Welcome to DARK2
(IT, MN)

Erik Hagersten
Uppsala University
www.it.uu.se/edu/course/homepage/dark2/ht07

**Literature**  Computer Architecture A Quantitative Approach (4th edition)  Hennesey & Pattersson

**Lecturer**  Erik Hagersten gives most lectures and is responsible for the course  Frédéric Haziza is responsible for the laborations and the hand-ins  Jakob Carlström, Xelerated guest lecturer in network processors  Sverker Holmgren guest lecturer in parallel programming

**Mandatory Assignment**  There are two lab assignments that all participants have to complete before a hard deadline. (+ a Microprocessor Report/Microbenchmark if you are doing the MN2 version)

**Optional Assignment**  There are three (optional) hand-in assignments: Memory, CPU, Multiprocessors. You will get extra credit at the exam …

**Examination**  Written exam at the end of the course. No books are allowed.
DARK2, Autumn 2007

Welcome!
News
Forms
Schedule
Slides
Papers
Assignments
Reading instructions
Exam

www.it.uu.se/edu/course/homepage/dark2/ht07
DARK2 in a nutshell

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

2. **Multiprocessors**
   TLP: coherence, interconnects, scalability, clusters, ...

3. **CPUs**
   ILP: pipelines, scheduling, superscalars, VLIWs, embedded, ...

4. **Widening + Future** (~Chapter 1 in 4th Ed)
   Technology impact, TLP+ILP in the CPU,...
# Part 1: Memory System

<table>
<thead>
<tr>
<th>Day</th>
<th>Time</th>
<th>Session</th>
<th>Instructor(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30/10</td>
<td>08.15-10.00</td>
<td>Welcome+Caches</td>
<td>EH</td>
</tr>
<tr>
<td>03/11</td>
<td>10.15-12.00</td>
<td>Caches and virtual memory</td>
<td>EH</td>
</tr>
<tr>
<td>04/10</td>
<td>15.15-17.00</td>
<td>Profiling and opt. + Lab I Intro</td>
<td>EH +FH</td>
</tr>
<tr>
<td>10/11</td>
<td>10.15-11.00</td>
<td>Statistical modeling</td>
<td>EH</td>
</tr>
</tbody>
</table>

### Lab 1

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>7/11</td>
<td>08.15-12.00</td>
<td>Group A</td>
</tr>
<tr>
<td>7/11</td>
<td>13.15-17.00</td>
<td>Group B</td>
</tr>
</tbody>
</table>

**THESE ARE HARD DEADLINES!**

- 10/11 12:01 Lab 1 (Use the lab occasions)
- 10/11 10:00 Handin 1 to FH (Leave them in FH's Mail Box on the 4th floor, Building 1).
Exam and bonus

- 3 Optional handins  (3 x 8p)
- 2 Optional Lab-bonus activity (2 x 4p)

= 32 p at the exam guaranteed (of 64p)
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
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
Execution in a CPU

"Machine Code"

"Data"

CPU
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]
How “long” is a CPU cycle?

- 1982: 5MHz
  200ns → 60 m (in vacum)

- 2002: 3GHz clock
  0.3ns → 10cm (in vacum)
  0.3ns → 3mm (on silicon)
Lifting the CPU hood (simplified...)

Instructions:

```
D
C
B
A

CPU

Mem
```
Pipeline

Instructions:

1. D
2. C
3. B
4. 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
Register Operations:

Add R1, R2, R3
Initially

DARK2
2008

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

PC

I  R  X  W
Regs

Mem
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

IF RegC < 100 GOTO A

RegC := RegC + 1

RegB := RegA + 1

LD RegA, (100 + RegC)
Cycle 4

LD RegA, (100 + RegC)

IF RegC < 100 GOTO A

RegC := RegC + 1

RegB := RegA + 1

LD RegA, (100 + RegC)
Today: ~10-20 stages and 4-6 pipes

+ Shorter cycle time (more GHz)
- Branch delay even more expensive
- Even harder to find "enough" independent instr.
Modern MEM: ~150 CPU cycles

+ Shorter cycle time (more GHz)
- Branch delay even more expensive
- Memory access even more expensive
- Even harder to find "enough" independent instr.
Connecting to the Memory System

Data Memory System

Instr Memory System

Instruction fetch

Instruction decode/ register fetch

Execute/ address calculation

Memory access

Add

NPG

pc

Zero?

Branch taken

Instr

Registers

ALU

AlU output

Data

st data

dest data

Mux

Data

I

R

X

M

W
Caches and more caches
or
spam, spam, spam and spam

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

- ~1 cycles
- ~32kB
- 250 cycles
- 1GB

Issue logic

Regs

I
R
B M M W

I
R
B M M W

I
R
B M M W

I
R
B M M W

Mem

$
Webster about “cache”

1. cache \'kash\ 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

<table>
<thead>
<tr>
<th>Year</th>
<th>SRAM</th>
<th>DRAM</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>1ns</td>
<td>10ns</td>
<td>5,000,000ns</td>
</tr>
<tr>
<td>1982</td>
<td>200ns</td>
<td>150ns</td>
<td>10,000,000ns</td>
</tr>
</tbody>
</table>

**SRAM**
- 1kB: 1ns
- 64kB: 1ns
- 4MB: 3ns

**DRAM**
- 1GB: 10ns
- 1TB: 150ns

**Disk**
- 5,000,000ns

(1982: 200ns, 200ns)
Address Book Cache
Looking for Tommy’s Telephone Number

One entry per page =>
Direct-mapped caches with 28 entries
# Address Book Cache

## Looking for Tommy’s Number

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>OMMY</td>
<td>12345</td>
</tr>
</tbody>
</table>

To find Tommy’s number, we perform an index lookup and then check if the name matches the query `TOMMY`. If there's a match, we return the number `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

Memory

address

address

hit

data (a word)

data
Cache Organization

Cache

TOMAS

index

Valid

Hit

Data (5 digits)

Data (5 digits)

1 OMAS 23457

Valid (1)

= (1)

& (1)

(4) Addr tag

(4)

(1)

(4)

(1)

(1)
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

Identifies the byte within a word

Mem
Overhead: 21/32 = 66%

Latency = SRAM+CMP+AND

1k entries of 4 bytes each
Cache

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)

\textbf{CPI} = 1 - mem_ratio +
mem_ratio * (hit_rate * hit_time +
(1 - hit_rate) * miss_time)
Example Numbers

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

\[
\begin{align*}
\text{mem\_ratio} &= 0.25 \\
\text{hit\_rate} &= 0.85 \\
\text{hit\_time} &= 3 \\
\text{miss\_time} &= 100
\end{align*}
\]

\[
\begin{align*}
CPI &= 0.75 + 0.25 \times 0.85 \times 3 + 0.25 \times 0.15 \times 100 = \\
&= 0.75 + 0.64 + 3.75 = 5.14
\end{align*}
\]

CPU     HIT     MISS
What if ...

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

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

\[\Rightarrow 0.75 + 0.64 + 3.75 = 5.14\]

- 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
- ...
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.

One entry per page => Direct-mapped caches with 784 (28 x 28) entries

“Address Tag”

“Data”

New Indexing function
Cache Organization

1MB, direct mapped

32 bit address

00100110000101001010011010100011

Identifies the byte within a word

Mem Overhead:
13/32 = 40%

Latency = SRAM + CMP + AND

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, ...
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:
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
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: index first, then search.

OMAS  23457
OMMY  12345

index

EQ?

EQ?
How should the select signal be produced?

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

Latency = SRAM+CMP+AND+LOGIC+MUX

Identifies a byte within a word

One “set”
128k “sets”

Multiplexer
(2.1-mux)

Hit?
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

00100110000101001010011010100011

(0) index
(2)
(28)
(13)

64k comparators

One “set”

16B 16B 16B

Multiplexer (256k:1 mux)

Select (16)

Hit?

4B Data

“logic”
Fully Associative

- Very expensive
- Only used for small caches

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

msb

lsb

(15)
index

(13)

(13)

&

&

“logic”

= 

= 

(2)

(1)

Select

Hit?

(32)
Data

Multiplexer
(8:1 mux)

(128)

(128)

(256)

(32k)
“sets”

001001100001010010100110101000110

0101001

0010011100101
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 ~4-way

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

- Pseudo-LRU
  - 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**

**Equake**
4-way sub-blocked cache
1MB, direct mapped, Block=64B, sub-block=16B

- Sub block within a block
- Identifies the word within a cache line
- Identifies a byte within a word

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

&

&

2:1mux

index

= =

f1

f2

>18

>18

(17)

(17)

(32)

(32)

(32)

= 0010011000101001010011010100011

128k entries

Identifies the byte within a word

function

&

&

2:1mux

index

= =

f1

f2

>18

>18

(17)

(17)

(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!!
Cache Hierarchy Latency

300:1 between on-chip SRAM - DRAM

- cache hierarchies

- L1: small on-chip cache
  - Runs in tandem with pipeline → small
  - VIPT caches adds constraints (more later...)

- L2: large SRAM on-chip
  - Communication latency becomes more important

- L3: Off-chip SRAM
  - Huge cache ~10x faster than DRAM
Cache Hierarchy

- CPU
- L1$: on-chip
- L2$: on-module
- L3$: on-board
- Memory
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
Common Cache Structure for Servers

L1: CL=32B, Size=32kB, 4-way, 1ns, split I/D
L2: CL=128B, Size= 1MB, 8-way, 4ns, unified
L3: CL=128B, Size= 32MB, 2-way, 15ns, unified
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 ld 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
DRAM organization

- 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: slow=50ns, fast=30ns, cycle time=60ns
The Endian Mess

Numbering the bytes

Big Endian

Little Endian

Store the value 0x5F

Store the string Hello

94
Virtual Memory System

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

- PROGRAM
- Physical Memory: 0, 64MB
- Disk
Virtual and Physical Memory

Context A

Context B

Physical Memory

Segments

Disk

Program

(Caches)

0

4GB

0

4GB

64MB

heap

data

text

heap

data

text

$1

$2

$
Translation & Protection

Virtual Memory

Context A

- Stack
- Heap
- Data
- Text

4GB

Context B

- Stack
- Heap
- Data
- Text

4GB

Physical Memory

- RW
- R

Disk

64MB
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</td>
<td>700K-6000K clock cycles</td>
</tr>
<tr>
<td></td>
<td>(6-60 clock cycles)</td>
<td>(500K-4000K clock cycles)</td>
</tr>
<tr>
<td></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

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

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

**Page Table Entry:** (translation & protection)
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
- \( L1\_cache\_size < (\text{page-size} \times \text{associativity}) \)
- 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…
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
- ...

DARK2 2008
Exploring the Memory of a Computer System

Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se
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 */
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 */
```
Stepping through the array

```c
def 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 */
```
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 */
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**

Mem+TLBmiss

L2$+TLBmiss

Mem=300ns

L2$hit=40ns

L1$ hit

L1$ block size=16B

L2$ block size=64B

Page size=8k

120

---

ArraySize=8MB

ArraySize=512kB

ArraySize=32kB-256kB

ArraySize=16kB

---

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...

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
Optimizing for cache performance

Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se
What is the potential gain?

- Latency difference L1$ and mem: \( \sim 50x \)
- Bandwidth difference L1$ and mem: \( \sim 20x \)
- Repeated TLB misses adds a factor \( \sim 2-3x \)
- Execute from L1$ instead from mem \( \Rightarrow 50-150x \) improvement
- At least a factor 2-4x is within reach
Optimizing for cache performance

- Keep the active footprint small
- Use the entire cache line once it has been brought into the cache
- Fetch a cache line prior to its usage
- Let the CPU that already has the data in its cache do the job
- ...

DARK2 2008
Example: Loop order

//Optimized Example A
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        A[i][j] = A[i-1][j-1];
    }
}

//Unoptimized Example A
for (j=0; j<N; j++) {
    for (i=0; i<N; i++) {
        A[i][j] = A[i-1][j-1];
    }
}
Performance Difference

![Graph showing speedup vs. unoptimized performance for different processors. The x-axis represents the array side size, and the y-axis shows the speedup. The graph compares Athlon64 x2, Pentium D, and Core 2 Duo processors.]
Example: Sparse data

// Optimized Example A
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        A_data[i][j] = A_data[i-1][j-1];
    }
}

// Unoptimized Example A
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        A[i][j].data = A[i-1][j-1].data;
    }
}
Performance Difference

- Athlon64 x2
- Pentium D
- Core 2 Duo

Speedup vs UnOPT vs Array side

16 32 64 128 256 512 1024 2048 4096
Loop Merging

/* Unoptimized */
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        a[i][j] = 2 * b[i][j];
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        c[i][j] = K * b[i][j] + d[i][j]/2

/* Optimized */
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        a[i][j] = 2 * b[i][j];
    c[i][j] = K * b[i][j] + d[i][j]/2;
Padding of data structures

```
A
A+1024*4
A+2048*4

00100110000101001010011010100011

lsb

index

(17)

= &

(13)

(13)

&

“logic”

= &

(1)

Select

(2:1 r)

Hit?

Data

Multip

(32)

(32)

0101001 001001100101
```

```
(1)

0101001
```

```
(32)

0101001
```
Padding of data structures

allocate more memory than needed

A
A + 1024 * 4 + padding
A + 2048 * 4 + 2 * padding

index

0010011000101001010011010100011

(17)

Hit?

S

“logic”

&

&
/* Unoptimized ARRAY: \( x = y \times z \) */

for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        \( r = 0; \)
        for (k = 0; k < N; k = k + 1)
            \( r = r + y[i][k] \times z[k][j]; \)
        \( x[i][j] = r; \)
/* Optimized ARRAY: \( X = Y \ast Z \) */

for (jj = 0; jj < N; jj = jj + B)
for (kk = 0; kk < N; kk = kk + B)
for (i = 0; i < N; i = i + 1)
  for (j = jj; j < min(jj+B,N); j = j + 1)
    \{ r = 0;
    for (k = kk; k < min(kk+B,N); k = k + 1)
      r = r + y[i][k] \ast z[k][j];
    x[i][j] += r;
  \};

Partial solution

First block

Second block
Blocking: the Movie!

/* Optimized ARRAY: \( X = Y \times Z \) */
for (\( jj = 0; \ jj < N; \ jj = \ jj + B \))
for (\( kk = 0; \ kk < N; \ kk = \ kk + B \))
for (\( i = 0; \ i < N; \ i = i + 1 \))
    for (\( j = \ jj; \ j < \ \min(\ jj+B, N); \ j = j + 1 \))
        \( r = 0 \);
        for (\( k = \ kk; \ k < \ \min(\ kk+B, N); \ k = k + 1 \))
            \( r = r + y[i][k] \times z[k][j]; \)
    \( x[i][j] += r; \);

Partial solution

First block

Second block

X:

Y:

Z:
Prefetching

/* Unoptimized */
for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
        x[i][j] = 2 * x[i][j];

/* Optimized */
for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
        PREFETCH x[i+8][j]
        x[i][j] = 2 * x[i][j];
Cache Affinity

- Schedule the process on the processor it last ran
- Allocate and free data buffers in a LIFO order
How are we doing?

Creating and exploring:

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

2) Parallelism
   a) Instruction level
   b) Loop level
   c) Thread level