Memory Hierarchies
&&
The New Bottleneck
==
Cache Conscious Data Access

Martin Grund
Agenda

• **Key Question:** *What is the memory hierarchy and how to exploit it?*

• **What to take home**
  
  • How is computer memory organized.
  
  • What should be considered when working with main memory.
  
  • How might future computer architectures look like.
Frequency + Bandwidth

![Graph showing the relationship between speed and year with lines for CPU Speed and Bus Speed.](image)
Transistors
What does this mean?
What does this mean?

- Memory gets slower for the CPU
- More memory channel, stalling latency
- Frequency is stalling, while # transistors increases
- Degree of parallelism increases
Back to Main Memory
The memory hierarchy has been extended with the evolution of CPUs from executing a single instruction to parallel execution; sometimes splitting instructions in more elided execution. A concrete example is work on MonetDB. Another facet is that predictable array-wise processing has profoundly influenced the database area and indeed our research highlights.

Typically, located on the CPU chip itself, the memory hierarchy models have been strongly favored in a string of recent CPU architectural innovations. While the rule "make the common things fast and cheap" has guided the evolution of CPUs, the difference in performance efficiency achieved by optimized code and intended use (e.g., "legacy database applications") versus nonoptimized code and nonintended use (e.g., "multimedia applications") has become very significant. A concrete example is work on MonetDB.

Typical system monitoring tools (top, or Windows Task manager) do not provide insight in this performance magnitude. Typical system monitoring tools (top, or Windows Task manager) do not provide insight in this performance magnitude. Typical system monitoring tools (top, or Windows Task manager) do not provide insight in this performance magnitude. Typical system monitoring tools (top, or Windows Task manager) do not provide insight in this performance magnitude.
Memory Hierarchy

Why not make larger caches?

- SRAM - complicated structure, bigger, more expensive, very fast
- DRAM - simpler structure, slower, constant refresh needed. But, can be read in parallel and exploited by applying sequential access patterns.

Organize memory in hierarchies where faster memories are used as caches for slower memory.
Memory Access

- Random, disk: 316 values/sec
- Sequential, disk: 53.2M values/sec
- Random, SSD: 1924 values/sec
- Sequential, SSD: 42.2M values/sec
- Random, memory: 36.7M values/sec
- Sequential, memory: 358.2M values/sec

* Disk tests were carried out on a freshly booted machine (a Windows 2003 server with 64GB RAM and eight 15,000RPM SAS disks in RAID5 configuration) to eliminate the effect of operating-system disk caching. SSD test used a latest generation Intel high-performance SATA SSD.
Cost of Memory Access

• Each memory access incurs a latency due to the organization of DRAM.

• The different caches try to hide this latency
  • Multi-level data caches
  • Instruction caches
  • Translation lookaside buffer (TLB) for virtual address translation

• Caches resemble features known from database systems (e.g. buffer pool) but are controlled by hardware.
Locality

- Caches exploit locality in programs
- **Spatial Locality** - related data is often spatially close
- **Temporal Locality** - programs tend to re-use data frequently
Writing I

- Processor caches are supposed to be inherent and should be completely transparent to user level code.

- Different write policies
  - **Write-through** - direct write to memory
  - **Write-back** - dirty flag per cache line, as soon as the cache line is evicted, data is written
  - **Write Combining** - combine multiple write operations per cache line and write then
  - **Uncacheable** - cache line is not stored in cache before write
For multi-processor systems cache coherency becomes increasingly complex

MESI protocol (modified, exclusive, shared, invalid) - 4 states to implement write-back with concurrent read-only access

With multiple threads, prefetching and write-back may saturate the bus very early
Cache Conscious - optimizing the program’s performance by changing the organization and layout of its data with additional knowledge of the cache properties ("Cache-conscious structure definition", Chilimbi et al., ACM SIGPLAN 1999).

Cache Oblivious - optimizing the program’s performance by changing the algorithms to adopt the underlying hardware properties without additional knowledge.
How to exploit this?
Allocation & Loading

Memory allocation can be crucial to program performance and it is important to understand its implementation.

- **Avoid cache line splits** - Padding
- **Avoid memory page splits** - Alignment / Padding
- **Heap contention** - multiple threads try to allocate from the same allocator with exclusive access

ftp://g.oswego.edu/pub/misc/malloc.c
Padding

Modify the data structure to match integer denominators of the size of a cache line.

```c
#include <stdio.h>

typedef struct _bad
{
    unsigned first;
    unsigned b;
    unsigned c;
    char second;
} bad;

int main(int argc, char* argv)
{
    printf("%ld\n", sizeof(bad));
    return 0;
}
```
Modify the data structure to match integer denominators of the size of a cache line.

```c
#include <stdio.h>

typedef struct _bad
{
    unsigned first;
    unsigned b;
    unsigned c;
    char second;
} bad;

int main(int argc, char* argv)
{
    printf("%ld\n", sizeof(bad));
    return 0;
}
```
Padding

Modify the data structure to match integer denominators of the size of a cache line.

```c
#include <stdio.h>

typedef struct _bad
{
    unsigned first;
    unsigned b;
    unsigned c;
    char second;
} bad;

int main(int argc, char* argv)
{
    printf("%ld\n", sizeof(bad));
    return 0;
}
```

`sizeof(bad) == ??`
Padding

Modify the data structure to match integer denominators of the size of a cache line.

```c
#include <stdio.h>

typedef struct _bad
{
    unsigned first;
    unsigned b;
    unsigned c;
    char second;
} bad;

int main(int argc, char* argv)
{
    printf("%ld\n", sizeof(bad));
    return 0;
}
```

```
sizeof(bad) == ??
13 16
-fpack-struct=1
```
Padding

Modify the data structure to match integer denominators of the size of a cache line.

```c
#include <stdio.h>

typedef struct _bad
{
    unsigned first;
    unsigned b;
    unsigned c;
    char second;
} bad;

int main(int argc, char* argv)
{
    printf("%ld\n", sizeof(bad));
    return 0;
}
```

sizeof(bad) == ??

13

-ffpack-struct=1

16

-ffpack-struct=4
Prefetching

Prefetching is used to asynchronously read co-located data (see Locality).

- Load adjacent cache line
- Pattern detection with stride based loading
No Prefetching

2 values per cache line
No Prefetching

2 values per cache line
Correct Prefetching
Correct Prefetching
Incorrect Prefetching
Incorrect Prefetching
Virtualization of resources becomes more and more important even for main memory databases:

- System consolidation,
- Administrative consolidation,
- Better provisioning and better scaling.
## Virtualized Access

<table>
<thead>
<tr>
<th>Description</th>
<th>Physical System</th>
<th>Virtualized System</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Miss Latency</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>L1 Replace Time</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>L2 Miss Latency</td>
<td>197</td>
<td>196</td>
</tr>
<tr>
<td>L2 Replace Time</td>
<td>334</td>
<td>333</td>
</tr>
<tr>
<td>TLB Miss Latency</td>
<td>-</td>
<td>23</td>
</tr>
</tbody>
</table>
## Virtualized Access

<table>
<thead>
<tr>
<th>Description</th>
<th>Physical System</th>
<th>Virtualized System</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Miss Latency</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>L1 Replace Time</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>L2 Miss Latency</td>
<td>197</td>
<td>196</td>
</tr>
<tr>
<td>L2 Replace Time</td>
<td>334</td>
<td>333</td>
</tr>
<tr>
<td>TLB Miss Latency</td>
<td>-</td>
<td><strong>23</strong></td>
</tr>
</tbody>
</table>
### Virtualized Access

<table>
<thead>
<tr>
<th>Description</th>
<th>Physical System</th>
<th>Virtualized System</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Miss Latency</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>L1 Replace Time</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>L2 Miss Latency</td>
<td>197</td>
<td>196</td>
</tr>
<tr>
<td>L2 Replace Time</td>
<td>334</td>
<td>333</td>
</tr>
<tr>
<td>TLB Miss Latency</td>
<td>-</td>
<td>23</td>
</tr>
</tbody>
</table>

First Conclusion: Memory Access does not incur a dedicated access latency!

Memory page change requires additional handling and thus incurs latency!
Measure Performance

To **understand** the system’s performance it is necessary to correctly **observe** its behavior.

- Identify relevant measures (CPU cycles, cache misses, resource stalls)
- Collect profiling data using **sampling** based or “real” **counting**
  - **oprofile** - provides sample-based performance evaluation for time based profiling.
  - **PAPI** - measure program performance based on hardware counters. Each CPU provides special registers to count profiling information based on special hardware events (CPU cycles, cache misses).
#include <stdio.h>
#include <papi.h>

int main()
{
    // PAPI events can be identified by name or const value
    char* event = "PAPI_TOT_CYC";
    // Events and results are identified
    // as array
    int events[1];
    long long result[1];

    PAPI_library_init(PAPI_VER_CURRENT);
    PAPI_event_name_to_code((char *) papi, &events[0]);
    PAPI_start_counters(events, 1);
    long long sum = 0;
    // Do something intensive here
    for (unsigned i=0; i < 1000; ++i)
        sum += i;
    PAPI_stop_counters(result, 1)
    printf("%lld\n", result[0]);
    return 0;
}
Example

• “Making B+-Trees Cache Conscious in Main Memory” SIGMOD 2000, J. Rao and K. A. Ross
Cache Sensitive B+ Tree

- B+ Tree - optimized search tree, typically used for indices, originally used to persist indexed data on disk!
- Cache Sensitive B+ Trees optimize cache performance by applying cache conscious techniques
  - Pointer elimination
  - Block structures
Tree Comparison

B+ Tree
Trees that further reduces split cost in Section 7. We describe another variant of CSB tuned to obtain good performance under particular workload. We achieve this by having fewer pointers per node than a B-tree. However, a CSB handles updates. However, a CSB doesn’t have to do any additional copying to form a new node. Since nodes in the same node group are not created sequentially, a more efficient bulkloading method can be expensive if used for CSB. In this section, we consider bulkload, search, insert, and delete operations on CSB. We get away with fewer pointers by using a child pointer explicitly. It can store more keys per node than a B-tree node needs to store just one key, tuple ID, and a list of keys. For example, if the node size is 32 bytes, a CSB+ tree node can only hold k keys per node whereas a CSB tree can store more keys per node than a B-tree. Hence better cache performance.

The number of pointers per node is a parameter that can be tuned to obtain good performance under particular context. In Section 7.0 we will describe variants with more pointers per node. The number of these pairs, and two sibling pointers, sets a cache line can satisfy. A cache line can satisfy almost one more level of comparisons and thus the two kinds of benefit: a cache line can contain both a key and a child pointer each occupies i bytes. Then a B-tree cache line size is: i bytes and a key and a child pointer each occupies i bytes. Then a B-tree child pointer explicitly. It can store more keys per node than a B-tree. A CSB tree comparison structure is a balanced multi-way search tree. For simplicity of presentation, we initially present a CSB tree exactly one pointer. Sometimes we simply use the average of CSB. The arrows from the internal nodes represent the rightmost path from the root. However, this method can be expensive if used for CSB. In this example, a node group can have no more than three nodes within it. Note that grouping is just a physical ordering property and does not have any association space overhead. Since a CSB tree has fewer pointers per node than a B-tree, we have more room for keys and hence better cache performance.

Every node in a CSB is a balanced multi-way search tree. We achieve this by copying the largest value in each node in the lower level to the rightmost key in the higher level. Then fill in the entries of nodes in the higher level. We then calculate how many nodes are needed in the higher level and then allocate a continuous chunk of space for all the nodes in this level. We then calculate how many nodes are needed in the higher level, and then allocate a continuous chunk of space for all the nodes in this level. We repeat the process until a node group is full. Then a node group is physically adjacent to each other. In this example, a node group can have no more than three nodes within it. For example, if a node group is full, we see Section 5 for further discussion of how leaf nodes can be implemented. Based on the fan out of each node is larger, which means the number of cache lines needed for a search is fewer.

A typical bulkloading algorithm for a B-tree is to build the index structure level by level. A more efficient bulkloading method can be used for CSB. Once we have determined how many nodes are needed in each level, we fill the entries of the next level of nodes. We then calculate how many nodes are needed in the next level and allocate a continuous chunk of space for all the nodes in this level. We repeat the process until the maximum number of levels is reached. Since all the nodes in the same node group are created sequentially, a more efficient bulkloading method can be used. However, this method can be expensive if used for CSB. In this example, a node group can have no more than three nodes within it. Note that grouping is just a physical ordering property and does not have any association space overhead. Since a CSB tree has fewer pointers per node than a B-tree, we have more room for keys and hence better cache performance.
be implemented. Based on the fan out of each node is larger, which means the number of cache lines needed for a search is fewer; almost one more level of comparisons and thus the two kinds of benefit: a cache line can satisfy can have $k$ keys per node. This gives CSB a variant can only hold $k$ keys per node whereas a CSB child pointer each occupies $i$ bytes. Then a B-tree cache line size is $i$ bytes and a key and a node than a B-tree has fewer pointers per node, we have more room for keys and handles updates. However, a CSB tree structure, which we call a node group are stored contiguously and can be accessed using an o-node in a CSB node group.

Each leaf node stores a list of keys, a pointer to the first child node, and two sibling pointers. This gives CSB a variant with more pointers per node. The number of these pairs, and two sibling pointers, is a parameter that can be tuned to obtain good performance under particular context. In Section 7.0 we will describe term CSB version CSB exactly one pointer. Sometimes we simply use the average of CSB.

For simplicity of presentation, we initially present to the first node in the group. Every node in a CSB B+ Tree is similar to a B+Tree node needs to store just one key and a tuple ID. The arrows from the internal nodes represent the leftmost key, the o-path from the root. However, this method can be expensive if used for CSB B+ Tree Comparison.

The arrows from the internal nodes represent the leftmost key, the o-path from the root. However, this method can be expensive if used for CSB B+ Tree Comparison.
Tree Comparison

B+ Tree

CSB+ Tree
3.1 Cache Sensitive B

Child Pointer

Each leaf node stores a list of keys. A CSB node has the following structure:

+ a list of keys
+ pointer to the first child node

A typical bulkloading algorithm for 6Trees is to keep inserting sorted leaf entries into the first node in the group. Every node in a CSB node group are physically adjacent to each other. In this example, a node group can have no more than three nodes within it. Note that grouping is just a convenience to improve I/O performance.

A more efficient bulkloading method can be expensive if used for CSB 6Trees. A typical bulkloading algorithm for CSB+ Tree is similar to B+ Tree in that it uses less space. Figure O shows a CSB+ Tree where nodes in the same node group are not created sequentially. A more efficient bulkloading method can be used for CSB+ Tree.

- Only 1 pointer
- Direct Offsets
- Segmented Structure with pointer traversal

Segmented Structure with pointer traversal

B+ Tree  

CSB+ Tree
Conclusion

• Observe system’s behavior
• Understand system’s performance
• Apply applicable optimization techniques.
The Future
Many-Core

- From single-core to multi-core to many-core!
- Frequency ~ Power Consumption ~ Moores Law [1]
- Underclocking a single core by 20 percent saves half the power while sacrificing just 13 percent of the performance.

Multiple Memory Channels

- Instead of increasing the memory bandwidth, increase the number of peers
- NUMA - Non Unified Memory Access
  - Locality revisited - each CPU has local and remote memory, remote memory incurs additional loading latency.
  - Intel Quick Path Interconnect - point to point communication protocol for memory access allows to connect multiple memory channels to the CPU.
• Experimental research platform for new concepts to evaluate the evolution from multi-core to many-core.

• 48 pentium style cores arranged in a two-dimensional array of 6 x 4 tiles with 2 cores each

• Each core has 16kB L1, and 256kB L2 cache private to the core

• All cores are connected to each other using an on-chip mesh network with 256 GB/s bisectional bandwidth
Intel SCC

- **Real NUMA**, measurable latency between memory access of cores

- **Dynamic lookup tables** - each core has a dynamical mapping of main memory to its visible, no cache coherency

- **Direct Message Passing** - new communication strategy for work partitioning and coherency protocols
The Angstrom Multi-Core Computing Project

- 1000 cores by 2014, the core is the logic gate of the 21st century

- **Spatial Problem** - huge near-neighbor bandwidth, low long distance bandwidth. Limited per-core on-chip memories, off-chip memory bandwidth is a big issue. Energy is new constraint.
Conclusion and Motivation

- For optimal performance it is crucial to **understand** the system and **observe** its behavior.
- Main memory based applications need to exploit this:
  - Sequential reading
  - Block sizes of the different caches
Recommended Readings

- **Latency lags bandwidth** - http://portal.acm.org/citation.cfm?id=1022594.1022596


- **DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing** - http://portal.acm.org/citation.cfm?id=1457150.1457160

- **Making B+-Trees Cache Conscious in Main Memory** - http://portal.acm.org/citation.cfm?id=335191.335449

- **Breaking the memory wall in MonetDB** - http://portal.acm.org/beta/citation.cfm?id=1409360.1409380


- **Memory system support for irregular applications** - http://www.springerlink.com/index/TQY3BCP1AEL3AQH6.pdf


- **Intel Tera Scale Research** - http://techresearch.intel.com/articles/Tera-Scale/1421.htm