Welcome to DARK2
(IT, MN and PhD)

Erik Hagersten
Uppsala University

Welcome to DARK2 On the web
www.it.uu.se/edu/course/homepage/dark2/2003ht

DARK2 in a nutshell
1. Memory Systems (caches, VM, DRAM, microbenchmarks, ...)
2. CPUs (ILP: pipelines, scheduling, superscalars, VLIWs, embedded, ...)
3. Multiprocessors (TLP: coherence, interconnects, scalability, clusters, ...)
4. Future: (physical limitations, TLP+ILP in the CPU,...)

Literature

Lecturer
Erik Hagersten gives most lectures and is responsible for the course
Håkan Zeffer is responsible for the laboratories 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. If you are following the MN2 version of the course you’ll also have to write a "Microprocessor Report" paper about a specific computer system and/or CPU chip.

Optional Assignment
There are three (optional) hand-in assignments: Memory, CPU, Multiprocessors. You will get extra credits on the exam for each hand-in that is completed (with a reasonable accuracy) before the announced deadlines. If you complete them all you will be guaranteed 24p out of 64p. 32p is needed for passing.

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

DARK2, Autumn 2003
Welcome!
News
Forms
Schedule
Slides
Papers
Assignments
Reading instructions
Exam

(Links
www.it.uu.se/edu/course/homepage/dark2/2003ht)
**Introduction to Computer Architecture**

Erik Hagersten
Uppsala University

---

**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

---

“Only” 20 years ago: APZ 212
“the AXE supercomputer”
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

How long is a CPU cycle?

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

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

Instructions:

- D
- C
- B
- A

CPU

Mem

Pipeline

Instructions:

- D
- C
- B
- A

I

R

X

W

Regs

Mem

Pipeline

I

R

X

W

Regs

Mem

Pipeline

I

R

X

W

Regs

Mem
Pipeline:

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

Register Operations:
Add R1, R2, R3

Initially

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

PC
Cycle 1

LD RegA, (100 + RegC)

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

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)
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)

Cycle 7

PC

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

Branch ➔ Next PC

Cycle 8

PC

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

Data dependency

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

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

Fix: Use a cache

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

Webster about “cache”

1. cache 
   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

Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se
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>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>1 ns</td>
<td>1kB</td>
</tr>
<tr>
<td>DRAM</td>
<td>10 ns</td>
<td>64k</td>
</tr>
<tr>
<td>Disk</td>
<td>150 ns</td>
<td>4MB</td>
</tr>
<tr>
<td>(1982)</td>
<td>200ns</td>
<td>10kB</td>
</tr>
</tbody>
</table>

Address Book Cache
Looking for Tommy's Telephone Number

“Address Tag”
“Data”

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

Miss!
Lookup Tomas’ number in the telephone directory

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

Cache

Cache Organization

CPU

Cache

Memory

Valid (1)
Hit (1)

Data (5 digits)

OAS

TOMAS

EQ?

OMMY 12345

OMMY

TOMAS

index

(1)

(4)

(1)

(4)

(1)

(1)

TOMAS

23457

OMAS

index

Hit (1)
Cache Organization (really)
4kB, direct mapped

Ordinary Memory

What is a good index function?

32 bit address identifying a byte in memory

1k entries of 4 bytes each

Hit? (1)

Valid (1)

Data (1)

Addr & (1)

Hit? (32)

Data (32)

Ordinary Memory

Mem Overhead: 21/32 = 66%

Latency = SRAM + CMP + AND

Cache

CPU → address → Cache

Hit: Use the data provided from the cache

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

Cache performance parameters

- Cache “hit rate” [%]
- Cache “miss rate” [%] (= 1 - hit_rate)
- Hit time [CPU cycles]
- Miss time [CPU cycles]
- Hit bandwidth
- Miss bandwidth
- Write strategy
- ....
How to rate architecture performance?

Marketing
- Frequency

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

\[
\text{CPI} = \frac{\text{NumberOfExecutionCycles}}{\#Instructions} =
= \frac{1 - \text{mem_ratio} + \text{mem_ratio} \times (\text{hit_rate} \times \text{hit_time}) +
(1 - \text{hit_rate}) \times \text{miss_time}}{\#Instructions}
\]

Example Numbers

\[
\text{CPI} = 1 - \text{mem_ratio} +
\left[ \begin{array}{c}
\text{mem_ratio} \times (\text{hit_rate} \times \text{hit_time}) + \\
\text{mem_ratio} \times (1 - \text{hit_rate}) \times \text{miss_time}
\end{array} \right]
\]

\[
\begin{align*}
\text{mem_ratio} &= 0.25 \\
\text{hit_rate} &= 0.85 \\
\text{hit_time} &= 3 \\
\text{miss_time} &= 100
\end{align*}
\]

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

What if ...

\[
\text{CPI} = 1 - \text{mem_ratio} +
\left[ \begin{array}{c}
\text{mem_ratio} \times (\text{hit_rate} \times \text{hit_time}) + \\
\text{mem_ratio} \times (1 - \text{hit_rate}) \times \text{miss_time}
\end{array} \right]
\]

\[
\begin{align*}
\text{mem_ratio} &= 0.25 \\
\text{hit_rate} &= 0.85 \\
\text{hit_time} &= 3 \\
\text{miss_time} &= 100
\end{align*}
\]

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

- Twice as fast CPU \( \Rightarrow 0.37 + 0.64 + 3.75 = 4.77 \)
- Faster memory (70c) \( \Rightarrow 0.75 + 0.64 + 2.62 = 4.01 \)
- Improve hit_rate (0.95) \( \Rightarrow 0.75 + 0.71 + 1.25 = 2.71 \)
How to get more effective caches:

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

Why do you miss in a cache

- Mark Hill’s three “Cs”
  - Compulsory miss (touching data for the first time)
  - Capacity miss (the cache is too small)
  - Conflict misses (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.

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

Mem Overhead: 13/32 = 40%

Cache Organization

1MB, direct mapped

32 bit address

Mem

Overhead:

256k entries

Latency = SRAM+CMP+AND

Identifies the byte within a word

Hit?

Data
**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, ...

---

**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: First index -- then search.

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

Latency = SRAM+CMP+AND+LOGIC+MUX

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

Hit? How should the select signal be produced?

Index

EQ?

EQ?
A combination thereof
1MB, 2-way, CL=16B

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

001001100001010010100110101000110
32k “sets”
(13) (13)
(15) index
msb lsb
(128) (128)
(15)
(13)
&
&
(2) Select
(32) Data
(256)

Multiplexer
(8:1 mux)

“logic”
Hit?

Least-recently used
- Considered the best algorithm
- Only practical up to 4-way (16 bits/CL)
- Not most recently used
  - Remember who used it last: 8-way -> 3 bits/CL
- Pseudo-LRU
  - Course Time stamps, used in the VM system
- Random replacement
  - Can’t continuously to have “bad luck...

Who to replace?
Picking a “victim”

UART research:
Data for a fully associative cache

Locality Analysis

Focusing on Datastruct X

Thanks: Erik Berg
4-way sub-blocked cache
1MB, direct mapped, Block=64B, sub-block=16B

Mem
Overhead:
16/512= 3%

Pros/Cons Sub-blocking
++ Needs much less address tags
++ Avoids problems with false sharing
++ Avoids problems with bandwidth waist
-- Will not explore as much spatial locality
-- Still poor utilization of SRAM
-- Fewer sparse “things”

Replacing dirty cache lines
- Write-back
  - A “dirty bit” indicates an altered cache line
  - Write dirty data back to memory (next level) at replacement
- Write-through
  - Always write through to the next level (as well)
  - Never dirty data to write back

Write Buffer/Store Buffer
- Do not need the old value for a store
- Write-through...
- Write around (no write allocate)
**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

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

32 bit address

Index

128k entries

UART: Elbow cache

Increase “associativity” when needed

If severe conflict: make room

Performs roughly the same as an 8-way cache

Slightly faster

Uses much less power!!
Cache Hierarchy Latency

300:1 between on-chip SRAM - DRAM ➔ cache hierarchies

- **L1**: small on-chip cache
  - Runs in tandem with pipeline ➔ small
  - VIPT caches adds constraints (more later...)
- **L2**: large SRAM on-chip
  - Communication latency becomes more important
- **L3**: Off-chip SRAM
  - Huge cache ~10x faster than DRAM

Cache Hierarchy

- **CPU**
  - **L1**: on-chip
  - **L2**: on-module
  - **L3**: on-board

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:
- **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**

- 4Mbit memory array
- One bit memory cell

- The address is multiplexed Row/Address Strobe (RAS/CAS)
- Thin organizations x4 or 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
- Refresh of memory cells decreases bandwidth
- Bit-error rate creates a need for error-correction (ECC)

**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
  - Mem controller provides strobe for next seq. access
- DDR-DRAM
  - 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)...?

Physical memory, little endian

Virtual Memory System

Erik Hagersten
Uppsala University, Sweden

 eh@it.uu.se
**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 takes $2^{48}/2^{13} \times 4B = 2^{37} = 128$ Gbyte!!!

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.

  - **kseg**
    - Used by OS.
    - Does not use virtual memory.
  - **seg1**
    - Used for stack.
  - **seg0**
    - 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 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 do 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

- 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

Virtual Memory System
- The “cache” language
- Virtual address
- Physical address
- Page
- Page fault
- Page-fault handler
- Page-out
- Write-back if dirty

Putting it all together

[Diagram showing CPU, TLB, L1 cache, L2 cache, and memory]

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
- ...

How are we doing?

Creating and exploring:
1) Locality
   - Spatial locality
   - Temporal locality
   - Geographical locality
2) Parallelism
   - Instruction level
   - 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
### 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)

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

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

### Merging arrays

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

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

### Loop Interchange (for C)

```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!
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];
    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;

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

/* Optimized */
for (i = 0; i < N/4; i = i + 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 ...

UART research: SIP

- “Source-Interdependence Profiler”
- The compiler is expected to make most optimizations
- Each performance problem is related to a combination of instructions
- SIP: Point out where the cache is poorly used
- SIP results: 60% speedup of SPEC2000 equake

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

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 */

<table>
<thead>
<tr>
<th>Stride(bytes)</th>
<th>Array Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>16 kB</td>
</tr>
<tr>
<td>16</td>
<td>32 kB</td>
</tr>
<tr>
<td>64</td>
<td>128 kB</td>
</tr>
<tr>
<td>256</td>
<td>512 kB</td>
</tr>
<tr>
<td>1K</td>
<td>1 MB</td>
</tr>
<tr>
<td>4K</td>
<td>4 MB</td>
</tr>
<tr>
<td>16K</td>
<td>16 MB</td>
</tr>
<tr>
<td>64K</td>
<td>64 MB</td>
</tr>
<tr>
<td>256K</td>
<td>256 MB</td>
</tr>
<tr>
<td>1M</td>
<td>1 MB</td>
</tr>
<tr>
<td>4M</td>
<td>4 MB</td>
</tr>
<tr>
<td>16M</td>
<td>16 MB</td>
</tr>
<tr>
<td>64M</td>
<td>64 MB</td>
</tr>
<tr>
<td>256M</td>
<td>256 MB</td>
</tr>
<tr>
<td>1GB</td>
<td>1 GB</td>
</tr>
</tbody>
</table>

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**

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

**Twice as large L2 cache...**

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

**Twice as large TLB...**

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

**SIG Challenge Memory System Performance**

```c
for (times = 0; times < Max; time++) /* many times*/
    for (i=0; i < ArraySize; i = i + Stride)
        dummy = A[i]; /* touch an item in the array */
```
Lab 1

- Run programs in a architecture simulator and measure the hit rate
  - Change the cache model
  - Change the program

Lab 2

- Write your own microbenchmark and check out your favorite computer’s memory system.