Exercise Session 1

Data Processing on Modern Hardware
263-3502-00L — Fall Semester 2012

Cagri Balkesen
cagri.balkesen@inf.ethz.ch

Department of Computer Science ETH Zurich, Switzerland

19 September 2012

About your TA

Cagri Balkesen
CAB E 77.2
+41 44 632 2496
`cagri.balkesen@inf.ethz.ch`

Biography

since July 2009	PhD Student, Systems Group, ETH Zurich
2007–2009	MSc CS, ETH Zurich
2003–2007	BSc CS, Middle East Technical University, Turkey

Research

Parallel data processing in database and data stream management systems by utilizing modern multi-core machines

More Info

`http://www.inf.ethz.ch/personal/cagri.balkesen/`
`Twitter:@cagri_balkesen`

Exercise Sessions: Provisional Schedule

- Assignments not mandatory but relevant for exam!

Date	Handout	Discussion
20.09.2012	Assignment 1	-
27.09.2012	Assignment 2	-
04.10.2012	-	Assignment 1
11.10.2012	Assignment 3	-
18.10.2012	-	Assignment 2
25.10.2012	Assignment 4	-
01.11.2012	-	Assignment 3
08.11.2012	Assignment 5	-
15.11.2012	-	Assignment 4
22.11.2012	Assignment 6	-
29.11.2012	-	Assignment 5
06.12.2012	-	Assignment 6
13.12.2012	Buffer	Buffer

Assignments 1-6: Goals

- Support your understanding of the lecture material

Assignments 1-6: Goals

- Support your understanding of the lecture material
- To get a *practical* understanding of how certain hardware aspects affects data processing algorithms and *experience* these effects on your own machine

Assignments 1-6: Goals

- Support your understanding of the lecture material
- To get a *practical* understanding of how certain hardware aspects affects data processing algorithms and *experience* these effects on your own machine
- Improve your low-level programming skills (mostly C)

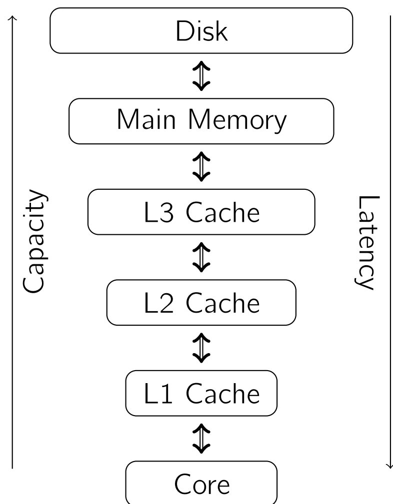
Assignments 1-6: Goals

- Support your understanding of the lecture material
- To get a *practical* understanding of how certain hardware aspects affects data processing algorithms and *experience* these effects on your own machine
- Improve your low-level programming skills (mostly C)
- Learn how to measure performance and profile your programs

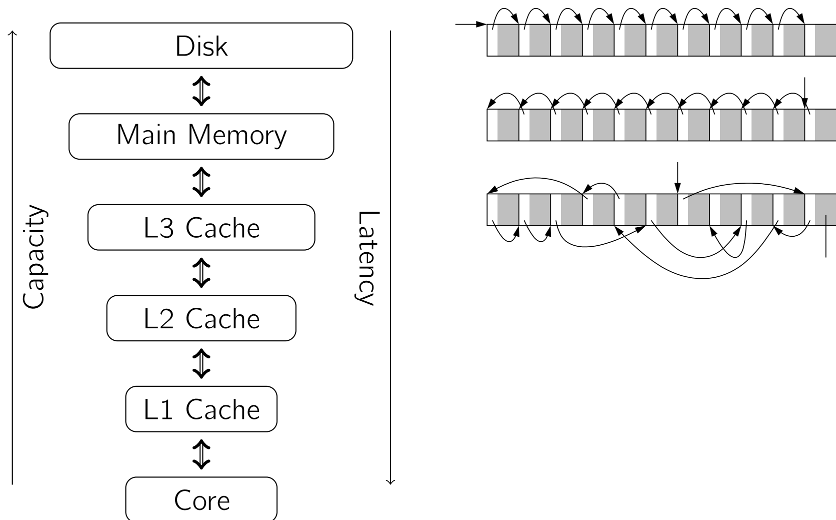
Assignments 1-6: Goals

- Support your understanding of the lecture material
- To get a *practical* understanding of how certain hardware aspects affects data processing algorithms and *experience* these effects on your own machine
- Improve your low-level programming skills (mostly C)
- Learn how to measure performance and profile your programs
- Get in contact with advanced hardware technologies like SSE and AVX, GPGPUs, FPGAs, etc.

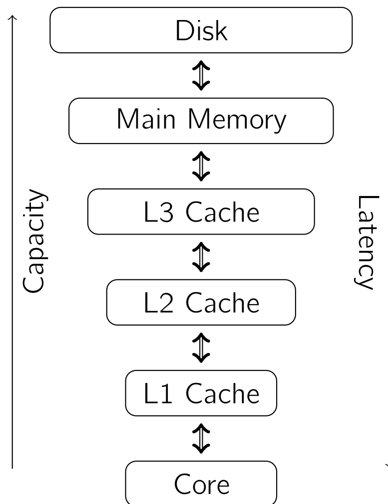
Overview: Assignment 1—Probing your Cache

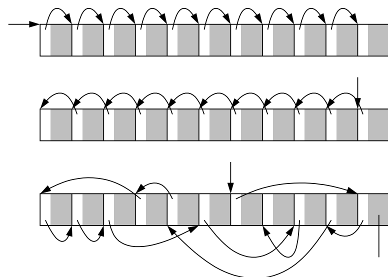


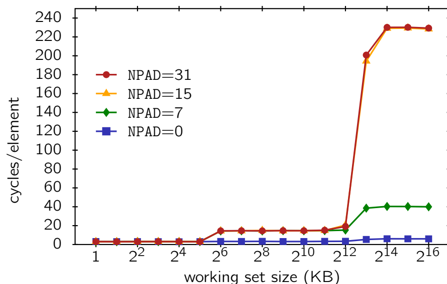
Overview: Assignment 1—Probing your Cache



Overview: Assignment 1—Probing your Cache







Overview: Assignment 2—Cache Awareness in DBs

Row-Store versus Column-Store → we evaluate the two approaches for main memory databases on the TPC-H benchmark.

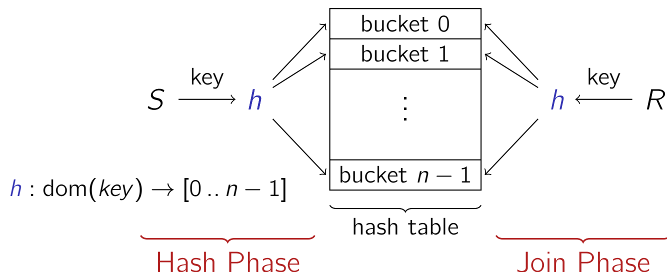
```
SELECT SUM(orderkey + linenumber * shipdate)
FROM   lineitem
```

```
/* row store */
lineitem_t relation[N];
```

```
/* column store */
uint32_t orderkey[N];
uint32_t partkey[N];
...
char     comment[45*N];
uint32_t shipdate[N];
```

Overview: Assignment 3—Hash Joins and Caches

A virtue of hashing is the $\mathcal{O}(1)$ access for every tuple.

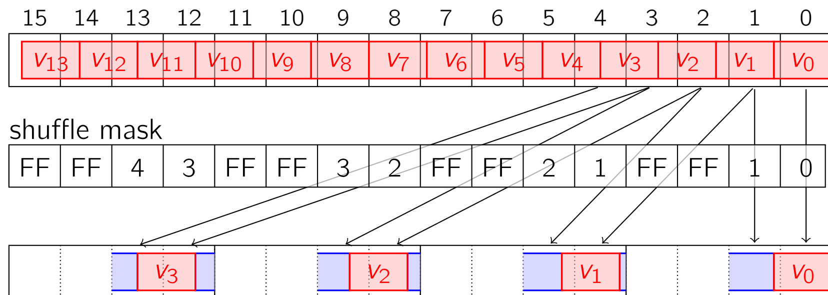


But that's also its Achilles' heel:

- Bucket access is inherently "random."
- Hash table access is going to become **expensive** when the size of the hash table exceeds the size of the cache.

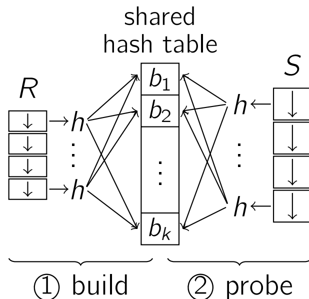
Overview: Assignment 4—Vectorized (De)compression

Parallel conversion of 9-bit integers into 32-bit integers.



Overview: Assignment 5—Parallel Multi-core Hash Joins

Parallel execution of hash joins (assignment 3) on multi-core machines.



- Parallel implementation of some of the hash join algorithms (i.e. no partitioning join shown in figure or radix join from assignment 3).

Overview: Assignment 6—Query Execution on GPGPUs

- Port column store (assignment 2) onto GPU using CUDA | OpenCL.
- Challenge: SQL query with aggregation → parallel reduction.

```
SELECT SUM(quantity * extendedprice)
FROM   lineitem
WHERE  suppkey<Z
```

- "Parallel Reduction: easy to implement in CUDA, harder to get it right" (Mark Harris, NVIDIA)

Assignment 1: Probing your Cache

Warm-up: Get to Know your Hardware

- Figure out what CPU and GPU you are working with
  - Read specs from manufacturers
  - Linux: `lshw` command
  - Linux: cache info at `/sys/devices/system/cpu/cpu*/cache/index*`

Warm-up: Get to Know your Hardware

- Figure out what CPU and GPU you are working with
  - Read specs from manufacturers
  - Linux: `lshw` command
  - Linux: cache info at `/sys/devices/system/cpu/cpu*/cache/index*`
- Do you have old hardware? Let me know

Warm-up: Get to Know your Hardware

- Figure out what CPU and GPU you are working with
  - ▷ Read specs from manufacturers
  - ▷ Linux: `lshw` command
  - ▷ Linux: cache info at `/sys/devices/system/cpu/cpu*/cache/index*`
- Do you have old hardware? Let me know
- Send me this information until next week (September 26, 2012)
  - ▷ E-Mail to cagri.balkesen@inf.ethz.ch
  - ▷ Subject = DPMH:assignment1_specs_{your netzname}
  - ▷ Body = Description of CPU and GPU
    1. CPU: Intel, Xeon QuadCore L5520, (4x 2267MHz),
       microarchitecture : Nehalem (Gainestown)
       L1D = 32KB, L1I = 32KB, L2 = 256KB, L3 = 8MB
    2. GPU: NVIDIA, GeForce 9500 GT,
       4 Multiprocessors (MP) x 8 (Cores/MP) = 32 CUDA Cores

Assignment 1: Task

1. Write a C program that probes your cache

Assignment 1: Task

1. Write a C program that probes your cache
2. Measure CPU cycles (or nanoseconds) for data access

Assignment 1: Task

1. Write a C program that probes your cache
2. Measure CPU cycles (or nanoseconds) for data access
3. Determine
   - size of your L1 D-cache
   - size of your L2 and L3 (if available) cache
   - size of a cache line

Assignment 1: Task

1. Write a C program that probes your cache
2. Measure CPU cycles (or nanoseconds) for data access
3. Determine
   - size of your L1 D-cache
   - size of your L2 and L3 (if available) cache
   - size of a cache line
4. Can you also figure out the size of your TLB?

Assignment 1: Task

1. Write a C program that probes your cache
2. Measure CPU cycles (or nanoseconds) for data access
3. Determine
   - size of your L1 D-cache
   - size of your L2 and L3 (if available) cache
   - size of a cache line
4. Can you also figure out the size of your TLB?

- Make cache access visible, *e.g.*, do not use the `CPUID` instr.

Assignment 1: Task

1. Write a C program that probes your cache
2. Measure CPU cycles (or nanoseconds) for data access
3. Determine
   - size of your L1 D-cache
   - size of your L2 and L3 (if available) cache
   - size of a cache line
4. Can you also figure out the size of your TLB?

- Make cache access visible, *e.g.*, do not use the `CPUID` instr.
- Prepare different memory access patterns (sequential, random)
  → How well does your CPU anticipate the access pattern?

Assignment 1: Task

1. Write a C program that probes your cache
2. Measure CPU cycles (or nanoseconds) for data access
3. Determine
   - size of your L1 D-cache
   - size of your L2 and L3 (if available) cache
   - size of a cache line
4. Can you also figure out the size of your TLB?

- Make cache access visible, *e.g.*, do not use the `CPUID` instr.
- Prepare different memory access patterns (sequential, random)
  $\rightarrow$ How well does your CPU anticipate the access pattern?
- Optional: what happens for write access?

Tricking the Compiler and the CPU

- Use *data-dependent code*, no simple `for` loops
- Choose code so that memory accessed next depends on the current memory access
- You may still want to use `-O2` optimization settings but watch out for “dead-code removal”

Tricking the Compiler and the CPU

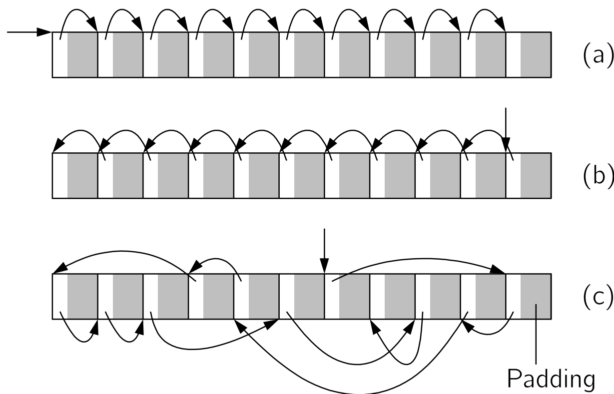
- Use *data-dependent code*, no simple `for` loops
- Choose code so that memory accessed next depends on the current memory access
- You may still want to use `-O2` optimization settings but watch out for "dead-code removal"

Bad example:

```
for(i=0; i<N; i++) {
    array[i]++;
}
```

- ☹ No data dependency
- ☹ Compiler can unroll and pipeline
- ☹ CPU can have multiple outstanding memory requests
- ☹ CPU can prefetch

Memory Traversal using Linked Lists



(a): increasing order, (b): decreasing order, (c): random order

Memory Traversal using Linked Lists (cont'd)

Linked list with padding array:

```
struct listelement {
    struct listelement *next;
    long int padding[NUMPAD];
}
```

$\rightarrow$ Use `NUMPAD` $= 0, \ldots, 31$

Note: array element size is 4 bytes on 32 bit and 8 bytes on 64 bit architectures

- Allocate array that holds all list elements
- Set up pointers according to traversal order
  - increasing : sequentially from low to high addresses
  - decreasing : sequentially form high to low addresses
  - random order
- Traverse linked list multiple times (e.g. 100,000) and measure (wall clock) time

Generating a Random Permutation

- Start with index array `index[i] = i`
- Generate a random permutation of the array using *Knuth Shuffle*
- Generate linked list using shuffled index array

```
/* Knuth Shuffle */
for(i=N-1; i>1; i--) {
    /* uniformly 0 ≤ j ≤ i */
    j = rand()%(i+1);
    tmp = index[j];
    index[j] = index[i];
    index[i] = tmp;
}
```

Measuring Time

- Measure wall clock for each single run or all runs
- Use `gettimeofday()` or `clock_gettime()` (compile with `-lrt`)

Function	**Resolution**
POSIX:	
`gettimeofday()`	$> 1\mu s$
`clock_gettime()`	`clock_getres()`
Windows:	
`GetTickCount()`	≈ 10 ms

Measuring Time

- Measure wall clock for each single run or all runs
- Use `gettimeofday()` or `clock_gettime()` (compile with `-lrt`)
- Or read **Time Stamp Counter**
  - ▷ 64-bit register on x86 CPUs
  - ▷ counts clock ticks since reset

Function	Resolution
POSIX:	
`gettimeofday()`	$> 1\mu s$
`clock_gettime()`	`clock_getres()`
Windows:	
`GetTickCount()`	$\approx$ 10 ms

Reading the CPU Time Stamp Counter

Read Time Stamp Counter (TSC):

```
static inline void
getcyclecount(uint64_t* cycles)
{
  __asm __volatile(
    "cpuid # force all prev. instr. to complete\n\t"
    "rdtsc              # TSC → edx:eax \n\t"
    "movl %%edx, 4(%0) # store edx\n\t"
    "movl %%eax, 0(%0) # store eax\n\t"
    : :"r"(cycles) : "eax", "ebx", "ecx", "edx");
}
```

Measuring Time — Caveats and Extra Tips

- If your results don't make sense

Measuring Time — Caveats and Extra Tips

- If your results don't make sense
  - Reduce noise by terminating interfering processes, e.g., updatedb

Measuring Time — Caveats and Extra Tips

- If your results don't make sense
  - ▷ Reduce noise by terminating interfering processes, e.g., updatedb
  - ▷ Consider using median of single runs instead of average (median is less sensitive to "outlier" noise than average)

Measuring Time — Caveats and Extra Tips

- If your results don't make sense
  - Reduce noise by terminating interfering processes, e.g., updatedb
  - Consider using median of single runs instead of average (median is less sensitive to "outlier" noise than average)
  - Watch out for frequency scaling (power management) when reading the cycle counter

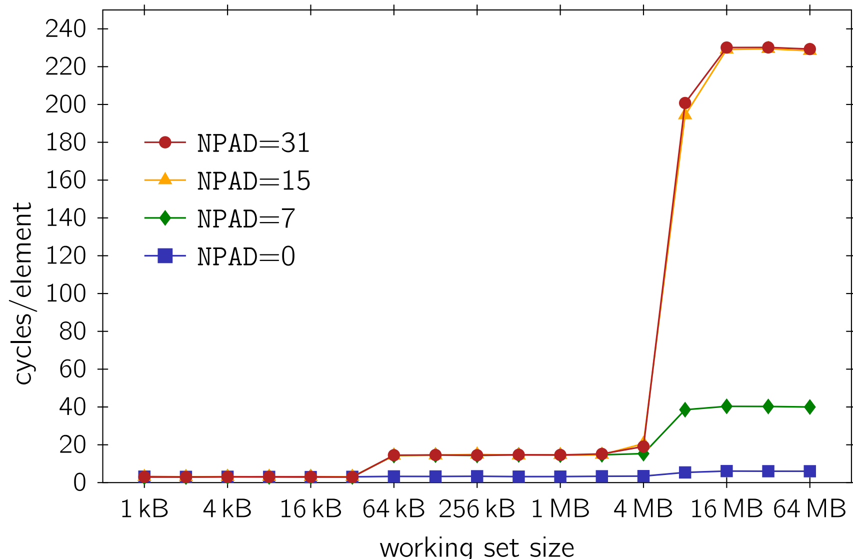
Measuring Time — Caveats and Extra Tips

- If your results don't make sense
  - ▷ Reduce noise by terminating interfering processes, e.g., updatedb
  - ▷ Consider using median of single runs instead of average (median is less sensitive to "outlier" noise than average)
  - ▷ Watch out for frequency scaling (power management) when reading the cycle counter
  - ▷ Multi-core : pin the process to a core: `sched_setaffinity()`

Measuring Time — Caveats and Extra Tips

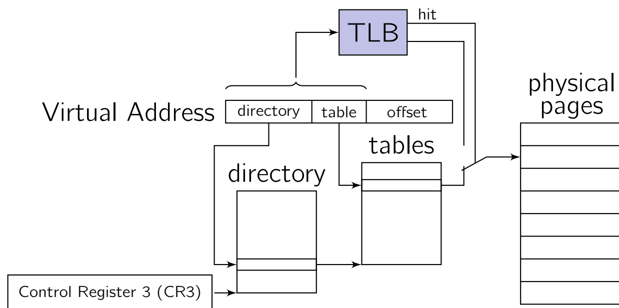
- If your results don't make sense
  - Reduce noise by terminating interfering processes, e.g., updatedb
  - Consider using median of single runs instead of average (median is less sensitive to "outlier" noise than average)
  - Watch out for frequency scaling (power management) when reading the cycle counter
  - Multi-core : pin the process to a core: `sched_setaffinity()`
  - The cycle counter is accurate enough to measure each run separately

Plot your Results and Analyze them : Example



Translation Lookaside Buffer (TLB)

Intel 32-bit Paging (4 kB pages):



TLB Experiment

1. Choose an element size so that each element occupies a single cache line

TLB Experiment

1. Choose an element size so that each element occupies a single cache line
2. Align the elements on cache-line boundaries

TLB Experiment

1. Choose an element size so that each element occupies a single cache line
2. Align the elements on cache-line boundaries
3. Align the elements on page-boundaries (e.g., 4 KB)

TLB Experiment

1. Choose an element size so that each element occupies a single cache line
2. Align the elements on cache-line boundaries
3. Align the elements on page-boundaries (e.g., 4 KB)
4. Compare the two results
   - ▷ Warning: You may have to allocate a large portion of your RAM. Do not choose too much memory (swapping to disk)
   - ▷ Can you estimate the size of your TLB cache?

Hand in your Results

- E-Mail to cagri.balkesen@inf.ethz.ch
- Subject = DPMH:assignment1_{your netzname}
- Body = Description of CPU you tested on, e.g., Xeon QuadCore L5520,(4x 2267MHz)
- Attach plots + raw data
- Attach source code (optional)

Literature

- This assignment is based on
  Ulrich Drepper. *What every programmer should know about memory*. LWN.net. 2007.
  `http://people.redhat.com/drepper/cpumemory.pdf`.
  → You might find Section 3.3.2 relevant for this exercise.

- Knuth Shuffle → Durstenfeld Richard. Algorithm 235: Random Permutation. Communications of the ACM **7**(7): 420. July, 1964.