Understanding, modeling, and improving main-memory database performance

Manegold, S.

Citation for published version (APA):
Chapter 3

Cost Factors in MMDBMS:
"No I/O" does not mean
"Only CPU"

In this chapter, we describe those aspects of hardware technology found in custom computer systems that are most relevant for the performance of main-memory query execution. We identify ongoing trends, and outline their consequences for database architecture. In addition, we describe our calibration tool which extracts the most important hardware characteristics like cache size, cache line size, and cache latency from any computer system, and provide results for our benchmark platforms (modern SGI, Sun, Intel, and AMD hardware).

3.1 Commodity Computer Architecture

Focusing on main-memory processing, we discuss the major technical issues of CPUs, main-memory, and hardware caches which are relevant for database performance.

3.1.1 CPUs

Concerning CPUs, two aspects are of primary interest for this thesis. First of all, we need to have a closer look at how modern CPUs are designed and how they do work. Secondly, we briefly introduce hardware counters that help us to monitor certain events within the CPU and thus understand their impact on the performance of programs.

3.1.1.1 Design & Working Methods

While CPU clock frequency has been following Moore's law (doubling every 18 months [Moo65]), CPUs have additionally become faster through parallelism within the processor. Scalar CPUs separate different execution stages for instructions, e.g.,
3 Cost Factors in MMDBMS

<table>
<thead>
<tr>
<th>Year</th>
<th>Computer Model</th>
<th>Processor Type</th>
<th>Processor Speed (MHz)</th>
<th>Memory STREAM/Copy Bandwidth (Mbps)</th>
<th>Memory Latency (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1989</td>
<td>Sun 380</td>
<td>68020</td>
<td>20</td>
<td>1</td>
<td>6.5</td>
</tr>
<tr>
<td>1990</td>
<td>Sun 380</td>
<td>68030</td>
<td>20</td>
<td>1</td>
<td>4.9</td>
</tr>
<tr>
<td>1991</td>
<td>Sun 40386</td>
<td>68021</td>
<td>17</td>
<td>1</td>
<td>9.6</td>
</tr>
<tr>
<td>1992</td>
<td>Sun Sparc</td>
<td>superSparc I</td>
<td>33</td>
<td>3</td>
<td>42.9</td>
</tr>
<tr>
<td>1993</td>
<td>Sun Sparc</td>
<td>superSparc I</td>
<td>40</td>
<td>3</td>
<td>48.0</td>
</tr>
<tr>
<td>1994</td>
<td>Sun Sparc</td>
<td>superSparc II</td>
<td>75</td>
<td>3</td>
<td>62.5</td>
</tr>
<tr>
<td>1995</td>
<td>Sun Ultra 170</td>
<td>ultraSparc I</td>
<td>167</td>
<td>5</td>
<td>225</td>
</tr>
<tr>
<td>1996</td>
<td>Sun Ultra 2000</td>
<td>ultraSparc II</td>
<td>300</td>
<td>5</td>
<td>228</td>
</tr>
<tr>
<td>1997</td>
<td>SGI PowerCh.</td>
<td>R10000</td>
<td>195</td>
<td>5</td>
<td>172</td>
</tr>
<tr>
<td>1998</td>
<td>SGI Origin 2000</td>
<td>R10000</td>
<td>250</td>
<td>5</td>
<td>332</td>
</tr>
<tr>
<td>1999</td>
<td>SGI Origin 2000</td>
<td>R12000</td>
<td>300</td>
<td>5</td>
<td>336</td>
</tr>
<tr>
<td>2000</td>
<td>Intel PC</td>
<td>80486</td>
<td>66</td>
<td>1</td>
<td>33.3</td>
</tr>
<tr>
<td>2001</td>
<td>Intel PC</td>
<td>Pentium</td>
<td>66</td>
<td>2</td>
<td>47.1</td>
</tr>
<tr>
<td>2002</td>
<td>Intel PC</td>
<td>Pentium</td>
<td>90</td>
<td>2</td>
<td>46.4</td>
</tr>
<tr>
<td>2003</td>
<td>Intel PC</td>
<td>Pentium</td>
<td>120</td>
<td>2</td>
<td>85.1</td>
</tr>
<tr>
<td>2004</td>
<td>Intel PC</td>
<td>Pentium</td>
<td>133</td>
<td>2</td>
<td>64.4</td>
</tr>
<tr>
<td>2005</td>
<td>Intel PC</td>
<td>PentiumPro</td>
<td>200</td>
<td>5</td>
<td>160</td>
</tr>
<tr>
<td>2006</td>
<td>Intel PC</td>
<td>PentiumUltraII</td>
<td>300</td>
<td>5</td>
<td>188</td>
</tr>
<tr>
<td>2007</td>
<td>Intel PC</td>
<td>PentiumUltraII</td>
<td>350</td>
<td>5</td>
<td>275</td>
</tr>
<tr>
<td>2008</td>
<td>Intel PC</td>
<td>PentiumUltraII</td>
<td>400</td>
<td>5</td>
<td>304</td>
</tr>
<tr>
<td>2009</td>
<td>Intel PC</td>
<td>PentiumUltraII</td>
<td>600</td>
<td>5</td>
<td>376</td>
</tr>
<tr>
<td>2010</td>
<td>Intel PC</td>
<td>PentiumUltraII</td>
<td>753</td>
<td>5</td>
<td>441</td>
</tr>
<tr>
<td>2011</td>
<td>AMD PC</td>
<td>Athlon</td>
<td>500</td>
<td>9</td>
<td>373</td>
</tr>
<tr>
<td>2012</td>
<td>AMD PC</td>
<td>Athlon</td>
<td>800</td>
<td>9</td>
<td>387</td>
</tr>
</tbody>
</table>

Figure 3.1: Trends in DRAM and CPU speed

Figure 3.2: Modern CPU and Hierarchical Memory Architecture
allowing a computation stage of one instruction to be overlapped with the decoding stage of the next instruction. Such a pipelined design allows for inter-stage parallelism. Modern super-scalar CPUs add intra-stage parallelism, as they have multiple copies of certain (pipelined) units that can be active simultaneously. Although CPUs are commonly classified as either RISC (reduced instruction set computer) or CISC (complex instruction set computing), modern CPUs combine successful features of both. Figure 3.2 shows a simplified schema that characterizes how modern CPUs work: instructions that need to be executed are loaded from memory by a fetch-and-decode unit. In order to speed up this process, multiple fetch-and-decode units may be present (e.g., Intel's PentiumIII and AMD's Athlon have three, the MIPS R10000 has two). Decoded instructions are placed in an instruction queue, from which they are executed by one of various functional units, which are sometimes specialized in integer-, floating-point, and load/store pipelines. The PentiumIII, for instance, has two such functional units, the R10000 has five, and the Athlon has even nine. To exploit this parallel potential, modern CPUs rely on techniques like branch prediction to predict which instruction will be next before the previous has finished. Also, the modern cache memories are non-blocking, which means that a cache miss does not stall the CPU. Such a design allows the pipelines to be filled with multiple instructions that will probably have to be executed (a.k.a. speculative execution), betting on yet unknown outcomes of previous instructions. All this goes accompanied by the necessary logic to restore order in case of mispredicted branches. As this can cost a significant penalty, and as it is very important to fill all pipelines to obtain the performance potential of the CPU, much attention is paid in hardware design to efficient branch prediction. CPUs work with prediction tables that record statistics about branches taken in the past.

3.1.1.2 Hardware Counters

Detailed insight into the behavior of CPUs while processing application code is a prerequisite to understand, and eventually model, the performance of application programs. To aid this process, many modern CPUs provide so-called hardware event counters or performance counters that allow to monitor certain performance-related events that occur within the CPU while processing user code. Examples are the MIPS R10k/R12k series, Sun's UltraSPARC family, all Intel Pentium and Itanium CPUs, AMD's Athlons, DEC's Alphas, IBM's and Motorola's PowerPC's. Usually, each counter can only monitor one event at a time, however with multiple counters present, several events can be monitored concurrently. The number of counters per CPU varies from 2 (e.g., MIPS R10k) to 8 (e.g., Intel Pentium 4).

Like the number of counters, also the number and kind of events that can be monitored vary significantly between the different CPUs. We omit the details here and refer the interested reader to the respective product manuals. Typically, the set of events includes events like cache misses (both instruction and data), instructions decoded and executed, branches executed, branch mispredictions, etc. We provide more information as required later when we use these features.

In contrary to software profiling as offered by certain compilers and/or profiling tools, using the hardware event counters has no impact on the execution performance.
Low-level access to the counters typically works via direct register access to select the events, start and stop monitoring, and finally read-out the results. More convenient high-level tools do exist but vary between hardware vendors and operating systems.

3.1.2 Main-Memory- & Cache-Systems

We now turn our attention to memory- and cache architectures. We explain the basic technical principles, discuss various aspects of memory access costs, and finally introduce a unified hardware model to be used in the remainder of this thesis.

3.1.2.1 Memory- & Cache-Architectures

Modern computer architectures have a hierarchical memory system as depicted in Figure 3.2. The main memory on the system board consists of DRAM chips (Dynamic Random Access Memory). While CPU speeds are increasing rapidly, DRAM access latency has hardly progressed through time. To narrow the exponentially growing performance gap between CPU speed and memory latency (cf., Figure 3.1), cache memories have been introduced, consisting of fast but expensive SRAM chips (Static Random Access Memory). SRAM cells are usually made-up from six transistors per memory bit, and hence, they consume a rather large area on the chips. DRAM cells require just a single transistor and a small capacitor to store a single bit. Thus, DRAMs can store much more data than SRAMs of equal (physical) size. But due to some leak current, the capacitor in DRAMs get discharged over time, and have to be recharged (refreshed) periodically to keep their information. These refreshes slowdown access.

The fundamental principle of all cache architectures is “reference locality”, i.e., the assumption that at any time the CPU, respectively the program, repeatedly accesses only a limited amount of data (i.e., memory) that fits in the cache. Only the first access is “slow”, as the data has to be loaded from main memory. We call this a compulsory cache miss (see below). Subsequent accesses (to the same data or memory addresses) are then “fast” as the data is then available in the cache. We call this a cache hit. The fraction of memory accesses that can be fulfilled from the cache is called cache hit rate; analogously, the fraction of memory accesses that cannot be fulfilled from the cache is called cache miss rate.

Cache memories are often organized in multiple cascading levels between the main memory and the CPU. They become faster, but smaller, the closer they are to the CPU. Originally, there was one level of typically 64 KB to 512 KB cache memory located on the system board. As the chip manufacturing processes improved, a small cache of about 4 KB to 16 KB got integrated on the CPU’s die itself, allowing much faster access. The on-board is typically not replaced by the on-chip cache, but rather both make up a cache hierarchy, with the one on chip called first level (L1) cache and the one on board called second level (L2) cache. Recently, also the L2 cache has been integrated on the CPU’s die (e.g., with Intel’s Pentium III “Coppermine”, or AMD’s Athlon “Thunderbird”). On PC systems, the on-board cache has since disappeared, keeping two cache levels. On other platforms, e.g., workstations based on Compaq’s
Commodity Computer Architecture

(formerly DEC's) Alpha CPU, the on-board cache is kept as third level (L3) cache, next to the two levels on the die.

To keep presentations from becoming complicated, we assume a typical system with two cache levels (L1 & L2) in most examples in the remainder of this work. However, our observations and results can easily be generalized to an arbitrary number of cascading cache levels in a straightforward way.

In practice, caches memories do not only cache the data used by an application, but also the program itself, more accurately, the instructions that are currently being executed. With respect to caching, there is one major difference between data and program. Usually, a program must not be modified while it is running, i.e., the caches may be read-only. Data, however, requires caches that also allow modification of the cached data. Therefore, almost all systems nowadays implement two separate L1 caches, a read-only one for instructions and a read-write one for data. The L2 cache, however, is usually a single "unified" read-write cache used for both instructions and data. Later in this thesis, we will see that instruction cache misses do not play a significant role in our scenario. Hence, we will not discuss instruction caches in more detail. Only where necessary, we will address them explicitly. Unless mentioned differently, we will refer to data caches simply as caches.

Caches are characterized by three major parameters: Capacity (C), Line Size (Z), and Associativity (A):

Capacity (C) A cache's capacity defines its total size in bytes. Typical cache sizes range from 8 KB to 8 MB.

Line Size (Z) Caches are organized in cache lines, which represent the smallest unit of transfer between adjacent cache levels. Whenever a cache miss occurs, a complete cache line (i.e., multiple consecutive words) is loaded from the next cache level or from main memory, transferring all bits in the cache line in parallel over a wide bus. This exploits spatial locality, increasing the chances of cache hits for future references to data that is "closed to" the reference that caused a cache miss. Typical cache line sizes range from 16 bytes to 128 bytes.

Dividing the cache capacity by the cache line size, we get the number of available cache lines in the cache: \( \# = C/Z \). Cache lines are often also called cache blocks. We use both terms as synonyms throughout this document.

Associativity (A) To which cache line the memory is loaded, depends on the memory address and on the cache’s associativity. An A-way set associative cache allows to load a line in A different positions. If A > 1, some cache replacement policy chooses one from the A candidates. Least Recently Used (LRU) is the most common replacement algorithm. In case A = 1, we call the cache direct-mapped. This organization causes the least (virtually no) overhead in determining the cache line candidate. However, it also offers the least flexibility and may cause a lot of conflict misses (see below). The other extreme case are fully associative caches. Here, each memory address can be loaded to any line in the cache (A = \#). This avoids conflict misses, and only capacity misses...
(see below) occur as the cache capacity gets exceeded. However, determining the cache line candidate in this strategy causes a relatively high overhead that increases with the cache size. Hence, it is feasible only for smaller caches. Current PCs and workstations typically implement 2-way to 8-way set associative caches.

With multiple cache levels, we further distinguish two types: inclusive and exclusive caches. With inclusive caches, all data stored in L1 is also stored in L2. As data is loaded from memory, it gets stored in all cache levels. Whenever a cache line needs to be replaced in L1 (because a mapping conflict occurs or as the capacity is exceeded), its original content can simply be discarded as another copy of that data still remains in the (usually larger) L2. The new content is then loaded from where it is found (either L2 or main memory). The total capacity of an inclusive cache hierarchy is hence determined by the largest level. With exclusive caches, all cached data is stored in exactly one cache level. As data is loaded from memory, it gets stored only in the L1 cache. When a cache lines needs to be replaced in L1, its original content is first written back to L2. If the new content is then found in L2, it is moved from L2 to L1, otherwise, it is copied from main memory to L1. Compared to inclusive cache hierarchies, exclusive cache hierarchies virtually extend the cache size, as the total capacity becomes the sum of all levels. However, the “swap” of cache lines between adjacent cache levels in case of a cache miss also causes more “traffic” on the bus and hence increases the cache miss latency. We will analyze this in more detail in Section 3.3.

Cache misses can be classified into the following disjoint types [HS89]:

- **Compulsory** The very first reference to a cache line always causes a cache miss, which is hence classified as a compulsory miss. The number of compulsory misses obviously depends only on the data volume and the cache line size.
- **Capacity** A reference that misses in a fully associative cache is classified as a capacity miss because the finite sized cache is unable to hold all the referenced data. Capacity misses can be minimized by increasing the temporal and spatial locality of references in the algorithm. Increasing cache size also reduces the capacity misses because it captures more locality.
- **Conflict** A reference that hits in a fully associative cache but misses in an A-way set associative cache is classified as a conflict miss. This is because even though the cache was large enough to hold all the recently accessed data, its associativity constraints force some of the required data out of the cache prematurely. For instance, alternately accessing just two memory addresses that “happen to be” mapped to the same cache line will cause a conflict cache miss with each access. Conflict misses are the hardest to remove because they occur due to address conflicts in the data structure layout and are specific to a cache size and associativity. Data structures would, in general, have to be remapped so as to minimize conflicting addresses. Increasing the associativity of a cache will decrease the conflict misses.
3.1.2.2 Memory Access Costs

We identify the following three aspects that determine memory access costs. For simplicity of presentation, we assume 2 cache levels in this section. Generalization to an arbitrary number of caches is straightforward.

**Latency** Latency is the time span that passes after issuing a data access until the requested data is available in the CPU. In hierarchical memory systems, the latency increases with the distance from the CPU. Accessing data that is already available in the L1 cache causes *L1 access latency* ($\lambda_{L1}$), which is typically rather small (1 or 2 CPU cycles). In case the requested data in not found in L1, an *L1 miss* occurs, additionally delaying the data access by *L2 access latency* ($\lambda_{L2}$) for accessing the L2 cache. Analogously, if the data is not yet available in L2, an *L2 miss* occurs, further delaying the access by *memory access latency* ($\lambda_{Mem}$) to finally load the data from main memory. Hence, the total latency to access data that is in neither cache is $\lambda_{Mem} + \lambda_{L2} + \lambda_{L1}$. As L1 accesses cannot be avoided, we assume in the remainder of this thesis, that L1 access latency is included in the pure CPU costs, and regard only memory access latency and L2 access latency as explicit memory access costs. As mentioned above, all current hardware actually transfers multiple consecutive words, i.e., a complete cache line, during this time.

When a CPU requests data from a certain memory address, modern DRAM chips supply not only the requested data, but also the data from subsequent addresses. The data is then available without additional address request. This feature is called *Extended Data Output* (EDO). Anticipating sequential memory access, EDO reduces the effective latency. Hence, we actually need to distinguish two types of latency for memory access. *Sequential access latency* ($\lambda^s$) occurs with sequential memory access, exploiting the EDO feature. With random memory access, EDO does not speed up memory access. Thus, *random access latency* ($\lambda^r$) is usually higher than sequential access latency.

**Bandwidth** Bandwidth is a metric for the data volume (in megabytes) that can be transferred between CPU and main memory per second. Bandwidth usually decreases with the distance from the CPU, i.e., between L1 and L2 more data can be transferred per time than between L2 and main memory. We refer to the different bandwidths as *L2 access bandwidth* ($\beta_{L2}$) and *memory access bandwidth* ($\beta_{Mem}$), respectively. In conventional hardware, the memory bandwidth used to be simply the cache line size divided by the memory latency. Modern multiprocessor systems typically provide excess bandwidth capacity $\beta^e \geq \beta$. To exploit this, caches need to be *non-blocking*, i.e., they need to allow more than one outstanding memory load at a time, and the CPU has to be able to issue subsequent load requests while waiting for the first one(s) to be resolved. Further, the access pattern needs to be sequential, in order to exploit the EDO feature as described above.

Indicating its dependency on sequential access, we refer to the excess bandwidth as *sequential access bandwidth* ($\beta^e = \beta^s$). We define the respective *sequential access latency* as $\lambda^s = Z/\beta^s$. For *random access latency* as described above, we define the
respective random access bandwidth as $\beta^R = Z\lambda^R$. For better readability, we will simply use plain $\lambda$ and $\beta$ (i.e., without superscripts respectively) whenever we refer to both sequential and random access without explicitly distinguishing between them.

On some architectures, there is a difference between read and write bandwidth, but this difference tends to be small. Therefore, we do not distinguish between read and write bandwidth in this article.

**Address Translation** For data access, logical virtual memory addresses used by application code have to be translated to physical page addresses in the main memory of the computer. In modern CPUs, a Translation Lookaside Buffer (TLB) is used as a cache for physical page addresses, holding the translation for the most recently used pages (typically 64). If a logical address is found in the TLB, the translation has no additional costs. Otherwise, a TLB miss occurs. The more pages an application uses (which also depends on the often configurable size of the memory pages), the higher the probability of TLB misses.

The actual TLB miss latency ($l_{TLB}$) depends on whether a system handles a TLB miss in hardware or in software. With software-handled TLB, TLB miss latency can be up to an order of magnitude larger than with hardware-handled TLB. Hardware-handled TLB fetches the translation from a fixed memory structure, which is just filled by the operating system. Software-handled TLB leaves the translation method entirely to the operating system, but requires trapping to a routine in the operating system kernel on each TLB miss. Depending on the implementation and hardware architecture, TLB misses can therefore be more costly even than a main-memory access. Moreover, as address translation often requires accessing some memory structure, this can in turn trigger additional memory cache misses.

We will treat TLBs just like memory caches, using the memory page size as their cache line size, and calculating their (virtual) capacity as $\text{number of entries} \times \text{page size}$. TLBs are usually fully associative. Like caches, TLBs can be organized in multiple cascading levels.

For TLBs, there is no difference between sequential and random access latency. Further, bandwidth is irrelevant for TLBs, because a TLB miss does not cause any data transfer.

### 3.1.2.3 Unified Hardware Model

Summarizing our previous discussion, we describe a computer's memory hardware as a cascading hierarchy of $N$ levels of caches (including TLBs). We add an index $i \in \{1, \ldots, N\}$ to the parameters described above to refer to the respective value of a specific level. The relation between access latency and access bandwidth then becomes $\lambda_{i+1} = Z_i/\beta_{i+1}$. To simplify the notation, we exploit the dualism that an access to level $i+1$ is caused by a miss on level $i$. Introducing the miss latency $l_i = \lambda_{i+1}$ and the respective miss bandwidth $b_i = \beta_{i+1}$, we get $l_i = Z_i/b_i$. Each cache level is characterized by the parameters given in Table 3.1.\(^1\) In Section 3.3, we will present a

\(^1\)We assume that costs for L1 cache accesses are included in the CPU costs, i.e., $\lambda_1$ and $\beta_1$ are not used and hence undefined.
3.2 The New Bottleneck: Memory Access

<table>
<thead>
<tr>
<th>description</th>
<th>unit</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache name (level)</td>
<td>-</td>
<td>Li</td>
</tr>
<tr>
<td>cache capacity</td>
<td>[bytes]</td>
<td>Ci</td>
</tr>
<tr>
<td>cache block size</td>
<td>[bytes]</td>
<td>Zi</td>
</tr>
<tr>
<td>number of cache lines</td>
<td>-</td>
<td>#i = Ci/Zi</td>
</tr>
<tr>
<td>cache associativity</td>
<td>-</td>
<td>Ai</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sequential access</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>access bandwidth</td>
<td>[bytes/ns]</td>
<td>\beta_{i+1}^s</td>
</tr>
<tr>
<td>access latency</td>
<td>[ns]</td>
<td>\lambda_{i+1}^s = Z_i/\beta_{i+1}^s</td>
</tr>
<tr>
<td>miss latency</td>
<td>[ns]</td>
<td>\beta_i^s = \lambda_{i+1}^s</td>
</tr>
<tr>
<td>miss bandwidth</td>
<td>[bytes/ns]</td>
<td>b_i^s = \beta_{i+1}^s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>random access</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>access latency</td>
<td>[ns]</td>
<td>\lambda_{i+1}^r</td>
</tr>
<tr>
<td>access bandwidth</td>
<td>[bytes/ns]</td>
<td>\beta_{i+1}^r = Z_i/\lambda_{i+1}^r</td>
</tr>
<tr>
<td>miss bandwidth</td>
<td>[bytes/ns]</td>
<td>b_i^r = \beta_{i+1}^r</td>
</tr>
<tr>
<td>miss latency</td>
<td>[ns]</td>
<td>\lambda_{i+1}^r</td>
</tr>
</tbody>
</table>

Table 3.1: Characteristic Parameters per Cache Level \((i \in \{1, \ldots, N\})^1\)

system independent C program called Calibrator to measure these parameters on any computer hardware. We point out, that these parameters also cover the cost-relevant characteristics of disk accesses. Hence, viewing main memory (e.g., a database system's buffer pool) as cache for I/O operations, it is straightforward to include disk access in this hardware model. Where appropriate, we use level \(N + 1\) as synonym for main memory respectively secondary storage.

Though the unified hardware model is convenient for our following analysis, it sometime makes real-life examples a bit hard to read. For this reason, we will also use symbolical indices like "L1", "L2", "TLB", "Mem", or "Disk" instead of \(i\) to indicate the various levels of caches and memories.

3.2 The New Bottleneck: Memory Access

In this section, we demonstrate the severe impact of memory access costs on the performance of elementary database operations. Using a traceable example, we first gather some general observations. Then, we analyze the results in detail and develop an analytical performance model. Finally, we present the results of our experiment on a number of machines and discuss them in a broader context.
3.2.1 Initial Example

As sample query, we use a simple aggregation (say, SELECT MAX(column) FROM table) on a one-byte attribute of an in-memory table. This query performs a sequential scan over the table. By varying the record width of the table, we vary the stride, i.e., the offset between two subsequently accessed memory addresses. We keep the cardinality of the table constant at 1,000,000 tuples. We use Monet to execute the experiment on an SGI Origin2000. This system uses the MIPS R10000 processor (250 MHz) with an L1 cache of 32KB (1024 lines of 32 bytes), and has an L2 cache of 4MB (32,768 lines of 128 bytes). The detailed hardware characteristics as derived by our Calibration Tool can be found in Section 3.3.

Figure 3.3 shows the results for various strides in terms of nanoseconds per iteration. We made sure that the table was in memory, but not in any of the memory caches, by first scanning the table in question, and then multiple times scanning some other table larger than the largest cache size.

3.2.2 General Observations

When the stride is small, successive iterations in the scan read bytes that are near to each other in memory, hitting the same cache line. The number of L1 and L2 cache misses is therefore low (cf., Figure 3.4)\(^2\). The L1 miss rate reaches its maximum of one miss per iteration as soon as the stride reaches the size of an L1 cache line (32 bytes). Only the L2 miss rate increases further, until the stride exceeds the size of an L2 cache line (128 bytes). Then, it is certain that every memory read is a cache miss. Performance cannot become any worse and stays constant.

\(^2\)We used the hardware counters provided by the MIPS R10000 CPU to measure the number of cache misses.
3.2.3 Detailed Analysis

In the following, we present a detailed analysis of our experiment. Though we use the SGI Origin2000 as sample machine, we keep the models applicable to other systems as well by using a set of specific parameters to describe the respective hardware characteristics. See Section 3.1.2 for a detailed description of these parameters. In Section 3.3, we will present our Calibration Tool to measure the parameters.

In general, the execution costs per iteration of our experiment—depending on the stride $s$—can be modeled in terms of pure CPU costs (including data accesses in the on-chip L1 cache) and additional costs due to L2 cache accesses and main-memory accesses.

To measure the pure CPU costs—i.e., without any memory access costs—, we reduce the problem size to fit in L1 cache and ensure that the table is cached in L1 before running the experiment. This way, we observed $T_{CPU} = 24$ns (6 cycles) per iteration for our experiment.

We model the costs for accessing data in the L2 cache and in main memory by scoring each access with the respective latency. As observed above, the number of L2 and main memory accesses (i.e., the number of L1 and L2 misses, respectively) depends on the access stride. With a stride $s$ smaller than the cache line size $Z$, the average number of cache misses per iteration is $M(s) = \frac{s}{Z}$. With a stride equal to or larger than the cache line size, a miss occurs with each iteration. In general, we get

$$
M_{Li}(s) = \begin{cases} \frac{s}{Z_{Li}}, & \text{if } s < Z_{Li} \\ 1, & \text{if } s \geq Z_{Li} \end{cases} = \min \left\{ \frac{s}{Z_{Li}}, 1 \right\}, \quad i \in \{1, 2\}
$$

(3.1)

with $M_{Li}$ and $Z_{Li}$ ($i \in \{1, 2\}$) denoting the number of cache misses and the cache line sizes for each level, respectively. Figure 3.5 compares $M_{L1}$ and $M_{L2}$ to the measured number of cache misses.

We get the total costs per iteration—depending on the access stride—by summing the CPU costs, the L2 access costs, and the main-memory access costs:

$$
T(s) = T_{CPU} + T_{L2}(s) + T_{Mem}(s)
$$

(“model 1”) with

$$
T_{L2}(s) = M_{L1}(s) \cdot \lambda_{L2}, \quad T_{Mem}(s) = M_{L2}(s) \cdot \lambda_{Mem},
$$

where $\lambda_x$ ($x \in \{L2, Mem\}$) denote the (cache) memory access latencies for each level, respectively. We measure the L2 and memory latency with our calibration tool presented in the next section (see Table 3.2). Figure 3.6 shows the resulting curve as “model 1”.

Obviously, this model does not match the experimental results. The reason is, that the R10000 processor is super scalar and can handle up to $q = 4$ active operations concurrently. Thus, the impact of memory access latency on the overall execution time may be reduced as (1) there must be four unresolved memory requests before the CPU stalls, and (2) up to $q$ L1 or L2 cache lines may be loaded in parallel. In other words, operations may (partly) overlap. Consequently, their costs must not simply be
added. Instead, we combine two cost components \( x \) and \( y \), given the degree \( o \in [0..1] \) they overlap, using the following function:

\[
O(o, x, y) = \max\{x, y\} + (1 - o) \min\{x, y\} = x + y - o \min\{x, y\}.
\]

This overlap function forms a linear interpolation between the two extreme cases
- no overlap \( (o = 0) \Rightarrow O(0, x, y) = x + y \), and
- full overlap \( (o = 1) \Rightarrow O(1, x, y) = \max\{x, y\} \).

Let \( o_1 \) and \( o_2 \) be the degrees of overlap for L2 access and main-memory access, respectively. Then, we get the total cost — considering overlap of CPU cost and memory access costs — as follows:

\[
T = O(o_1 \cdot o_2, T_{\text{CPU}}, T_{L2} + T_{\text{Mem}}).
\]

The following consideration will help us to determine \( o_1 \) and \( o_2 \). In our experiments, we have a pure sequential memory access pattern. Up to a stride of 8 bytes, 4 subsequent memory references refer to the same 32-bytes L1 line, i.e., only one L1 line is loaded at a time, not allowing any overlap of pure calculation and memory access \( (o_1 = o_2 = 0) \). With strides between 8 and 32 bytes, \( o_1 \) linearly increases toward its maximum. The same holds for \( o_2 \) with strides between 32 and 128 bytes, as L2 lines...
contain 128 bytes on the R10000. Thus, we get

\[
o_l(s) = \max \left\{ 0, \min \left\{ 1, \frac{s - \frac{Z_{L1}}{q}}{\frac{Z_{L1}}{q}} \right\} \right\}
\]

\[
= \begin{cases} 
0, & \text{if } s \leq \frac{Z_{L1}}{q} \\
\frac{s - \frac{Z_{L1}}{q}}{\frac{Z_{L1}}{q}}, & \text{if } \frac{Z_{L1}}{q} < s < \frac{Z_{L1}}{q} \\
1, & \text{if } s \geq \frac{Z_{L1}}{q}
\end{cases} 
\]

(i \in \{1, 2\})

Similarly, up to \( q = 4 \) cache lines can be loaded concurrently during a single latency period, reducing the effective latency per cache miss to \( \frac{1}{q} \)-th. Following the previous overlap considerations, we model the effective latency depending on the stride:

\[
\lambda'_{L2}(s) = \lambda_{L2}^{\min} + (1 - o_1(s)) \cdot \left( \lambda_{L2}^{\min} - \lambda_{L2}^{\min} \right), \quad \lambda_{L2}^{\min} = \frac{\lambda_{L2}}{q}
\]

\[
\lambda'_{\text{Mem}}(s) = \lambda_{\text{Mem}}^{\min} + (1 - o_2(s)) \cdot \left( \lambda_{\text{Mem}}^{\min} - \lambda_{\text{Mem}}^{\min} \right), \quad \lambda_{\text{Mem}}^{\min} = \frac{\lambda_{\text{Mem}}}{q}
\]

Now, we can refine our model as follows:

\[
T(s) = O \left( o_1(s) \cdot o_2(s), T_{\text{CPU}}, T'_{L2}(s) + T'_{\text{Mem}}(s) \right) \tag{"model 2"}
\]

with

\[
T'_{L2}(s) = M_{L1}(s) \cdot \lambda'_{L2}(s), \quad T'_{\text{Mem}}(s) = M_{L2}(s) \cdot \lambda'_{\text{Mem}}(s),
\]

and \( T_{\text{CPU}}, M_{L1}, M_{L2}, \lambda'_{L2}, \lambda'_{\text{Mem}} \) as above.

Figure 3.6 depicts the resulting curve as "model 2". The curve fits the experimental results almost exactly for smaller strides up to \( s = 32 \). For larger strides, however, the modeled costs are significantly lower than the measured costs. When loading several cache lines concurrently we have to consider another limit: bandwidth. L2 bandwidth is large enough to allow \( q = 4 \) concurrent L1 loads within a single L2 latency period (4 \* 32 bytes within 24ns (6 cycles), i.e., \( \sim 5\text{GB/s} \)). Memory bandwidth, however, is limited to 555MB/s.\(^3\) Hence, loading four L2 lines (4 \* 128 bytes) in parallel takes at least 880ns (220 cycles), or on average \( \lambda_{\text{Mem}}^{bw} = 220\text{ns} \) (55 cycles) per line.

Replacing \( \lambda_{\text{Mem}}^{\min} \) by \( \lambda_{\text{Mem}}^{bw} \) in "model 2" yields our final "model 3". As Figure 3.6 shows, "model 3" fits the experimental curve pretty well. In this scenario, the "ideal" performance of

\[
T(s) = \max(T_{\text{CPU}}, T'_{L2}(s), T'_{\text{Mem}}(s)), \tag{"ideal"}
\]

i.e., with \( o_1 = o_2 = 1 \), is not reached (cf., "ideal" in Fig. 3.6), because the whole memory bandwidth cannot be utilized automatically for smaller strides, i.e., when several memory references refer to a single L2 line.

\(^3\)See Section 3.3, Table 3.2.
3.2.4 Discussion

The detailed analysis and the models derived, show how hardware specific parameters such as cache line sizes, cache miss penalties, and degree of CPU-inherent parallelism determine the performance of our scan experiment. We will now discuss the experiment in a broader context.

Figure 3.7 shows results of the above experiment on a number of popular workstations of the past decade. The X-axis shows the different systems ordered by their age, and per system the different strides tested. The Y-axis shows the absolute elapsed time for the experiments. For each system, the graph is split up to show which part of the elapsed time is spent waiting for memory (upper), and which part with CPU processing (lower, gray-shaded).

While all machines in Figure 3.7 exhibit the same pattern of performance degradation with decreasing data locality, Figure 3.7 clearly shows that the penalty for poor memory cache usage has dramatically increased in the last ten years. The CPU speed has improved by at least an order of magnitude, both through higher clock frequencies and through increased CPU-inherent parallelism. However, the memory cost trend in Figure 3.7 shows a mixed picture, and has clearly not kept up with the advances in CPU power. Consequently, while our experiment was still largely CPU-bound on the Sun from 1992, it is dominated by memory access costs on the modern machines (even the PentiumIII with fast memory is 75% of the time waiting for memory). Note that the later machines from Sun, Silicon Graphics and DEC actually have memory access costs that in absolute numbers are even higher than on the Sun from 1992. This can be attributed to the complex memory subsystem that comes with SMP architectures, resulting in a high memory latency. These machines do provide a high memory bandwidth—thanks to the ever growing cache line sizes\textsuperscript{4}—but this does not solve the

\textsuperscript{4}In one cache miss, the Origin2000 fetches 128 bytes, whereas the Sun LX fetches only 16; an improvement of factor 8.
latency problem if data locality is low. In fact, we must draw the sad conclusion that if no attention is paid in query processing to data locality, all advances in CPU power are neutralized due to the memory access bottleneck caused by memory latency.

The trend of improvement in bandwidth but standstill in latency [Ram96, SLD97] is expected to continue, with no real solutions in sight. The work in [Mow94] has proposed to hide memory latency behind CPU work by issuing *prefetch* instructions, before data is going to be accessed. The effectiveness of this technique for database applications is, however, limited due to the fact that the amount of CPU work per memory access tends to be small in database operations (e.g., the CPU work in our select-experiment requires only 4 cycles on the Origin2000). Another proposal [MKW+98] has been to make the caching system of a computer configurable, allowing the programmer to give a “cache-hint” by specifying the memory-access stride that is going to be used on a region. Only the specified data would then be fetched; hence optimizing bandwidth usage. Such a proposal has not yet been considered for custom hardware, however, let alone in OS and compiler tools that would need to provide the possibility to incorporate such hints for user-programs.

Our simple experiment makes clear why database systems are quickly constrained by memory access, even on simple tasks like scanning, that seem to have an access pattern that is easy to cache (sequential). The default physical representation of a tuple is a consecutive byte sequence (a “record”), which must always be accessed by the bottom operators in a query evaluation tree (typically selections or projections). The record byte-width of typical relational table amounts to some hundreds of bytes. Figure 3.7 makes clear that such large strides lead to worst-case performance, such that the memory access bottleneck kills all CPU performance advances.

To improve performance, we strongly recommend using *vertically fragmented* data structures. In Monet, we *fully* decompose relational tables on all columns, storing each in a separate Binary Association Tables (BAT). This approach is known in literature as the Decomposed Storage Model [CK85]. A BAT is represented in memory as an array of fixed-size two-field records [OID,value]—called Binary UNits (BUN)—where the OIDs are used to link together the tuples that are decomposed across different BATs. Full vertical fragmentation keeps the database records thin (8 bytes or less) and is therefore the key for reducing memory access costs (staying on the left side of the graphs in Figure 3.7). In Section 2.7, we presented specific implementation details of Monet.

### 3.2.5 Implications for Data Structures

In terms of data structures for query processing, we already noted from the simple scan experiment in Figure 3.7 that *full vertical table fragmentation* optimizes column-wise memory access to table data. This is particularly beneficial if the table is accessed in a sequential scan that reads a minority of all columns. Such table scans very often occur in both OLAP and Data Mining workloads. When record-oriented (i.e., non-fragmented) physical storage is used, such an access leads to data of the non-used columns being loaded into the cache lines, wasting memory bandwidth. In case of a vertically fragmented table, the table scan just needs to load the vertical fragments
pertaining to the columns of interest. Reading those vertical fragments sequentially achieves a 100% hit rate on all cache levels, exploiting optimal bandwidth on any hardware, including parallel memory access.

There are various ways to incorporate vertical fragmentation in database technology. In Monet, which we designed for OLAP and Data Mining workloads, vertical fragmentation is the basic building block of all physical storage, as Monet fully fragments all relations into Binary Association Tables (BATs) (see Figure 3.8). Flat binary tables are a simple set-oriented physical representation, that is not tied to a particular logical data model, yet is sufficiently powerful to represent, e.g., join indices [Val87]. Monet has successfully been used to store and query relational, object-oriented and network data structures, using this very simple data model and a small kernel of algebraic operations on it [BK99]. In Monet, we applied two additional optimizations that further reduce the per-tuple memory requirements in its BATs:

- **virtual-OIDs.** Generally, when decomposing a relational table, we get an identical system-generated column of OIDs in all decomposition BATs, which is dense and ascending (e.g., 1000, 1001, ..., 1007). In such BATs, Monet computes the OID-values on-the-fly when they are accessed using positional lookup of the BUN, and avoids allocating the 4-byte OID field. This is called a “virtual-OID” or VOID column. Apart from reducing memory requirements by half, this optimization is also beneficial when joins or semi-joins are performed on
3.3 The Calibrator: Quantification of Memory Access Costs

OID columns.\(^5\) When one of the join columns is VOID, Monet uses positional lookup instead of, e.g., hash-lookup; effectively eliminating all join costs.

- byte-encodings. Database columns often have a low domain cardinality. For such columns, Monet uses fixed-size encodings in 1- or 2-byte integer values. This simple technique was chosen because it does not require decoding effort when the values are used (e.g., a selection on a string “MAIL” can be re-mapped to a selection on a byte with value 3). A more complex scheme (e.g., using bit-compression) might yield even more memory savings, but the decoding-step required whenever values are accessed can quickly become counterproductive due to extra CPU effort. Even if decoding would just cost a handful of cycles per tuple, this would more than double the amount of CPU effort in simple database operations, like a simple aggregation from Section 3.2.1, which takes just 2 cycles of CPU work per tuple.

Figure 3.8 shows that when applying both techniques, the storage needed for 1 BUN in the “shipmode” column is reduced from 8 bytes to just one. Reducing the stride from 8 to 1 byte significantly enhances performance in the scan experiment from Figure 3.7, eliminating all memory access costs.

Alternative ways of using vertical table fragmentation in a database system are to offer the logical abstraction of relational tables but employ physically fragmentation in transposed files [Bat79] on the physical level (like in NonStopSQL [CDH*99]), or to use vertically fragmented data as a search accelerator structure, similar to a B-tree. Sybase IQ uses this approach, as it automatically creates projection indices on each table column [Syb96]. In the end, however, all these approaches lead to the same kind and degree of fragmentation.

3.3 The Calibrator: Quantification of Memory Access Costs

In order to model memory access costs in detail, we need to know the characteristic parameters of the memory system, including memory sizes, cache sizes, cache line sizes, and access latencies. Often, not all these parameters are (correctly) listed in the hardware manuals. In the following, we describe a simple but powerful calibration tool to measure the (cache) memory characteristics of an arbitrary machine.

3.3.1 Calibrating the (Cache-) Memory System

The idea underlying our calibrator tool is to have a micro benchmark whose performance only depends on the frequency of cache misses that occur. Our calibrator is a simple C program, mainly a small loop that executes a million memory reads, repeatedly sweeping over an array stored in main memory. By changing the **stride** (i.e., the

\(^{5}\)In Monet, the projection phase in query processing typically leads to additional “tuple-reconstruction” joins on OID columns that are caused by the fact that tuples are decomposed into multiple BATs.
offset between two subsequent memory accesses) and the *array size*, we force varying cache miss rates.

In principle, the occurrence of cache misses is determined by the array size. Accessing an array that fits into the L1 cache does not generate any cache misses once the data is loaded into the cache. Analogously, sweeping over an array that exceeds the L1 cache size, but still fits into L2, will cause L1 misses but no L2 misses. Finally, using an array larger than L2 causes both L1 and L2 misses.

The frequency of cache misses depends on the access stride and the cache line size. With strides equal to or larger than the cache line size, a cache miss occurs with every iteration. With strides smaller than the cache line size, a cache miss occurs only every \( n \) iterations (on average), where \( n \) is the ratio of cache line size to stride. In this latter case, each miss causes a complete cache line to be loaded into the cache, providing the data for both the request that triggered the miss and the subsequent \( n - 1 \) requests that access data within the same cache line.

Thus, we can calculate the latency for a cache miss by comparing the execution time without misses to the execution time with exactly one miss per iteration. This approach only works, if memory accesses are executed purely sequential, i.e., we have to ensure that neither two or more load instructions nor memory access and pure CPU work can overlap. We use a simple pointer chasing mechanism to achieve this: the memory area we access is initialized such that each load returns the address for the subsequent load in the next iteration. Thus, super-scalar CPUs cannot benefit from their ability to hide memory access latency by speculative execution. Further, we need to avoid that the system can benefit from prefetching. *Prefetching* depicts a mechanism where CPUs do not only load the demanded cache line, but also some cache line ahead (i.e., the subsequent cache lines in memory) although they are not requested, yet. With a sequential “forward-oriented” memory access pattern, this technique allows to (partly) overlap CPU processing and memory accesses (even without speculative execution), and hence may reduce the effective memory access latency. To disable prefetching or at least make it “useless”, the calibration tool walks “backward” through the memory. Figure 3.9 illustrates this.

To measure the cache characteristics, we run our experiment several times, varying the stride and the array size. We make sure that the stride varies at least between
3.3 The Calibrator: Quantification of Memory Access Costs

4 bytes and twice the maximal expected cache line size, and that the array size varies from half the minimal expected cache size to at least ten times the maximal expected cache size. In case the array is so big that the default 1 million iterations using the given stride do not cover the whole array, we increase the number of iterations accordingly.

We run two experiments. In the first, the insert a delay of about 100 CPU cycles between to subsequent memory accesses. Thus, we give the whole cache-memory-system and the connecting bus some time to "calm down". This tries to mimic a "once-a-while" kind of access. In the second experiment, we run without the delay, continuously issuing memory accesses. Comparing the two experiments, we find out, whether the cache and memory latencies differ between "once-a-while" and continuous memory access. To distinguish the results derived from both experiments, we call the first cache miss latencies and the latter cache replace times (derived from the fact that exclusive cache hierarchies actually do swap cache lines between adjacent cache levels; cf., Section 3.1.2).

Figure 3.10a depicts the resulting execution time (in nanoseconds) per iteration of the first experiment for different array sizes on an Origin2000 (MIPS R10000, 250 MHz = 4ns per cycle). Each curve represents a different stride. From this figure, we can derive the desired parameters as follows: Up to an array size of 32 KB, one iteration takes 8 nanoseconds (i.e., 2 cycles), independent on the stride. Here, no cache misses occur once the data is loaded, as the array completely fits in L1 cache. One of the two cycles accounts for executing the load instruction, the other one accounts for the latency to access data in L1. With array sizes between 32 KB and 4 MB, the array exceeds L1, but still fits in L2. Thus, L1 misses occur. In other words, the two steps in the curves at array sizes of 32 KB and 4 MB, respectively, indicate that there are two cache levels, L1 and L2, with sizes of 32 KB and 4 MB, respectively. The miss rate (i.e., the number of misses per iteration) depends on the stride (s) and the L1 cache line size (Z_{L1}). With \( s < \frac{Z_{L1}}{Z_{L2}} \), L1 misses occur per iteration (or one L1 miss occurs every \( \frac{Z_{L1}}{s} \) iterations). With \( s \geq Z_{L1} \), each load causes an L1 miss. Figure 3.10a shows that the execution time increases with the stride, up to a stride of 32. Then, it stays constant. Hence, L1 line size is 32 byte. Further, L1 miss latency (i.e., L2 access latency) is 32ns - 8ns = 24ns, or 6 cycles. Similarly, when the array size exceeds L2 size (4 MB), L2 misses occur. Here, the L2 line size is 128 byte, and the L2 miss latency (memory access latency) is 324ns - 32ns = 292ns, or 73 cycles.

Analogously, Figures 3.10b through 3.10d show the results for a Sun Ultra (Sun UltraSPARC, 200 MHz = 5ns per cycle), an Intel PC (Intel PentiumIII, 450 MHz = 2.22ns per cycle), and an AMD PC (AMD Athlon, 600 MHz = 1.67ns per cycle). All curves show two steps, indicating the existence of two cache levels and their sizes.

The random access memory bandwidth for our systems, listed in Table 3.2, is computed from the cache line sizes and the latencies as follows:

\[
\beta_{Mem}^{RF} = \frac{\lambda_{L2}}{\lambda_{Mem} + \frac{Z_{L2}}{Z_{L1}} \cdot \lambda_{L2}}.
\]
3 Cost Factors in MMDBMS

![Graphs showing cache sizes, line sizes, and miss latencies for different platforms: Origin 2000, Sun Ultra, Intel PC, and AMD PC.](image)

(Vertical grid lines indicate derived cache sizes, horizontal grid lines indicate derived latencies.)

Figure 3.10: Calibration Tool: Cache sizes, line sizes, and miss latencies
3.3 The Calibrator: Quantification of Memory Access Costs

(Vertical grid lines indicate derived cache sizes, horizontal grid lines indicate derived latencies.)

Figure 3.11: Calibration Tool: Cache sizes, line sizes, and replace times
The sequential access memory bandwidth $\beta^s$ is derived from the scan experiment of the previous section using a stride $s > Z_{l2}$.

Figures 3.11a through 3.11d depict the results of the second experiments for all four machines. As expected, the derived cache sizes and cache line sizes are identical to those from the first experiment. Comparing the replacement times here with the latencies above, we observe the following. For L1 misses (L2 accesses), latency and replacement time are equal on each of the four machines. All four machines use inclusive cache systems, hence we did not expect any other result. For L2 misses (i.e., main memory accesses), however, we see a different image. On the Origin2000, the Sun, and the AMD PC, replacement time is higher than latency. On the Sun and the AMD PC, the difference is just about 3% to 4% (195ns vs. 188ns and 180ns vs. 175ns, respectively); on the Origin2000, however, it is 40% (406ns vs. 292ns). On the Intel PC, replacement time is 20% less than latency (102ns vs. 123ns).

**Cache Associativity** Above, we forced capacity misses in order to measure cache sizes, line sizes and latencies. Now, to be able to measure the cache associativity, we need to force conflict misses. Assuming the usually only the lower bits of a memory address are used to determine the cache line to be used, we use rather large strides when walking through the memory array. Further using strides that are powers of 2, we ensure that the lower bits of subsequently accesses memory addresses are equal. Successively increasing the stride from 1024 through array size and varying the array size as above, conflict misses will occur as soon as the number of distinct spots accessed in the array exceeds the cache’s associativity.

Figures 3.11a through 3.11d depict the respective results for all four machines. The X-axis now gives the number of spots accessed, i.e., array size divided by stride. Again, each curve represents a different stride. We can derive the following associations. On the Origin2000, L1 is direct-mapped (1-way associative) and L2 is 2-way associative; on the Sun, L1 is direct-mapped (1-way associative) and L2 is 4-way associative; on the Intel PC, L1 and L2 are both 4-way associative; and on the AMD PC, L1 is 2-way associative and L2 is 4-way associative.

### 3.3.2 Calibrating the TLB

We use a similar approach as above to measure TLB miss costs. The idea here is to force one TLB miss per iteration, but to avoid any cache misses. We force TLB misses by using a stride that is larger than the system’s page size, and by choosing the array size such that we access more distinct spots than there are TLB entries. Cache misses will occur at least as soon as the number of spots accessed equals the number of cache lines. We cannot avoid that. But even with less spots accessed, two or more spots might be mapped to the same cache line, causing conflict misses. To avoid this, we use strides that are not exactly powers of 2, but slightly bigger, shifted by L2 cache line size, i.e., $s = 2^r + Z_{l2}$.

Figure 3.13 shows the results for four machines. Again, the X-axis gives the number of spots accessed, and each curve represents a different stride. From Figure 3.13a (Origin2000), e.g., we derive the following: Like above, we observe the base line of
3.3 The Calibrator: Quantification of Memory Access Costs

Figur ee 3.12: Calibratio n Tool: Cache associativitie s

(Vertical grid lines indicate derived associativities, horizontal grid lines indicate cache miss latencies.)
Figure 3.13: Calibration Tool: TLB entries and TLB miss costs
3.3 The Calibrator: Quantification of Memory Access Costs

8 nanoseconds (i.e., 2 cycles) per iteration. The smallest number of spots where the performance decreases due to TLB misses is 64, hence, there must be 64 TLB entries. The decrease at 64 spots occurs with strides of 32 KB or more, thus, the page size is 32 KB. Further, TLB miss latency is $236\text{ns} - 8\text{ns} = 228\text{ns}$, or 57 cycles. Figure 3.13d correctly reflects the Athlon's two TLB levels with 32 and 256 entries, respectively. The third step in the curves at 1024 spots is caused by L1 misses as L1 latency is 5 times higher than TLB latency on the Athlon. The same holds for the second step in the PentiumIII curves (Figure 3.13c) at 512 spots. On the Origin2000 and on the Sun, L1 misses also occur with more than 1024 spots accessed, but their impact is negligible as TLB latency is almost 10 times higher than L1 latency on these machines.

Due to their small size, TLBs are usually fully associative. Hence, we omit testing the TLBs' associativity.

3.3.3 Summary

Next to producing the graphs as depicted above, our calibration tool automatically analyzes the measured data and derives the desired parameters. The final output looks as follows (here: Origin2000):

<table>
<thead>
<tr>
<th>CPU loop + L1 access:</th>
<th>8.18 ns = 2 cy</th>
</tr>
</thead>
<tbody>
<tr>
<td>delay:</td>
<td>401.74 ns = 100 cy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>caches:</th>
</tr>
</thead>
<tbody>
<tr>
<td>level</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TLBs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>level</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.2 gathers the results for all four machines. The PCs have the highest L2 access latencies, probably as their L2 caches are running at only half the CPUs' clock speed. Main-memory access, however, is faster on the PCs than it is on the SGI and the Sun. The TLB miss latency of the PentiumIII and the Athlon (TLB\textsubscript{1}) are very low, as their TLB management is implemented in hardware. This avoids the costs of trapping to the operating system on a TLB miss, that is necessary in the software controlled TLBs of the other systems. The TLB\textsubscript{2} miss latency on the Athlon is comparable to that on the R10000 and the UltraSPARC. The Origin2000 has the highest memory latency, but due to its large cache lines, it achieves better sequential memory bandwidth than the Sun and the Intel PC.

The calibration tool and results for a large number of different hardware platforms are available on our web site: http://www.cwi.nl/~monet/.
### 3.4 Further Observations

In the remainder of this chapter, we discuss further aspects of main-memory access, such as parallel access and prefetching, and sketch some future hardware trends.

#### 3.4.1 Parallel Memory Access

It is interesting to note that the calibrated latencies in Table 3.2 do not always confirm the suggested latencies in the sequential scan experiment from Figure 3.7. For the PentiumIII, the access costs per memory read of 52ns at a stride of 32 bytes, and 204ns at a stride of 128 bytes for the Origin2000, are considerably lower than their memory latencies (135ns resp. 424ns), whereas in the case of the Sun Ultra, the scan measurement at L2 line size almost coincides with the calibrated memory latency. The discrepancies are caused by parallel memory access that can occur on CPUs that feature both speculative execution and a non-blocking memory system. This allows a CPU to execute multiple memory load instructions in parallel, potentially enhancing memory bandwidth above the level of cache-line size divided by latency. Prerequisites for this technique are a bus system with excess transport capacity and a non-blocking cache system that allows multiple outstanding cache misses.
3.4 Further Observations

<table>
<thead>
<tr>
<th>normal loop</th>
<th>multi-cursor</th>
<th>prefetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (int tot=i=0; i&lt;N; i++) {</td>
<td>for (int tot0=tot1=i=0, C=N/2; i&lt;C; i++) {</td>
<td>for (int tot=i=0; i&lt;N; i++) {</td>
</tr>
<tr>
<td>tot += buf[i];</td>
<td>tot0 += buf[i]; tot1 += buf[i+C];</td>
<td>#prefetch buf[i+32] freq=32</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>tot += buf[i];</td>
</tr>
</tbody>
</table>

| 5.88 cycles/addition | 3.75 cycles/addition | 3.88 → 2 cycles/addition |

Figure 3.14: Three ways to add a buffer of integers, and costs per addition on the Origin2000

To answer the question what needs to be done by an application programmer to achieve these parallel memory loads, let us consider a simple programming loop that sums an array of integers. Figure 3.14 shows three implementations, where the leftmost column contains the standard approach that results in sequential memory loads into the buf[size] array. An R10000 processor can continue executing memory load instructions speculatively until four of them are stalled. In this loop, that will indeed happen if buf[i], buf[i+1], buf[i+2], and buf[i+3] are not in the (L2) cache. However, due to the fact that our loop accesses consecutive locations in the buf array, these four memory references request the same 128-byte L2 cache line. Consequently, no parallel memory access takes place. If we assume that this loop takes 2 cycles per iteration\(^6\), we can calculate that 32 iterations cost 32*2 + 124 = 188 cycles (where 124 is the memory latency on our Origin2000); a total mean cost of 5.88 cycles per addition.

Parallel memory access can be enforced by having one loop that iterates two cursors through the buf[size] array (see the middle column of Figure 3.14). This causes 2 parallel 128-byte (=32 integer) L2 cache line fetches from memory per 32 iterations, for a total of 64 additions. On the R10000, the measured maximum memory bandwidth of the bus is 555MB/s, so fetching two 128-byte cache lines in parallel costs only 112 cycles (instead of 124 + 124). The mean cost per addition is hence 2 + 112/64 = 3.75 cycles.

It is important to note that parallel memory access is achieved only if the ability of the CPU to execute multiple instructions speculatively spans multiple memory references in the application code. In other words, the parallel effect disappears if there is too much CPU work between two memory fetches (more than 124 cycles on the R10000) or if the instructions are interdependent, causing a CPU stall before reaching the next memory reference. For database algorithms this means that random access operations like hashing will not profit from parallel memory access, as following a linked list (hash bucket chain) causes one iteration to depend on the previous; hence a

\(^6\)As each iteration of our loop consists of a memory load (buf[i]), an integer addition (of “total” with this value), an integer increment (of i), a comparison, and a branch, the R10000 manual suggests a total cost of minimally 6 cycles. However, due to the speculative execution in the R10000 processor, this is reduced to 2 cycles on the average.
memory miss will block execution. Only sequential algorithms with CPU processing costs less than the memory latency will profit, like in the simple scan experiment from Figure 3.7. This experiment reaches optimal parallel bandwidth when the stride is equal to this L2 cache line size. As each loop iteration then requests one subsequent cache line, modern CPUs will have multiple memory loads outstanding, executing them in parallel. Results are summarized at the bottom of Table 3.2, showing the parallel effect to be especially strong on the Origin2000, the PentiumIII, and the Athlon. In other words, if the memory access pattern is not sequential (like in equi-join), the memory access penalty paid on these systems is actually much higher than suggested by Figure 3.7, but determined by the latencies from Table 3.2.

3.4.2 Prefetched Memory Access

Computer systems with a non-blocking cache can shadow memory latency by performing a memory fetch well before it is actually needed. CPUs like the R10000, the PentiumIII, the Athlon, and the newer SPARC Ultra2 models have special prefetching instructions for this purpose. These instructions can be thought of as memory load instructions that do not deliver a result. Their only side effect is a modification of the status of the caches. Mowry describes compiler techniques to generate these prefetching instructions automatically [Mow94]. These techniques optimize array accesses from within loops when most loop information and dependencies are statically available, and as such are very appropriate for scientific code written in FORTRAN. Database code written in C/C++, however, does not profit from these techniques as even the most simple table scan implementation will typically result in a loop with both a dynamic stride and length, as these are (dynamically) determined by the width and length of the table that is being scanned. Also, if table values are compared or manipulated within the loop using a function call (e.g., comparing two values for equality using a C function looked up from some ADT table, or a C++ method with late binding), the unprotected pointer model of the C/C++ languages forces the compiler to consider the possibility of side effects from within that function; eliminating the possibility of optimization.

In order to provide the opportunity to still enforce memory prefetching in such situations, the MipsPRO compiler for the R10000 systems of Silicon Graphics allows passing of explicit prefetching hints by use of pragma’s, as depicted in the rightmost column of Figure 3.14. This pragma tells the compiler to request the next cache line once in every 32 iterations. Such a prefetch-frequency is generated by the compiler by applying loop unrolling (it unrolls the loop 32 times and inserts one prefetch instruction). By hiding the memory prefetch behind 64 cycles of work, the mean cost per addition in this routine is reduced to $2 + ((124-64)/32) = 3.88$ cycles. Optimal performance is achieved in this case when prefetching two cache lines ahead every 32 iterations (#prefetch buf[i+64] freq=32). The 124 cycles of latency are then totally hidden behind 128 cycles of CPU work, and a new cache line is requested every 64 cycles. This setting effectively combines prefetching with parallel memory access (two cache lines in 128 cycles instead of 248), and reduces the mean cost per addition to the minimum 2 cycles; three times faster than the simple approach.
3.4 Further Observations

3.4.3 Future Hardware Features

In spite of memory latency staying constant, hardware manufacturers have been able to increase memory bandwidth in line with the performance improvements of CPUs, by working with ever wider lines in the L1 and L2 caches. As cache lines grew wider, buses also did. The latest Sun UltraII workstations, for instance, have a 64-byte L2 cache line which is filled in parallel using a 576 bits wide PCI bus ($576 = 64 \times 8$ plus 64 bits overhead). The strategy of doubling memory bandwidth by doubling the number of DRAM chips and bus lines is now seriously complicating system board design. The Rambus [Ram96] memory standard eliminates this problem by providing an “protocol-driven memory bus”. Instead of designating one bit in the bus for one bit of data transported to the cache line, this new technology serializes the DRAM data into packets using a protocol and sends these packets over a thin (16-bit) bus that runs at very high speeds (up to 800MHz). While this allows for continued growth in memory bandwidth, it does not provide the same perspective for memory latency, as Rambus still needs to access DRAM chips, and there will still be the relatively long distance for the signals to travel between the CPU and these memory chips on the system board; both factors ensuring a fixed startup cost (latency) for any memory traffic.

A radical way around the high latencies mandated by off-CPU DRAM systems is presented in the proposal to integrate DRAM and CPU in a single chip called IRAM (Intelligent RAM) [PAC+97]. Powerful computer systems could then be built using many such chips. Finding a good model for programming such a highly parallel systems seems one of the biggest challenges of this approach. Another interesting proposal worth mentioning here has been “smarter memory” [MKW+98], which would allow the programmer to give a “cache-hint” by specifying the access pattern that is going to be used on a memory region in advance. This way, the programmer is no longer obliged to organize his data structures around the size of a cache line. Instead, the cache adapts its behavior to the needs of the application. Such a configurable system is in some sense a protocol-driven bus system, so Rambus is a step in this direction. However, both configurable memory access and IRAM have not yet been implemented in custom hardware, let alone in OS and compiler tools that would be needed to program them usefully.

Recent developments concerning memory caches are to move the L2 cache closer to the CPU, either locate it on the same multi-chip module (e.g., Intel’s first PentiumII “Katmai”, or AMD’s first Athlon generation) or even include it on the CPU’s die (e.g., Intel’s latest PentiumII “Coppermine”, or AMD’s latest Athlon “Thunderbird”). While reducing L2 latency — the L2 caches now operate at half or even full CPU speed — these trends do not reduce the memory latency. Further, on-chip caches are usually smaller than off-chip caches and hence provide even less potential to avoid memory accesses. Similarly, additional L3 caches — although increasing the total cache capacity — cannot reduce memory latency, but rather might even increase it due to an increased management overhead.

Concerning CPU technology, it is anticipated [Sem97] that the performance advances dictated by Moore’s law [Moo65] will continue well into the millennium.
However, performance increase will also be brought by more parallelism within the CPU. The upcoming IA-64 architecture has a design called Explicitly Parallel Instruction Computing (EPIC) [ACM*98], which allows instructions to be combined in bundles, explicitly telling the CPU that they are independent. The IA-64 is specifically designed to be scalable in the number of functional units, so while newer versions are released, more and more parallel units will be added. This means that while current PC hardware uses less parallel CPU execution than the RISC systems, this will most probably change in the new 64-bit PC generation.

Summarizing, we have identified the following ongoing trends in modern hardware:

- CPU performance keeps growing with Moore’s law for years to come.
- A growing part of this performance increase will come from parallelism within the CPU.
- New bus technology will provide sufficient growth in memory bandwidth.
- Memory latency will not improve significantly.

This means that the failure of current DBMS technology to properly exploiting memory and CPU resources of modern hardware [ADHW99, KPH’98, BGB98, TLPZT97] will grow worse. Modern database architecture should therefore take these new hardware issues into account. With this motivation, we investigate in the following new approaches to large main-memory equi-joins, that specifically aim at optimizing resource utilization of modern hardware.