Welcome to AVDARK

Erik Hagersten
Uppsala University
Ericsson, CPU designer 1982-84: APZ212
MIT 1984-85: Dataflow parallel architecture
Ericsson computer science lab 1985-1988: NetInsight, (Erlang)
SICS, parallel architectures 1988-1993: COMA, (Virtutech)
Professor Uppsala University 1999 – New modeling + Acumem
Startup Acumem 2006 – 2010: ThreadSpotter
Chief scientist Rogue Wave Software 2011 –
Goal for this course

- Understand **how and why** modern computer systems are designed the way they are:
  - pipelines
  - memory organization
  - virtual/physical memory ...

- Understand **how and why** multiprocessors are built
  - Cache coherence
  - Memory models
  - Synchronization...

- Understand **how and why** parallelism is created and
  - Instruction-level parallelism
  - Memory-level parallelism
  - Thread-level parallelism...

- Understand **how and why** multiprocessors of combined SIMD/MIMD type are built
  - GPU
  - Vector processing...

- Understand **how** computer systems are adopted to different usage areas
  - General-purpose processors
  - Embedded/network processors...

- Understand the physical limitation of modern computers
  - Bandwidth
  - Energy
  - Cooling...
Welcome!
News
FAQ
Schedule
Slides
New Papers
Assignments
Reading instr 4:ed
Exam
AVDARK in a nutshell

**Literature**  
John Hennesey & David Pattersson

**Lecturer**  
Erik Hagersten gives most lectures and is responsible for the course.  
Andreas Sandberg is responsible for the labs and the hand-ins.  
Jakob Carlström from Xelerated will teach network processors.  
Sverker Holmgren will teach parallel programming.  
David Black-Schaffer will teach about graphics processors.

**Mandatory Assignment**  
There are four lab assignments that all participants have to complete before a hard deadline. Each can earn you a bonus point.

**Optional Assignment**  
There are four (optional) hand-in assignments. Each can earn you a bonus point.

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

**Bonus system**  
64p max/32p to pass. For each bonus point, there is a corresponding question 4p bonus question. Full bonus ➔ Pass.
Schedule in a nutshell

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

2. Multiprocessors
   TLP: coherence, memory models, interconnects, scalability, clusters, ...

3. Scalable Multiprocessors
   Scalability, synchronization, clusters, ...

4. CPUs
   ILP: pipelines, scheduling, superscalars, VLIWs, Vector instructions...

5. Widening + Future (~Chapter 1 in 4th Ed)
   Technology impact, GPUs, Network processors, Multicores (!!!)
Exam and bonus structure

- 4 Mandatory labs
- 4 Hand-in (optional)
- Written Exam

How to get a bonus point:
- Complete extra bonus activity at lab occasion
- Complete optional bonus hand-in [with a reasonable accuracy] before a hard deadline

⇒ 32p/64p at the exam = PASS
Crash Course in Computer Architecture
(covering the course in 45 min)

Erik Hagersten
Uppsala University
≈30 years ago: APZ 212 @ 5MHz
"the AXE supercomputer"
APZ 212
marketing brochure quotes:

- "Very compact"
  - 6 times the performance
  - 1/6:th the size
  - 1/5 the power consumption
- "A breakthrough in computer science"
- "Why more CPU power?"
- "All the power needed for future development"
- "...800,000 BHCA, should that ever be needed"
- "SPC computer science at its most elegance"
- "Using 64 kbit memory chips"
- "1500W power consumption"
CPU Improvements
Relative Performance
[log scale]

Historical rate: 55%/year
How to get efficient architectures...

- Higher clock rate
- Create and explore locality:
  a) Spatial locality
  b) Temporal locality
  c) Geographical locality
- Create and explore parallelism
  a) Instruction level parallelism (ILP)
  b) Thread level parallelism (TLP)
  c) Memory level parallelism (MLP)
Load/Store architecture (e.g., "RISC")

ALU ops: Reg --> Reg
Mem ops: Reg <--> Mem

Example: C = A + B

Compiler

Load R1, [A]
Load R3, [B]
Add R2, R1, R3
Store R2, [C]
Lifting the CPU hood (simplified...)

Instructions:

```
  B
 / 
A
```

CPU

Mem
Pipeline

Instructions:

\[
\begin{array}{cccc}
D & C & B & A \\
\end{array}
\]

\[
\begin{array}{cccc}
I & R & X & W \\
\end{array}
\]

Regs

\[
\begin{array}{cccc}
\end{array}
\]

Mem
Pipeline
Pipeline
Pipeline
Pipeline:

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

Regs

Mem
Example ALU operation:

R1 := R2 op R3
(ADD R1, R2, R3)

Ifetch

OP
e.g., +, -, *, /

I = Instruction fetch
R = Read register
X = Execute
W = Write register/mem
Initially

IF RegC < 100 GOTO A

RegC := RegC + 1

RegB := RegA + 1

LD RegA, (100 + RegC)
Cycle 1

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

PC →

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
Cycle 2

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
Cycle 3

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
Cycle 4

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
Cycle 5

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
Data dependency 😞
Previous execution example wrong!

IF RegC < 100 GOTO A
RegB := RegC + 1
RegC := RegC + 1
LD RegA, (100 + RegC)
Data dependency fix: pipeline delays

IF RegC < 100 GOTO A

"Stall"

"Stall"

RegC := RegC + 1

RegB := RegA + 1

"Stall"

"Stall"

LD RegA, (100 + RegC)
Branch Delay: Each Iteration takes 11c 😞

PC →

“Stall”

“Stall”

“Stall”

IF RegC < 100 GOTO A

“Stall”

“Stall”

RegC := RegC + 1

RegB := RegA + 1

“Stall”

“Stall”

LD RegA, (100 + RegC)

Need to find Instruction-Level Parallelism (ILP) to avoid stalls!
It is actually a lot worse!

Modern CPUs: “superscalars” with ~4 parallel pipelines

- Higher throughput
- More complicated architecture
- Branch delay more expensive (more instr. missed)
- Harder to find “enough” independent instr. (need 8 instr. between write and use)
It is actually a lot worse:
Modern CPUs: ~10-20 pipeline stages

More pipeline stages ➔ can run at higher freq 😊 But will need to find ~10x more ILP 😞

+ Shorter cycletime (higher GHz)
- Branch delay even more expensive
- Even harder to find "enough" independent instr.
Fix: Out-of order execution: Improving ILP

The HW may execute instructions in a different order, but will make the “side-effects” of the instructions appear in order.

Assume that LD takes a long time. The ADD is dependent on the LD ⊗ Start the SUB and ST before the ADD Update R5 and M(100) after R3
Fix: Branch prediction

The HW can guess if the branch is taken or not and avoid branch stalls if the guess is correct.

Assume the guess is “Y”.

The HW can start to execute these instruction before the outcome the branch is known, but cannot allow any “side-effect” to take place until the outcome is known.
Fix: Scheduling Past Branches

Improving ILP

All instructions along the predicted path can be executed out-of-order

"Predict taken"
Fix: Scheduling Past Branches
Improving ILP

Actual path!

Wrong Prediction!!!

Throw away i.e., no side effects!

=0?

>=0?

LD ADD SUBST

>1?

LD ADD SUBST

<2?

LD ADD SUBST

Y

Y

Y

Y
It is actually a lot worse:
Modern MEM: ~200 CPU cycles

Need more ILP again!

200 cycles 😞
Fix: Use cache(s)

Still need some more ILP, but also Memory-Level Parallelism (MLP)

1-30 cycles

< < 1 GB

200 cycles

Mem

1 GB
Woops, using too much power 2007

- Running at 2x the frequency will use much more than 2x the power

- It is also really hard to find enough ILP
But now we also need to find Thread-Level Parallelism (TLP)
Example: Intel i7 “Nehalem”
How is the silicon used (i7-Ex)?

Source: JSSC Jan 2010, Rusu et. al
How is the silicon used?

Quick Path Interconnect

Memory Interface

Source: JSSC Jan 2010, Rusu et. al
Example: Intel i7 "Nehalem"

Coherence = NUMA + funky memory models
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, ...}
Caches and more caches

or

spam, spam, spam and spam

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

~1 cycles

~32kB

200 cycles

1 GB

Regs

B M M M W

I

R

I

R

I

R

I

R

B M M M W

B M M M W

B M M M W

B M M M W
Webster about “cache”

1. cache "kash" n [F, fr. cacher to press, hide, fr. (assumed) VL coactivare 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

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

(1982: 200ns 200ns 1 ns 3ns 10ns 10 000 000ns)
Address Book Cache

Looking for Tommy’s Telephone Number

“Address Tag”

“Data”

Indexing function

One entry per page =>

Direct-mapped caches with 28 entries
Address Book Cache

Looking for Tommy’s Number

TOMMY

EQ?

index

OMMY 12345

T U V W X Y Z Ä Ö
Address Book Cache
Looking for Tomas’ Number

Miss!
Lookup Tomas’ number in the telephone directory
Address Book Cache
Looking for Tomas’ Number

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

CPU

address

Memory

Cache

hit

data (a word)

data
Cache Organization

Cache

TOMAS

OMAS 23457

Index (1)

Valid (1)

Hit (1)

Data (5 digits)

28 entries

Upptäck av Erik Hagersten | http://user.it.uu.se/~eh
Cache Organization (really)

4kB, direct mapped

Ordinary Memory

1k entries of 4 bytes each

32 bit address identifying a byte in memory

What is a good index function

Valid (1)

& (1)

Hit? (1)

Data (32)

Dept of Information Technology| www.it.uu.se © Erik Hagersten| http://user.it.uu.se/~eh
Cache Organization
4kB, direct mapped

32 bit address

Identifies the byte within a word

1k entries of 4 bytes each

Mem Overhead: 21/32 = 66%

Latency = SRAM+CMP+AND

Dept of Information Technology | www.it.uu.se © Erik Hagersten | http://user.it.uu.se/~eh
Cache

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

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

Marketing:
- Frequency / Number of cores...

Architecture “goodness”:
- CPI = Cycles Per Instruction
- IPC = Instructions Per Cycle

Benchmarking:
- SPEC-fp, SPEC-int, ...
- TPC-C, TPC-D, ...
Cache performance example

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

**Total number of cycles** =
#Instr. * ( (1 - mem\_ratio) * 1 +
mem\_ratio * avg\_mem\_accesstime) =

= #Instr * ( (1 - mem\_ratio) +
mem\_ratio * (hit\_rate * hit\_time +
(1 - hit\_rate) * miss\_time)

**CPI** = 1 -mem\_ratio +
mem\_ratio * (hit\_rate * hit\_time +
(1 - hit\_rate) * miss\_time)
Example Numbers

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

\[
\begin{array}{l}
\text{mem\_ratio} = 0.25 \\
\text{hit\_rate} = 0.85 \\
\text{hit\_time} = 3 \\
\text{miss\_time} = 100
\end{array}
\]

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

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

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

CPU HIT MISS

\[
\begin{align*}
0.75 & + 0.64 & + 3.75 = 5.14 \\
0.37 & + 0.64 & + 3.75 = 4.77 \\
0.75 & + 0.64 & + 2.62 = 4.01 \\
0.75 & + 0.71 & + 1.25 = 2.71 \\
\end{align*}
\]
How to get more effective caches:

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

- Mark Hill’s three “Cs”
  - Compulsory miss (touching data for the first time)
  - Capacity miss (the cache is too small)
  - Conflict misses (non-ideal cache implementation)
    (too many names starting with “H”)

- (Multiprocessors)
  - Communication (imposed by communication)
  - False sharing (side-effect from large cache blocks)
Avoiding Capacity Misses –
a huge address book
Lots of pages. One entry per page.

One entry per page =>
Direct-mapped caches with $28^2$ entries $\rightarrow$ 784 entries
Cache Organization
1MB, direct mapped

32 bit address

Identifies the byte within a word

Mem Overhead: 13/32 = 40%

Latency = SRAM+CMP+AND

256k entries
Pros/Cons Large Caches

++ The safest way to get improved hit rate
-- SRAMs are very expensive!!
-- Larger size ==> slower speed
  more load on “signals”
  longer distances
-- (power consumption)
-- (reliability)
Why do you hit in a cache?

- Temporal locality
  - Likely to access the same data again soon

- Spatial locality
  - Likely to access nearby data again soon

**Typical access pattern:**
(in 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 in the world: 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)
Small cache: Short cache lines are better

Large caches: Longer cache lines are better

Huge caches: Everything fits regardless of CL size

Note: this is just a single example, but the conclusion typically holds for most applications.

Thanks: Dr. Erik Berg
Cache Conflicts

Typical access pattern:
(inner loop stepping through an array)
A, B, C, A+1, B, C, A+2, B, C, ...

What if B and C index to the same cache location
Conflict misses -- big time!
Potential performance loss 10-100x
Address Book Cache
Two names per page: index first, then search.

TOMMY

EQ?

OMMY

EQ?

OMAS

T

23457

UV

X

Y

Z

Ä

Å

EQ?

2011
Avoiding conflict: More associativity
1MB, 2-way set-associative, CL=4B

Identifies a byte within a word

Latency = SRAM+CMP+AND+LOGIC+MUX

How should the select signal be produced?
Pros/Cons Associativity

++ Avoids conflict misses
-- Slower access time
-- More complex implementation comparators, muxes, ...
-- Requires more pins (for external SRAM...)
Going all the way...!
1MB, fully associative, CL=16B

Identifies the word within a cache line

Identifies a byte within a word

One “set”

64k comparators

Multiplexer
(256k:1 mux)

“logic”

Select (16)

Hit?

Data

4B

© Erik Hagersten | http://user.it.uu.se/~eh
Fully Associative

- Very expensive
- Only used for small caches (and sometimes TLBs)

CAM = Contents-addressable memory

- ~Fully-associative cache storing key+data
- Provide key to CAM and get the associated data
A combination thereof
1MB, 2-way, CL=16B

Identifies the word within a cache line

Identifies a byte within a word

msb

lsb

001001100001010010100110101000110
(15)

(13)

index

(13)

=  

=  

&  

&  

“logic”

Hit?

(1)

Select

(2)

Multiplexer (8:1 mux)

(32)

Data

(256)

(128)

(128)

32k “sets”

Dept of Information Technology | www.it.uu.se © Erik Hagersten | http://user.it.uu.se/~eh
Example in Class

- Cache size = 2 MB
- Cache line = 64 B
- Word size = 8B (64 bits)
- 4-way set associative
- 32 bits address (byte addressable)
Who to replace?

Picking a “victim”

- Least-recently used (aka LRU)
  - Considered the “best” algorithm (which is not always true...)
  - Only practical up to limited number of ways

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

- Pseudo-LRU
  - E.g., 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 (SPEC 2000)  
equake (SPEC 2000)
4-way sub-blocked cache
1MB, direct mapped, Block=64B, sub-block=16B

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

Mem Overhead: 16/512 = 3%
Pros/Cons Sub-blocking

++ Lowers the memory overhead
++ (Avoids problems with false sharing -- MP)
++ Avoids problems with bandwidth waste
-- Will not explore as much spatial locality
-- Still poor utilization of SRAM
-- Fewer sparse “things” allocated
Replacing dirty cache lines

- **Write-back**
  - Write dirty data back to memory (next level) at replacement
  - A “dirty bit” indicates an altered cache line

- **Write-through**
  - Always write through to the next level (as well)
  - data will never be dirty ➔ no write-backs
Write Buffer/Store Buffer

- Do not need the old value for a store

- One option: Write around (no write allocate in caches) used for lower level smaller caches
Victim Cache (VC): a small, fairly associative cache (~10s of entries)

Lookup: search cache and VC in parallel

Cache replacement: move victim to the VC and replace in VC

VC hit: swap VC data with the corresponding data in Cache

“A second life 😊”
Skewed Associative Cache
A, B and C have a three-way conflict

It has been shown that 2-way skewed performs roughly the same as 4-way caches
Skewed-associative cache: Different indexing functions

32 bit address

Identifies the byte within a word

128k entries

function

2:1mux

(32)

(32)

(32)

Dept of Information Technology | www.it.uu.se © Erik Hagersten | http://user.it.uu.se/~eh
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!!
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
Cache Hierarchy of Today

- DRAM Memory
- Off-chip SRAM (today more commonly on-chip)
- Use whatever transistors there i left for the Last-Level Cache (LLC)
- Small enough to keep up with the CPU speed
- Separate I cache to allow for instruction fetch and data fetch in parallel
HW prefetching

...a little green man that anticipates your next memory access and prefetches the data to the cache.

Implements MLP!

- Sequential prefetching: Sequential streams [to a page]. Some number of prefetch streams supported. Often only for L2 and L3.
- PC-based prefetching: Detects strides from the same PC. Often also for L1.
- Adjacent prefetching: On a miss, also bring in the “next” cache line. Often only for L2 and L3.
Hardware prefetching

- Hardware “monitor” looking for patterns in memory accesses
- Brings data of anticipated future accesses into the cache prior to their usage
- Two major types:
  - Sequential prefetching (typically page-based, 2nd level cache and higher). Detects sequential cache lines missing in the cache.
  - PC-based prefetching, integrated with the pipeline. Finds per-PC strides. Can find more complicated patterns.
Cache Capacity/Latency/BW
Cache implementation

Caches at all level roughly work like this:

- **L3**: 24MB
- **L2**: 256kB
- **D1**: 64kB
- **I1**: 64kB

**Generic Cache:**

Input address: `Addr [63..0]`

Output:

- **Hit Select way "6"**
- **SRAM:**

**Cacheline, here 64B:**

- **AT**
- **S**
- **Data = 64B**

Data = 64B
Address Book Analogy
Two names per page: index first, then search.

TOMMY

OMAS 23457
OMMY 12345

Select the second entry!
Cache lingo

Cacheline: Data chunk move to/from a cache
Cache set: Fraction of the cache identified by the index
Associativity: Number of alternative storage places for a cacheline
Replacement policy: picking the victim to throw out from a set (LRU/Random/Nehalem)
Temporal locality: Likelihood to access the same data again soon
Spatial locality: Likelihood to access nearby data again soon

Typical access pattern:
(inner loop stepping through an array)
A, B, C, A+4, B, C, A+8, B, C, ...

Dept of Information Technology | www.it.uu.se © Erik Hagersten | http://user.it.uu.se/~eh
Cache Lingo Picture

Cacheline, here 64B:

AT S Data = 64B

Generic Cache:

Addr [63..0]

Cache Set:

SRAM:

Associativity = 8-way

Hit Sel way "6"

Data = 64B
Exemple Nehalem i7 (one example)

- Core
- L1$ 32kB 8-way pLRU
- L2$ 256kB 8-way pLRU non-incl
- L3$ 8MB 16-way Neh.-repl non-incl
- DRAM
- Latency (cycles)
Take-away message: Caches

- Cache are fast but small
- Cache space in cache-line chunks (~64 bytes)
- LSB part of the address is used to find the "set" (aka, indexing)
- There is a limited number of cache lines per set (associativity)
- Typically, several levels of caches
- The most important target for optimizations
How are we doing?

- Create and explore locality:
  a) Spatial locality
  b) Temporal locality
  c) Geographical locality

- Create and explore parallelism
  a) Instruction level parallelism (ILP)
  b) Thread level parallelism (TLP)
  c) Memory level parallelism (MLP)