Understanding, modeling, and improving main-memory database performance

Manegold, S.

Citation for published version (APA):
Chapter 5
Self-tuning Cache-conscious Join Algorithms

In the past decade, the exponential growth in commodity CPUs’ speed has far outpaced advances in memory latency. A second trend is that CPU performance advances are not only brought by increased clock rate, but also by increasing parallelism inside the CPU. Current database systems have not yet adapted to these trends, and show poor utilization of both CPU and memory resources on current hardware. We discussed these issues in detail in Chapter 3. In this chapter, we show how these resources can be optimized for large joins. We refine the partitioned hash-join with a new partitioning algorithm called radix-cluster, which is specifically designed to optimize memory access. The algorithms are designed to be tuned at runtime to achieve the optimal performance given the underlying hardware and the actual data to be processed. Tuning is done by means of just three parameters. We will demonstrate, how the cost models developed in Chapter 4 allow us to determine the optimal values for these parameters automatically at runtime. Finally, we investigate the effect of implementation techniques that optimize CPU resource usage. It turns out, that the full benefit of memory access optimization can only be achieved, if also the CPU resource usage is minimized. Exhaustive experiments on four different architectures show that large joins can be accelerated almost an order of magnitude on modern RISC hardware when both memory and CPU resources are optimized.

5.1 Introduction

Custom hardware—from workstations to PCs—has experienced tremendous performance improvements in the past decades. Unfortunately, these improvements are not equally distributed over all aspects of hardware performance and capacity. Figure 3.1 shows that the speed of commercial microprocessors has increased roughly 50% every year, while the access latency of commodity DRAM has improved by little more than 10% over the past decade [Mow94]. One reason for this is that there is a direct trade-
off between capacity and speed in DRAM chips, and the highest priority has been for increasing capacity. The result is that from the perspective of the processor, memory is getting slower at a dramatic rate, making it increasingly difficult to achieve high processor efficiencies. Another trend is the ever increasing number of inter-stage and intra-stage parallel execution opportunities provided by multiple execution pipelines and speculative execution in modern CPUs. Current database systems on the market make poor use of these new features; studies on several DBMS products on a variety of workloads [ADHW99, BGB98, KPH*98, TLPZT97] consistently show modern CPUs to be stalled (i.e., non-working) most of the execution time.

In this chapter, we show how large main-memory joins can be accelerated by optimizing memory and CPU resource utilization on modern hardware. These optimizations involve radical changes in database architecture, encompassing new data structures, query processing algorithms, and implementation techniques. Our findings are summarized as follows:

- **Memory access is a bottleneck to query processing.** We demonstrated in Section 3.2 that the performance of even simple database operations is nowadays severely constrained by memory access costs. For example, a simple in-memory table scan runs on Sun hardware from the year 2000 in roughly the same absolute time as on a Sun from 1992, now spending 95\% of its cycles waiting for memory (see Section 3.2). It is important to note that this bottleneck affects database performance in general, not only main-memory database systems.

- **Data structures and algorithms should be tuned for memory access.** We discuss database techniques to avoid the memory access bottleneck, both in the fields of data structures and query processing algorithms. The key issue is to optimize the use of the various caches of the memory subsystem. We show how vertical table fragmentation optimizes sequential memory access to column data. For equi-join, which has a random access pattern, we refine partitioned hash-join with a new radix-cluster algorithm which makes its memory access pattern more easy to cache. Our experiments indicate that large joins can strongly benefit from these techniques.

- **Memory access costs can be modeled precisely.** Cache-aware algorithms and data structures must be tuned to the memory access pattern imposed by a query and hardware characteristics such as cache sizes and miss penalties, just like traditional query optimization tunes the I/O pattern imposed by a query to the size of the buffers available and I/O cost parameters. Therefore it is necessary to have models that predict memory access costs in detail. We apply the techniques presented in Chapter 4 to provide such detailed models for our partitioned hash-join algorithms. These models use an analytical framework that predicts the number of hardware events (e.g., cache misses and CPU cycles), and scores them with hardware parameters. The experiments in this chapter confirm both the usability and the accuracy of our generic cost models.

- **Memory optimization and efficient coding techniques boost each others effects.** CPU resource utilization can be optimized using implementation techniques
known from high-performance computing [Sil97] and main-memory database systems [Ker89, BK99]. We observe that applying these optimizations in combination with memory optimizations yields a higher performance increase than applying them without memory optimizations. The same is also the case for memory optimizations: they turn out to be more effective on CPU-optimized code than on non-optimized code. Our experiments show that database performance can be improved by an order of magnitude applying both CPU and memory optimization techniques.

Our research group has studied large main-memory database systems for the past 10 years. This research started in the PRISMA project [AvdBF+92], focusing on massive parallelism, and is now centered around Monet: a high-performance system targeted to query-intensive application areas like OLAP and data mining (cf., Section 2.7). We use Monet as our experimentation platform.

### 5.1.1 Related Work

Database system research into the design of algorithms and data structures that optimize memory access, has been relatively scarce. Our major reference is the work by Shatdal et al. [SKN94], which shows that join performance can be improved using a main-memory variant of Grace Join, in which both relations are first hash-partitioned in chunks that fit the (L2) memory cache. There were various reasons that lead us to explore this direction of research further. First, after its publication, the observed trends in custom hardware have continued, deepening the memory access bottleneck. For instance, the authors list a mean performance penalty for a cache miss of 20-30 cycles in 1994, while a range of 100-300 is typical in 2002 (and rising). This increases the benefits of cache optimizations, and possibly changes the trade-offs. Another development has been the introduction of so-called level-one (L1) caches, which are typically very small regions on the CPU chip that can be accessed at almost CPU clock-speed. The authors of [SKN94] provide algorithms that are only feasible for the relatively larger off-chip L2 caches. Finally, this previous work uses standard relational data structures, while we argue, that the impact of memory access is so severe that vertically fragmented data structures should be applied at the physical level of database storage.

Though we consider memory-access optimization to be relevant for database performance in general, it is especially important for main-memory databases, a field that through time has received fluctuating interest within the database research community. In the 1980s [LC86a, LC86b, Eic89, Wil91, AP92, GMS92], when falling DRAM prices seemed to suggest that most data would soon be memory-resident, its popularity diminished in the 1990s, narrowing its field of application to real-time systems only. Currently, interest has revived into applications for small and distributed database systems, but also in high performance systems for query-intensive applications, like data mining and OLAP. In our research, we focus on this latter category. Example commercial systems are the Times-Ten product [Tea99], Sybase IQ [Syb96], and Compaq’s Infocharger [Com98], which is based on an early version of Monet (cf.,
Section 2.7).

Past work on main-memory query optimization [LN96, WK90] models the main-memory costs of query processing operators on the coarse level of procedure calls, using profiling to obtain some 'magical' constants. As such, these models do not provide insight in individual components that make up query costs, limiting their predictive value. Conventional (i.e., non main-memory) cost modeling, in contrast, has I/O as the dominant cost aspect, making it possible to formulate accurate models based on the amount of predicted I/O work. Calibrating such models is relatively easy, as statistics on the I/O accesses caused during an experiment are readily available in a database system. Past work on main-memory systems was unable to provide such cost models on a similarly detailed level, for two reasons. First, it was difficult to model the interaction between low-level hardware components like CPU, Memory Management Unit, bus, and memory caches. Second, it was impossible to measure the status of these components during experiments, which is necessary for tuning and calibration of models. Modern CPUs, however, contain performance counters for events like cache misses, and exact CPU cycles [BZ98, ZLT96, Yea96]. This enabled us to develop a new main-memory cost modeling methodology that first mimics the memory access pattern of an algorithm, yielding a number of CPU cycle and memory cache events, and then scores this pattern with an exact cost prediction (see Chapter 4). Therefore, the contribution of the algorithms, models, and experiments presented here is to demonstrate that detailed cost modeling of main-memory performance is both important and feasible.

5.1.2 Outline

In Section 5.2, we introduce the radix-cluster algorithm, which improves the partitioning phase in partitioned hash-join by trading memory access costs for extra CPU processing. We perform exhaustive experiments where we use CPU event counters to obtain detailed insight in the performance of this algorithm. First, we vary the partition sizes, to show the effect of tuning the memory access pattern to the memory cache sizes. Second, we investigate the impact of code optimization techniques for main-memory databases. These experiments show that improvements of almost an order of magnitude can be obtained by combining both techniques (cache tuning and code optimization) rather than by each one individually. Our results are fully explained by detailed models of both the partition (radix-cluster) and join phase of partitioned hash-join. We show how performance can exactly be predicted from hardware events like cache and TLB misses, and thus validate the generic cost modeling techniques we developed in the previous chapter. The design of our algorithms paired with the ability to accurately predict there performance build the foundation for automatically tune the algorithms at runtime to yield the efficient utilization of CPU and memory resources.

In Section 5.3, we present radix-join as an alternative to partitioned hash-join, and compare the performance of both algorithms.

In Section 5.4, we evaluate our findings and show how they support the choices we made back in 1994 when designing Monet, which uses full vertical fragmentation and
implementation techniques optimized for main memory to achieve high performance on modern hardware. We conclude with recommendations for future systems.

5.2 Partitioned Hash-Join

Shatdal et al. [SKN94] showed that a main-memory variant of Grace Join, in which both relations are first partitioned on hash-number into $H$ separate clusters, that each fit the memory cache, performs better than normal bucket-chained hash join. This work employs a straightforward clustering-algorithm that simply scans the relation to be clustered once, inserting each scanned tuple in one of the clusters, as depicted in Figure 5.1. This constitutes a random access pattern that writes into $H$ separate locations. If $H$ is too large, there are two factors that degrade performance. First, if $H$ exceeds the number of TLB entries each memory reference will become a TLB miss. Second, if $H$ exceeds the number of available cache lines (L1 or L2), cache thrashing occurs, causing the number of cache misses to explode.

As an improvement over this straightforward algorithm, we propose a clustering algorithm that has a memory access pattern that requires less random-access, even for high values of $H$.

---

1 If the relation is very small and fits the total number of TLB entries times the page size, multiple clusters will fit into the same page and this effect will not occur.
5.2.1 Radix-Cluster Algorithm

The \textit{radix-cluster} algorithm divides a relation $U$ into $H$ clusters using multiple passes (see Figure 5.2). Radix-clustering on the lower $B$ bits of the integer hash-value of a column is achieved in $P$ sequential passes, in which each pass clusters tuples on $B_p$ bits, starting with the leftmost bits ($\sum_1^P B_p = B$). The number of clusters created by the radix-cluster is $H = \prod_1^P H_p$, where each pass subdivides each cluster into $H_p = 2^{B_p}$ new ones. When the algorithm starts, the entire relation is considered one single cluster, and is subdivided into $H_1 = 2^{B_1}$ clusters. The next pass takes these clusters and subdivides each into $H_2 = 2^{B_2}$ new ones, yielding $H_1 \times H_2$ clusters in total, etc.

Note that with $P = 1$, radix-cluster behaves like the straightforward algorithm.

For ease of presentation, we did not use a hash function in the table of integer values displayed in Figure 5.2. In practice, though, it is better to use such a function even on integers in order to ensure that all bits of the table values play a role in the lower bits of the radix number.

The interesting property of the radix-cluster is that the number of randomly accessed regions $H_x$ can be kept low; while still a high overall number of $H$ clusters can be achieved using multiple passes. More specifically, if we keep $H_x = 2^{B_x}$ smaller than the number of cache lines and the number of TLB entries, we totally avoid both TLB and cache thrashing.

After radix-clustering a column on $B$ bits, all tuples that have the same $B$ lowest bits in its column hash-value, appear consecutively in the relation, typically forming chunks of $|U|/2^B$ tuples (with $|U|$ denoting the cardinality of the entire relation). It is therefore not strictly necessary to store the cluster boundaries in some additional data structure; an algorithm scanning a radix-clustered relation can determine the cluster boundaries by looking at these lower $B$ “radix-bits”. This allows very fine clusterings without introducing overhead by large boundary structures. It is interesting to note that a radix-clustered relation is in fact \textit{ordered} on radix-bits. When using this algorithm in the partitioned hash-join, we exploit this property, by performing a merge step on the radix-bits of both radix-clustered relations to get the pairs of clusters that should be hash-joined with each other.

5.2.2 Quantitative Assessment

The radix-cluster algorithm presented in the previous section provides three tuning parameters:

1. the number of radix-bits used for clustering ($B$), implying the number of clusters $H = 2^B$,

2. the number of passes used during clustering ($P$),

3. the number of radix-bits used per clustering pass ($B_p$).

In the following, we present an exhaustive series of experiments to analyze the performance impact of different settings of these parameters. After establishing which
5.2 Partitioned Hash-Join

Parameter settings are optimal for radix-clustering a relation on $B$ radix-bits, we turn our attention to the performance of the join algorithm with varying values of $B$. For both phases, clustering and joining, we investigate how appropriate implementations techniques can improve the performance even further. Finally, these two experiments are combined to gain insight in the overall join performance.

5.2.2.1 Experimental Setup

In our experiments, we use binary relations (BATs) of 8 bytes wide tuples and varying cardinalities ($|U|$), consisting of uniformly distributed random numbers. Each value occurs three times. Hence, in the join-experiments, the join hit-rate is three. The result of a join is a BAT that contains the [OID,OID] combinations of matching tuples (i.e., a join-index [Val87]). Subsequent tuple reconstruction is cheap in Monet, and equal for all algorithms, so just like in [SKN94] we do not include it in our comparison. The experiments were carried out on the machines presented in Section 3.3, an SGI Origin2000, a Sun Ultra, an Intel PC, and an AMD PC (cf., Table 3.2).

To analyze the performance behavior of our algorithms in detail, we break down the overall execution time into the following major categories of costs:

- **Memory access** In addition to memory access costs for data as analyzed above, this category also contains memory access costs caused by instruction cache misses.

- **CPU stalls** Beyond memory access, there are other events that make the CPU stall, like branch mispredictions or other so-called resource related stalls.

- **Divisions** We treat integer divisions separately, as they play a significant role in our hash-join (see below).

- **Real CPU** This is the remaining time, i.e., the time the CPU is indeed busy executing the algorithms.

The four architectures we investigate, provide different hardware counters [BZ98] that enable us to measure each of these cost factors accurately. Table 5.1 gives an overview of the counters used. Some counters yield the actual CPU cycles spent during a certain event, others just return the number of events that occurred. In the latter case, we multiply the counters by the penalties of the events (as calibrated in Section 3.3). None of the architectures provides a counter for the pure CPU activity. Hence, we subtract the cycles spent on memory access, CPU stalls, and integer division from the overall number of cycles and assume the rest to be pure CPU costs.

In our experiments, we found that in our algorithms, branch mispredictions and instruction cache misses do not play a role on either architecture. The reason is that, in contrast to most commercial DBMSs, Monet’s code base is designed for efficient main-memory processing. Monet uses a very large grain size for buffer management in its operators (an entire BAT), processing therefore exhibits much code locality during execution, and hence avoids instruction cache misses and branch mispredictions. Thus, for simplicity of presentation, we omit these events in our evaluation.
Self-tuning Cache-conscious Join Algorithms

<table>
<thead>
<tr>
<th>category</th>
<th>MIPS R10k</th>
<th>Sun UltraSPARC</th>
<th>Intel PentiumIII</th>
<th>AMD Athlon</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory access</td>
<td>L1.d.miss <em>6cy L2.d.miss</em>100cy TLB.miss *57cy</td>
<td>DC.miss*6cy</td>
<td>DCU.miss.. outstanding MTLB *54cy</td>
<td>DC.refill.L2 *27cy</td>
</tr>
<tr>
<td></td>
<td>L1.i.miss <em>6cy L2.i.miss</em>100cy</td>
<td>EC.miss*39cy MTLB *5cy</td>
<td>DC.refill.sys *103cy L1.DTLB.miss <em>5cy L2.DTLB.miss</em>52cy</td>
<td></td>
</tr>
<tr>
<td>CPU stalls</td>
<td>br.mispred *4cy stall.mispred stall.fdep</td>
<td>stall.IC.miss</td>
<td>IFU.mem.stall ITLB.miss *32cy</td>
<td></td>
</tr>
<tr>
<td>divisions</td>
<td></td>
<td></td>
<td>branches.mispred</td>
<td></td>
</tr>
</tbody>
</table>

| | | | |
| | | | |

\[ DC_{miss} = DC_{read} - DC_{read.hit} + DC_{write} - DC_{write.hit} \]
\[ EC_{miss} = EC_{ref} - EC_{hit} \]
\[ Taken from [ADHW99] \]
\[ Taken from [ADHW99] \]
\[ This counter originally includes "DCU.miss.outstanding". We use only the remaining part after subtracting "DCU.miss.outstanding", here. \]

Table 5.1: Hardware Counters used for Execution Time Breakdown

5.2.2.2 Radix Cluster

To analyze the impact of all three parameters \((B, P, B_p)\) on radix clustering, we conduct two series of experiments, keeping one parameter fixed and varying the remaining two.

First, we conduct experiments with various numbers of radix-bits and passes, distributing the radix-bits evenly across the passes. Figure 5.3 shows an execution time breakdown for 1-pass radix-cluster \(|U| = 8M\) on each architecture. The pure CPU costs are nearly constant across all numbers of radix-bits, taking about 3 seconds on the Origin, 5.5 seconds on the Sun, 2.5 seconds on the PentiumIII, and a about 1.7 seconds on the Athlon. Memory and TLB costs are low with small numbers of radix-bits, but grow significantly with rising numbers of radix-bits. With more than 6 radix-bits, the number of clusters to be filled concurrently exceeds the number of TLB entries (64), causing the number of TLB misses to increase significantly. On the Origin and on the Sun, the execution time increases significantly due to their rather high TLB miss penalties. On the PentiumIII however, the impact of TLB misses is hardly visible due to its very low TLB miss penalty. The same holds for TLB\(_1\) misses on the Athlon, while the impact of the more expensive TLB\(_2\) misses is clearly visible. Analogously, the memory costs increase as soon as the number of clusters exceeds the number of L1 and L2 cache lines, respectively. Further, on the PentiumIII, "resource related stalls" (i.e., stalls due to functional unit unavailability) play a significant role. They make up one fourth of the execution time when the memory costs are low. When the memory
5.2 Partitioned Hash-Join

<table>
<thead>
<tr>
<th>a) Origin2000</th>
<th>b) Sun Ultra</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>L1</td>
</tr>
<tr>
<td>seconds</td>
<td>seconds</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>clocks (in billions)</td>
<td>clocks (in billions)</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>5.0</td>
<td>4.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c) Intel PC</th>
<th>d) AMD PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>L1</td>
</tr>
<tr>
<td>seconds</td>
<td>seconds</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>clocks (in billions)</td>
<td>clocks (in billions)</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>9.0</td>
<td>8.0</td>
</tr>
</tbody>
</table>

(resource stalls | DCU misses | TLB (model) | CPU | resource stalls | DCU misses | TLB (model) | CPU)

(Vertical grid lines indicate, where $H = 2^8$ equals $\#_{TLB}$, $\#_{L1}$, or $\#_{L2}$, respectively.)

Figure 5.3: Execution Time Breakdown of Radix-Cluster using one pass ($|U| = 8M$)
Figure 5.4: Execution Time Breakdown of Radix-Cluster using optimal number of passes (|U| = 8M)

c) Intel PC

d) AMD PC

costs rise, the resource related stalls decrease and finally vanish completely, reducing the impact of the memory penalty. In other words, minimizing the memory access costs does not fully pay back on the PentiumIII, as the resource related stalls partly take over their part. The Athlon, however, does not seem to suffer from such "resource related stalls".

Figure 5.4 depicts the breakdown for radix-cluster using the optimal number of
5.2 Partitioned Hash-Join

```c
#define HASH(v) ((v>>7) XOR (v>>13) XOR (v>>21) XOR v)

typedef struct {
  int v1,v2; /* simplified binary tuple */
} bun;

radix_cluster(bun *dst[2^D], bun *dst_end[2^D], /* output buffers for created clusters */
  bun *rel, bun *rel_end, /* input relation */
  int R, int D /* radix and cluster bits */
) {
  int M = (2^D - 1) << R;
  for(bun*cur=rel; cur<rel_end; cur++) {
    int idx = (*hashFcn)(cur->v2)&M;
    memcpy(dst[idx], cur, sizeof(bun));
    if (dst[idx]>dst_end[idx]) REALLOC(dst[idx],dst_end[idx]);
    *dst[idx] = *cur;
  }
}
```

Figure 5.5: C language radix-cluster with annotated CPU optimizations (right)

pass. The idea of multi-pass radix-cluster is to keep the number of clusters generated per pass low—and thus the memory costs—at the expense of increased CPU costs. Obviously, the CPU costs are too high to avoid the TLB costs by using two passes with more than 6 radix-bits. Only with more than 15 radix-bits—i.e., when the memory costs exceed the CPU costs—two passes win over one pass. The only exception is the Athlon, where multi-pass radix-cluster benefits from the high clock speed, and hence, two passes outperform one pass already from 11 radix-bits onward.

The only way to improve this situation is to reduce the CPU costs. Figure 5.5 shows the source code of our radix-cluster routine. It performs a single-pass clustering on the $D$ bits that start $R$ bits from the right (multi-pass clustering in $P > 1$ passes on $B = P \times D$ bits is done by making subsequent calls to this function for pass $p = 1$ through $p = P$ with parameters $D_p = D$ and $R_p = (p - 1) \times D$, starting with the input relation and using the output of the previous pass as input for the next). As the algorithm itself is already very simple, improvement can only be achieved by means of implementation techniques. We replaced the generic ADT-like implementation by a specialized one for each data type. Thus, we could inline the hash function and replace the `memcpy` by a simple assignment, saving two function calls per iteration.

Figure 5.6 shows the execution time breakdown for the optimized 1-pass radix-cluster. The CPU costs have reduced significantly, by almost a factor 4. Replacing the two function calls has two effects. First, some CPU cycles are saved. Second, the CPUs can benefit more from the internal parallel capabilities using speculative execution, as the code has become simpler and parallelization options more predictable. On the PentiumIII, the resource stalls have doubled, neutralizing the CPU improvement partly. We think the simple loop does not consist of enough instructions to fill the relatively long pipelines of the PentiumIII efficiently.

The results in Figure 5.7 indicate that with the optimized code, multi-pass radix-cluster is feasible already with smaller numbers of radix-bits. On the Origin, two passes win with more than 6 radix-bits, and three passes win with more than 13 radix-
Figure 5.6: Execution Time Breakdown of optimized Radix-Cluster using one pass 
\(|U| = 8M\)
bits, thus avoiding TLB thrashing nearly completely. Analogously, the algorithm creates at most 512 clusters per pass on the AMD PC, avoiding L1 thrashing which is expensive due to the rather high L1 miss penalty on the Athlon. For the Pentium III, however, the improvement is marginal. The severe impact of resource stalls with low numbers of radix-bits makes the memory optimization of multi-pass radix-cluster almost ineffective.

In order to effectively exploit the performance potential of our radix-cluster algorithm, we need to know the optimal number of passes to be used with a given number of radix-bits on a given machine. Obviously, we could give a rule of thumb for each machine we discussed here, like “Never use more than 6 bits (64 clusters) per pass on an Origin2000.” However, a more general approach is clearly desirable. Our proposal is to build a cost function for radix-cluster, that allows us to estimate and compare the performance for various numbers of passes. The best performance determines the optimal numbers of passes. We use the techniques we presented in Chapter 4 to create a cost function for radix-cluster estimating the total execution time as sum of pure CPU time and memory access time. With $U$ describing the input data region, $B$ denoting
the requested number of radix-bits, and \( P \) being the number of passes, we get

\[
T_c(U, B, P) = T_{CPU}^c(U, P) + T_{Mem}^c(U, B, P).
\]

The pure CPU costs are equal for all passes and independent of the number of bits used. For each pass, the pure CPU costs consist of a fixed start-up cost \( c_0^c \) and the per-tuple costs \( c_1^c \). We determine \( c_0^c \) and \( c_1^c \) using calibration as described in Section 4.6. Hence, we have

\[
T_{CPU}^c(U, P) = P \cdot (c_0^c + |U| \cdot c_1^c).
\]

The memory access cost is determined by the data access pattern. With each pass, radix-cluster sequentially reads the input and puts each tuple into one of \( H_p = 2^B_p \) output clusters. Within each cluster, the access is sequential, but the clusters are accessed in a random order. Hence, we model the data access pattern of radix-cluster by

\[
\{U_j\}_{j=1}^{2^B} \leftarrow \text{radix_cluster}(U, B, P):
\]

\[
\oplus \big|_{p=1}^P \text{s_trav}^s(U) \circ \text{nest} \left( \{U_j\}_{j=1}^{2^B}, 2^B, \text{s_trav}^s(U_j), \text{ran} \right).
\]

\( T_{Mem}^c(U, B, P) \) is then calculated by estimating the number of cache misses and scoring them with their latency as described in Chapter 4.

Figure 5.8 compares the predicted events (lines) with the events observed during our experiments (points) on the Origin2000 for different cardinalities. The model accurately predicts the performance crucial behavior of radix-cluster, i.e., the steep raise in cache/TLB misses as soon as we generate more clusters per pass than there are cache lines respectively TLB entries. According to Figure 5.9, scoring the misses with their latency yields a reasonably accurate prediction of the total execution time on all architectures. The plots clearly reflect the impact of increasing cache and TLB misses on the execution time. Hence, our model provides sufficiently accurate information to determine the optimal number of passes for a given number of radix-bits. The effort needed to do so is quite limited. Consider the largest table in our experiments, containing 64,000,000 tuples. In the worst case, we need to create 64,000,000 clusters, each containing only a single tuple. Hence, we use at most 26 radix-bits. This means that theoretically at most 26 alternatives need to be compared. However, our experiments indicate, that we do not have to use more passes than we need to stay just within the smallest cache. With the 64-entry TLBs being the smallest caches in our case, there is no need to use less than 5 bits per pass, hence we can restrict the search to just \( \left\lceil \frac{26}{5} \right\rceil = 6 \) alternatives.

The question remaining is how to distribute the number of radix-bits over the passes. Of course, we could also use our cost model, here. However, with \( \binom{B-1}{P-1} \) ways to distribute \( B \) radix-bits on \( P \) passes, the number of alternatives that need to be explored grows rapidly. For instance in our previous example, we need to distribute 26 bits on up to 6 passes, hence a total of

\[
\sum_{j=1}^{6} \binom{26 - 1}{j - 1} = 68,406
\]
5.2 Partitioned Hash-Join

![Graphs showing L1, L2, and TLB misses for different numbers of radix-bits](image)

- (Point types indicate cardinalities, line types indicate number of passes; vertical grid lines indicate, where the number of clusters created equals the number of TLB entries, L1, or L2 cache lines, respectively.)

Figure 5.8: Measured (points) and Modeled (lines) Events of Radix-Cluster (Origin2000)

Candidates would need to be checked to find the best values for $P, B_1, \ldots, B_P$.

Alternatively, we consider the following empirical approach. We conducted another number of experiments, using a fix number of passes, but varying the number of radix-bits per pass. Figure 5.10 depicts the respective results for 4, 8, 12, 16, 20, and 24 radix-bits, using 2 passes. The x-axis shows $B + \frac{B_1}{5}$, hence, for each number of radix-bits ($B = B_1 + B_2$) there is a short line segment consisting of $B - 1$ points. The first (leftmost) point of each segment represents $B_1 = 1, B_2 = B - 1$, the last (rightmost) point represents $B_1 = B - 1, B_2 = 1$. The results show that even distribution of radix-bits ($B_1 \approx B_2 \approx \frac{B}{2}$) achieves the best performance. Indeed, these results are quite intuitive, as even distribution of radix-bits minimizes the maximum number of bits per pass.
(Point types indicate cardinalities, line types indicate number of passes; vertical grid lines indicate, where the number of clusters created equals the number of TLB entries, L1, or L2 cache lines, respectively.)

Figure 5.9: Measured (points) and Modeled (lines) Performance of Radix-Cluster
5.2.2.3 Isolated Partitioned Hash-Join Performance

We now analyze the impact of the number of radix-bits on the pure join performance, not including the clustering costs. With 0 radix-bits, the join algorithm behaves like a simple non-partitioned hash-join.

The partitioned hash-join exhibits increased performance with increasing number of radix-bits. Figure 5.11 shows that this behavior is mainly caused by the memory costs. While the CPU costs are almost independent of the number of radix-bits, the memory costs decrease with increasing number of radix-bits. The performance increase flattens past the point where the entire inner cluster (including its hash table) consists of less pages than there are TLB entries (64). Then, it also fits the L2 cache comfortably. Thereafter, performance increases only slightly until the point that the inner cluster fits the L1 cache. Here, performance reaches its maximum. The fixed overhead by allocation of the hash-table structure causes performance to decrease when the cluster sizes get too small and clusters get very numerous. Again, the PentiumIII shows a slightly different behavior. TLB costs do not play any role, but “partial stalls” (i.e., stalls due to dependencies among instructions) are significant with small numbers of radix-bits. With increasing numbers of clusters, the partial stalls decrease, but then, resource stalls increase, almost neutralizing the memory optimization.

Like with radix-cluster, once the memory access is optimized, the execution of partitioned hash-join is dominated by CPU costs. Hence, we applied the same optimizations as above. We inlined the hash-function calls during hash build and hash probe as well as the compare-function call during hash probe and replaced two memcpy by simple assignments, saving five function calls per iteration. Further, we replaced the modulo division (“%”) for calculating the hash index by a bit operation (“&”). Figure 5.12 depicts the original implementation of our hash-join routine and the opti-
Figure 5.11: Execution Time Breakdown of Partitioned Hash-Join (|U| = |V| = 8M)
5.2 Partitioned Hash-Join

<table>
<thead>
<tr>
<th>hash_join(bun *dst, bun <em>end /</em> start and end of result buffer */</th>
<th>bun *outer, bun <em>outer_end, bun <em>inner, bun</em> inner_end, /</em> inner and outer relations */</th>
<th>int R /* radix bits */</th>
</tr>
</thead>
<tbody>
<tr>
<td>/* build hash table on inner */</td>
<td>int pos=0, S=inner_end-inner, H=log2(S), N=2^H, M=(N-1)/R;</td>
<td>int idx = ((hashFcn)(cur-&gt;v2)&gt;&gt;R) % N;</td>
</tr>
<tr>
<td>int next[S], bucket[N] = { -1 }; /* hash bucket array and chain-lists */</td>
<td>/* hash table */</td>
<td>int idx = HASH(cur-&gt;v2) &amp; M;</td>
</tr>
<tr>
<td>for(bun *cur=inner; cur&lt;inner_end; cur++) {</td>
<td></td>
<td>next[pos] = bucket[idx];</td>
</tr>
<tr>
<td>int idx = ((hashFcn)(cur-&gt;v2)&gt;&gt;R) % N;</td>
<td></td>
<td>bucket[idx] = pos++;</td>
</tr>
<tr>
<td>}</td>
<td>/* probe hash table with outer */</td>
<td>/* probe hash table with outer */</td>
</tr>
<tr>
<td>for(bun *cur=outer; cur&lt;outer_end; cur++) {</td>
<td></td>
<td>int idx = HASH(cur-&gt;v2) &amp; M;</td>
</tr>
<tr>
<td>int idx = ((hashFcn)(cur-&gt;v2)&gt;&gt;R) % N;</td>
<td></td>
<td>for(int hit=bucket[idx]; hit&gt;=0; hit=next[hit]) {</td>
</tr>
<tr>
<td>if ((compareFcn)(cur-&gt;v2,inner[hit].v2)==0) {</td>
<td></td>
<td>if ((cur-&gt;v2 == inner[hit].v2)) {</td>
</tr>
<tr>
<td>memcpy(&amp;dst-&gt;vl,&amp;cur-&gt;vl,sizeof(int));</td>
<td></td>
<td>dst-&gt;v1 = cur-&gt;v1;</td>
</tr>
<tr>
<td>memcpy(&amp;dst-&gt;v2,inner[hit].v1,sizeof(int));</td>
<td></td>
<td>dst-&gt;v2 = inner[hit].v1;</td>
</tr>
<tr>
<td>if (++dst&gt;=end) REALLOC(dst,end);</td>
<td></td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

Figure 5.12: C language hash-join with annotated CPU optimizations (right)

mizations we applied.

Figure 5.13 shows the execution time breakdown for the optimized partitioned hash-join. For the same reasons as with radix-cluster, the CPU costs are reduced by almost a factor 4 on the Origin and the Sun, by factor 3 on the PentiumIII, and by factor 2 on the Athlon. The expensive divisions have vanished completely. Additionally, the dependency stalls on the PentiumIII have disappeared, but the functional unit stalls remain almost unchanged, now taking about half of the execution time. It is interesting to note the 450 MHz PC outperforms the 250 MHz Origin on non-optimized code, but on CPU optimized code, where both RISC chips execute without any overhead, the PC actually becomes slower due to this phenomenon of resource stalls.

Like with radix-cluster, we will now create a cost function for partitioned hash-join that estimates the total execution time as sum of pure CPU time and memory access time. Let $U$ and $V$ describe the left (outer) and right (inner) input data region, respectively, and $B$ denote the number of radix-bits that both $U$ and $V$ are clustered on. Further let $W$ represent the output data region$^2$

$$T_{\text{phj}}^{\text{CPU}}(U, V, B, W) = T_{\text{phj}}^{\text{CPU}}(U, V, B, W) + T_{\text{mem}}^{\text{phj}}(U, V, B, W).$$

On each of the $H = 2^B$ pairs $(U_p, V_p)$ of matching partitions, partitioned hash-join

---

$^2$Here, we use $W$ to convey the results of a logical cost model, such as the estimates result size, to the physical cost model.
performs a simple hash-join. The latter in turn consists of two phases, first building the hash table on the inner partition \( V_p \), and then probing the outer partition \( U_p \) against the hash table. The pure CPU cost of the first phase is linear in the cardinality of \( V_p \). The pure CPU cost of the second phase consists of two components. The first represents the actual hash lookup, and is in our case linear in the cardinality of \( U_p \) (i.e.,
independent of the size of the hash table). The second component reflects the creation of the actual result tuples, and is hence linear in the cardinality of \( W_p \). Altogether, we get

\[
T_{CPU}^{bb}(V_p) = c_0^{bb} + |V_p| \cdot c_1^{bb}
\]

\[
T_{CPU}^{hp}(U_p, W_p) = c_0^{hp} + |U_p| \cdot c_1^{hp} + |W_p| \cdot c_2^{hp}
\]

\[
T_{CPU}^{hij}(U_p, V_p, W_p) = T_{CPU}^{bb}(V_p) + T_{CPU}^{hp}(U_p, W_p)
\]

\[
T_{CPU}^{phj}(U, V, B, W) = \sum_{p=1}^{2^g} T_{CPU}^{hij}(U_p, V_p, W_p)
\]

We determine the respective cost constants using calibration as described in Section 4.6.

To obtain the memory access costs, we describe the memory access pattern of partitioned hash-join by combining the patterns of the single phases as follows:

\[
V'_p \leftarrow hash\_build(V_p) :
\]

\[
s\_trav(V_p) \odot r\_trav(V'_p)
\]

\[= build\_hash(V_p, V'_p)\]

\[
W_p \leftarrow hash\_probe(U_p, V'_p) :
\]

\[
s\_trav(U_p) \odot r\_acc(|U_p|, V'_p) \odot s\_trav(W_p)
\]

\[= probe\_hash(U_p, V'_p, W_p)\]

\[
W_p \leftarrow hash\_join(U_p, V_p) :
\]

\[
build\_hash(V_p, V'_p) \oplus probe\_hash(U_p, V'_p, W_p)
\]

\[= hash\_join(U_p, V_p, W_p)\]

\[
\{W_p\}_{p=1}^{2^g} \leftarrow part\_hash\_join(\{U_p\}_{p=1}^{2^g}, \{V_p\}_{p=1}^{2^g}, B) :
\]

\[\oplus |\{W_p\}_{p=1}^{2^g}| (hash\_join(U_p, V_p, W_p))\]

The actual cost functions are then derived using the techniques of Chapter 4.

Figure 5.14 compares the predicted events (lines) with the events observed during our experiments (points) on the Origin2000 for different cardinalities. The model accurately predicts the performance crucial behavior of partitioned hash-join, i.e., the significantly increased number of cache/TLB misses when using partition sizes that exceed the respective caches capacities. The plots in Figure 5.15 confirm that the estimated execution times match the actual performance on all architectures reasonably well.


5 Self-tuning Cache-conscious Join Algorithms

5.2.2.4 Overall Partitioned Hash-Join Performance

After having analyzed the impact of the tuning parameters on the clustering phase and the joining phase separately, we now turn our attention to the combined cluster and join costs. Radix-cluster gets cheaper for fewer radix-bits, whereas partitioned hash-join gets more expensive. Putting together the experimental data we obtained on both cluster- and join-performance, we determine the optimum number of $B$ for given relation cardinality.

It turns out that there are three possible strategies, which correspond to the diagonals in Figure 5.15:

**phash L2** partitioned hash-join on $B = \log_2(|V'|*\bar{V}'/||L2||)$ clustered bits, so the inner relation plus hash-table fits the L2 cache. This strategy was used in the work of Shatdal et al. [SKN94] in their partitioned hash-join experiments.

**phash TLB** partitioned hash-join on $B = \log_2(|V'|*\bar{V}'/||TLB||)$ clustered bits, so the
5.2 Partitioned Hash-Join

(Point types indicate cardinalities; diagonal lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)

Figure 5.15: Measured (points) and Modeled (lines) Time of Partitioned Hash-Join
inner relation plus hash-table spans at most \(|TLB|\) pages. Our experiments show a significant improvement of the pure join performance between phash L2 and phash TLB.

**phash L1** partitioned hash-join on \(B = \log_2(|V'| \cdot \frac{\overline{V}}{|L1|})\) clustered bits, so the inner relation plus hash-table fits the L1 cache. This algorithm uses more clustered bits than the previous ones, hence it really needs the multi-pass radix-cluster algorithm (a straightforward 1-pass cluster would cause cache thrashing on this many clusters).

Figure 5.16 shows the overall performance for the original (thin lines) and the CPU-optimized (thick lines) versions of our algorithms, using 1-pass and multi-pass clustering. In most cases, phash TLB is the best strategy, performing significantly better than phash L2. On the Origin2000 and the Sun, the differences between phash TLB and phash L1 are negligible. On the PCs, phash L1 performs sightly better than phash TLB. With very small cardinalities, i.e., when the relations do not span more memory pages than there are TLB entries, clustering is not necessary, and the non-partitioned hash-join ("simple hash") performs best.

Using phash TLB seems to be a good choice the achieve reasonably good performance on all architectures without any optimization effort. However, our goal is to find and use the optimal strategy, also on architectures we have not evaluated here. Our cost models provide the necessary tools to achieve this goal. Using the cost models, we can—at runtime—estimate the costs of radix-cluster and partitioned hash-join for various numbers radix-bits (i.e., partition sizes). Thus, we can find the best setting for number of radix-bits and number of cluster passes to be used in the actual evaluation of the algorithms. Figure 5.17 compares the estimated performance (lines) with the results of our experiments (points) for 8M tuples. Our models tend to overestimate the “worst-case” costs, i.e., when using too large partitions for partitioned hash-join or too many partitions for radix-cluster. But in the crucial area around the optimal performance, they are rather accurate. Moreover, the models correctly predict the optimum in almost all cases. In case they do not, the cost of the estimated optimum only marginally differ from the actual optimum.

Further, the results in Figure 5.17 show that CPU and memory optimization support each other and boost their effects. The gain of CPU optimization for phash TLB is bigger than that for simple hash, and the gain of memory optimization for the CPU-optimized implementation is bigger than that for the non-optimized implementation. For example, for large relations on the Origin 2000, CPU optimization improves the execution time of simple hash by approximately a factor 1.25, whereas it yields a factor 3 with phash TLB. Analogously, memory optimization achieves an improvement of slightly less than a factor 2.5 for the original implementation, but more than a factor 5 for the optimized implementation. Combining both optimizations improves the execution time by almost a factor 10.

There are two reasons for the boosting effect to occur. First, modern CPUs try to overlap memory access with other useful CPU computations by allowing independent instructions to continue execution while other instructions wait for memory. In a
5.2 Partitioned Hash-Join

![Graphs showing performance comparison between non-optimized and optimized implementations of Partitioned Hash-Join for different computer models: Origin2000, Sun Ultra, Intel PC, and AMD PC.]

Figure 5.16: Overall Performance of Partitioned Hash-Join: non-optimized (thin lines) vs. optimized (thick lines) implementation. (Line types indicate cardinalities, point types indicate number of passes; diagonal lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)
5 Self-tuning Cache-conscious Join Algorithms

Figure 5.17: Measured (points) and Modeled (lines) Overall Performance of Partitioned Hash-Join ($|U| = |V| = 8M$)

(VERTICAL grid lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)
memory-bound load, much CPU computation is overlapped with memory access time, hence optimizing these computations has no overall performance effect (while it does when the memory access would be eliminated by memory optimizations). Second, an algorithm that allows memory access to be traded for more CPU processing (like radix-cluster), can actually trade more CPU for memory when CPU-cost are reduced, reducing the impact of memory access costs even more.

The Sun Ultra and the AMD PC achieve similar results like the Origin2000, although the absolute gains are somewhat smaller. With the Ultra, the CPU is so slow that trading memory for CPU less beneficial on this platform; with the AMD PC, the memory access costs are somewhat lower than on the Origin2000, thus offering less potential for improvements.

The overall effect of our optimizations on the PentiumIII is just over a factor 2. One cause of this is the low memory latency on the Intel PC, that limits the gains when memory access is optimized. The second cause is the appearance of the “resource-stalls”, which surge in situations where all other stalls are eliminated (and both RISC architectures are really steaming). We expect, though, that future PC hardware with highly parallel IA-64 processors and new Rambus memory systems (that offer high bandwidth but high latencies) will show a more RISC-like performance on our algorithms.

5.3 Radix-Join

In this section, we present our *radix-join* algorithm as an alternative for the partitioned hash-join. Radix-join makes use of the very fine clustering capabilities of radix-cluster. If the number of clusters $H$ is high, the radix-clustering has brought the potentially matching tuples near to each other. As cluster sizes are small, a simple nested loop is then sufficient to filter out the matching tuples. Radix-join is similar to hash-join in the sense that the number $H$ should be tuned to be the relation cardinality $|U|$ divided by a small constant; just like the length of the bucket-chain in a hash-table. If this constant gets down to 1, radix-join degenerates to sort/merge-join, with radix-sort [Knu68] employed in the sorting phase.

5.3.1 Isolated Radix-Join Performance

Figure 5.18 shows the execution time breakdown for our radix-join algorithm ($|U| = 1M$). On all three architectures\(^3\), radix-join performs the better the more radix-bits are used, i.e., the smaller the clusters are. With increasing cluster size, execution time increases rapidly due to the nested-loop characteristic of radix-join. Only cluster sizes that fit into the L1 cache are reasonable. Hence, memory access costs are small, and the performance is dominated by CPU costs.

Like with radix-cluster and partitioned hash-join, we optimize our radix-join implementation (cf., Figure 5.19) in order to reduce the CPU costs. Figure 5.20 depicts the results for the optimized radix-join. For very small clusters, the optimizations

---

\(^3\)Due to hardware problems, we have no results of radix-join on the AMD PC.
yield an improvement of factor 1.25 (on the PC) to 1.5 (on the Origin2000). For larger clusters, we observe similar improvements as with partitioned hash-join: factor 4.5 on the RISC architectures and factor 2.5 on the PC. We also note that functional unit stalls (resource stalls) become clearly visible on the PC, as soon as the cluster size reaches and exceeds L1 cache size.

We follow our usual schema to model the cost of radix-join.

\[ T_{ij}^d(U, V, B, W) = T_{CPU}^{ij}(U, V, B, W) + T_{Mem}^{ij}(U, V, B, W). \]

The total CPU costs are simply the sum of the CPU costs of executing a simple nested-loop-join on all matching pairs of partitions. A simple nested-loop-join incurs constant start-up cost \( (c_0^{ij}) \), quadratic comparison cost \( (c_1^{ij}) \), and linear result creation cost
5.3 Radix-Join

```
5.3.3 Radix-Join

```nested_loop(bun *dst, bun *end
bun *outer, bun *outer_end, bun *inner, bun * inner_end,
) {
  for(bun *outer_cur=outer; outer.cur < outer.end; outer.cur++) {
    for(bun *innerCur=inner; inner.cur < inner.end; inner.cur++) {
      if (**start and end of result buffer */
      / start and end of result buffer */
      / inner and outer relations */
      if ((*compareFcn)(outer.cur->v2,inner.cur->v2)==0) {
        memcpy(&dst->v1,&outer.cur->v1,sizeof(int));
        dst->v1 = outer.cur->v1;
        dst->v22 = inner.cur->v2;
      }
      if (++dst[end]) REALLOC(dst,end);
    }
  }
}

Figure 5.19: C language nested-loop with annotated CPU optimizations (right)

Figure 5.20: Execution Time Breakdown of optimized Radix-Join (|U| = |V| = 1M)

\[ T_{CPU}^{nij}(U_p, V_p, W_p) = c_0^{nij} + |U_p| \cdot |V_p| \cdot c_1^{nij} + |W_p| \cdot c_2^{nij} \]

\[ T_{CPU}^i(U, V, B, W) = \sum_{p=1}^{2^8} T_{CPU}^{nij}(U_p, V_p, W_p) \]

We note that calibrating the CPU cost factors as described in Section 4.6 ensures that
the resource stalls on the PC (see above) are properly included in the cost factors.

To get the memory access cost function for radix-join, we model its memory access pattern as follows.

\[ W_p \leftarrow \text{nested}\_loop\_join(U_p, V_p) : \]
\[ s\_trav^* (U_p) \odot rs\_trav^* (|U_p|, \text{uni}, V_p) \odot s\_trav^* (W_p) =: n\_join(U_p, V_p, U_p) \]

\[ \{W_p\}_{p=1}^{2^n} \leftarrow \text{radix}\_join(\{U_p\}_{p=1}^{2^n}, \{V_p\}_{p=1}^{2^n}, B) : \]
\[ \odot \{n\_join(U_p, V_p, W_p)\}_{p=1}^{2^n} \]

Figure 5.21 confirms the accuracy of our model (lines) for the number of L1, L2, and TLB misses on the Origin2000, and for the elapsed time on all architectures.

### 5.3.2 Overall Radix-Join Performance

Figure 5.22 shows the overall performance of radix-join, i.e., including radix-cluster of both input relations. For larger clusters, the performance is dominated by the high CPU costs of radix-join. Only very small clusters with up to 8 tuples are reasonable, requiring multi-pass radix-clustering in most cases.

### 5.3.3 Partitioned Hash-Join vs. Radix-Join

Finally, Figure 5.23 compares the overall performance of partitioned hash-join and radix-join. With the original non-optimized implementation, the optimal performance of radix-join (circled points on thin lines) is almost the same as the optimal partitioned hash-join performance (circled points on thick lines). With code optimizations applied, optimal radix-join performance is never better than partitioned hash-join using the same number of radix-bits.

### 5.4 Evaluation

In the previous sections, we have demonstrated that performance of large equi-joins can be strongly improved by combining techniques that optimize memory access and CPU resource usage. As discussed in Section 3.4.3, hardware trends indicate that the effects of such optimizations will become even larger in the future, as the memory access bottleneck will deepen and future CPUs will have even more parallel resources. In the following, we discuss the more general implications of these findings to the field of database architecture.
5.4 Evaluation

Figure 5.21: Measured (points) and Modeled (lines) Events (Origin2000)
Performance of Radix-Join

(Events are in absolute numbers; times are in milliseconds. Point types indicate cardinalities; diagonal lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)
5.4.1 Implications for Implementation Techniques

Implementation techniques strongly determine how CPU and memory are used in query processing, and have been the subject of study in the field of main-memory database engineering [DKO'84], where query processing costs are dominated by CPU processing. First, we present some rules of thumb, that specifically take into account the modern hardware optimization aspects, then we explain how they were implemented in Monet:

- **use the most efficient algorithm.** Even the most efficient implementation will not make a sub-optimal algorithm perform well. A more subtle issue is tuning algorithms with the optimal parameters.

- **minimize memory copying.** Buffer copying should be minimized, as it both wastes CPU cycles and also causes spurious main-memory access. As function
5.4 Evaluation

Non-optimized implementation

Optimized implementation

Times are in milliseconds. Line types indicate cardinalities, point types indicate number of passes.

Figure 5.23: Partitioned Hash-Join (thick lines) vs. Radix-Join (thin lines): non-optimized (top) and optimized (bottom)
calls copy their parameters on the stack, they are also a source of memory copying, and should be avoided in the innermost loops that iterate over all tuples. A typical function call overhead is about 20 CPU cycles.

- *allow compiler optimizations*. Techniques like memory prefetching, and generation of parallel EPIC code in the IA-64, rely on compilers to detect independence of certain statements. These compiler optimizations work especially well if the hotspot of the algorithm is one simple loop that is easily analyzable for the compiler. Again, performing function calls in these loops, force the compiler to assume the worst (side effects) and prevent optimizations from taking place. This especially holds in database code, where those function calls cannot be analyzed at compile time, since the database atomic type interface makes use of C dereferenced calls on a function-pointer looked up in an ADT table, or C++ late-binding methods.

As an example of correctly tuning algorithms, we discuss the (non-partitioned) hash-join implementation of Monet that uses a simple bucket-chained hash-table. In a past implementation, it used a default mean bucket chain length of four [BMK99], where actually a length of one is optimal (perfect hashing). Also, we had used integer division (modulo) by a prime-number (the number of hash buckets) to obtain a hash-bucket number, while integer division costs 40-80 cycles on current CPUs. Later, we changed the number of hash buckets to be a power of 2 (i.e., \( N = 2^k \)), and hence, we could replace the expensive modulo division by a much cheaper bit-wise AND with \( N-1 \). Such simple tuning made the algorithm more than 4 times faster.

In order to minimize copying, Monet does not do explicit buffer management, rather it uses virtual memory to leave this to the OS. This avoids having to copy tuple segments in and out of a buffer manager, whenever the DBMS accesses data. Monet maps large relations stored in a file into virtual memory and accesses it directly. Minimizing memory copying also means that pointer swizzling is avoided at all time by not having hard pointers and value-packing in any data representation.

Functions calls are minimized in Monet by applying *logarithmic code expansion* [Ker89]. Performance-critical pieces of code, like the hash-join implementation, are replicated in specific functions for the most commonly used types. For example, the hash-join is separated in an integer-join, a string-join, etc., and an ADT join (that handles all other types). The specific integer-join processes the table values directly as C integers, without calling a hash-function for hashing, or calling a comparison function when comparing two values. The same technique is applied for constructing the result relation, eliminating function calls for inserting the matching values in the result relation. To make this possible, the type-optimized join implementations require the result to have a fixed format: a join index containing OIDs (in Monet the result of joining two BATs is again a BAT, so it has a fixed binary format, and typical invocations produce a BAT with matching OID pairs). In this way, all function calls can be removed from an algorithm in the optimized cases. For the non-optimized cases, the (slower) but equivalent implementation is employed that uses ADT method calls for manipulating values. The Monet source code is kept small by generating both
the optimized and ADT code instantiations with a macro package from one template algorithm. We refer to [BK99] for a detailed discussion of this subject.

5.4.2 Implications for Query Processing Algorithms

Our join experiments demonstrated that performance can strongly improve when algorithms that have a random memory access pattern are tuned, in order to ensure that the randomly accessed region does not exceed the cache size (be it L1, L2, or TLB). In the case of join, we confirmed results of Shatdal et al. who had proposed a partitioned hash-join such that each partition joined fits the L2 cache [SKN94], and showed that the beneficial effect of this algorithm is even stronger on modern hardware. Secondly, we introduced a new partitioning algorithm called radix-cluster that performs multiple passes over the data to be partitioned but earns back this extra CPU work with much less memory access costs when the number of partitions gets large.

We believe that similar approaches can be used to optimize algorithms other than equi-join. For instance, Ronström [Ron98] states, that a B-tree with a block-size equal to the L2 cache line size as a main-memory search accelerator, now outperforms the traditionally known-best main-memory T-tree search structure [LC86a]. As another example, memory cost optimizations can be applied to sorting algorithms (e.g., radix-cluster followed by quick-sort on the partitions), and might well change the trade-offs for other well-known main-memory algorithms (e.g., radix-sort has a highly cachable memory access pattern and is likely to outperform quick-sort).

Main-memory cost models are a prerequisite for tuning the behavior of an algorithm to optimize memory cache usage, as they allow to make good optimization decisions. Our work shows that such models can be obtained and how to do it. First, we show with our calibration tool how all relevant hardware characteristics can be retrieved from a computer system automatically. This calibrator does not need any OS support whatsoever, and should in our opinion be used in modern DBMS query optimizers. Secondly, we present a methodological framework that first characterizes the memory access pattern of an algorithm to be modeled in a formula that counts certain hardware events. These computed events are then scored with the calibrated hardware parameters to obtain a full cost model. This methodology represents an important improvement over previous work on main-memory cost models [LN96, WK90], where performance is characterized on the coarse level of a procedure call with "magical" cost factors obtained by profiling. We were helped in formulating this methodology through our usage of hardware event counters present in modern CPUs.

5.4.3 Implications for Disk Resident Systems

We think our findings are not only relevant to main-memory databases engineers. Vertical fragmentation and memory access costs have a strong impact on performance of database systems at a macro level, including those that manage disk-resident data. Nyberg et al. [NBC+'94] stated that techniques like software assisted disk-striping have reduced the I/O bottleneck; i.e., queries that analyze large relations (like in OLAP or Data Mining) now read their data faster than it can be processed. Hence the main
performance bottleneck for such applications is shifting from I/O to memory access. We therefore think that, as the I/O bottleneck decreases and the memory access bottleneck increases, main-memory optimization of both data structures and algorithms—like described in this paper—will become a prerequisite to any DBMS for exploiting the power of custom hardware.

In Monet, we delegate I/O buffering to the OS by mapping large data files into virtual memory, hence treat management of disk-resident data as memory with a large granularity (a memory page is like a large cache line). This is in line with the consideration that disk-resident data is the bottom level of a memory hierarchy that goes up from the virtual memory, to the main memory through the cache memories up to the CPU registers (Figure 3.2). Algorithms that are tuned to run well on one level of the memory, also exhibit good performance on the lower levels.

5.5 Conclusion

We have shown what steps are taken in order to optimize the performance of large main-memory joins on modern hardware. To achieve better usage of scarce memory bandwidth, we recommend using vertically fragmented data structures. We refined partitioned hash-join with a new partitioning algorithm called radix-cluster, that prevents performance becoming dominated by memory latency (avoiding the memory access bottleneck). Exhaustive equi-join experiments were conducted on modern SGI, Sun, Intel, and AMD hardware. We formulated detailed analytical cost models that explain why this algorithm makes optimal use of hierarchical memory systems found in modern computer hardware and very accurately predict performance on all three platforms. Further, we showed that once memory access is optimized, CPU resource usage becomes crucial for the performance. We demonstrated, how CPU resource usage can be improved by using appropriate implementation techniques. The overall speedup obtained by our techniques can be almost an order of magnitude. Finally, we discussed the consequences of our results in a broader context of database architecture, and made recommendations for future systems.