Welcome to DARK2 (IT, MN and PhD)

Erik Hagersten
Uppsala University

DARK2 On the web

www.it.uu.se/edu/course/homepage/dark2/ht06

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,...

Literature
Computer Architecture A Quantitative Approach (3rd or 4th edition)

Lecturer
Erik Hagersten gives most lectures and is responsible for the course
Frédéric Haziza is responsible for the lab assignments and the hand-ins
Jakob Carlström 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.
Part I: Memory Systems

<table>
<thead>
<tr>
<th>Day</th>
<th>Room</th>
<th>Time</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>06/11</td>
<td>1211</td>
<td>15.15-17.00</td>
<td>Welcome+Caches</td>
</tr>
<tr>
<td>08/11</td>
<td>1311</td>
<td>08.15-10.00</td>
<td>Caches and virtual memory</td>
</tr>
<tr>
<td>08/11</td>
<td>1311</td>
<td>10.15-12.00</td>
<td>Profiling and optimizing for the memory system</td>
</tr>
<tr>
<td>10/11</td>
<td>1211</td>
<td>08.15-10.00</td>
<td>Statistical modelling + Lab 1 introduction</td>
</tr>
</tbody>
</table>

Lab 1

<table>
<thead>
<tr>
<th>Day</th>
<th>Room</th>
<th>Time</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>13/11</td>
<td>1515</td>
<td>15.15-19.00</td>
<td>Group A</td>
</tr>
<tr>
<td>15/11</td>
<td>1515</td>
<td>08.15-12.00</td>
<td>Group B</td>
</tr>
</tbody>
</table>

Deadlines

THESE ARE HARD DEADLINES!

- 17/11 17:00 Handin 1 to FH (Leave them in FH's Mail Box).
- 15/11 12:01 Lab 1 (Use the lab occasions)

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”

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

CPU

"Machine Code"

"Data"
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:

```
A
B
C
D
```

```
CPU
```

Pipeline

Instructions:

```
I
R
X
W
```

```
Mem
```

Regs
Pipeline

A

I
R
X
W
Regs

Mem

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

Register Operations:
Add R1, R2, R3

Initially

Cycle 1
Cycle 2

PC ➔

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

Cycle 3

PC ➔

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

Cycle 4

PC ➔

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

Cycle 5

PC ➔

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
Cycle 6

PC ➔

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

<

I R X W
Regs

Mem

Cycle 7

PC ➔

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

I R X W
Regs

Branch ➔ Next PC

Mem

Cycle 8

PC ➔

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

I R X W
Regs

Mem

Pipelining: a great idea??

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

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

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.

Instruction-Level Parallelism (ILP) in Superscalar Pipelines
Instruction-Level Parallelism (ILP) in Superscalar Pipelines

\[
\begin{align*}
J &= K + L \\
G &= H + I \\
D &= E + F \\
A &= B + C \\
MEM[X] &= MEM[X] + 14 \\
X &= X + 1 \\
\end{align*}
\]

If \( X < 1000 \) GOTO \( \text{START:} \)

Connecting to the Memory System

Fix: Use a cache

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

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

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>sram</td>
<td>1ns</td>
</tr>
<tr>
<td>dram</td>
<td>10ns</td>
</tr>
<tr>
<td>disk</td>
<td>150ns</td>
</tr>
<tr>
<td>(1982)</td>
<td>200ns</td>
</tr>
<tr>
<td></td>
<td>200ns</td>
</tr>
<tr>
<td></td>
<td>10ns</td>
</tr>
<tr>
<td></td>
<td>4MB</td>
</tr>
<tr>
<td></td>
<td>5,000,000ns</td>
</tr>
<tr>
<td></td>
<td>1 TB</td>
</tr>
<tr>
<td></td>
<td>2000ns</td>
</tr>
</tbody>
</table>

Address Book Cache

Looking for Tommy’s Telephone Number

Indexing function

One entry per page => Direct-mapped caches with 28 entries
Address Book Cache Looking for Tommy’s Number

Looking for Tomas’ Number

Address Book Cache

Lookup Tomas’ number in the telephone directory

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

Cache Organization (really)
4kB, direct mapped

What is a good index function

Ordinary Memory

Cache Organization
4kB, direct mapped

Mem Overhead: 21/32 = 66%
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
- ...

DARK2 2006

DARK2 2006

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 Cycles Per Instruction (CPI) CPU
100% instruction cache hit rate

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

= #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

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

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

CPI = 0.75 + 0.25 * 0.85 * 3 + 0.25 * 0.15 * 100 =

0.75 + 0.64 + 3.75 = 5.14

CPU HIT MISS
What if ...

\[ CPI = 1 - \text{mem}_\text{ratio} + \]
\[ \text{mem}_\text{ratio} \times (\text{hit}_\text{rate} \times \text{hit}_\text{time}) + \]
\[ \text{mem}_\text{ratio} \times (1 - \text{hit}_\text{rate}) \times \text{miss}_\text{time} \]

<table>
<thead>
<tr>
<th>CPU</th>
<th>HIT</th>
<th>MISS</th>
</tr>
</thead>
</table>
| 0.75 | 0.64 | 3.75 | 5.14

- 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
- ...

Avoiding Capacity Misses –
a huge address book
Lots of pages. One entry per page.

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)
**Cache Organization**

1MB, direct mapped

- 32 bit address
- Identifies the byte within a word
- Mem Overhead: 13/32 = 40%
- Latency = SRAM+CMP+AND

**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)

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.

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

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

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

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
  - 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

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

---

**Skewed Associative Cache**

A, B and C have a three-way conflict

- 2-way
- 4-way
- 2-way skewed

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
- 128k entries
- 2:1mux
- function
- (32)
**UART: Elbow cache**

Increase “associativity” when needed

If severe conflict: make room

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

- **L1**: small on-chip cache
  - Runs in tandem with pipeline
  - 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

- 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 book...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

---

**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!!

---

**SRAM organization**

- Address is typically not multiplexed
- Each cell consists of about 4-6 transistors
- Wider organization (x16 or x36), typically few chips
- Often parity protected (ECC becoming more common)

---

**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)
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 Store the value 0x5F

Store the string Hello

Big Endian

Little Endian

0 1 2 3 4

64MB msb lsb

0 1 2 3 4

64MB msb lsb

Word

Virtual Memory System

Erik Hagersten

Uppsala University, Sweden

eh@it.uu.se

Physical Memory

Virtual and Physical Memory

Physical Memory

Disk

Program

4GB

64MB

Disk

Context A

Context B

Segments
**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>(Transfer time)</td>
<td>(2-40 clock cycles)</td>
<td>(500K-4000K 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 $2^{38}/2^{13} 4B = 2^{37} = 128GB!!$

**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 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)
- => 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

- \( L_1 \text{ cache size} < (\text{page-size} \times \text{associativity}) \)
- 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**!
- 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
  - Physical address
  - Page
  - Page fault
  - Page-fault handler
  - Page-out
- Cache address
- Cache location
- Huge cache block
- Extremely painful $miss
- The software filling the $
- Write-back if dirty
### Putting it all together

**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

### 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; 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 */

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

Can software help us?
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

Loop Interchange (for C)
/* Unoptimized */
for (j = 0; j < N; j = j + 1)
  for (i = 0; i < N; i = i + 1)
    x[i][j] = 2 * x[i][j];

/* Optimized */
for (i = 0; i < N; i = i + 1)
  for (j = 0; j < N; j = j + 1)
    x[i][j] = 2 * x[i][j];

(FORTRAN: The other way around!)
Merging arrays (if both members accessed at the same time)

/* Unoptimized */
int record[MAX]
int key[MAX]

/* Optimized */
struct merge {
    int record;
    int key;
};
struct merge merge_array[size];

Frequent member in one struct

/* Unoptimized*/
struct array_data {
    int a, b, c;
    int debug1, debug2, error;
};

/* Optimized*/
struct array_data_freq{
    int a, b, c;
};
struct array_data_infreq{
    int debug1, debug2, error;
};

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

01001000010001001001010101010101

and

index
(13)
(17)

i

1024

and

Select
(32)

Hit?
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;
  }

/* Optimized ARRAY: X = Y * Z */
for (jj = 0; jj < N; jj = jj + B) /* Loop 5 */
  for (kk = 0; kk < N; kk = kk + B) /* Loop 4 */
    for (i = 0; i < N; i = i + 1) /* Loop 3 */
      for (j = jj; j < min(jj+B,N); j = j + 1) /* Loop 2 */
        {r = 0;
         for (k = kk; k < min(kk+B,N); k = k + 1) /* Loop 1 */
           r = r + y[i][k] * z[k][j];
        x[i][j] += r;
      }

Partial solution
First block
Second block

Example in Class

/* Optimized ARRAY: X = Y * Z */
for (jj = 0; jj < N; jj = jj + B) /* Loop 5 */
  for (kk = 0; kk < N; kk = kk + B) /* Loop 4 */
    for (i = 0; i < N; i = i + 1) /* Loop 3 */
      for (j = jj; j < min(jj+B,N); j = j + 1) /* Loop 2 */
        {r = 0;
         for (k = kk; k < min(kk+B,N); k = k + 1) /* Loop 1 */
           r = r + y[i][k] * z[k][j];
        x[i][j] += r;
      }

Partial solution
First block
Second block

Second block
Prefetching

/* Unoptimized */
for (j = 0; j < N; j = j + 1)
    for (i = 0; i < N; i = i + 1)
        x[i][j] = 2 * x[i][j];

/* Optimized */
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N/4; j = j + 4)
        PREFETCH x[i][j+8]
        x[i][j] = 2 * x[i][j];

Cache Affinity

- Schedule the process on the processor it last ran
- Caches are warmed up ...

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

Lab1

- Compile and run programs in an architecture simulator modelling cache and memory
- Study performance when you:
  - change the cache model
  - change the program
StatCache - a locality tool for SW developers

Erik {Berg, Hagersten}
Uppsala University
Sweden

Caches – a huge cludge
++ hides latency, - - requires locality!

A = B + C:
Read B
Read C
Add B & C
Write A

Latency
0.3 -- 100 ns
0.3 -- 100 ns
0.3 ns
0.3 -- 100 ns

Traditional Simulation
Slowdown: ≈100x

CPU-sim
Level-1 Cache
Level-n Cache
Memory

Simulated CPU

Code:
set A,%r1
ld [%r1],%r0
st %r0,[%r1+8]
add %r1,1,%r1
ld [%r1+16],%r0
add %r0,%r5,%r5
st %r5,[%r1+8]

Memory ref:
1:read A
2:write B
3:read C
4:write B

Simulated Memory System

Hardware Counters
Slowdown: ≈ 0%

CPU
Level-1 Cache
Level-n Cache
Memory

Simulated Cache

•No flexibility
•Limited insight

Host Computer

Hardware stat

Simulated Memory System

Simulated Memory System
**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*

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

- repl = #cache misses since last touched
- L = #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 $R$)

$p_{\text{miss}} = f(5 \times R)$

# repl $\approx 3 \times Mr$

Tot_{misses} = R \times N = \sum_{i=0}^{N} f(d(i) \times R)$

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

$h(1)f(R) + h(2)f(2R) + h(3)f(3R) + ... \approx RN$

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)

# Samples

Reuse distance histogram:

Reuse Distance Histogram

Estimated by Sampling
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
\(\Sigma\)tatCach\(\varepsilon\): How accurate?

Results from a traditional Simulator

Why is speed so important?

Implementing \(\Sigma\)tatCach\(\varepsilon\)

Three steps:

1. Select samples: Overflow trap from HW perf.counter (DC\_rd)
2. Detect reuse: Solaris watchpoint support:
   \(\text{write("/proc/self/ctl", addr,$linesize)}\)
3. Measure reuse distance: Using the perf.counter again (DC\_rd)
StatCache Model

Slowdown: ≈ 30%

Modelling of many memory systems

Sample every 10^7-th

dist=5

dist=3

Level-1 Cache

Level-n Cache

Memory

CPU Mem ref:
1:read A
2:read B
3:read C
4:write C
5:read B
6:read D
7:read A
8:read E
9:read B

Spy prog

Mem

unmod binary

Host computer

New statistical cache model

ε

prototype

xterm

% cc hello.c

% a.out

Hello World
Using $\Sigma$tatCach$\varepsilon$

- Evaluate and compare optimizations
- Measure data locality: spatial and temporal
- Identify poor data structure layout and/or access patterns
- Locate code with poor cache behavior
- Workload characterization

Per-datastructure info: Multiplication: $X = Y \cdot Z$

Varying Cache Line Size

Spatial locality measurement

Spatial Locality for 32B @ 64kB
We introduce Spatial Use as a measure of Spatial Locality based on this difference
Equake Spatial Locality

Spatial Optimization of Equake

Unoptimized memory layout:
- Useful data
- Unused

Optimized memory layout:
- Useful data
- Unused

Equake: Spatial Locality