CMSC724: Modern Hardware

Amol Deshpande

University of Maryland, College Park

May 8, 2013
Motivation

- DBMSs built for 70’s-80’s hardware
- Current hardware is much much different
  - Need to rethink the design
- Key issues:
  - Pipelining $\rightarrow$ dependent code, branches bad
  - Multi-core
  - Caches
  - GPUs: lots of processing power, not clear how to use it
  - Increasingly NUMA architectures
  - FPGAs
Overview of "Modern" CPUs

- Discussion from: "MonetDB/X100: Hyper-Pipelining Query Execution"; CIDR 2005
- Heavy use of instruction pipelining
  - Split a CPU instruction into large number of stages
    - 1993 Pentium: 5-stage pipeline, 2004 Pentium4: 31 pipeline stages
    - Example stages: IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, etc...
  - More stages → simpler architecture
  - More stages necessitates speculative execution
  - More stages → Wasted work because of dependent instructions and branch misprediction
Overview of "Modern" CPUs

- Heavy use of instruction pipelining

<table>
<thead>
<tr>
<th>Instr. No.</th>
<th>Pipeline Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>2</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>3</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>4</td>
<td>IF ID EX MEM</td>
</tr>
<tr>
<td>5</td>
<td>IF ID EX</td>
</tr>
<tr>
<td>Clock Cycle</td>
<td>1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>
Overview of "Modern" CPUs

- Heavy use of instruction pipelining

<table>
<thead>
<tr>
<th>Instr. No.</th>
<th>Pipeline Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>2</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>3</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>4</td>
<td>IF ID EX MEM</td>
</tr>
<tr>
<td>5</td>
<td>IF ID EX</td>
</tr>
</tbody>
</table>

- Super-scalar architectures
  - Large number of independent pipelines
  - Hard to keep feeding data into them in many cases
Overview of "Modern" CPUs

- Heavy use of instruction pipelining

<table>
<thead>
<tr>
<th>Instr. No.</th>
<th>Pipeline Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td>2</td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td>3</td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td>4</td>
<td>IF  ID  EX  MEM</td>
</tr>
<tr>
<td>5</td>
<td>IF  ID  EX</td>
</tr>
</tbody>
</table>

- Super-scalar architectures
  - Large number of independent pipelines
  - Hard to keep feeding data into them in many cases
Discussion from: "MonetDB/X100: Hyper-Pipelining Query Execution"; CIDR 2005

```c
int sel_lt_int_col_int_val(int n, int* res, int* in, int V) {
    for(int i=0, j=0; i<n; i++){
        /* branch version */
        if (src[i] < V)
            out[j++] = i;
        /* predicated version */
        bool b = (src[i] < V);
        out[j] = i;
        j += b;
    }
    return j;
}
```

Figure 2: Itanium Hardware Predication Eliminates Branch Mispredictions
Overview of "Modern" CPUs

- Discussion from: "Breaking the Memory Wall in MonetDB"; CACM 2008
- Memory Hierarchy

![Hierarchical memory architecture](image)

- On-chip caches (SRAM, vs DRAM for main memory)
  - Increasingly overall execution CPU bound, rather than I/O bound
  - Cache misses quite critical
Memory Hierarchy

Issues:

- DRAM latencies: 1-2 cycles in 80’s, 300 cycles today (i.e., 2009)
- L1 cache: usually split between instructions and data; L2, L3: unified
- Capacity = 30KB-4MB; Line size = 64 bytes
- Associativity: higher associative ==> better performance, but much higher cost (i.e., silicon cost)
  - Compulsory cache misses vs conflict misses
- Translation lookaside buffer (TLB) (Note: paper called it "transition" – wrong)
  - Mapping between virtual addresses and real addresses
  - Double penalty if you get a TLB miss
Overview of "Modern" CPUs

- Discussion from: "Breaking the Memory Wall in MonetDB"; CACM 2008
- Key innovations that help:
  - Vertical storage: Better utilization of cache lines
  - Bulk query algebra: Simplified algebra
  - Cache-conscious algorithms: Re-implementations to minimize cache misses
  - Memory access cost modeling
- Aside: always keep in mind Amdahl’s law
  - In many of these cases, the maximum benefits are limited
  - Often not orders of magnitude
DBMSs on Modern Processors: Where does the time go?

- VLDB 1999
- Detailed study comparing multiple commercial DBMSs
  - Couldn’t name the DBMSs because of the "DeWitt" clause

**Figure 5.1:** Query execution time breakdown into the four time components.
VLDB 1999

Figure 5.2: Contributions of the five memory components to the memory stall time ($T_M$)
There are some techniques to reduce the I-cache stall time [6] and use the L1 I-cache more effectively. Unfortunately, the first-level cache size is not expected to increase at the same rate as the second-level cache size, because large L1 caches are not as fast and may slow down the processor clock. Some new processors use a larger (64-KB) L1 I-cache that is accessed through multiple pipeline stages, but the trade-off between size and latency still exists. Consequently, the DBMSs must improve spatial locality in the instruction stream. Possible techniques include storing together frequently accessed instructions while pushing instructions that are not used that often, like error-handling routines, to different locations.

An additional, somewhat surprising, observation was that increasing data record size increases L1 I-cache misses (and, of course, L1 D-cache misses). It is natural that larger data records would cause both more L1 and L2 data misses. Since the L2 cache is unified, the interference from more L2 data misses could cause more L2 instruction misses. But how do larger data records cause more L1 instruction misses? On certain machines, an explanation would be inclusion (i.e., an L1 cache may only contain blocks present in an L2 cache). Inclusion is often enforced by making L2 cache replacements force L1 cache replacements. Thus, increased L2 interference could lead to more L1 instruction misses. The Xeon processor, however, does not enforce inclusion. Another possible explanation is interference of the NT operating system [19]. NT interrupts the processor periodically for context switching, and upon each interrupt the contents of L1 I-cache are replaced with operating system code. As the DBMS resumes execution, it fetches its instructions back into the L1 I-cache. As the record size varies between 20 and 200 bytes, the execution time per record increases by a factor of 2.5 to 4, depending on the DBMS. Therefore, larger records incur more operating system interrupts and this could explain increased L1 I-cache misses. Finally, a third explanation is that larger records incur more frequent page boundary crossings. Upon each crossing the DBMS executes buffer pool management instructions. However, more experiments are needed to test these hypotheses.

5.3 Branch mispredictions

As was explained in Section 3.2, branch mispredictions have serious performance implications, because (a) they cause a serial bottleneck in the pipeline and (b) they cause instruction cache misses, which in turn incur additional stalls. Branch instructions account for 20% of the total instructions retired in all of the experiments. Even with our simple workload, three out of the four DBMSs tested suffer significantly from branch misprediction stalls. Branch mispredictions depend upon query selectivity.

**Figure 5.4:** Left: Branch misprediction rates. SRS: sequential selection, IRS: indexed selection, SJ: join. Right: System D running a sequential selection. $T_B$ and $T_{L1I}$ both increase as a function of an increase in the selectivity.
MonetDB/X100

- Discussion from: "MonetDB/X100: Hyper-Pipelining Query Execution"; CIDR 2005
- Detailed experiments comparing where time went in a relational DBMS
- Even simple queries on MySQL: most time is not spent doing useful work
  - 62% spread over functions dealing with MySQL’s record representation
- MonetDB (old version) suffered from memory bandwidth limitations
- X100: New vectorized query processor
Volcano iterator model

- Discussion from notes by Jens Teubner (ETH)
- Data passed from operator to operator using `next()`
- Problems:
  - Operators tightly interleaved $\rightarrow$ instruction cache misses
  - Large function call overhead
  - Combined state too large to fit into caches (data cache misses)
  - Single-tuple functions hard to optimize by compiler
MonetDB Column-at-a-time processing

- Discussion from notes by Jens Teubner (ETH)
- Operators consume and produce full columns
- Each sub-result fully materialized
- No pipelining
- Example:
  - `sel-age := people-age.select(30, nil)`
  - `sel-id := sel-age.mirror().join(people-age)`
  - `tmp := [-] (sel-age, 30)`

Advantages:
- Tight loops conveniently fit into instruction caches,
- Can be effectively optimized by modern compilers,
  - loop unrolling, vectorization (use of SIMD instructions)
- Function calls are now out of the critical code path.
MonetDB Column-at-a-time processing

- Discussion from notes by Jens Teubner (ETH)
- Operators consume and produce full columns
- Each sub-result fully materialized
- No pipelining

Example:

- `sel-age := people-age.select(30, nil)`
- `sel-id := sel-age.mirror().join(people-age)`
- `tmp := [-] (sel-age, 30)`

Advantages:

- Tight loops conveniently fit into instruction caches,
- Can be optimized effectively by modern compilers,
  - loop unrolling
  - vectorization (use of SIMD instructions)
- Function calls are now out of the critical code path.
Still not ideal
- Data may not fit in cache
- Problems when intermediate results don’t fit in main memory

MonetDB aims for a middle ground
- Still use iterator model, but pass vectors of tuples around
- Large enough that the overheads are amortized

**Figure 2: MonetDB: a BAT algebra machine.**
Cache-conscious algorithms - Selections

- Conjunctive selection conditions; Ross; PODS 2001
- Need to apply $k$ predicates, and find all tuples that satisfy all of them
  - Predicates are: $f_1, \ldots, f_k$
  - $r1[i] = \text{the } i\text{'th attribute of } r_1$.

```c
/* Algorithm Branching-And */
for(i=0;i<number_of_records;i++) { 
  if(f1(r1[i]) && \ldots && fk(rk[i]))
    {answer[j++] = i;}
}

/* Algorithm Logical-And */
for(i=0;i<number_of_records;i++) { 
  if(f1(r1[i]) && \ldots & fk(rk[i]))
    {answer[j++] = i;}
}

/* Algorithm No-Branch */
for(i=0;i<number_of_records;i++) { 
  answer[j] = i;
  j += (f1(r1[i]) && \ldots & fk(rk[i]));
}
Cache-conscious algorithms - Selections

- Conjunctive selection conditions; Ross; PODS 2001
- Need to apply $k$ predicates, and find all tuples that satisfy all of them
  - Predicates are: $f_1, ..., f_k$
  - $r_1[i] =$ the $i$'th attribute of $r_1$.
- Can’t expect an optimizer to transform the three plans into each other
  - Different semantics/results in general

![Figure 1: Three implementations: Pentium.](image-url)
Cache-conscious algorithms - Selections

- Conjunctive selection conditions; Ross; PODS 2001
- Mixing the plans superior in many cases

```c
/* A Mixed Algorithm (loop code omitted) */
if((f1(r1[i]) & f2(r2[i])) & f3(r3[i]))
    { answer[j] = i;
      j += (f4(r4[i]) & ... & f_k(r_k[i]));
    }
```
Cache-conscious algorithms - Hash Joins

- Discussion from: "Breaking the Memory Wall in MonetDB"; CACM 2008
- First idea: Make each partition fit into a cache line, say a total of $H$ clusters
- Problem: The partitioning itself creates a huge random access pattern
- Radix cluster:
  - A multi-pass algorithm to do the partitioning into $H$ clusters
The number of clusters created by the Radix-Cluster is clusters. The next pass takes these clusters and subdivides considered one single cluster, and is subdivided into new ones. When the algorithm starts, the entire relation is ward algorithm.

Two-pa...
Cache-conscious algorithms - Prefetching in Hash Joins

- Improving Hash Join Performance through Prefetching; TODS 2007
- Standard hash joins spend 80% time stalled on CPU cache misses
- Try to use "pre-fetching" to hide the cache misses
  - Most standard algorithms rely on identifying predictable patterns
  - Hash joins don’t generate predictable patterns, but we know the pattern
  - So optimizing at the algorithm level helps
Focus on cache misses and storage layout within a page
### Current Scheme: Slotted Pages

Formal name: NSM (N-ary Storage Model)

<table>
<thead>
<tr>
<th>RID</th>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1237</td>
<td>Jane</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>4322</td>
<td>John</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>1563</td>
<td>Jim</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>7658</td>
<td>Susan</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>2534</td>
<td>Leon</td>
<td>43</td>
</tr>
<tr>
<td>6</td>
<td>8791</td>
<td>Dan</td>
<td>37</td>
</tr>
</tbody>
</table>

- Records are stored sequentially
- Offsets to start of each record at end of page
Predicate Evaluation using NSM

<table>
<thead>
<tr>
<th>PAGE HEADER</th>
<th>RH1</th>
<th>RH2</th>
<th>RH3</th>
<th>RH4</th>
<th>MAIN MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>30</td>
<td>RH1</td>
<td>4322</td>
<td>John</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>RH3</td>
<td>1563</td>
<td>Jim</td>
<td>20</td>
<td>RH4</td>
</tr>
<tr>
<td>7658</td>
<td>Susan</td>
<td>52</td>
<td>2534</td>
<td>Leon</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>block 1</th>
<th>block 2</th>
<th>block 3</th>
<th>block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>30</td>
<td>RH</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>RH3</td>
<td>1563</td>
<td>Leon</td>
</tr>
<tr>
<td>Jim</td>
<td>20</td>
<td>RH4</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>2534</td>
<td>Leon</td>
<td></td>
</tr>
</tbody>
</table>

select name from R
where age > 50

NSM pushes non-referenced data to the cache
Partition Attributes Across (PAX)

Partition data *within* the page for spatial locality
A Real NSM Record

NSM: All fields of record stored together + slots
PAX: Detailed Design

PAX: Group fields + amortizes record headers
Multi-core

- Trend towards adding many cores on a single chip
  - Not really possible to increase the performance of a single CPU much more
- Typical today: many CPUs (multi-socket), each many-core
  - Often end up with a NUMA architecture
- Cores share L2, L3 caches
  - Can lead to cache pollution, contention
- Programming can be headache
- New bottlenecks not seen in previous CPUs with few cores
- Transactions are more problematic than read-only queries
  - Database operations are highly parallelizable so having enough work to do not a problem
  - But still need to be careful about NUMA architecture
Multi-core

Discussion from: "Shore-MT: A scalable storage manager for the multicore era"; Johnson et al.; 2009

Scalability experiments on 4 systems

- No contention (each client creates a private table and inserts into it)

Principles for scalable storage managers

- Efficient synchronization primitives needed
- Shorten or remove critical sections
- Eliminate hot-spots, even read-only
GPUs

- Discussion from notes by Jens Teubner (ETH)
- GPUs have increasingly become general-purpose
- Programmability has increased dramatically
  - Originally: hard-coded fix-function pipeline
  - Today: C-like languages (e.g., CUDA, OpenCL)
- An early paper (SIGMOD 2004; Govindaraju et al.)
  - Use "stencils" (ability to render only parts of a screen) to evaluate Boolean predicates
Discussion from notes by Jens Teubner (ETH)

Issues:
- No direct access to GPU buffers from CPU
- Data movement from host to GPU expensive
- Limited memory on GPU
- Programming often convoluted
- Limited support for data types

Many of these are addressed by now
GPUs

- Discussion from notes by Jens Teubner (ETH)
- GPUs provide data parallelism
  - Lightweight threads
  - 10,000 of threads of 100s of cores
  - All threads run the same code (SIMD-like)
- How to use GPUs?
  - Must combine CPUs and GPUs
  - CPUs copy data into GPU buffers
  - Invoke compute functions on GPUs
Sorting with GPUs

- Current GPUs have 10x higher main memory bandwidth and high data parallelism
- Sorting
  - Key problem, and served as a benchmark for many years
  - Originally Sort Benchmark, then MinuteSort, PennySort, JouleSort (how much per Joule of energy used)
- External sorting
  - Distributed-based: disjointly partition input file by sort attribute, sort each partition separately
  - Merge-based: create sorted runs from contiguous chunks, do a merge
  - In either case, need about $\sqrt{\text{memory (in blocks)}}$ to sort a large file in two passes
    - Can sort a terabyte file in few GBs of RAM
- Issues with CPU
  - Cache misses, compute intensive
GPUs

- Designed to execute geometric transformations on rectangular pixel array

Capabilities

- Data parallelism
  - 96 comparisons per clock cycles (NVIDIA 7800 GTX)
- Instruction parallelism
  - 7800 GTX has 313 GFLOPS
- Dedicated memory interface
  - Much higher peak bandwidths
- Low memory latencies
Use both CPUs and GPUs for different things

Figure 3: Flow Diagram of Phase 1 of GPU TeraSort Architecture using GPUs and CPUs.
GPUTeraSort

- Uses Bitonic sort
  - Highly parallel sorting algorithm
  - Standard algorithms like quicksort not a good fit because of dependencies
  - With bitonic sort, the same operations are executed independent of the data

![Bitonic Sorting Network](image)

Figure 4: This figure illustrates a bitonic sorting network on 8 data values. The sorting algorithm proceeds in 3 stages. The output of each stage is the input to the next stage. In each stage, the array is conceptually divided into sorted data chunks or regions highlighted in green and red. Elements of adjacent chunks are merged as indicated by arrows. The minimum element is moved to the green region and the maximum is stored in the red colored regions producing larger sorted chunk.

- Shows huge benefits over CPU sorting