dept of Information Technology | www.it.uu.se

© Erik Hagersten | http://user.it.uu.se/~eh

AVDA 201
Common speculations in CPUs

- Caches [next]
- Address translation caches (TLBs) [later]
- Prefetching (SW&HW, Data & Instr.) [much later]
- Branch prediction [later]
- Execute ahead [not covered in this course]

More complications

- Execute instructions out-of-order, but still make it look like an in-order execution [much later]
- Multithreading (much later)
- Multicore
- ...
Caches and more caches
or
spam, spam, spam and spam

Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se
Fix: Use a cache

Issue logic

~1 cycles

$\sim 32kB$

200 cycles

Mem

1GB
Webster about “cache”

1. cache ˈkash  n [F, fr. cacher to press, hide, fr. (assumed) VL coacticare to press] together, fr. L coactare to compel, fr. coactus, pp. of cogere to compel - more at COGENT 1a: a hiding place esp. for concealing and preserving provisions or implements 1b: a secure place of storage 2: something hidden or stored in a cache
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

2000: 1ns 1ns 3ns 10ns 150ns 5,000,000ns
1kB 64k 4MB 1GB 1TB

1982: 200ns 200ns 43200ns 10,000,000ns
Address Book Cache
Looking for Tommy’s Telephone Number

“Address Tag”
“Data”

One entry per page $\Rightarrow$
Direct-mapped caches with 28 entries

Indexing function
Address Book Cache
Looking for Tommy’s Number

EQ?

TOMMY

index

OMMY 12345
Address Book Cache
Looking for Tomas’ Number

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 → Cache: address
Cache → Memory: data
CPU ← Memory: data

hit
data (a word)
Cache Organization

Cache

TOMAS

Index

OMAS

23457

Valid

(1)

(1)

(4)

(4)

Address tag

Data (5 digits)

Hit (1)

&

49
Cache Organization (really)
4kB, direct mapped

What is a good index function?

32 bit address identifying a byte in memory

1k entries of 4 bytes each

Ordinary Memory
Cache Organization
4kB, direct mapped

32 bit address

Mem Overhead: 21/32 = 66%

Latency = SRAM+CMP+AND

Identifies the byte within a word

1k entries of 4 bytes each

index

Valid

Data

Overhead:

Latency =

SRAM+CMP+AND
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
- ....
How to rate architecture performance?

Marketing:
- Frequency / Number of cores...

Architecture “goodness”:
- CPI = Cycles Per Instruction
- IPC = Instructions Per Cycle

Benchmarking:
- SPEC-fp, SPEC-int, ...
- TPC-C, TPC-D, ...
Cache performance example

Assumption:
Infinite bandwidth
A perfect 1.0 CyclesPerInstruction (CPI) CPU
100% instruction cache hit rate

**Total number of cycles** =
#Instr. * ( (1 - mem_ratio) * 1 +
   mem_ratio * avg_mem_accesstime) =

= #Instr * ( (1 - mem_ratio) +
   mem_ratio * (hit_rate * hit_time +
         (1 - hit_rate) * miss_time)

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

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

<table>
<thead>
<tr>
<th>mem_ratio = 0.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>hit_rate = 0.85</td>
</tr>
<tr>
<td>hit_time = 3</td>
</tr>
<tr>
<td>miss_time = 100</td>
</tr>
</tbody>
</table>

\[
\text{CPI} = 0.75 + 0.25 \times 0.85 \times 3 + 0.25 \times 0.15 \times 100 = \\
0.75 + 0.64 + 3.75 = 5.14
\]

CPU     HIT     MISS
What if ...

\[
\begin{align*}
\text{CPI} &= 1 - \text{mem}_\text{ratio} + \\
&\quad \text{mem}_\text{ratio} \times (\text{hit}_\text{rate} \times \text{hit}_\text{time}) + \\
&\quad \text{mem}_\text{ratio} \times (1 - \text{hit}_\text{rate}) \times \text{miss}_\text{time})
\end{align*}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{mem}_\text{ratio} &= 0.25 \\
\text{hit}_\text{rate} &= 0.85 \\
\text{hit}_\text{time} &= 3 \\
\text{miss}_\text{time} &= 100 \\
\hline
\end{array}
\]

\[
\begin{align*}
\text{CPU} &\quad \text{HIT} &\quad \text{MISS} \\
0.75 &+ 0.64 &+ 3.75 &= 5.14
\end{align*}
\]

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

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

• Improve hit_rate (0.95) \[\Rightarrow 0.75 + 0.71 + 1.25 = 2.71\]
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 avoiding (cache bypass)
- Optimized application/compiler
- ...

AVDARK 2010
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 (non-ideal cache implementation) (too many names starting with “H”)

- (Multiprocessors)
  - Communication (imposed by communication)
  - False sharing (side-effect from large cache blocks)
Avoiding Capacity Misses –
a huge address book
Lots of pages. One entry per page.

“Address Tag”

“Data”

One entry per page =>
Direct-mapped caches with 784 (28 x 28) entries
Cache Organization
1MB, direct mapped

32 bit address
00100110000101001010011010100011

Identifies the byte within a word

Mem Overhead: 13/32 = 40%

Latency = SRAM + CMP + AND

Mem
0101001100101

Overhead:
13/32 = 40%

Latency = SRAM + CMP + AND

Hit?
61

Data
(32)

Valid
(1)

Addr
tag
(12)

Index
(18)

256k entries
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)
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)

A, B, C, A+1, B, C, A+2, B, C, ...

![Diagram showing temporal and spatial locality](image)
Fetch more than a word: cache blocks (a.k.a cache line)
1MB, direct mapped, CacheLine=16B

Identifies the word within a cache line

Identifies a byte within a word

Mem Overhead: 13/128 = 10%

Latency = SRAM + CMP + AND

Overhead: 13/128 = 10%
Example in Class

Direct mapped cache:

- Cache size = 64 kB
- Cache line = 16 B
- Word size = 4B
- 32 bits address (byte addressable)

“There are 10 kinds of people in the world: Those who understand binary number and those who do not.”
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)
UART: StatCache Graph
app=matrix multiply

Note: this is just a single example, but the conclusion typically holds for most applications.

Thanks: Dr. Erik Berg
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 10-100×
Address Book Cache

Two names per page: index first, then search.

OMAS  23457
OMMY  12345

index

EQ?

EQ?
Avoiding conflict: More associativity
1MB, 2-way set-associative, CL=4B

Latency = SRAM+CMP+AND+LOGIC+MUX

How should the select signal be produced?
Pros/Cons Associativity

++ Avoids conflict misses
-- Slower access time
-- More complex implementation
    comparators, muxes, ...
-- Requires more pins (for external SRAM...)
Going all the way…!
1MB, fully associative, CL=16B

Identifies the word within a cache line

Identifies a byte within a word

One “set”

64k comparators

“logic”

Hit?

Select (16)

4B Data

Multiplexer (256k:1 mux)
Fully Associative

- Very expensive
- Only used for small caches (and sometimes TLBs)

CAM = Contents-addressable memory
- ~Fully-associative cache storing key+data
- Provide key to CAM and get the associated data
A combination thereof
1MB, 2-way, CL=16B

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

"logic"

Hit?

Multiplexer (8:1 mux)

Select

Data

001001100001010010100110101000110

0101001

0010011100101

(15)

index

(13)

(13)

&

&

(2)

(1)

(256)

(128)

(128)

32k "sets"

msb

lsb

Multiplexer (8:1 mux)

Select

Data
Example in Class

- Cache size = 2 MB
- Cache line = 64 B
- Word size = 8B (64 bits)
- 4-way set associative
- 32 bits address (byte addressable)
Who to replace? 
Picking a “victim”

- Least-recently used (aka LRU)
  - Considered the “best” algorithm (which is not always true...)
  - Only practical up to limited number of ways

- Not most recently used
  - Remember who used it last: 8-way -> 3 bits/CL

- Pseudo-LRU
  - E.g., based on course time stamps.
  - Used in the VM system

- Random replacement
  - Can’t continuously to have “bad luck...
Cache Model: Random vs. LRU

art (SPEC 2000)  
equate (SPEC 2000)
4-way sub-blocked cache
1MB, direct mapped, Block=64B, sub-block=16B

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

Mem Overhead: 16/512 = 3%
Pros/Cons Sub-blocking

++ Lowers the memory overhead
++ (Avoids problems with false sharing -- MP)
++ Avoids problems with bandwidth waste
-- Will not explore as much spatial locality
-- Still poor utilization of SRAM
-- Fewer sparse “things” allocated
Replacing dirty cache lines

- **Write-back**
  - Write dirty data back to memory (next level) at replacement
  - A “dirty bit” indicates an altered cache line

- **Write-through**
  - Always write through to the next level (as well)
  - data will never be dirty ➔ no write-backs
Write Buffer/Store Buffer

- Do not need the old value for a store

- One option: Write around (no write allocate in caches) used for lower level smaller caches
Innovative cache: Victim cache

Victim Cache (VC): a small, fairly associative cache (~10s of entries)

Lookup: search cache and VC in parallel

Cache replacement: move victim to the VC and replace in VC

VC hit: swap VC data with the corresponding data in Cache

“A second life 😊”
Skewed Associative Cache

A, B and C have a three-way conflict

It has been shown that 2-way skewed performs roughly the same as 4-way caches
Skewed-associative cache:
Different indexing functions

32 bit address

Identifies the byte within a word

128k entries

function

&

&

0010011000101001010011010100011

0101001 001001100101

0101001 0010011100101

1 0101001 0010011100101

1 0101001 0010011100101

2:1mux

(32)

(32)

(32)
UART: Elbow cache
Increase “associativity” when needed

If severe conflict: make room

Performs roughly the same as an 8-way cache
Slightly faster
Uses much less power!!
Topology of caches: Harvard Arch

- CPU needs a new instruction each cycle
- 25% of instruction LD/ST
- Data and Instr. have different access patterns
  ==> Separate D and I first level cache
  ==> Unified 2nd and 3rd level caches
Cache Hierarchy of Today

- DRAM Memory
  - L3$ off-chip
    - Off-chip SRAM (less common today)
  - L2$ on-chip
  - L1 D$ on-chip
  - L1 I$ on-chip
- CPU

Additional notes:
- Use whatever transistors there left for the Last-Level Cache (LLC)
- Small enough to keep up with the CPU speed
- Separate I cache to allow for instruction fetch and data fetch in parallel
Hardware prefetching

- Hardware “monitor” looking for patterns in memory accesses
- Brings data of anticipated future accesses into the cache prior to their usage
- Two major types:
  - Sequential prefetching (typically page-based, 2nd level cache and higher). Detects sequential cache lines missing in the cache.
  - PC-based prefetching, integrated with the pipeline. Finds per-PC strides. Can find more complicated patterns.
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)

(Multiprocessors)
- Communication (imposed by communication)
- False sharing (side-effect from large cache blocks)
How are we doing?

- Creating and exploring:
  1) Locality
     a) Spatial locality
     b) Temporal locality
     c) Geographical locality
  2) Parallelism
     a) Instruction level
     b) Thread level
Memory Technology

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

Performance of main memory (from 3rd Ed... faster today)

- **Access time**: time between address is latched and data is available (~50ns)
- **Cycle time**: time between requests (~100 ns)
- **Total access time**: from Id to REG valid (~150ns)

- Main memory is built from **DRAM**: Dynamic RAM
- 1 transistor/bit ==> more error prune and slow
- Refresh and precharge

- Cache memory is built from **SRAM**: Static RAM
  - about 4-6 transistors/bit
The address is multiplexed Row/Address Strobe (RAS/CAS)

“Thin” organizations (between x16 and x1) to decrease pin load

Refresh of memory cells decreases bandwidth

Bit-error rate creates a need for error-correction (ECC)
Address is typically not multiplexed
Each cell consists of about 4-6 transistors
Wider organization (x18 or x36), typically few chips
Often parity protected (ECC becoming more common)
Error Detection and Correction

Error-correction and detection
- E.g., 64 bit data protected by 8 bits of ECC
  - Protects DRAM and high-availability SRAM applications
  - Double bit error detection ("crash and burn")
  - Chip kill detection (all bits of one chip stuck at all-1 or all-0)
  - Single bit correction
  - Need “memory scrubbing” in order to get good coverage

Parity
- E.g., 8 bit data protected by 1 bit parity
  - Protects SRAM and data paths
  - Single-bit "crash and burn" detection
  - Not sufficient for large SRAMs today!!
Correcting the Error

- Correction on the fly by hardware
  - no performance-glitch
  - great for cycle-level redundancy
  - fixes the problem for now...

- Trap to software
  - correct the data value and write back to memory

- Memory scrubber
  - kernel process that periodically touches all of memory
Improving main memory performance

- Page-mode => faster access within a small distance
- Improves bandwidth per pin -- not time to critical word
- Single wide bank improves access time to the complete CL
- Multiple banks improves bandwidth
Newer kind of DRAM...

- SDRAM (5-1-1-1 @100 MHz)
  - Mem controller provides strobe for next seq. access
- DDR-DRAM (5-½-½-½)
  - Transfer data on both edges
- RAMBUS
  - Fast unidirectional circular bus
  - Split transaction addr/data
  - Each DRAM devices implements RAS/CAS/refresh... internally
- CPU and DRAM on the same chip?? (IMEM)...
Newer DRAMs ...
(Several DRAM arrays on a die)

<table>
<thead>
<tr>
<th>Name</th>
<th>Clock rate (MHz)</th>
<th>BW (GB/s per DIMM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDR-260</td>
<td>133</td>
<td>2,1</td>
</tr>
<tr>
<td>DDR-300</td>
<td>150</td>
<td>2,4</td>
</tr>
<tr>
<td>DDR2-533</td>
<td>266</td>
<td>4,3</td>
</tr>
<tr>
<td>DDR2-800</td>
<td>400</td>
<td>6,4</td>
</tr>
<tr>
<td>DDR3-1066</td>
<td>533</td>
<td>8,5</td>
</tr>
<tr>
<td>DDR3-1600</td>
<td>800</td>
<td>12,8</td>
</tr>
</tbody>
</table>

2006 access latency: slow=50ns, fast=30ns, cycle time=60ns
Prefetch buffer on DRAM chips
The Endian Mess

Big Endian

Numbering the bytes

Store the value 0x5F

Store the string Hello

Little Endian

64MB
msb lsb

0  1  2  3
4  5  6  7

0  3  2  1  0
3  2  1  0

64MB
msb lsb

00 00 00 5f

00 00 00 5f

Hello

00 00 00 5f

H e l l o

Word

100
Virtual Memory System

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

Diagram showing a program connected to physical memory and a disk, with physical memory containing 64MB.
Virtual and Physical Memory

Virtual Memory

Context A

Context B

Physical Memory

(Caches)

Disk

4GB

64MB

Segments

$1

$2
Translation & Protection

Virtual Memory

Physical Memory

Disk

4GB 4GB
Context A Context B

stack heap data text

stack heap data text

64MB

R RW R RW

Disk

R

R

RW

RW

0

0

0

© Erik Hagersten | http://user.it.uu.se/~eh
Virtual memory — parameters
Compared to first-level cache parameters

- Replacement in cache handled by HW. Replacement in VM handled by SW
- VM hit latency very low (often zero cycles)
- VM miss latency huge (several kinds of misses)
- Allocation size is one “page” 4kB and up)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>First-level cache</th>
<th>Virtual memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block (page) size</td>
<td>16-128 bytes</td>
<td>4K-64K bytes</td>
</tr>
<tr>
<td>Hit time</td>
<td>1-2 clock cycles</td>
<td>40-100 clock cycles</td>
</tr>
<tr>
<td>Miss penalty (Access time)</td>
<td>8-100 clock cycles (6-60 clock cycles)</td>
<td>700K-6000K clock cycles (500K-4000K clock cycles)</td>
</tr>
<tr>
<td>(Transfer time)</td>
<td>(2-40 clock cycles)</td>
<td>(200K-2000K clock cycles)</td>
</tr>
<tr>
<td>Miss rate</td>
<td>0.5%-10%</td>
<td>0.00001%-0.001%</td>
</tr>
<tr>
<td>Data memory size</td>
<td>16 Kbyte - 1 Mbyte</td>
<td>16 Mbyte - 8 Gbyte</td>
</tr>
</tbody>
</table>
VM: Block placement

Where can a block (page) be placed in main memory?
What is the organization of the VM?

- The high miss penalty makes SW solutions to implement a **fully associative address mapping** feasible at page faults
- A page from disk may occupy any pageframe in PA
- Some restriction can be helpful (page coloring)
VM: Block identification

Use a page table stored in main memory:

- Suppose 8 Kbyte pages, 48 bit virtual address
- Page table occupies $2^{48}/2^{13} \times 4B = 2^{37} = 128$GB!!!

Solutions:

- Only one entry per physical page is needed
- Multi-level page table (dynamic)
- Inverted page table (~hashing)
Address translation

- Multi-level table: The Alpha 21064

Segment is selected by bit 62 & 63 in addr.

Kernel segment
Used by OS.
Does not use virtual memory.

User segment 1
Used for stack.

User segment 0
Used for instr. & static data & heap
Protection mechanisms

The address translation mechanism can be used to provide memory protection:

- Use **protection attribute bits** for each page
- Stored **in the page table entry** (PTE) (and TLB...)
- Each physical page gets its own **per process protection**
- **Violations** detected during the address translation **cause exceptions** (i.e., SW trap)
- **Supervisor/user modes** necessary to prevent user processes from changing e.g. PTEs
Fast address translation

How can we avoid three extra memory references for each original memory reference?

- Store the most commonly used address translations in a cache—*Translation Look-aside Buffer* (TLB)

==> *The caches rears their ugly faces again!*
Do we need a fast TLB?

- Why do a TLB lookup for every L1 access?
- Why not cache virtual addresses instead?
  - Move the TLB on the other side of the cache
  - It is only needed for finding stuff in Memory anyhow
  - The TLB can be made larger and slower – or can it?
Aliasing Problem

The same physical page may be accessed using different virtual addresses

- A virtual cache will cause confusion -- a write by one process may not be observed
- Flushing the cache on each process switch is slow (and may only help partly)
- \( \Rightarrow \) VIPT (VirtuallyIndexedPhysicallyTagged) is the answer
  - Direct-mapped cache no larger than a page
  - No more sets than there are cache lines on a page + logic
  - Page coloring can be used to guarantee correspondence between more PA and VA bits (e.g., Sun Microsystems)
Virtually Indexed Physically Tagged = VIPT

Have to guarantee that all aliases have the same index
- L1_cache_size < (page-size * associativity)
- Page coloring can help further
What is the capacity of the TLB

Typical TLB size = 0.5 - 2kB
Each translation entry 4 - 8B ==> 32 - 500 entries
Typical page size = 4kB - 16kB

**TLB-reach** = 0.1MB - 8MB

**FIX:**

- *Multiple page sizes, e.g., 8kB and 8 MB*
- *TSB -- A direct-mapped translation in memory as a “second-level TLB”*
VM: Page replacement

Most important: minimize number of page faults

Page replacement strategies:

• FIFO—First-In-First-Out

• LRU—Least Recently Used

• Approximation to LRU
  • Each page has a reference bit that is set on a reference
  • The OS periodically resets the reference bits
  • When a page is replaced, a page with a reference bit that is not set is chosen
So far...

Diagram showing the flow of data through the CPU, TLB, and memory, including TLB fill and TLB miss events. The diagram also includes a page fault handler and interactions with the disk.
Adding TSB (software TLB cache)
VM: Write strategy

Write back or Write through?

- **Write back!**
- Write through is impossible to use:
  - Too long access time to disk
  - The write buffer would need to be *prohibitively* large
  - The I/O system would need an extremely high bandwidth
# VM dictionary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Memory System</td>
<td>The “cache” language</td>
</tr>
<tr>
<td>Virtual address</td>
<td>~Cache address</td>
</tr>
<tr>
<td>Physical address</td>
<td>~Cache location</td>
</tr>
<tr>
<td>Page</td>
<td>~Huge cache block</td>
</tr>
<tr>
<td>Page fault</td>
<td>~Extremely painfull $miss</td>
</tr>
<tr>
<td>Page-fault handler</td>
<td>~The software filling the $</td>
</tr>
<tr>
<td>Page-out</td>
<td>Write-back if dirty</td>
</tr>
</tbody>
</table>
Caches Everywhere...

- D cache
- I cache
- L2 cache
- L3 cache
- ITLB
- DTLB
- TSB
- Virtual memory system
- Branch predictors
- Directory cache
- ...

120
Exploring the Memory of a Computer System

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

```c
for (times = 0; times < Max; times++) /* many times*/

    for (i=0; i < ArraySize; i = i + Stride)
        dummy = A[i]; /* touch an item in the array */
```

Measuring the average access time to memory, while varying ArraySize and Stride, will allow us to reverse-engineer the memory system.
(need to turn off HW prefetching...)
Micro Benchmark Signature

for (times = 0; times < Max; times++) /* many times*/
    for (i=0; i < ArraySize; i = i + Stride)
        dummy = A[i]; /* touch an item in the array */
Stepping through the array

```c
for (times = 0; times < Max; times++) /* many times*/
    for (i=0; i < ArraySize; i = i + Stride)
        dummy = A[i]; /* touch an item in the array */
```

- **Array Size = 16, Stride=4**
- **Array Size = 16, Stride=8**
- **Array Size = 32, Stride=4**
- **Array Size = 32, Stride=8**
Micro Benchmark Signature

for (times = 0; times < Max; time++) /* many times*/

    for (i=0; i < ArraySize; i = i + Stride)
        dummy = A[i]; /* touch an item in the array */
Micro Benchmark Signature

for (times = 0; times < Max; time++) /* many times*/
    for (i=0; i < ArraySize; i = i + Stride)
        dummy = A[i]; /* touch an item in the array */

Mem+TLBmiss
L2$+TLBmiss
Mem=300ns
L2$hit=40ns
L1$ hit
L1$ block size=16B
L2$ block size=64B
Page size=8k 126

ArraySize=8MB
ArraySize=512kB
ArraySize=32kB-256kB
ArraySize=16kB

L1$ hit
Pagesize=8k==> #TLB entries = 32-64
(56 normal+8 large)
Twice as large L2 cache ???

for (times = 0; times < Max; time++) /* many times*/

    for (i=0; i < ArraySize; i = i + Stride)
        dummy = A[i]; /* touch an item in the array */
Twice as large TLB...

```c
for (times = 0; times < Max; time++) /* many times*/
    for (i=0; i < ArraySize; i = i + Stride)
        dummy = A[i]; /* touch an item in the array */
```
How are we doing?

Creating and exploring:

1) Locality
   a) Spatial locality
   b) Temporal locality
   c) Geographical locality

2) Parallelism
   a) Instruction level
   b) Thread level