Welcome to DARK2
(IT, MN and PhD)

Erik Hagersten
Uppsala University

Literature
Computer Architecture A Quantitative Approach (3rd edition)

Lecturer
Erik Hagersten gives most lectures and is responsible for the course
Frédéric Haziza is responsible for the laborations and the hand-ins
Jakob Engblom guest lecturer in embedded systems
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 on the exam …

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

DARK2 On the web
www.it.uu.se/edu/course/homepage/dark2/ht05

DARK2, Autumn 2005
Welcome!
News
Forms
Schedule
Slides
Papers
Assignments
Reading instructions
Exam

DARK2 in a nutshell

1. Memory Systems
Caches, VM, DRAM, microbenchmarks, optimizing SW

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

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

4. Widening + Future
Technology impact, TLP+ILP in the CPU,…
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, ...}

...
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
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...)

Pipeline

Instructions:

```
D
C
B
A
```

CPU

Mem

Instructions:

```
I R X W
```

Mem

Regs
Pipeline system in the Book

Pipeline

Pipeline

Pipeline
Pipeline:

I = Instruction fetch  
R = Read register  
X = Execute  
W = Write register

Register Operations:
Add R1, R2, R3

Register Operations:
Add R1, R2, R3

Initially

DARK2 2005

Cycle 1
Cycle 2

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

Cycle 3

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

Cycle 4

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

Cycle 5

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

- 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

K+L START:

Issue Logic

Connecting to the Memory System

Data 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

Mem

250 cycles

1 GB

10 cycles

64kB

Fix: Use a cache
Webster about “cache”

1. cache \(\text{kash} \) 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

Address Book Cache

Looking for Tommy’s Telephone Number

<table>
<thead>
<tr>
<th>Indexing function</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Address Tag”</td>
</tr>
<tr>
<td>“Data”</td>
</tr>
</tbody>
</table>

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

Memory/storage

- sram
- dram
- disk

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

(1982: 200ns 200ns 10 000 000ns)
Address Book Cache
Looking for Tommy’s Number

OMMY 12345

EQ?

TOMMY

Miss!
Lookup Tomas’ number in the telephone directory

OMMY 12345

TOMAS

EQ?

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

TOMAS 23457

Address Book Cache
Looking for Tomas’ Number

OMMY 12345

TOMAS

Cache

address
data (a word)

data

CPU

Memory
Cache Organization

4kB, direct mapped

Valid (1)  = Hit (1) 
&  

Data (5 digits)  49

Ordinary Memory

Hit? (1)  Data  
&  

1k entries of 4 bytes each

Ordinary Memory

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

Mem Overhead: 21/32 = 66% 
Latency = SRAM+CMP+AND

What is a good index function

32 bit address identifying a byte in memory
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

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 * (
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
What if ...

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

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

• Twice as fast CPU  ==>  0.37 + 0.64 + 3.75 = 4.77

• Faster memory (70c) ==> 0.75 + 0.64 + 2.62 = 4.01

• Improve hit_rate (0.95) => 0.75 + 0.71 + 1.25 = 2.71

How to get more effective caches:

- Larger cache (more capacity)
- Cache block size (larger cache lines)
- More placement choice (more associativity)
- Innovative caches (victim, skewed, ...)
- Cache hierarchies (L1, L2, L3, CMR)
- Latency-hiding (weaker memory models)
- Latency-avoiding (prefetching)
- Cache 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 (imperfect cache implementation)

- (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.

New Indexing function

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

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, ...

temporal
spatial

Fetch more than a word:
cache blocks (a.k.a cache line)
1MB, direct mapped, CacheLine=16B

Mem
Overhead: 13/128 = 10%

Latency = SRAM+CMP+AND

Mem
Overhead: 13/32 = 40%

Latency = SRAM+CMP+AND
Example in Class
Direct mapped cache:

- Cache size = 64 kB
- Cache line = 16 B
- Word size = 4B

“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)

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

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

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”

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

Cache Model: Random vs. LRU
- Write-back
- Write-through
- A “dirty bit” indicates an altered cache line
- 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
- 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
**UART: Elbow cache**

Increase “associativity” when needed

If severe conflict: make room

Conflicts!!

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**

Topologies 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

---

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

The Endian Mess

Numbering the bytes
- Store the value 0x5F
- Store the string Hello
Virtual Memory System

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

Virtual and Physical Memory

Translation & Protection

Program

Physical Memory

Disk

Virtual Memory

Physical Memory

Disk

Virtual Memory

Disk
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>8-20 clock cycles</td>
<td>700K-60000K clock cycles</td>
</tr>
<tr>
<td>(Access time)</td>
<td>(6-20 clock cycles)</td>
<td>(500K-4000K clock cycles)</td>
</tr>
<tr>
<td>(Transfer time)</td>
<td>(2-40 clock cycles)</td>
<td>(200K-2000K clock cycles)</td>
</tr>
<tr>
<td>Miss rate</td>
<td>0.5%-10%</td>
<td>0.00001%-0.001%</td>
</tr>
<tr>
<td>Data memory size</td>
<td>16 Kbyte - 1 Mbyte</td>
<td>16 Mbyte - 8 Gbyte</td>
</tr>
</tbody>
</table>

VM: Block placement

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

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

VM: Block identification

Use a page table stored in main

- 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 Look-aside Buffer** (TLB)

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 (VirtuallyIndexedPhysicallyTagged) is the answer
  - Direct-mapped cache no larger than a page
  - No more sets than there are cache lines on a page + logic
  - Page coloring can be used to guarantee correspondence between more PA and VA bits (e.g., Sun Microsystems)
Virtually Indexed Physically Tagged = VIPT

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

What is the capacity of the TLB

Typical TLB size = 0.5 - 2kB
Each translation entry 4 - 8B ==> 32 - 500 entries
Typical page size = 4kB - 16kB
**TLB-reach** = 0.1MB - 8MB

**FIX:**
- Multiple page sizes, e.g., 8kB and 8 MB
- TSB -- A direct-mapped translation in memory as a "second-level TLB"

VM: Page replacement

Most important: *minimize number of page faults*

Page replacement strategies:
- FIFO—First-In-First-Out
- LRU—Least Recently Used
- Approximation to LRU
  - Each page has a *reference bit* that is set on a reference
  - The OS periodically resets the reference bits
  - When a page is replaced, a page with a reference bit that is not set is chosen

So far...
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>Virtual Memory System</th>
<th>The “cache” language</th>
</tr>
</thead>
<tbody>
<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>

Putting it all together
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
- ...

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

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; 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 */

Micro Benchmark Signature

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

Micro Benchmark Signature

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

SGI Challenge Memory System Performance

```
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
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
- ...

Merging arrays (align)

/* Unoptimized */
int info1[MAX]
int info2[MAX]
int key[MAX]

/* Optimized */
struct merged_a {
    align ...
    int key, info1, info2;
};
struct merged_a merge_array[size];

/* AltOptimized */
struct merged_a {
    int key, info1, info2, dummy;
};
struct merge merge_array[size];
**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**

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

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

**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 (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**

Padding examples:

1. A+1024*4
2. A+2048*4

Diagram:

- Index
- Data
- Multiplication (2:1 ratio)
- Logic
- Hit/Select
- Output
Padding of data structures

allocate more memory than needed

allocate more memory than needed

Example in Class

How many misses assuming a cache size $S$, such that:

$2 \times \text{sizeof(data)} \times N < S$

and

$\text{sizeof(data)} \times N \times N > S$

Blocking

/* 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)

$r = r + y[i][k] \times z[k][j];$

$x[i][j] = r;$

};

j

First block

Second block

/* 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)$

$r = r + y[i][k] \times z[k][j];$

$x[i][j] += r;$

};

j

Partial solution
Example in Class

- Pick a good value for B
- How many misses?

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

ΣtatCachε
a locality tool for SW developers
Erik {Berg, Hagersten}
Uppsala University
Sweden

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

Sloooow Memory

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

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

Traditional Simulation
Slowdown: ≈100x

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

Code:
set A,%r1
ld [%r1],%r0
st %r0,[%r1+8]
add %r1,1,%r1
ld [%r1+16],%r0
add %r0,5,%r5
st %r5,[%r1+8]
[...]

Memory ref:
1:read A
2:write B
3:read C
4:write B
[...]
Hardware Counters

timeout: \approx 0\%

- No flexibility
- Limited insight

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{hardware_counters.png}
\caption{Hardware Counters Diagram}
\end{figure}

Statistical Cache Model

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

- \(\text{hit}(\text{repl}) = (1 - 1/L)^{\text{repl}}\)
- \(\text{repl} = \#\text{cache misses since last touched}\)
- \(L = \#\text{cache lines in the cache}\)

Modelling of many memory systems

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{statistical_cache_model.png}
\caption{Statistical Cache Model Diagram}
\end{figure}

Hit function

\textit{Fully assoc, random replacement}

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

Miss probability: \( f(n) = 1 - (1 - 1/L)^n \)
\( n = \text{#cache misses since last touched} \)
\( L = \text{#cache lines in the cache} \)

\[ f(n) = \text{miss probability} \]

After \( L \) cache misses

Probabilistic Cache Model

(assumpt: “const” miss rate MR)

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

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

\[ \# \text{Samples} \]

Solve for \( R \) to get miss ratio.
(With numerical method)
Reuse Distance Histogram
Estimated by Sampling

Probabilistic Cache Model

Hit ratio (%)
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:
  \[ h(1)f(R) + h(2)f(2R) + h(3)f(3R) + ... = RN \]
- Take average miss ratio of all time slots

\[ \sum_{i} \text{tatCach} \epsilon \]

How accurate?

Results from a traditional Simulator

Why is speed so important?

Implementing StatCacheε

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:
   \texttt{write(”/proc/self/ctl”, addr,$linesize)}
3 Measure reuse distance: Using the perf.counter again (DC_rd)

ΣstatCacheε Model

Slowdown: \(\approx 30\%\)

- Modelling of many memory systems
- New statistical cache model
- Sample every 10\(^7\):th
- \(5, 3, \ldots\)

ΣstatCacheε prototype

- Example output of the program
- Sample graph showing cache utilization
Using StatCache

- 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=YZ

Total blocksize = 60

Total blocksize = 20

Varying Cache Line Size
Spatial locality measurement

We introduce Spatial Use as a measure of Spatial Locality based on this difference.

Spatial Optimization of Equake

Unoptimized memory layout:

- Useful data
- Unused

Optimized memory layout:

- Useful data
- Unused

Equake: Spatial Locality

Graph showing Spatial Locality for different cache line sizes.

Equake Spatial Locality

Graph showing Spatial Locality for Equake.
New stuff:
- Cold misses
- Footprint
- Monitor multiple threads in shared memory
- Model shared caches

Future:
- Stabilize tool ➔ open source
- Model limited associativity
http://www.it.uu.se/research/group/uart