Understanding, modeling, and improving main-memory database performance

Manegold, S.

Citation for published version (APA):
Chapter 4

Generic Database Cost Models for Hierarchical Memory Systems

Accurate prediction of operator execution time is a prerequisite for database query optimization. Although extensively studied for conventional disk-based DBMSs, cost modeling in main-memory DBMSs is still an open issue. Recent database research has demonstrated that memory access is more and more becoming a significant—if not the major—cost component of database operations. If used properly, fast but small cache memories—usually organized in cascading hierarchy between CPU and main memory—can help to reduce memory access costs. However, they make the cost estimation problem more complex.

In this chapter, we propose a generic technique to create accurate cost functions for database operations. We identify a few basic memory access patterns and provide cost functions that estimate their access costs for each level of the memory hierarchy. The cost functions are parameterized to accommodate various hardware characteristics appropriately. Combining the basic patterns, we can describe the memory access patterns of database operations. The cost functions of database operations can automatically be derived by combining the basic patterns’ cost functions accordingly.

To validate our approach, we performed experiments using our DBMS prototype Monet. The results presented here confirm the accuracy of our cost models for different operations.

Aside from being useful for query optimization, our models provide insight to tune algorithms not only in a main-memory DBMS, but also in a disk-based DBMS with a large main-memory buffer cache.
4.1 Related Work and Historical Development

Database cost models provide the foundation for query optimizers to derive an efficient execution plan. Such models consist of two parts: a logical and a physical component. The former is geared toward estimation of the data volumes involved. Usually, statistics about the data stored in the database are used to predict the amount of data that each operator has to process. The underlying assumption is that a query plan that has to process less data will also consume less resources and/or take less time to be evaluated. The logical cost component depends only on the data stored in the database, the operators in the query, and the order in which these operators are to be evaluated (as specified by the query execution plan). Hence, the logical cost component is independent of the algorithm and/or implementation used for each operator.

The problem of (intermediate) result size estimation has been intensively studied in literature (cf., Section 2.2). In this thesis, we focus on the physical cost component. Therefore, we assume a perfect oracle to predict the data volumes.

Given the data volumes, the physical cost component is needed to discriminate the costs of the various algorithms and implementations of each operator. The query optimizer uses this information to choose the most suitable algorithm and/or implementation for each operator.

Given the fact that disk-access used to be the predominant cost factor, early physical cost functions just counted the number of I/O operations to be executed by each algorithm [Gra93]. Any operation that loads a page from disk into the in-memory buffer pool or writes a page from the buffer back to disk is counted as an I/O operation. However, disk systems depict significant differences in cost (in terms of time) per I/O operation depending on the access pattern. Sequentially reading or writing consecutive pages causes less cost per page than accessing scattered pages in a random order. Hence, more accurate cost models discriminate between random and sequential I/O. The cost for sequential I/O is calculated as the data volume\(^1\) divided by the I/O bandwidth. The cost for random I/O additionally considers the seek latency per operation.

With memory chips dropping in price while growing in capacity, main memory sizes grow as well. Hence, more and more query processing work is done in main memory, trying to minimize disk access as far as possible in order to avoid the I/O bottleneck. Consequently, the contribution of pure CPU time to the overall query evaluation time becomes more important. Cost models are extended to model CPU costs, usually in terms of CPU cycles (scored by the CPU's clock speed to obtain the elapsed time).

CPU cost used to cover memory access costs [LN96, WK90]. This implicitly assumes that main memory access costs are uniform, i.e., independent of the memory address being accessed and the order in which different data items are accessed. However, recent database research has demonstrated that this assumption does not hold (anymore) [ADHW99, BMK99]. With hierarchical memory systems being used, access latency varies significantly, depending on whether the requested data can be

\(^1\)i.e., number of sequential I/O operations multiplied by the page size
found in (any) cache, or has to be fetch from main memory. The state (or contents) of the cache(s) in turn depends on the applications’ access patterns, i.e., the order in which the required data items are accessed. Furthermore, while CPU speed is continuously experiencing an exponential growth, memory latency has hardly improved over the last decade.\(^2\) Our detailed analysis of these issues in Section 3.2 comes to the conclusion that memory access has become a significant cost factor—not only for main memory databases—which cost models need to reflect.

In query execution, the memory access issue has been addressed by designing new cache-conscious data structures [RR99, RR00, ADHS01] and algorithms [SKN94, MBK00b]. On the modeling side, however, nothing has been published yet considering memory access appropriately.

### 4.2 Outline

In this chapter, we address the problem of how to model memory access costs of database operators appropriately. As it turns out to be quite complicated to derive proper memory access cost functions for various operations, we develop a new technique to automatically derive such cost functions. The basic idea is to describe the data access behavior of an algorithm in terms of a combination of basic access patterns (such as "sequential" or "random"). The actual cost function is then obtained by combining the patterns’ cost functions (as derived in this chapter) appropriately. Using a unified hardware model that covers the cost-related characteristics of both main memory and disk access, it is straightforward to extend our approach to consider I/O cost as well. Gathering I/O and memory cost models into a single common framework is a new approach that simplifies the task of generating accurate cost functions.

Section 4.3 presents a simplified abstract representation of data structures and identifies a number of basic access patterns to be performed on such data structures. Equipped with these tools, we show how to specify the data access patterns of database algorithms by combining basic patterns. In Section 4.4, we derive the cost function for our basic access patterns and Section 4.5 provides rules how to obtain the cost functions of database algorithms from their representation introduced in Section 4.3. Section 4.7 contains some experimental results validating the obtained cost functions and Section 4.8 will draw some conclusions.

### 4.3 The Idea

Our recent work on main-memory database algorithms suggests that memory access cost can be modeled by estimating the number of cache misses \(M\) and scoring them with their respective miss latency \(l\) [MBK02]. This approach is similar to the one used for detailed I/O cost models. The hardware discussion in Section 3.1 shows, that

\(^2\)Wide buses and raised clock speeds, such as with DDR-SDRAM or RAMBUS, help to keep memory bandwidth growing at almost the pace of CPU speed, however, these techniques do not improve memory access latency. See also Section 3.2.
also for main-memory access, we have to distinguish between sequential and random access patterns. However, in contrary to disk access, we now have multiple levels of cache with varying characteristics. Hence, the challenge is to predict the number and kind of cache misses for all cache levels. Our hypothesis is, that we can treat all cache levels individually, though equally, and calculate the total cost as the sum of the cost for all levels:

$$T_{Mem} = \sum_{i=1}^{N} (M_i^s \cdot l_i^s + M_i^f \cdot l_i^f).$$  \hspace{1cm} (4.1)

With the hardware modeled as described in Section 3.1 and the hardware parameters measured by our calibration tool (see Section 3.3), the remaining challenge is to estimate the number and kind of cache misses per cache level for various database algorithms. The task is similar to estimating the number and kind of I/O operations in traditional cost models. However, our goal is to provide a generic technique for predicting cache miss rates of various database algorithms. Nevertheless, we want to sacrifice as little accuracy as possible to this generalization.

To achieve the generalization, we introduce two abstractions. Our first abstraction is a unified description of data structures. We call it data regions. The second are basic data access patterns. Both of them are driven by the goal to keep the models as simple as possible, but as detailed as necessary. Hence, we try to ignore any details that are not significant for our purpose (predicting cache miss rates) and only focus on the relevant parameters. The following paragraphs will present both abstractions in detail.

4.3.1 Data Regions

We model data structures as data regions. $D$ denotes the set of data regions. A data region $R \in D$ consists of $|R|$ data items of size $\overline{R}$ (in bytes). We call $|R|$ the length of region $R$, $\overline{R}$ its width and $||R|| = |R| \cdot \overline{R}$ its size. Further, we define the number of cache lines covered by $R$ as $|C_R|_Z = ||R||/Z$, and the number of data items that fit in the cache as $|C_R|_Z = \lceil C/\overline{R} \rceil$.

A (relational) database table is hence represented by a region $R$ with $|R|$ being the table’s cardinality and $\overline{R}$ being the tuple size (or width). Similarly, more complex structures like trees are modeled by regions with $|R|$ representing the number of nodes and $\overline{R}$ representing the size (width) of a single node.

4.3.2 Basic Access Patterns

Data access patterns vary in their referential locality and hence in their cache behavior. Thus, not only the cost (latency) of cache misses depend on the access pattern, but also the number of cache misses that occur. Each database algorithm describes a different data access pattern. This means, each algorithm requires an individual cost function to predict its cache misses. Deriving each cost function "by hand" is not only exhaustive and time consuming, but also error-prone. Our hypothesis is that we only need to specify the cost functions of a few basic access patterns. Given
these basic patterns and their cost functions, we could describe the access patterns of
database operations as combinations of basic access patterns, and derive the resulting
cost functions automatically.

In order to identify the relevant basic access patterns, we first have to analyze
the data access characteristics of database operators. We classify database operations
according to the number of operands.

Unary operators—such as, e.g., table scan, selection, projection, sorting, hashing,
aggregation, or duplicate elimination—read data from one input region and write data
to one output region. Data access can hence be modeled by two cursors, one for the
input and one for the output. The input cursor traverses the input region sequentially.
For table scan, selection, and projection, the output cursor also simply progresses
sequentially with each output item. When building a hash table, the output cursor
"hops back and forth" in a non-sequential way. In practice, the actual pattern is not
completely random, but rather depends on the physical order and attribute value distri-
bution of the input data as well as on the hash function. In our case, i.e., knowing
only the algorithm, but not the actual data, it is not possible to make more accurate
(and usable) assumptions about the pattern described by the output cursor. Hence,
we assume that the output region is accessed in a completely random manner. This
assumption should not be too bad, as a "good" hash function typically destroys any
sorting order and tends/tries to level out skew data distributions.

Sort algorithms typically perform a more complicated data access pattern. In Sec-
tion 4.7.2, we will present quick-sort as an example to demonstrate how such patterns
can be specified as combinations of basic patterns. Aggregation and duplicate elimi-
nation are usually implemented using sorting or hashing. Thus, they incur the respective
patterns.

Though also a unary operation, data partitioning takes a separate role. Again, the
input region is traversed sequentially. However, modeling the output cursor's access
pattern as purely random is too simple. In fact, we can do better. Suppose, we want
to partition the input region into \( m \) output regions. Then, we know that the access
within each region is sequential. Hence, we model the output access as a nested pat-
tern. Each region is a separate local cursor, performing a sequential pattern. A single
global cursor hops back and forth between the regions. Similar to the hashing scenario
described before, the order in which the different region-cursors are accessed—i.e., the
global pattern—depends on the partitioning criterion (e.g., hash- or range-based) and
the physical order and attribute value distribution of the input data. Again, it is not
possible to model these dependencies in a general way without detailed knowledge
about the actual data to process. Purely from the algorithm, we can only deduce a
random order.

Concerning binary operations, we focus our discussion on join. The appropriate
treatment of union, intersection and set-difference can be derived respectively. Binary
operators have two inputs and a single output. In most cases, one input—we call it left
or outer input—is traversed sequentially. Access to the other—right or inner—input
depends on the algorithm and the data of the left input. A nested loop join performs
a complete sequential traversal over the whole inner input for each outer data item. A
merge join—assuming both inputs are already sorted—sequentially traverses the inner
input once while the outer input is traversed. A hash join—provided there is already a hash table on the inner input—performs an "un-ordered" access pattern on the inner input’s hash table. As discussed above, we assume a uniform random access.

From this discussion, we identify the following basic access patterns as eminent in the majority of relational algebra implementations. Let $R \in \mathbb{D}$ be a data region.

**single sequential traversal:** $s_{\text{trav}}(R, u)$

A sequential traversal sequentially sweeps over $R$, accessing each data item in $R$ exactly once. The optional parameter $u$ gives the number of bytes that are actually used of each data item. If not specified, we assume that all bytes are used, i.e., $u = R$. If specified, we require $0 < u \leq R$. $u$ is used to model the fact that an operator, e.g., an aggregation or a projection (either as separate operator or in-lined with another operator), accesses only a subset of its input’s attributes. For simplicity of presentation, we assume that we always access $u$ consecutive bytes. Though not completely accurate, this is a reasonable abstraction in our case.\(^3\) Figure 4.1 shows a sample sequential traversal.

**repetitive sequential traversal:** $r_{s_{\text{trav}}}(r, d, R, [, u])$

A repetitive sequential traversal performs $r$ sequential traversals over $R$ after another. $d$ specifies, whether all traversals sweep over $R$ in the same direction, or whether subsequent traversals go in alternating directions. The first case—*uni-directional*—is specified by $d = \text{uni}$. The second case—*bi-directional*—is

---

\(^3\)In case the $u$ bytes are rather somehow spread across the whole item width $R$, say as $k$ times $u'$ bytes ($k \cdot u' = u$), one can replace $s_{\text{trav}}(R, u)$ by $s_{\text{trav}}(R', u')$ with $R' = R/k$ and $|R'| = |R| \cdot k$. 

---

Figure 4.1: Single Sequential Traversal: $s_{\text{trav}}(R, u)$

Figure 4.2: Single Random Traversal: $r_{\text{trav}}(R, u)$
4.3 The Idea

Figure 4.3: Interleaved Multi-Cursor Access: \( \text{nest}(R, m, \text{s.trav}(R, u), \text{seq}, \text{bi}) \)

specified by \( d = \text{bi} \).

**single random traversal:** \( \text{r.trav}(R[, u]) \)
Like a sequential traversal, a random traversal accesses each data item in \( R \) exactly once, reading or writing \( u \) bytes. However, the data items are not accessed in the order they are stored, but rather randomly. Figure 4.2 depicts a sample random traversal.

**repetitive random traversal:** \( \text{rr.trav}(r, R[, u]) \)
A repetitive random traversal performs \( r \) random traversals over \( R \) after another. We assume that the permutation orders of two subsequent traversals are independent of each other. Hence, there is no point in discriminating uni-directional and bi-directional accesses, here. Therefore, we omit parameter \( d \).

**random access:** \( \text{r.acc}(r, R[, u]) \)
Random access hits \( r \) randomly chosen data items in \( R \) after another. We assume, that each data item may be hit more than once, and that the choices are independent of each other. Even with \( r \geq |R| \) we do not require that each data item is accessed at least once.

**interleaved multi-cursor access:** \( \text{nest}(R, m, \mathcal{P}, O[, D]) \)
A nested multi-cursor access models a pattern where \( R \) is divided into \( m \) (equal-sized) sub-regions. Each sub-region has its own local cursor. All local cursors perform the same basic pattern, given by \( \mathcal{P} \). \( O \) specifies, whether the global cursor picks the local cursors randomly \( (O = \text{ran}) \) or sequentially \( (O = \text{seq}) \). In
In the latter case, $D$ specifies, whether all traversals of the global cursor across the local cursors use the same direction ($D = \text{uni}$), or whether subsequent traversals use alternating directions ($D = \text{bi}$). Figure 4.3 shows a sample interleaved multi-cursor access.

A similar idea has been used by Chou and DeWitt in their buffer management algorithm $\text{DBMIN}$ [CD85]. $\text{DBMIN}$ is based on a model for relational query behavior called $\text{query locality set model (QLSM)}$. QLSM is founded on the observation that basic database operations (like scans, index scans, joins, etc.) could be characterized by a limited number of reference patterns to database pages. Chou and DeWitt propose three classes of reference patterns: sequential, random, and hierarchical. $\text{DBMIN}$ exploits the information provided by QLSM to choose the most suitable pages replace strategy and estimate the proper buffer size to be used for each relation in a given query.

### 4.3.3 Compound Access Patterns

Database operations access more than one data region, usually at least their input(s) and their output. This means, they perform more complex data access patterns than the basic ones we introduced in the previous section. In order to model these complex patterns, we now introduce $\text{compound data access patterns}$. Unless we need to explicitly distinguish between basic and compound data access patterns, we refer to both as data access patterns, or simply patterns. We use $P_b, P_c,$ and $P = P_b \cup P_c$ to denote the set of basic access patterns, compound access patterns, and all access patterns, respectively. We require $P_b \cap P_c = \emptyset$.

Be $P_1, \ldots, P_p \in P$ ($p > 1$) data access patterns. There are two principle ways to combine two or more patterns. Either the patterns are executed one after the other or they are executed concurrently. We call the first combination $\text{sequential execution}$ and denote it by operator $\oplus : P \rightarrow P$; the second combination represents $\text{concurrent execution}$ and is denoted by operator $\circ : P \rightarrow P$. The result of either combination is again a (compound) data access pattern. Hence, we can apply $\oplus$ and $\circ$ repeatedly to describe complex patterns. By definition, $\circ$ is commutative, while $\oplus$ is not. In case both $\circ$ and $\oplus$ are used to describe a complex pattern, $\circ$ has precedence over $\oplus$, i.e.,

$$P_1 \circ P_2 \circ P_3 \oplus P_4 \circ P_5 \oplus P_6 \equiv ((P_1 \circ P_2 \circ P_3) \oplus (P_4 \circ P_5) \oplus P_6).$$

We use bracketing to overrule these assumptions or to avoid ambiguity. Further, we use the following notation to simplify complex terms where necessary and appropriate:

$$\oplus \in \{\oplus, \circ\} : \quad P_1 \circ \ldots \circ P_p \equiv \oplus(P_1, \ldots, P_p) \equiv \bigoplus_{q=1}^{p}(P_q).$$

Table 4.1 gives some examples how to describe the access patterns of some typical database algorithms as compound patterns. For convenience, some re-occurring compound access patterns are assigned a new name.
4.4 Deriving Cost Functions

<table>
<thead>
<tr>
<th>algorithm</th>
<th>pattern description</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W \leftarrow \text{select}(U)$</td>
<td>$s_{m_{i}}(U) \circ s_{m_{i}}(W)$</td>
<td></td>
</tr>
<tr>
<td>$W \leftarrow \text{nested}<em>\text{loop}</em>\text{join}(U, V)$</td>
<td>$s_{m_{i}}(U) \circ rs_{m_{i}}(</td>
<td>U</td>
</tr>
<tr>
<td>$W \leftarrow \text{zick}<em>\text{zack}</em>\text{join}(U, V)$</td>
<td>$s_{m_{i}}(U) \circ rs_{m_{i}}(</td>
<td>U</td>
</tr>
<tr>
<td>$V' \leftarrow \text{hash}_\text{build}(V)$</td>
<td>$s_{m_{i}}(V) \circ r_{m_{i}}(V')$</td>
<td>$\Rightarrow \text{build}_\text{hash}(V, V')$</td>
</tr>
<tr>
<td>$W \leftarrow \text{hash}_\text{probe}(U, V')$</td>
<td>$s_{m_{i}}(U) \circ r_{m_{i}}(</td>
<td>U</td>
</tr>
<tr>
<td>$W \leftarrow \text{hash}_\text{join}(U, V)$</td>
<td>$\text{build}<em>\text{hash}(V, V') \circ \text{probe}</em>\text{hash}(U, V', W)$</td>
<td>$\Rightarrow \text{h}_\text{join}(U, V, W)$</td>
</tr>
<tr>
<td>$</td>
<td>U</td>
<td>_{j=1} \sim \text{cluster}(U, m)$</td>
</tr>
<tr>
<td>$W \leftarrow \text{part}<em>\text{nl}</em>\text{join}(U, V, m)$</td>
<td>$\text{part}(U, m,</td>
<td>U</td>
</tr>
<tr>
<td>$W \leftarrow \text{part}<em>\text{h}</em>\text{join}(U, V, m)$</td>
<td>$\text{part}(U, m,</td>
<td>U</td>
</tr>
</tbody>
</table>

*nested-loop-join with alternating traversal direction on inner table, aka. "boustrophedonism"

Table 4.1: Sample Data Access Patterns $(U, V, V', W \in \mathbb{D})$

Our hypothesis is, that we only need to provide an access pattern description as depicted in Table 4.1 for each operation we want to model. The actual cost function can then be created automatically, provided we know the cost functions for the basic patterns, and the rules how to combine them. To verify this hypothesis, we will now first estimate the cache miss rates of the basic access patterns and then derive rules how to calculate the cache miss rates of compound access patterns.

4.4 Deriving Cost Functions

In the following sections, $N$ depicts the number of cache levels and $i$ iterates over all levels: $i \in \{1, \ldots, N\}$. For better readability, we will omit the index $i$ wherever we do not refer to a specific cache level, but rather to all or any.

4.4.1 Preliminaries

For each basic pattern, we need to estimate both sequential and random cache misses for each cache level. Given an access pattern $\mathcal{P} \in \mathcal{P}$, we describe the number of misses per cache level as pair

$$\tilde{M}_i(\mathcal{P}) = (M^s_i(\mathcal{P}), M^r_i(\mathcal{P})) \in \mathbb{N} \times \mathbb{N}$$  \hspace{1cm} (4.2)
containing the number of sequential and random cache misses. Obviously, the random patterns cause only random misses, but no sequential misses. Consequently, we always set

\[ M^s_i(T) = 0 \quad \text{for} \quad T \in \{r_{\text{trav}}, rr_{\text{trav}}, r_{\text{acc}}\}. \]

Sequential traversals can achieve sequential latency (i.e., exploit full excess bandwidth), only if all the requirements listed in Section 3.1.2.2 are fulfilled. Sequential access is fulfilled by definition. The hardware requirements (non-blocking caches and super-scalar CPUs allowing speculative execution) are covered by the results of our calibration tool. In case these properties are not given, sequential latency will be the same as random latency. However, the pure existence of these hardware features is not sufficient to achieve sequential latency. Rather, the implementation needs to be able to exploit these features. Data dependencies in the code may keep the CPU from issuing multiple memory requests concurrently. It is not possible to deduce this information only from the algorithm without knowing the actual implementation. But even without data dependencies, multiple concurrent memory requests may hit the same cache line. In case the number of concurrent hits to a single cache line is lower than the maximal number of outstanding memory references allowed by the CPU, only one cache line is loaded at a time.\(^4\) Though we can say how many subsequent references hit the same cache line (see below), we do not know how many outstanding memory references the CPU can handle without stalling.\(^5\) Hence, it is not possible to automatically guess, whether a sequential traversal can achieve sequential latency or not. For this reason, we offer two variants of \(s_{\text{trav}}\) and \(rs_{\text{trav}}\). \(s_{\text{trav}}\) and \(rs_{\text{trav}}\) assume a scenario that can achieve sequential latency while \(s_{\text{trav}}^r\) and \(rs_{\text{trav}}^r\) do not. The actual number of misses is equal in both cases. However, in the first case, we get only sequential but no random misses, while the second case causes only random but no sequential misses:

\[ M^s_i(T^s) = M^s_i(T^r) = 0 \quad \text{for} \quad T \in \{s_{\text{trav}}, rs_{\text{trav}}\}. \]

Unless we need to explicitly distinguish between both variants, we will use \(s_{\text{trav}}^r\) respectively \(rs_{\text{trav}}^r\) to refer to both \((x \in \{s, r\})\). When describing the access pattern of a certain algorithm, we will use the variant that fits to the actual code.

### 4.4.2 Single Sequential Traversal

Be \(R\) a data region and \(P = s_{\text{trav}}^x(R, u) \ (x \in \{s, r\})\) a sequential traversal over \(R\). As mentioned above, we have

\[ M^x_i(s_{\text{trav}}^x(R, u)) = 0 \quad \text{and} \quad M^x_i(s_{\text{trav}}^x(R, u)) = 0. \]

To calculate \(M^x_i(s_{\text{trav}}^x(R, u))\), we distinguish two cases: \(\overline{R} - u < Z\) and \(\overline{R} - u \geq Z\).

\(^4\)For a more detailed discussion, we refer the interested reader to Section 3.2.

\(^5\)Our calibration results can only indicate, whether the CPU can handle outstanding memory references without stalling, but not how many it can handle concurrently.
4.4 Deriving Cost Functions

Figure 4.4: Impact of \( \overline{R} - u \) on the Number of Cache Misses

**Case** \( \overline{R} - u < Z \). In this case, the gap between two adjacent accesses that is not touched at all is smaller than a single cache line. Hence, the cache line containing this gap is loaded to serve at least one of the two adjacent accesses (cf., Fig. 4.4a). Thus, during a sweep over \( R \) with \( \overline{R} - u < Z \) all cache lines covered by \( R \) have to be loaded, i.e.,

\[
M^*_i(s_{,\text{trav}}^i(R, u)) = |R| Z_i.
\]  

**Case** \( \overline{R} - u \geq Z \). In this case, the gap between two adjacent accesses that is not touched at all spans at least a complete cache line. Hence, not all cache lines covered by \( R \) have to be loaded during a sweep over \( R \) with \( \overline{R} - u \geq Z \) (cf., Fig. 4.4b). Further, no access can benefit from a cache line already loaded by a previous access to another spot. Thus, each access to an item in \( R \) requires at least \( \left\lceil \frac{u}{Z} \right\rceil \) cache lines to be loaded. We get

\[
M^*_i(s_{,\text{trav}}^i(R, u)) \geq |R| \cdot \left\lceil \frac{u}{Z_i} \right\rceil.
\]

However, with \( u > 1 \) it may happen that — depending on the alignment of \( u \) within a cache line — one additional cache line has to be loaded per access. Figure 4.5 depicts such a scenario.

The actual alignment of each \( u \) in a sweep is determined by two parameters. First, it of course depends on the alignment of the first item in \( R \), i.e., the alignment of \( R \) itself. Assuming a 1 byte access granularity, \( R \) can be aligned on \( Z \) places within a cache line. Second, \( \overline{R} \) determines whether all items in \( R \) are aligned equally, or whether their alignment changes throughout \( R \). In case \( \overline{R} \) is a multiple of \( Z \), all items in \( R \) are equally aligned as the first one. Otherwise, the alignment varies throughout \( R \), but picking only \( \frac{Z}{\gcd(|Z|, R)} \) out of the \( Z \) theoretically possible places. As we do not know anything about the alignment of \( R \), there is no way of reasonably exploiting the information we just learned about the impact of \( \overline{R} \) on the alignment shift. Hence, all we can do is assuming that all \( Z \) possibilities occur equally often. All we need to do now, is count how many of these \( Z \) possibilities yield an additional cache miss. For
convenience, we define

\[
x \mod y = \begin{cases} 
y, & \text{if } x \mod y = 0, \\
x \mod y, & \text{else.}
\end{cases}
\]

When \( u \) is aligned on the first position (i.e., the first byte) in a cache line, no more than \( \left\lfloor \frac{u}{Z} \right\rfloor \) cache lines have to be loaded to access whole \( u \) (cf., Fig. 4.5). In this case, the last \( Z - (u \mod Z) \) bytes in the last cache line loaded for \( u \) are not used by \( u \). Hence, shifting \( u \)'s position by up to \( Z - (u \mod Z) \) also does not require any additional cache miss. Only the remaining

\[
Z - 1 - (Z - (u \mod Z)) \\
= (u \mod Z) - 1 \\
= \begin{cases} 
Z - 1, & \text{if } u \mod Z = 0 \\
(u \mod Z) - 1, & \text{else}
\end{cases}
\]

positions will yield one additional cache miss (cf., Fig. 4.5). Putting all pieces together, we get:
4.4 Deriving Cost Functions

\[ M_f^f(s_{\text{trav}}(R, u)) = |R| \cdot \left(\left\lfloor \frac{u}{Z_t} \right\rfloor + \frac{(u - 1) \mod Z_t}{Z_t} \right). \quad (4.4) \]

Figure 4.6 demonstrates the impact of \( u \) on the number of cache misses. The points show the number of cache misses measured with various alignments. "align = 0" and "align = -1" make up the two extreme cases. In the first case, \( u \) is aligned on the first byte of a cache line; in the second case, \( u \) starts on the last byte of a cache line. "average" depicts the average over all possible alignments. The dotted curve and the dashed curve represent Equations (4.5) and (4.3), respectively, which ignore \( u \) and assume that all \( R \) bytes of each item are touched. The solid curve represents the identical Equations (4.4) and (4.6) which consider \( u \). The graphs show that \( u \) has a significant impact on the number of cache misses, and that our formulas correctly predict the average impact.

4.4.3 Single Random Traversal

Be \( R \) a data region and \( P = r_{\text{trav}}(R, u) \) a random traversal over \( R \). As mentioned above, we have

\[ M_f^F(r_{\text{trav}}(R, u)) = 0. \]

Like with sequential traversal, we distinguish two cases: \( \overline{R} - u < Z \) and \( \overline{R} - u \geq Z \).

**Case \( \overline{R} - u < Z \).** With the untouched gaps being smaller than cache line size, again all cache lines covered by \( R \) have to be accessed. Hence, \( M_f^F(P) \geq |R|Z \). But due to the random access pattern, two locally adjacent accesses are not temporally adjacent. Thus, if \( ||R|| \) exceeds the cache size, a cache line that serves two or more (locally adjacent) accesses may be replaced by another cache line before all accesses that require it actually took place. This in turn causes an additional cache miss, once the original cache line is accessed again. Of course, such additional cache misses only occur, once the cache capacity is exceeded, i.e., after \( \min \{\#i, |C_i|_R\} \) spots have been accessed. The probability that a cache line is removed from the cache although it will be used for another access increases with the size of \( R \). In the worst case, each access causes an additional cache miss. Hence, we get

\[ M_f^f(r_{\text{trav}}(R, u)) = |R|Z_t + \left( |R| - \min \{\#_i, |C_i|_R\} \right) \cdot \left( 1 - \min \left\{1, \frac{|C_i|_R}{||R||} \right\} \right). \quad (4.5) \]

**Case \( \overline{R} - u \geq Z \).** Each spot is touched exactly once, and as adjacent accesses cannot benefit from previously loaded cache lines, we get the same formula as for sequential access:

\[ M_f^F(r_{\text{trav}}(R, u)) = |R| \cdot \left(\left\lfloor \frac{u}{Z_t} \right\rfloor + \frac{(u - 1) \mod Z_t}{Z_t} \right). \quad (4.6) \]
Figure 4.6: Impact of $u$ and its Alignment on the Number of Cache Misses

($|R| = 1,000,000$, $\bar{R} = 320$ B, $Z_1 = 32$ B, $Z_2 = 128$ B)
4.4 Deriving Cost Functions

4.4.4 Discussion

Comparing the final formulas for sequential and random traversals, we can derive the following relationships and invariants. Figure 4.7 visualizes some of the effects. The points represent the measured cache misses, while the lines represent the estimations of our formulas.

\[ R - u < Z_i \land ||R|| \leq C_i \Rightarrow M^A_t(s_{trav^x}(R, u)) = M^F_t(r_{trav}(R, u)); \]
\[ R - u < Z_i \land ||R|| > C_i \Rightarrow M^A_t(s_{trav^x}(R, u)) < M^F_t(r_{trav}(R, u)). \]

With untouched gaps smaller than cache lines, random traversals cause as many misses as sequential traversals as long as \( R \) fits in the cache, but more, if \( R \) exceeds the cache (cf., Figure 4.7a vs. 4.7c & 4.7b vs. 4.7d).

\[ R - u \geq Z_i \Rightarrow M^A_t(s_{trav^x}(R, u)) = M^F_t(r_{trav}(R, u)). \]

With untouched gaps larger than cache lines, random traversals cause as many misses as sequential traversals.

\[ R - u < Z_i \Rightarrow M^A_t(s_{trav^x}(R, u)) = M^F_t(s_{trav^x}(R', u')) \forall R', u' \text{ with } ||R'|| = ||R|| \land \overline{R} - u' < Z_i. \]

With untouched gaps smaller than cache lines, sequential traversals depend only on the size of \( R \), but are invariant to varying item size (and hence number of items) and bytes touched per item (cf., Figure 4.7a & 4.7b).

\[ R - u < Z_i \land ||R|| \leq C_i \Rightarrow M^F_t(r_{trav}(R, u)) = M^F_t(r_{trav}(R', u')) \forall R', u' \text{ with } ||R'|| = ||R|| \land \overline{R} - u' < Z_i; \]
\[ R - u < Z_i \land ||R|| > C_i \Rightarrow M^F_t(r_{trav}(R, u)) = M^F_t(r_{trav}(R, u')) \forall u' \text{ with } R - u' < Z_i. \]

For random traversals, the invariance to item size holds only if \( R \) entirely fits in the cache (cf., Figure 4.7c & 4.7d).

\[ R - u \geq Z_i \Rightarrow M^A_t(T(R, u)) = M^A_t(T(R', u)) \forall R' \text{ with } |R'| = |R| \land \overline{R} - u \geq Z_i, \]
\[ T \in \{s_{trav^x}, r_{trav}\}. \]

With untouched gaps larger than cache lines, the number of misses of all traversals depend only on the number of items accessed and the number of bytes touched per item.
Figure 4.7: Impact of $|R|$ and $\overline{R}$ on the Number of Cache Misses

$(u = \overline{R}, \ C_1 = 32 \ \text{kB}, \ Z_1 = 32 \ \text{B}, \ C_2 = 8 \ \text{MB}, \ Z_2 = 128 \ \text{B})$
4.4.5 Repetitive Traversals

With repetitive traversals, cache re-usage comes into play. We assume initially empty caches.\(^6\) Hence, the first traversal requires as many cache misses as estimated above. But the subsequent traversals may benefit from the data already present in the cache after the first access. We will analyze this in detail for both sequential and random traversals.

4.4.5.1 Repetitive Sequential Traversal

Be \( R \) a data region, \( \mathcal{P} = \text{rs	extunderscore trav}^\ell(r, d, R, u) \) a repetitive sequential traversal over \( R \), and \( \mathcal{P}^* = \text{s	extunderscore trav}^\ell(R, u) \) a single sequential traversal over \( R \). Two parameters determine the caching behavior of a repetitive sequential traversal: the number \( M^\ell(\mathcal{P}^*) \) of cache lines touched during the first traversal and the direction \( d \) in which subsequent traversals sweep over \( R \).

In case \( M^\ell(\mathcal{P}^*) \) is smaller than the total number of available cache lines, only the first traversal causes cache misses, loading all required data. All \( r - 1 \) subsequent traversals then just access the cache, causing no further cache misses.

In case \( M^\ell(\mathcal{P}^*) \) exceeds the number of available cache lines, the end of a traversal pushes the data read at the begin of the traversal out of the cache. If the next traversal then again starts at the begin of \( R \), it cannot benefit from any data in the cache. Hence, with \( d = \text{uni} \), each sweep causes the full amount of cache misses. Only if a subsequent sweep starts where the previous one stopped, i.e., it traverses \( R \) in the opposite direction as its predecessor, it can benefit from the data stored in the cache. Thus, with \( d = \text{bi} \), only the first sweep causes the full amount of cache misses. The \( r - 1 \) remaining sweeps cause cache misses only for the fraction of \( R \) that does not fit into the cache.

In total, we get

\[
M^\ell_i(\text{rs	extunderscore trav}^\ell(r, d, R, u)) = \begin{cases} 
M^\ell_i(\mathcal{P}^*), & \text{if } M^\ell_i(\mathcal{P}^*) \leq #_i \\
M^\ell_i(\mathcal{P}^*) + (r - 1) \cdot (M^\ell_i(\mathcal{P}^*) - #_i), & \text{if } M^\ell_i(\mathcal{P}^*) > #_i \land d = \text{uni} \\
M^\ell_j(\mathcal{P}^*) + (r - 1) \cdot (M^\ell_j(\mathcal{P}^*) - #_i), & \text{if } M^\ell_j(\mathcal{P}^*) > #_i \land d = \text{bi}.
\end{cases}
\]

4.4.5.2 Repetitive Random Traversal

Be \( R \) a data region, \( \mathcal{P} = \text{rr	extunderscore trav}(r, R, u) \) a repetitive random traversal over \( R \), and \( \mathcal{P}^* = \text{r	extunderscore trav}(R, u) \) a single random traversal over \( R \). With random memory access, \( d \) is not defined, hence, we need to consider only \( M^\ell(\mathcal{P}^*) \) to determine to which extend repetitive accesses can benefit from cached data.

When \( M^\ell(\mathcal{P}^*) \) is smaller than the number of available cache lines, we get the same effect as above. Only the first sweep causes cache misses, loading all required data. All \( r - 1 \) subsequent sweeps then just access the cache, causing no further cache misses.

---

\(^6\)Section 4.5 will discuss how to consider pre-loaded caches.
In case $M^f(P')$ exceeds the number of available cache lines, the most recently accessed data remains in the cache at the end of a sweep. Hence, there is a certain probability that the first accesses of the following sweep might re-use (some of) these $\#$ cache lines. This probability decreases as $M^f(P')$ increases. We estimate the probability with $\#/M^f(P')$.

Analogously to the sequential case, we get

$$M^r_i(\text{trav}(r, R, u)) = \begin{cases} M^f_i(P') & \text{if } M^f_i(P') \leq \#_i \\ M^f_i(P') + (r - 1) \cdot \left( M^f_i(P') - \frac{\#_i}{M^f_i(P')} \cdot \#_i \right) & \text{if } M^f_i(P') > \#_i. \end{cases} \quad (4.8)$$

### 4.4.6 Random Access

Be $R$ a data region and $P = \text{acc}(r, R, u)$ a random access pattern on $R$. As in Section 4.4.3, we have

$$M^p_i(\text{acc}(r, R, u)) = 0.$$ 

In contrary to a single random traversal, where each data item of $R$ is touched exactly once, we do not know exactly, how many distinct data items are actually touched with random access. However, knowing that there are $r$ independent random accesses to the $|R|$ data items in $R$, we can estimate the average/expected number $I$ of distinct data items that are indeed touched. Be $E$ the number of all different outcomes of picking $r$ times one of the $|R|$ data items allowing multiple accesses to each data item. Further be $E_j$ the number of outcomes containing exactly $1 \leq j \leq \min\{r, |R|\}$ distinct data items. If we respect ordering, all outcomes are equally likely to occur, hence, we have

$$\sum_{j=1}^{\min\{r, |R|\}} E_j(r, |R|) = E(r, |R|),$$

with

$$\sum_{j=1}^{\min\{r, |R|\}} E_j(r, |R|) = E(r, |R|).$$

Calculating $E$ is straight forward:

$$E(r, |R|) = |R|^r.$$ 

Calculating $E_j$ turns out to be a bit more difficult. Be

$$\binom{x}{y} = \frac{x!}{(x-y)! \cdot y!}$$

the binomial coefficient, i.e., the number of ways of picking $y$ unordered outcomes from $x$ possibilities. Further be

$$\binom{x}{y} = \frac{1}{y!} \cdot \sum_{j=0}^{y-1} (-1)^j \cdot \binom{y}{j} \cdot (y-j)^x$$
the Stirling number of second kind, i.e., the number of ways of partitioning a set of \( x \) elements into \( y \) nonempty sets [Sti30]. Then, we have

\[
E_j(r, |R|) = \binom{|R|}{j} \cdot \binom{r}{j} \cdot j!.
\]

First of all, there are \( \binom{|R|}{j} \) ways to choose \( j \) distinct data items from the available \( |R| \) data items. Then, there are \( \binom{r}{j} \) ways to partition the \( r \) accesses into \( j \) groups, one for each distinct data item. Finally, we have to consider all \( j! \) permutations to get equally likely outcomes.

Knowing the number \( I \) of distinct data items that are touched by \( r_{\text{acc}}(r, R, u) \) on average, we can now calculate the number \( C \) of distinct cache lines touched. Again, we distinguish two cases, depending on the size of the (minimal) untouched gaps between two adjacent accesses.

**Case** \( \overline{R} - u \geq Z \). With the (minimal) untouched gaps larger than cache line size, no cache line is used by more than one data item. Following the discussion in Section 4.4.2, we get

\[
C_i(r_{\text{acc}}(r, R, u)) = I(r_{\text{acc}}(r, R, u)) \cdot \left( \left\lfloor \frac{u}{Z_i} \right\rfloor + \frac{(u - 1) \mod Z_i}{Z_i} \right).
\]

**Case** \( \overline{R} - u \geq Z \). With the (minimal) untouched gaps smaller than cache line size, (some) cache lines might be used by more than one data item. In case all \( I \) touched data items are pair-wise adjacent, we get

\[
\hat{C}_i(r_{\text{acc}}(r, R, u)) = \left\lfloor \frac{I(r_{\text{acc}}(r, R, u)) \cdot \overline{R}}{Z_i} \right\rfloor.
\]

However, if \( I \ll |R| \), the actual untouched gap might still be larger than cache line size, hence

\[
\hat{C}_i(r_{\text{acc}}(r, R, u)) = \min \left\{ I(r_{\text{acc}}(r, R, u)) \cdot \left( \left\lfloor \frac{u}{Z_i} \right\rfloor + \frac{(u - 1) \mod Z_i}{Z_i} \right), |R|/Z_i \right\}.
\]

\( \hat{C} \) is more likely with large \( I \), while \( \hat{C} \) is more likely with small \( I \). Hence, we calculate the average \( C \) as a linear combination of \( \hat{C} \) and \( \hat{C} \):

\[
C_i(r_{\text{acc}}(r, R, u)) = \frac{I(r_{\text{acc}}(r, R, u))}{|R|} \cdot \hat{C}_i(r_{\text{acc}}(r, R, u))
\]

\[
+ \left( 1 - \frac{I(r_{\text{acc}}(r, R, u))}{|R|} \right) \hat{C}_i(r_{\text{acc}}(r, R, u)).
\]
Knowing the number $C$ of distinct cache lines touched, we can finally calculate the number of cache misses. With $r$ accesses spread over $I$ distinct data items, each item is touched $r/I$ times on average. Analogously to Equation (4.8), we get ($\mathcal{P} = r_{\text{acc}}(r, R, u)$)

$$M^*_f(r_{\text{acc}}(r, R, u)) = \begin{cases} C_l(\mathcal{P}), & \text{if } C_l(\mathcal{P}) \leq \#_i \\ C_l(\mathcal{P}) + \left( \frac{r}{I(\mathcal{P})} - 1 \right) \left( C_l(\mathcal{P}) - \frac{\#_i}{C_l(\mathcal{P})} \cdot \#_i \right), & \text{if } C_l(\mathcal{P}) > \#_i \end{cases} \quad (4.9)$$

### 4.4.7 Interleaved Multi-Cursor Access

Be $R = \{R_j\}_{j=1}^m$ a data region divided into $m \leq |R|$ sub-regions $R_j$ with

$$\overline{R_j} = \overline{R} \quad \text{and} \quad k = |R_j| = \frac{|R|}{m}. \quad (\ast)$$

Further be $\mathcal{P} = \text{nest}(R, m, T([r, ]R_j, u), O, D)$ with $T \in \{s_{\text{trav}}^*, r_{\text{trav}}, r_{\text{acc}}\}$ an interleaved multi-cursor access. We inspect local random access ($T \in \{r_{\text{acc}}, r_{\text{trav}}\}$) and local sequential access ($T = s_{\text{trav}}^*$) separately.

#### 4.4.7.1 Local Random Access

With $T \in \{r_{\text{acc}}, r_{\text{trav}}\}$, $\mathcal{P}$ behaves like a single traversal $\mathcal{P}' = T'([m \cdot r, ]R, u)$. For $k = 1$ (i.e., $m = |R|$), the new order is the original global order, otherwise, it is the original local order, i.e., we get

$$T' = \begin{cases} s_{\text{trav}}^*, & \text{if } k = 1 \land O = \text{seq} \\ T, & \text{else} \end{cases}$$

and consequently

$$\bar{M}_i(\text{nest}(R, m, T([r, ]R_j, u), O, D)) = \bar{M}_i(T'([m \cdot r, ]R, u)).$$

#### 4.4.7.2 Local Sequential Access

For $T = s_{\text{trav}}^*$, we distinguish three cases:

- $\overline{R} - u > Z$,
- $\overline{R} - u \leq Z \land m \cdot \left\lfloor \frac{u}{2} \right\rfloor \leq \#$,
- $\overline{R} - u \leq Z \land m \cdot \left\lceil \frac{u}{2} \right\rceil > \#$. 
Case \( \overline{R} - u > Z \). In this case, \( \mathcal{P} \) means \( k = |R_j| \) times traversing across all \( R_j \) in order \( O \). Hence, each traversal performs \( m \) accesses (one to each \( R_j \)). The distance between adjacent accesses within each traversal is \( ||R_j|| = k \cdot \overline{R} \). We describe these traversals by \( T'(R', u) \) where \( R' \) is a data region with

\[
|R'| = m \quad \text{and} \quad \overline{R'} = ||R||, \tag{\dagger}
\]

and

\[
T' = \begin{cases} 
  r_{\text{trav}}, & \text{if } O = \text{ran} \\
  T, & \text{else.}
\end{cases}
\]

As the non-touched gap between adjacent accesses within each \( R_j \) is larger than a cache line \((\overline{R} - u > Z)\), no cache line is shared by two or more accesses. Thus, the total number of cache misses is the sum of the cache misses caused by the \( k \) traversals:

\[
\tilde{M}_i(\text{nest}(R, m, T(R_j, u), O, D)) = \sum_{h=1}^{k} \tilde{M}_i(T'(R', u)) \]

\[
= k \cdot \tilde{M}_i(T'(R', u)) \]

\[
= \begin{cases} 
  k \cdot |R'| \cdot \left( \left\lceil \frac{u}{Z} \right\rceil + \frac{(u - 1) \mod Z_i}{Z_i} \right), & \text{if } T' = s_{\text{trav}}^* \\
  \left(0, k \cdot |R'| \cdot \left( \left\lceil \frac{u}{Z} \right\rceil + \frac{(u - 1) \mod Z_i}{Z_i} \right) \right), & \text{else}
\end{cases} \]

\[
\overset{(4.4/4.6)}{=} \tilde{M}_i(T'(R, u)). \]

Case \( \overline{R} - u \leq Z \land m \cdot \left\lceil \frac{u}{Z} \right\rceil \leq \# \). With the non-touched gaps being smaller than cache line size \((\overline{R} - u \leq Z)\), adjacent accesses within each \( R_j \) might share a cache line, and hence benefit from previous accesses. With one traversal across all \( R_j \) touching less cache lines than there are in total \((m \cdot \left\lceil \frac{u}{Z} \right\rceil \leq \#)\), the subsequent traversal does not have to reload the shared cache lines. Hence, the total number of cache misses is just the sum of all local patterns. Though these are sequential, a global random pattern will avoid sequential latency. We take this into account when defining \( T' \) and get

\[
\tilde{M}_i(\text{nest}(R, m, T(R_j, u), O, D)) \]

\[
= \sum_{j=1}^{m} \tilde{M}_i(T'(R_j, u)) \]

\[
= m \cdot \tilde{M}_i(T'(R_j, u)) \]

\[
\overset{(4.3, \#)}{=} \tilde{M}_i(T'(R, u)). \]
with
\[ T' = \begin{cases} 
    \text{s.trav}^x, & \text{if } O = \text{ran} \\
    T, & \text{else}.
\end{cases} \]

Case \( R - u < Z \land m \cdot \left\lceil \frac{n}{Z} \right\rceil > \# \). With one traversal across all \( R_j \) touching more cache lines than there are in total \( (m \cdot \left\lceil \frac{n}{Z} \right\rceil > \#) \), only \( h = \# / \left\lceil \frac{n}{Z} \right\rceil < m \) of the \( m \) shared cache lines remain in the cache for potential re-use. The number \( h' \) of cache lines that is actually re-used depends on \( O \) and \( D \) and is calculated similarly as for the repetitive traversals in Section 4.4.5:
\[ h_i' = \begin{cases} 
    0, & \text{if } O = \text{seq} \land D = \text{uni} \\
    h_i, & \text{if } O = \text{seq} \land D = \text{bi} \\
    \frac{h}{m} \cdot h_i, & \text{if } O = \text{ran}
\end{cases} \]
\[ h_i = \# / \left\lceil \frac{n}{Z} \right\rceil. \]

Hence, the total number of cache misses is the same as in the previous case, plus the \( m - h' \) cache lines that have to be reloaded during all but the first traversal. These additional misses cause random latency, i.e., we get
\[ \tilde{M}_i(\text{nest}(R, m, T(R_j, u), O, D)) = \tilde{M}_i(T'(R, u)) + \tilde{X}_i \]
with
\[ \tilde{X}_i = (0, (k-1) \cdot (m-h'_i)) \] (4.10)
and \( T' \) as before.

### 4.4.7.3 Summary

Gathering the results from all the different cases discussed above, we get
\[ \tilde{M}_i(\text{nest}(R, m, T([r, ]R_j, u), O, D)) \]
\[ = \begin{cases} 
    \tilde{M}_i(T'([m \cdot r, ]R, u)) + \tilde{X}_i, & \text{if } T = \text{s.trav}^x \\
    \tilde{M}_i(T'([m \cdot r, ]R, u)), & \text{if } T = \text{r.trav} \\
    \tilde{M}_i(T'([m \cdot r, ]R, u)), & \text{if } T = \text{s.trav}^y
\end{cases} \] (4.10)

with \( \tilde{X}_i \) as in (4.10) and
\[ T' = \begin{cases} 
    \text{s.trav}^x, & \text{if } T \in \{ \text{r.acc, r.trav} \} \land O = \text{seq} \land k = 1 \\
    \text{r.trav}, & \text{if } T = \text{s.trav}^x \land O = \text{ran} \land R - u > Z_i \\
    \text{s.trav}^y, & \text{if } T = \text{s.trav}^y \land O = \text{ran} \land R - u \leq Z_i \\
    T, & \text{else}.
\end{cases} \]
In other words, an interleaved multi-cursor access pattern causes at least as many cache misses as some simple traversal pattern on the same data region. However, it might cause random misses though the local pattern is expected to cause sequential misses. Further, if the cross-traversal requires more cache lines than available, $X_r = (k - 1) \cdot (m - h'_r)$ additional random misses will occur.

## 4.5 Combining Cost Functions

Given the cache misses for basic patterns, we will now discuss how to derive the resulting cache misses of compound patterns. The major problem is to model cache interference that occurs among the basic patterns.

### 4.5.1 Sequential Execution

Be $P_1, \ldots, P_p \in \mathcal{P}$ ($p > 1$) access patterns. $\oplus(P_1, \ldots, P_p)$ then denotes that $P_{q+1}$ is executed after $P_q$ is finished (cf., Sec. 4.3.3). Obviously, the patterns do not interfere in this case. Consequently, the resulting total number of cache misses is at most the sum of the cache misses of all $p$ patterns. However, if two subsequent patterns operate on the same data region, the second might benefit from the data that the first one leaves in the cache. It depends on the cache size, the data sizes, and the characteristics of the individual patterns, how many cache misses may be saved this way.

To model this effect, we need to consider the contents or state of the caches. We describe the state of a cache as a set $S$ of pairs $\langle R, \rho \rangle \in \mathbb{D} \times [0, 1]$, stating for each data region $R$ the fraction $\rho$ that is available in the cache. For convenience, we omit data regions that are not cached at all, i.e., those with $\rho = 0$. In order to appropriately consider the caches' initial states when calculating the cache misses of a basic pattern $P = T([.., ]R[..]) \in \mathcal{P}_b$, we define

$$M_i(S, P) = \begin{cases} (0, 0), & \text{if } \langle R, 1 \rangle \in S_i \\ M_i(P) - \left(0, \frac{\rho \cdot |R|_Z}{M_i(P)} \cdot \rho \cdot |R|_Z\right), & \text{if } T \in \{r\_trav, r\_rr\_trav, r\_acc\} \\ M_i(P), & \text{else} \\ \end{cases}$$

(4.11)

with $M_i(P)$ as defined in Equations (4.3) through (4.10). In case $R$ is already entirely available in the cache, no cache misses will occur during $P$. In case only a fraction of $R$ is available in the cache, there is a certain chance, that random patterns might (partially) benefit from this fraction. Sequential patterns, however, would only benefit if this fraction makes up the "head" of $R$. As we do not know whether this is true, we assume that sequential patterns can only benefit, if $R$ is already entirely in the cache. For convenience, we write

$$M_i(\emptyset, P) = M_i(P) \quad \forall P \in \mathcal{P}. $$
Additionally, we need to know the caches’ resulting states \( S(\mathcal{P}) \) after a pattern \( \mathcal{P} \) has been performed. For basic patterns \( \mathcal{P} = T([.., R[..]) \in \mathcal{P}_b \), we define

\[
S_i(\mathcal{P}) = \left\{ R, \min \left\{ \frac{C_i}{\| R \|}, 1 \right\} \right\}.
\]

For compound patterns \( \oplus(\mathcal{P}_1, \ldots, \mathcal{P}_p) \) with \( \mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathcal{P}, p > 1 \), we define

\[
S_i(\oplus(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = S_i(\mathcal{P}_p).
\]

Here, we assume that only that last data region (partially) remains in the cache. In case that \( R \) is smaller that the cache, (parts) of the previous data regions might also remain in the cache. However, we ignore this case here, and leave it for future research.

Equipped with these tools, we can finally calculate the number of cache misses that occur when executing patterns \( \mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathcal{P}, p > 1 \) sequentially, given an initial cache state \( S^0 \):

\[
\tilde{M}_i(S^0, \oplus(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \tilde{M}_i(S^0, \mathcal{P}_1) + \sum_{q=2}^{p} \tilde{M}_i(S_i(\mathcal{P}_{q-1}), \mathcal{P}_q). \tag{4.12}
\]

### 4.5.2 Concurrent Execution

When executing two or more patterns concurrently, we actually have to consider the fact that they are competing for the same cache. The number of total cache misses will be higher than just the sum of the individual cache miss rates. The reason for this is, that the patterns will mutually evict cache lines from the cache due to alignment conflicts. To which extend such conflict misses occur does not only depend on the patterns themselves, but also on the data placement and details of the cache alignment. Unfortunately, these parameters are not know during cost evaluation.

Hence, we model the impact of the cache interference between concurrent patterns by dividing the cache among all patterns. Each individual pattern gets only a fraction of the cache according to its footprint size. We define a pattern’s footprint size \( F \) as the number of cache lines that it potentially revisits.

With single sequential traversals, a cache line is never visited again once access has moved on to the next cache line. Hence, simple sequential patterns virtually occupy only one cache line a at time. Or in other words, the number of cache misses is independent of the available cache size. The same holds for single random traversals with \( \bar{R} - u \geq Z \). In all other cases, basic access patterns (potentially) revisit all cache lines covered by their respective data region. We define \( F \) as follows.

Be \( \mathcal{P} = T([.., R, u] \in \mathcal{P}_b \), a basic access pattern, then

\[
F_i(\mathcal{P}) = \begin{cases} 1, & \text{if } T = s.\text{trav}^r \\ 1, & \text{if } T = r.\text{trav} \land \bar{R} - u \geq Z_i \\ |R|_{Z_i}, & \text{else.} \end{cases}
\]
Be $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathcal{P}$ ($p > 1$) access patterns, then

$$F_i(\ominus(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \max\{F_i(\mathcal{P}_1), \ldots, F_i(\mathcal{P}_p)\},$$

$$F_i(\odot(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \sum_{q=1}^{p} F_i(\mathcal{P}_q).$$

Further, we use $M_{\phi_v}$ with $v \geq 1$ to denote the number of misses with only $\frac{1}{v}$th of the total cache size available. To calculate $M_{\phi_v}$, we simply replace $C$ and $\#$ by $\frac{C}{v}$ and $\frac{\#}{v}$, respectively, in the formulas in Sections 4.4 and 4.5.1. Likewise, we define $S_{\phi_v}(\mathcal{P})$. We write $M_i = M_{\phi_1}$ and $S_i = S_{\phi_1}$.

With these tools at hand, we calculate the cache misses for concurrent execution of patterns $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathcal{P}$ ($p > 1$) given an initial cache state $S^0$ as

$$M_{\phi_v}(S^0, \ominus(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \sum_{q=1}^{p} M_{\phi_{v_q}}(S^0, \mathcal{P}_q)$$

(4.13)

with

$$v_q = \frac{F(\ominus(\mathcal{P}_1, \ldots, \mathcal{P}_p))}{F(\mathcal{P}_q)} \cdot v.$$

After executing $\ominus(\mathcal{P}_1, \ldots, \mathcal{P}_p)$, the cache contains a fraction of each data region involved, proportional to its footprint size:

$$S_i(\ominus(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \bigcup_{q=1}^{p} S_{\phi_{v_q}}(\mathcal{P}_q)$$

with $v_q$ as defined before.

### 4.5.3 Query Execution Plans

With the techniques discussed in the previous sections, we have the basic tools at hand to also estimate the number and kind of cache misses of complete query plans, and hence to predict their memory access costs. The various operators in a query plan are combined in the same way the basic patterns are combined to form compound patterns. Basically, the query plan describes, which operators are executed one after the other and which are executed concurrently. Here, we view pipelining as concurrent execution of data-dependent operators. Hence, we can derive the complex memory access pattern of a query plan by combining the compound patterns of the operators as discussed above. Considering the caches' states as introduced before takes care of properly recognizing data dependencies.

### 4.6 CPU Costs

Next to memory access costs, we need to know the pure CPU processing costs in order to estimate the total execution costs. We now present a simple but effective method to
acquire CPU costs using a calibration approach. As we focus on memory access costs in this work, a more detailed model of CPU costs is beyond the scope of this thesis and left for future work.

### 4.6.1 What to calibrate?

Calibrating CPU costs means to actually measure the costs — i.e., execution time — of all algorithms in a laboratory setting. A prerequisite for this approach is that we know the complexity of the algorithms in terms of input and output cardinalities. In other words, we assume we know the principle CPU cost functions of our algorithms. For instance, we know that the CPU cost of a scan-select can be described as

\[
T_{\text{CPU}} = c_0 + c_1 \cdot n + c_2 \cdot m
\]

where \(c_0\) represents the fix startup costs, \(n\) and \(m\) are the input and output cardinality, respectively, and \(c_1\), \(c_2\) represent the per tuple costs for processing input tuples and producing output tuples, respectively. As we are talking about our own algorithms, the assumption that we know such functions is reasonable. In case of doubt, we can use the Software Testpilot [KK93, AKK95] to experimentally derive these functions.

Obviously, these functions only depend on the algorithm itself. All implementations details like coding style, code optimizations, and compiler optimizations are covered by the constants \(c_i\) in the above formula. This approach implies that the constants \(c_i\) are indeed independent of the data volume. But this does hold for the pure CPU costs. The impact of data volume is already covered by the memory access costs. However, the \(c_i\) are typically not independent of other parameters like data types, and respective code optimizations in the algorithms. Here, again, our code expansion technique pays back. Within each expanded implementation variant of our algorithm, these parameters are constant, and thus, the \(c_i\) are indeed simple constants for each physical implementation.

### 4.6.2 How to calibrate?

Hence, calibrating the \(c_i\) can be done by some simple experiments. For instance, in our example case, the scan-select, we use the following experiments:

First, we need to measure the fix (start-up) costs \(c_0\). The start-up costs cover all the "administrative" work that is done only once per algorithm. Such work contains among other

- parsing the MIL command;
- performing on-the-fly tactical/operational optimization, i.e., choosing the most adequate algorithm/implementation according to the current properties of the inputs and the current state of the system;
- overhead for calling the function that implements the operator, including possible instruction cache misses (i.e., memory access) to load the respective code and the necessary stack management;
• creating, initializing, and removing temporary data structures;

• creating the output/result data structure, usually a BAT.

We measure \( c_0 \) by measuring the time needed to execute the operation on empty inputs. Typically, the time needed for this task is so small, that the resolution of the timing function is just not small enough to ensure accurate and reliable measurements. We prevent such problems by repeating the empty call several times and measure the total time. The actual costs for a single call are then calculated by dividing the total time by the number of calls.

Once we know \( c_0 \), we can measure \( c_1 \) by performing the operation in question on a non-empty input that produces no output. In case of our example (select), for instance, this can be achieved by using a predicate the does not match any tuple of the input BAT. \( c_1 \) is then to be calculated by subtracting \( c_0 \) from the measured time and dividing the remainder by the cardinality of the input. Two things have to be considered. On the one hand, we would like to exclude any interference with memory access. One way to do this is to use an input table that is so small that all processing takes place in L1 cache. This requires an initial not-measured run to pre-load the cache. Again, we might need to measure several subsequent runs to ensure stable results. On the other hand, using too little input data might result in inaccurate times for the per tuple costs. The fix costs are likely to be dominant, as we might need to use really small inputs to be sure that all processing indeed is limited to L1. Using large inputs, however, implies that memory costs will be included in the measurement. But we do know the memory costs, and hence, we can easily subtract them from the measured times to get the pure CPU costs. We propose to use the second technique.

Actually, the second technique has another advantage. As we learned in Chapter 3, CPU costs and memory access costs may overlap. Hence, simply adding-up the CPU costs as calibrated without any memory access by the first technique and the memory costs as estimated by our models would result in too high overall costs. Accurately predicting the degree of overlap between CPU costs and memory access costs, however, depends on various parameters and is hence very difficult, if not impossible. Our second technique, however, implicitly considers the actual overlap by measuring only that part of the CPU costs that does not overlap with memory access costs. As we are interested in the CPU costs only to add them to the estimated memory access costs, and thus yield the total costs, the second technique makes-up a feasible solution.

Knowing both \( c_0 \) and \( c_1 \), we can measure \( c_2 \) in a third and final experiment. We modify the second experiment to produce output that is as big as the input. In case of our example (select), for instance, this can be achieved by using a predicate such that all input tuples do match. Subtracting fix costs, input processing costs (as measured before), and memory access costs from the measured time, we can calculate the output creation costs.

For other algorithms, the calibration procedure follows the same schema, though it gets more complex for more complex algorithms.
4.6.3 When to calibrate?

For the memory access cost models, we proposed to measure the required hardware parameters once when the DBMS is installed on a new (hardware-)system. Likewise, the CPU costs also need to be measured only once per system. However, as we need to measure the costs for each physical implementation of each algorithm, this might be a rather complex and long running task. One way to speed-up the installation process is to restrict this task (at installation time) to the most popular MIL operations and only their most important variants. The remaining costs can then be calibrated "on-the-fly" only as soon as they are needed by the system.

4.7 Experimental Validation

To validate our cost model, we compare the estimated costs with experimental results. We focus on characteristic operations, here. The data access pattern of each operation is a combination of several basic patterns. The operations are chosen so that each basic pattern occurs at least once. Extension to further operations and whole queries, however, is straightforward, as it just means applying the same techniques to combine access patterns and derive their cost functions.

4.7.1 Setup

We implemented our cost functions and used our main-memory DBMS prototype Monet (see Section 2.7) as experimentation platform. We ran our experiments on an SGI Origin2000 and on an AMD PC. Table 4.2 lists the relevant hardware features of the machines. The cache characteristics are measured with our calibration tool. We use the CPU's hardware counters to get the exact number of cache and TLB misses while running our experiments. Thus, we can validate the estimated cache miss rates. Validating the resulting total memory access cost (i.e., miss rates scored by their latencies) is more complicated, as there is no way to measure the time spent on memory access. We can only measure the total elapsed time, and this includes the (pure) CPU costs as well. Hence, we extend our model to estimate the total execution time $T$ as sum of memory access time and pure CPU time

$$T = T_{\text{Mem}} + T_{\text{CPU}}$$

with $T_{\text{Mem}}$ as in Equation (4.1). We calibrate $T_{\text{CPU}}$ for each algorithm as described in the previous section.

4.7.2 Results

Figures 4.8 through 4.11 gather our experimental results. Each plot represents one algorithm. The cache misses and times measured during execution are depicted as points. The respective cost estimations are plotted as lines. Cache misses are depicted

---

7We present more examples and validation in Chapter 5.
### 4.7 Experimental Validation

<table>
<thead>
<tr>
<th>machine type</th>
<th>SGI Origin2000</th>
<th>AMD PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>IRIX64 6.5</td>
<td>Linux 2.2.14</td>
</tr>
<tr>
<td>CPU</td>
<td>MIPS R10000</td>
<td>AMD Athlon</td>
</tr>
<tr>
<td>CPU speed</td>
<td>250 MHz</td>
<td>600 MHz</td>
</tr>
<tr>
<td>main-memory size</td>
<td>48 GB (4 GB local)</td>
<td>384 MB</td>
</tr>
<tr>
<td>cache &amp; TLB levels</td>
<td>$N$</td>
<td>3</td>
</tr>
<tr>
<td>L1 cache capacity</td>
<td>$C_1$</td>
<td>32 KB</td>
</tr>
<tr>
<td>L1 cache line size</td>
<td>$Z_1$</td>
<td>32 bytes</td>
</tr>
<tr>
<td>L1 cache lines</td>
<td>$N_1$</td>
<td>1,024</td>
</tr>
<tr>
<td>L2 cache capacity</td>
<td>$C_2$</td>
<td>4 MB</td>
</tr>
<tr>
<td>L2 cache line size</td>
<td>$Z_2$</td>
<td>128 bytes</td>
</tr>
<tr>
<td>L2 lines</td>
<td>$N_2$</td>
<td>32,768</td>
</tr>
<tr>
<td>TLB entries</td>
<td>$N_3$</td>
<td>64</td>
</tr>
<tr>
<td>page size</td>
<td>$Z_3$</td>
<td>16 KB</td>
</tr>
<tr>
<td>TLB capacity (#3 · $Z_3$)</td>
<td>$C_3$</td>
<td>1 MB</td>
</tr>
<tr>
<td>TLB miss latency</td>
<td>$\ell_1$ = $\ell_3$</td>
<td>228 ns = 57 cycles</td>
</tr>
</tbody>
</table>

**Table 4.2: Hardware Characteristics**

...in absolute numbers. Times are depicted in milliseconds. We will now discuss each algorithm in detail.

**Quick-Sort** Our first experiment is sorting. We use quick-sort to sort a table in-place. Quick-sort uses two cursors, one starting at the front and the other starting at the end. Both cursors sequentially walk toward each other swapping data items where necessary, until they meet in the middle. We model this as two concurrent sequential traversals, each sweeping over one half of the table: $s_{\text{trav}}(U/2) \odot s_{\text{trav}}(U/2)$. At the meeting point, the table is split in two parts and quick-sort recursively proceeds depth-first on each part. With $n$ being the table's cardinality, the depth of the recursion...
Figure 4.8: Measured (points) and Predicted (lines) Cache Misses and Execution Time of Quick-Sort

is \( \log_2 n \). In total, we model the data access pattern of quick-sort as

\[
U \leftarrow \text{quick\textunderscore sort}(U) : \quad \oplus_{i=1}^{\lceil \log_2 n \rceil} \left( \oplus_{j=i}^{\lceil \log_2 n \rceil} (s\text\_trav}^s(U/2^i) \odot s\text\_trav}^s(U/2^i)) \right).
\]

We varied the table sizes from 128 KB to 128 MB and the tables contained randomly distributed (numerical) data. Figure 4.8 shows that the models accurately predict the actual behavior. Only the start-up overhead of about 100 TLB misses is not covered, but this is negligible. The step in the L2 misses-curve depicts the effect of caching on repeated sequential access: Tables that fit into the cache have to be loaded only once during the top-level iteration of quick-sort. Subsequent iterations operate on the cached data, causing no additional cache misses.

**Merge-Join** Our next candidate is merge-join. Assuming both operands are already sorted, merge-join simply performs three concurrent sequential patterns, one on each input and one on the output:

\[
W \leftarrow \text{merge\textunderscore join}(U, V) : \quad s\text\_trav}^s(U) \odot s\text\_trav}^s(V) \odot s\text\_trav}^s(W).
\]

Again, we use randomly distributed data and table sizes as before. In all experiments, both operands are of equal size, and the join is a 1:1-match. The respective
results in Figure 4.9 demonstrate the accuracy of our cost functions. Further, we see that single sequential access is not affected by cache sizes. The costs are proportional to the data sizes.

**Hash-Join** While the previous operations perform only sequential patterns, we now turn our attention to hash-join. Hash-join performs random access to the hash-table, both while building it and while probing the other input against it. We model the data access pattern of hash-join as

\[ W \leftarrow hash\_join(U, V) : \]

\[ s\_trav^2(V) \circ r\_trav(V') \oplus s\_trav^2(U) \circ r\_acc(|U|, V') \circ s\_trav^2(W). \]

The plots in Figure 4.10 clearly show the significant increase in L2 and TLB misses, once the hash-table size \(|V'|\) exceeds the respective cache size.\(^8\) Our cost model correctly predicts these effects and the resulting execution time.

**Partitioning** One way to prevent the performance decrease of hash-join on large tables is to partition both operands on the join attribute and then hash-join the matching partitions [SKN94, MBK00a]. If each partition fits into the cache, no additional cache misses will occur during hash-join.

\(^8\)The plots show no such effect for L1 misses, as all hash-tables are larger than the L1 cache, here.
Figure 4.10: Measured (points) and Predicted (lines) Cache Misses and Execution Time of Hash-Join

Figure 4.11: Measured (points) and Predicted (lines) Cache Misses and Execution Time of Partitioning and Partitioned Hash-Join
Partitioning algorithms typically maintain a separate output buffer for each result partition. The input is read sequentially, and each tuple is written to its output partition. Data access within each output partition is also sequential. Hence, we model partitioning using a sequential traversal for the input and an interleaved multi-cursor access for the output:

\[
\{U_j\}_{j=1}^m \leftarrow \text{cluster}(U, m) : \text{s.trav}^a(U) \circ \text{nest}(\{U_j\}_{j=1}^m, m, \text{s.trav}^a(U_j, \text{ran})).
\]

The curves in Figure 4.11a demonstrate the effect we discussed in Section 4.4.7: The number of cache misses increases significantly, once the number of output buffers \(m\) exceeds the number of available cache blocks \#. Though they tend to under estimate the costs for very high numbers of partitions, our models accurately predict the crucial points.

**Partitioned Hash-Join** Once the inputs are partitioned, we can join them by performing a hash-join on each pair of matching partitions. We model the data access pattern of partitioned hash-join as

\[
\{W_j\}_{j=1}^m \leftarrow \text{part.hash.join}(\{U_j\}_{j=1}^m, \{V_j\}_{j=1}^m, m) :
\]

\[
\otimes \{j=1 (\text{hash.join}(V_j, U_j, W_j)).
\]

Figure 4.11b shows that the cache miss rates, and thus the total costs, decrease significantly, once each partition (respectively its hash-table) fits into the cache.

### 4.8 Conclusion

We presented a new generic approach to build generic database cost models for hierarchical memory systems.

We extended the knowledge base on analytical cost-models for query optimization with a strategy derived from our experimentation with main-memory database technology. The approach taken shows that we can achieve hardware-independence by modeling hierarchical memory systems as multiple level of caches. Each level is characterized by a few parameters describing its sizes and timings. This abstract hardware model is not restricted to main-memory caches. As we pointed out, the characteristics of main-memory access are very similar to those of disk access. Viewing main-memory (e.g., a database system’s buffer pool) as cache for disk access, it is obvious that our approach also covers I/O. As such, the model presented provides a valuable addition to the core of cost-models for disk-resident databases as well.

Adaptation of the model to a specific hardware is done by instantiating the parameters with the respective values of the very hardware. Our Calibrator, a software tool to measure these values on arbitrary systems, is available for download from our website http://monetdb.cwi.nl.
We identified a few key access patterns eminent in the majority of relational algebra implementations. The key patterns fulfill two major requirements: they are simple and they have a relevant impact on data access costs. For these basic patterns, we developed cost functions that estimate the respective access cost in terms of cache misses scored by there latency. To maintain hardware-independence, the functions are parameterized with the hardware characteristics.

We introduced two operators to combine simple patterns to more complex patterns and developed rules how to generate the respective cost functions.

With our approach, building physical costs function for database operations boils down to describing the algorithms' data access in a kind of "pattern language" as presented in Section 4.3.3. This task requires only information that can be derived from the algorithm. Especially, no knowledge about the hardware is needed, here. The detailed cost function are than automatically derived from the pattern descriptions.

Though focusing on data access costs, our model does not ignore CPU costs. We presented a simple but effective calibration approach that allows to automatically measure the CPU costs of each algorithm and its various implementations. An investigation as to whether and how CPU costs can be modeled in more details is left to future research.