Welcome to DARK2 (IT, MN)

Erik Hagersten
Uppsala University

DARK2 On the web

www.it.uu.se/edu/course/homepage/dark2/ht07

DARK2, Autumn 2006
- Welcome!
- News
- Forms
- Schedule
- Slides
- Papers
- Assignments
- Reading instructions
- Exam

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

Day  Room  Time               Topic /Lecturer
24/10  1211  10.15-12.00  Welcome+Caches EH
24/10  1211  13.15-15.00  Caches and virtual memory EH
29/10  1211  13.15-15.00  Profiling and optimizing for the mem EH
30/10  1211  13.15-15.00  Statistical modeling + Lab 1 intro EH+FH

Lab 1
31/10  1515  8:15-12.00  Group A
31/10  1515  13.15-17.00  Group B

THESE ARE HARD DEADLINES!
7/11 12:01 Lab 1 (Use the lab occasions)
7/11 17:00 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)

Part 1: Memory System

Day  Room  Time               Topic /Lecturer
24/10  1211  10.15-12.00  Welcome+Caches EH
24/10  1211  13.15-15.00  Caches and virtual memory EH
29/10  1211  13.15-15.00  Profiling and optimizing for the mem EH
30/10  1211  13.15-15.00  Statistical modeling + Lab 1 intro EH+FH

Lab 1
31/10  1515  8:15-12.00  Group A
31/10  1515  13.15-17.00  Group B

THESE ARE HARD DEADLINES!
7/11 12:01 Lab 1 (Use the lab occasions)
7/11 17:00 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)

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]

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"  CPU  "Data"
```

Register-based machine

```
Example: C := A + B
```

```
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:
```

```
CPU

Mem
```
Pipeline

Instructions:

A

I R X W

Regs

Mem

A

I R X W

Regs

Mem
Pipeline:

- Instruction fetch (I)
- Read register (R)
- Execute (X)
- Write register (W)

Register Operations:

Add R1, R2, R3

Initially

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

Pipeline system in the book
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

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
Cycle 5

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

Cycle 6

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

Cycle 7

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

Cycle 8

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

- Great instruction throughput (one/cycle)!
- Explored instruction-level parallelism (ILP)!
- Requires "enough" "independent" instructions
  - Control dependence
  - Data dependence

Data dependency

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

Today: ~10-20 stages and 4-6 pipes

+ Shorter cycletime (more MHz)
+ Even more ILP (parallel pipelines)
- Branch delay even more expensive
- Even harder to find "enough" independent instr.

Modern MEM: ~150 CPU cycles

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

Instruction-Level Parallelism (ILP) in Superscalar Pipelines

START:
J=K+L
G=H+I
D=E+F
A=B+C
MEM[X]=MEM[X]+14
X=X+1
IF X < 1000 GOTO START:

K+L START:
Issue Logic

Connecting to the Memory System

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

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

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>R</th>
<th>B</th>
<th>M</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>R</td>
<td>B</td>
<td>M</td>
<td>M</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>R</td>
<td>B</td>
<td>M</td>
<td>M</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>R</td>
<td>B</td>
<td>M</td>
<td>M</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>R</td>
<td>B</td>
<td>M</td>
<td>M</td>
<td>W</td>
<td></td>
</tr>
</tbody>
</table>

Issue logic → Regs

~1 cycles → ~32kB

250cycles → Mem → 1GB

---

**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>Storage Type</th>
<th>Access Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>sram</td>
<td>1ns 1ns 3ns</td>
</tr>
<tr>
<td></td>
<td>dram</td>
<td>150ns</td>
</tr>
<tr>
<td></td>
<td>disk</td>
<td>5,000,000ns</td>
</tr>
</tbody>
</table>

(1982: 200ns 200ns)

---
Address Book Cache
Looking for Tommy’s Telephone Number

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

Indexing function

“Address Tag”
“Data”

Address Book Cache
Looking for Tommy’s Number

Miss! Lookup Tommy’s number in the telephone directory

Replace Tommy’s data with Tomas’ data. There is no other choice (direct mapped)
Cache Organization (really)

4kB, direct mapped

What is a good index function?

1k entries of 4 bytes each

Identifies the byte within a word

Mem

Overhead: 21/32 = 66%

Latency = SRAM + CMP + AND

32 bit address identifying a byte in memory

Ordinary Memory

32 bit address

Identifies the byte within a word
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
- …. 

Cache performance example

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

Total number of cycles =
\[
#Instr. \times \left( 1 - \text{mem_ratio} \right) * 1 + \text{mem_ratio} \times \text{avg_mem_accessstime} = \]
\[
= #Instr \times \left( 1 - \text{mem_ratio} \right) + \text{mem_ratio} \times (\text{hit_rate} \times \text{hit_time} + \left( 1 - \text{hit_rate} \right) \times \text{miss_time})
\]

CPI = \[ \frac{1 - \text{mem_ratio} + \text{mem_ratio} \times (\text{hit_rate} \times \text{hit_time} + \left( 1 - \text{hit_rate} \right) \times \text{miss_time})}{#Instr} \]
Example Numbers

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

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

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

What if ... 

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

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

\[ \text{CPU HIT MISS } \]

\[
\begin{array}{ccc}
\text{CPU} & \text{HIT} & \text{MISS} \\
0.75 & 0.64 & 3.75 = 5.14
\end{array}
\]

- 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 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.

“Address Tag” “Data”

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

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

Latency Analysis

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 10-100x

Address Book Cache
Two names per page: index first, then search.

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

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

Latency = SRAM+CMP+AND+LOGIC+MUX
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

Hit? 4B

Data

00100110000101001010110110100011

Identifies a byte within a word

Identifies the word within a cache line

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

32k “sets”

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

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

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

Innovative cache: Victim cache

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

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

Different indexing functions

Identifies the byte within a word

32 bit address

Identifies the byte within a word

Index = \text{index} = f_1(>18) \& f_2(17)

128k entries

1:1mux

2:1mux

UART: Elbow cache
Increase “associativity” when needed

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

Memory

L3$ on-board

L2$ on-module

L1$ on-chip

CPU
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

<table>
<thead>
<tr>
<th>Level</th>
<th>Cache Line (CL)</th>
<th>Size</th>
<th>Way</th>
<th>Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>32B</td>
<td>32kB</td>
<td>4-way</td>
<td>1ns, split I/D</td>
</tr>
<tr>
<td>L2</td>
<td>128B</td>
<td>1MB</td>
<td>8-way</td>
<td>4ns, unified</td>
</tr>
<tr>
<td>L3</td>
<td>128B</td>
<td>32MB</td>
<td>2-way</td>
<td>15ns, unified</td>
</tr>
</tbody>
</table>

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 prone 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)

SRAM organization

- 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

Big Endian

Little Endian

Numbering the bytes

Physical Memory

Virtual Memory System

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

Translation & Protection

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</td>
<td>8-100 clock cycles</td>
<td>700K-6000K clock cycles</td>
</tr>
<tr>
<td>Access (time)</td>
<td>(6-40 clock cycles)</td>
<td>(500K-8000 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>0.000001%-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 identification

Use a page table stored in main

- Suppose 8 Kbyte pages, 48 bit virtual address
- Page table occupies \(2^{48}/2^{13}\times 4B = 2^{35} = 128GB!!!\)
- 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 Lookaside 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)
- $\implies$ VIPT (Virtually Indexed Physically Tagged) 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
- $L1\_cache\_size < (page-size \times associativity)$
- Page coloring can help further

What is the capacity of the TLB

Typical TLB size = 0.5 - 2kB
Each translation entry 4 - 8B $\implies$ 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

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**

Virtual Memory System: The “cache” language
- Virtual address: ~Cache address
- Physical address: ~Cache location
- Page: ~Huge cache block
- Page fault: ~Extremely painful $miss
- Page-fault handler: ~The software filling the $
- Page-out: Write-back if dirty

**Putting it all together**

**TLB**
- TLB fill
- PF handler
- PT
- I
- D
- Memory
- Disk
- 1-2ns 2-4ns 10-20ns
- 150ns
- 500ns
- 2-10ms
- TLBI
- Atrans$
- L2 miss
- L1S
- Data L1S
- Instr L1S
- Unified L2S

**Summary**

Cache memories:
- HW-management
- Separate instruction and data caches permits simultaneous instruction fetch and data access
- Four questions:
  - Block placement
  - Block identification
  - Block replacement
  - Write strategy

Virtual memory:
- Software-management
- Very high miss penalty => miss rate must be very low
- Also supports:
  - memory protection
  - multiprogramming

**Caches Everywhere...**

- D cache
- I cache
- L2 cache
- L3 cache
- ITLB
- DTLB
- TSB
- Virtual memory system
- Branch predictors
- Directory cache
- ...
Exploring the Memory of a Computer System

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

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

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 */

Mem+TLBmiss
Mem=300ns
L2$+TLBmiss
L2$hit=40ns
L1$ hit

Avg time (ns)

Time (ns)

Stride (bytes)

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 */
Can software help us?

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: ~50x
- Bandwidth difference L1$ and mem: ~20x
- Repeated TLB misses adds a factor ~2-3x
- Execute from L1$ instead from mem ==> 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
- ...
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];
    }
}

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;
    }
}
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

allocate more memory than needed

Blocking

/* Unoptimized ARRAY: x = y * 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] * z[k][j];
        x[i][j] = r;
Blocking

/* Optimized ARRAY: X = Y * 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] * z[k][j];
 x[i][j] += r;
};

/* Optimized ARRAY: X = Y * 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] * z[k][j];
 x[i][j] += r;
};

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

Virtual Performance Expert

```
/* Unoptimized Array Multiplication: x = y * z    N = 1024 */
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] * z[k][j];
x[i][j] = r;
```

Analyze

You can improve the performance by 45%

#1: Reverse loop order <main.c:45> (15%) (How to)
#2: Hot/cold usage of data <sort.c:56> (12%) (How to)
#3: Sparsely allocated data struct <qs.h:8> (8%) (How to)
#4: Loop fusion <qs.h:12> (10%) (How to)
**Lab1**

- Compile and run programs in an architecture simulator modelling cache and memory
- Study performance when you:
  - change the cache model
  - change the program

**StatCacheε**

a locality tool for SW developers

Erik {Berg, Hagersten}
Uppsala University
Sweden
Caches – a huge cludge
++ hides latency, - - requires locality!

Latency

A = B + C:
Read B 0.3 -- 100 ns
Read C 0.3 -- 100 ns
Add B & C 0.3 -- 100 ns
Write A 0.3 -- 100 ns

Traditional Simulation
Slowdown: ≈100x

Simulated CPU

Simulated Memory System

Hardware Counters
Slowdown: ≈ 0%

ΣstatCacheε Model
Slowdown: ≈ 30%

Hardware Counters

Host Computer

New statistical cache model

Modelling of many memory systems
**Statistical Cache Model**

Find the probability that a load or store instruction causes a cache miss without knowledge of exact cache content.

**Line of reasoning:**
1) How many accesses have occurred since "C" was last touched? 
a.k.a. reuse distance \(d\)
2) How many of them are likely to miss?
3) How likely is it that "C" still resides in the cache after that many misses?

**Hit function**

*Fully assoc, random replacement*

\[
\text{hit}(\text{repl}) = (1 - 1/L)^{\text{repl}}
\]

\(\text{repl} = \#\text{cache misses since last touched}\)
\(L = \#\text{cache lines in the cache}\)

**Miss probability function**

Miss probability: \(f(n) = 1 - (1 - 1/L)^n\)

\(n = \#\text{cache misses since last touched}\)
\(L = \#\text{cache lines in the cache}\)

**Probabilistic Cache Model**

(assumt: "const" miss rate \(R\))

\[
\text{Tot}_\text{misses} = R \times N = \sum_{i=0}^{N} f(d(i) \times R)
\]

\(\#\text{repl} \approx 3 \times M_r\)

\(p_{\text{miss}} = f(3 \times R)\)

\(p_{\text{miss}} = f(5 \times R)\)
Probabilistic Cache Model

By reordering the elements in the sum and using \( h(i) \) instead of \( A(i) \) we get:

\[
h(1)f(R) + h(2)f(2R) + h(3)f(3R) + \ldots \approx RN
\]

This can be solved for \( R \) given a histogram \( h \) obtained by sampling.

The formula only works if the miss ratio is approximately constant. What if the miss ratio changes over time?

Miss Ratio Formula

Solve for \( R \) to get miss ratio. (With numerical method)

\[ R \cdot N = h(1)f(R) + h(2)f(2R) + h(3)f(3R) + \ldots \]

Reuse Distance Histogram

Estimated by Sampling

Probabilistic Cache Model

Hit ratio (%) vs. Time (cycles)
Probabilistic Cache Model

- Split time in time slots
- Generate histogram for each time slot at run-time
- Calculate the miss ratio for each time slot:
  \[ h(1)f(R) + h(2)f(2R) + h(3)f(3R) + \ldots = RN \]
- Take average miss ratio of all time slots

\[ \sum_{tatCach} \varepsilon \]: How accurate?

Results from a traditional Simulator
Why is speed so important?

Miss ratio (%)

Cache size (bytes)

Implementing $\Sigma$tatCach$_E$

Three steps:
1 Select samples:
2 Detect reuse:
3 Measure reuse distance:

Implementation

1 Select samples: Overflow trap from HW perf. counter (DC_rd)
2 Detect reuse: Solaris watchpoint support:
   `write("/proc/self/ctl", addr, $linesize)`
3 Measure reuse distance: Using the perf. counter again (DC_rd)