The VLDB Journal (2000) 9: 231–246

The VLDB Journal

Optimizing database architecture for the new bottleneck: memory access

Stefan Manegold[1], **Peter A. Boncz**[2,*], **Martin L. Kersten**[1]

[1] CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands; E-mail: {S.Manegold,M.L.Kersten}@cwi.nl
[2] Data Distilleries B.V., Kruislaan 402, 1098 SM Amsterdam, The Netherlands; E-mail: P.Boncz@ddi.nl

Edited by M.P. Atkinson. Received April 20, 2000 / Accepted June 23, 2000

Abstract. In the past decade, advances in the speed of commodity CPUs have far out-paced advances in memory latency. Main-memory access is therefore increasingly a performance bottleneck for many computer applications, including database systems. In this article, we use a simple scan test to show the severe impact of this bottleneck. The insights gained are translated into guidelines for database architecture, in terms of both data structures and algorithms. We discuss how vertically fragmented data structures optimize cache performance on sequential data access. We then focus on equi-join, typically a random-access operation, and introduce radix algorithms for partitioned hash-join. The performance of these algorithms is quantified using a detailed analytical model that incorporates memory access cost. Experiments that validate this model were performed on the Monet database system. We obtained exact statistics on events such as TLB misses and L1 and L2 cache misses by using hardware performance counters found in modern CPUs. Using our cost model, we show how the carefully tuned memory access pattern of our radix algorithms makes them perform well, which is confirmed by experimental results.

Key words: Main-memory databases – Query processing – Memory access optimization – Decomposed storage model – Join algorithms – Implementation techniques

1 Introduction

Custom hardware – from workstations to PCs – has experienced tremendous improvements in the past decades. Unfortunately, this growth has not been equally distributed over all aspects of hardware performance and capacity. Figure 1 shows that the speed of commercial microprocessors has increased roughly 70% every year, while the speed of commodity DRAM has improved by little more than 50% over the past decade [Mow94]. Part of the reason for this is that there is a direct tradeoff 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 has been getting slower at a dramatic rate. This affects all computer systems, making it increasingly difficult to achieve high processor efficiencies.

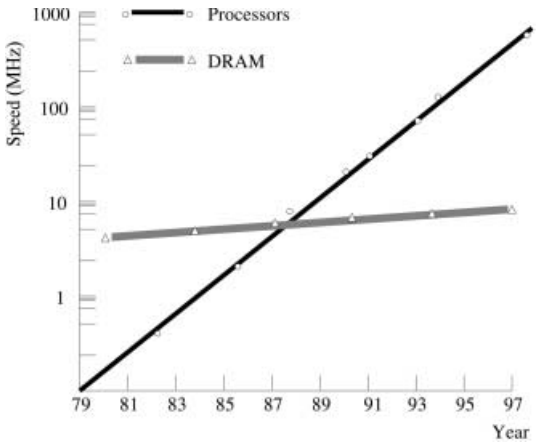

Fig. 1. Hardware trends in DRAM and CPU speed

Three aspects of memory performance are of interest: bandwidth, latency, and address translation. The only way to reduce effective memory latency for applications has been to incorporate *cache memories* in the memory subsystem. Fast and more expensive SRAM memory chips found their way to computer boards, to be used as L2 caches. Due to the ever-rising CPU clock-speeds, the time to bridge the physical distance between such chips and the CPU became a problem; so modern CPUs come with an on-chip L1 cache (see Fig. 2). This physical distance is actually a major complication for designs trying to reduce main-memory latency. The new DRAM standards Rambus [Ram96] and SLDRAM [SLD97] therefore concentrate on fixing the memory bandwidth bottleneck [McC95], rather than the latency problem.

Cache memories can reduce the memory latency only when the requested data is found in the cache. Their effectiveness depends on the memory access pattern of the application. Thus, unless special care is taken, memory latency becomes an increasing performance bottleneck, preventing applications –

A preliminary version of this paper has been published as [BMK99].

* This work was carried out while the author was at the University of Amsterdam, supported by SION grant 612-23-431.

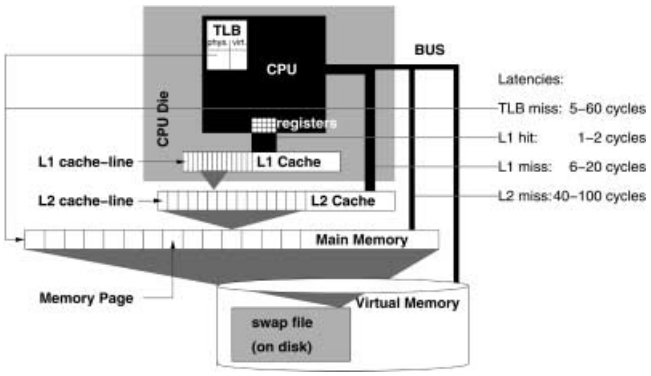

Fig. 2. Hierarchical memory system

including database systems – from fully exploiting the power of modern hardware.

Besides memory latency and memory bandwidth, translation of logical virtual memory addresses to physical page addresses can also have a severe impact on memory access performance. The memory management unit (MMU) of all modern CPUs has a translation lookaside buffer (TLB), a kind of cache that holds the translation for (typically) the 64 most recently used pages. If a logical address is found in the TLB, the translation has no additional cost. Otherwise, a *TLB miss* occurs. A TLB miss is handled by trapping to a routine in the operating system kernel, which translates the address and places it in the TLB. Depending on the implementation and hardware architecture, TLB misses can be even more costly than a main memory access.

For a more detailed discussion of the hardware background, we refer the interested reader to [MBK00].

1.1 Overview

In this article we investigate the effect of memory access cost on database performance by looking in detail at the main-memory cost of typical database applications. 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 [BQK96, BWK98], a high-performance system targeted to query-intensive application areas like OLAP and Data Mining. For the research presented here, we use Monet as our experimentation platform.

The rest of this paper is organized as follows. In Sect. 2, we analyze the impact of memory access costs on basic database operations. We show that, unless special care is taken, a database server running even a simple sequential scan on a table will spend 95% of its cycles waiting for memory to be accessed. We present a detailed analytical cost model that describes how hardware characteristics like cache line sizes and cache miss latencies determine the performance of a sequential scan. This memory access bottleneck is even more difficult to avoid in more complex database operations such as sorting, aggregation and join, which exhibit a random access pattern.

In Sect. 3, we discuss the consequences of this bottleneck for data structures and algorithms to be used in database systems. We identify vertical fragmentation as the solution for database data structures that leads to optimal memory cache usage. Concerning query processing algorithms, we focus on equi-join and introduce new radix algorithms for partitioned hash-join.

In Sect. 4, we analyze the properties of these algorithms with detailed analytical cost models that quantify query costs in terms of CPU cycles, TLB misses, and cache misses. These models show how memory access determines the performance of database algorithms, and they enable us to tune the memory access pattern of our algorithms carefully to achieve optimal performance. Exhaustive experiments using the Monet system confirm the significant performance improvement that our memory-conscious algorithms achieve over standard algorithms.

Finally, we evaluate our findings and conclude that the hard data obtained in our experiments justify the basic architectural choices of the Monet system, which back in 1992 were mostly based on intuition.

2 Initial experiment

In this section, we demonstrate the severe impact of memory access cost on the performance of elementary database operations. Figure 3 shows results of a simple scan test on a number of popular workstations of the past decade. In this test, we sequentially scan an in-memory buffer, by iteratively reading one byte with a varying *stride*, i.e., the offset between two subsequently accessed memory addresses. This experiment mimics what happens if a database server performs a read-only scan of a one-byte column in an in-memory table with a certain record-width (the stride); as would happen in a selection on a column with zero selectivity or in a simple aggregation (e.g., Max or Sum). The y-axis in Fig. 3 shows the cost of 200 000 iterations in elapsed time, and the x-axis shows the stride used. We made sure that the buffer was in memory, but not in any of the memory caches.

2.1 General observations

When the stride is small, successive iterations in the scan read bytes that are near to each other in memory, hitting the same cache line. The number of L1 and L2 cache misses is therefore low. The L1 miss rate reaches its maximum of one miss per iteration as soon as the stride reaches the size of an L1 cache line (16 to 32 bytes). Only the L2 miss rate increases further, until the stride exceeds the size of an L2 cache line (16 to 128 bytes). Then, it is certain that every memory read is a cache miss. Performance cannot become any worse and stays constant.

In the following, we first present a detailed analysis of our experiment in order to understand the impact of various parts of the hardware system on the performance of (basic) database operations, such as a sequential in-memory scan. We use an SGI Origin2000 with MIPS R10000 processors (250 MHz) as a sample machine, but we keep the models applicable to other systems as well by using a set of specific parameters to describe the respective hardware characteristics. Table 1 lists the parameters for the Origin2000. In [MBK00], we present a *calibration tool* to automatically extract these parameters on any computer hardware. The software is freely available for

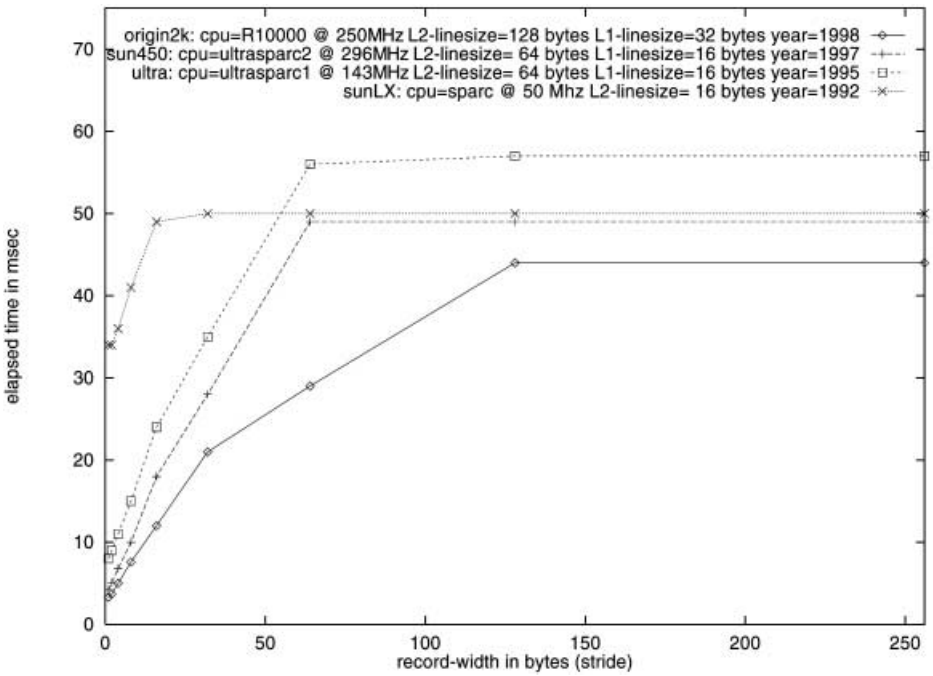

Fig. 3. Reality check: simple in-memory scan of 200 000 tuples

Table 1. Hardware characteristics

Description		Value
Machine type		SGI Origin2000
OS		IRIX64 6.5
CPU		MIPS R10000
CPU speed		250 MHz
CPU-inherent parallelism	q	4
Main-memory size		48 GB (4 GB local)
L1 cache size	$\|\|L1\|\|$	32 KB
L1 cache line size	LS_{L1}	32 bytes
L1 cache lines	$\|L1\|_{\mathrm{L1}}$	1024
L2 cache size	$\|\|L2\|\|$	4 MB
L2 cache line size	LS_{L2}	128 bytes
L2 lines	$\|L2\|_{\mathrm{L2}}$	32 768
TLB entries	$\|TLB\|$	64
Page size	$\|\|Pg\|\|$	16 KB
TLB size ($\|TLB\| \cdot \|\|Pg\|\|$)	$\|\|TLB\|\|$	1 MB
L1 miss latency	l_{L2}	24 ns = 6 cycles
L2 miss latency	l_{Mem}	400 ns = 100 cycles
TLB miss latency	l_{TLB}	228 ns = 57 cycles
Memory bandwidth	bw_{Mem}	555 MB/s

download from http://www.cwi.nl/∼monet/. There, the calibrated results for a large number of hardware platforms are available, too.

After the detailed analysis, we discuss our scan experiment in a broader context.

2.2 Detailed analysis

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

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

We model the costs for accessing data in the L2 cache and in main memory by scoring each access with the respective latency. As observed above, the number of L2 and main memory accesses (i.e., the number of L1 and L2 misses, respectively) depends on the access stride. With a stride s smaller than the cache line size LS, the average number of cache misses per iteration is $M(s) = \frac{s}{LS}$. With a stride equal to or larger than the cache line size, a miss occurs with each iteration. In general, we get ($i \in \{1, 2\}$)

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

with M_{Li} and LS_{Li} ($i \in \{1, 2\}$) denoting the number of cache misses and the cache line sizes for each level, respectively. Figure 4 compares M_{L1} and M_{L2} to the measured number of cache misses.

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

$$T(s) = T_{\mathrm{CPU}} + T_{\mathrm{L2}}(s) + T_{\mathrm{Mem}}(s), \qquad \text{(“model 1”)}$$

with

$$T_{\mathrm{L2}}(s) = M_{\mathrm{L1}}(s) \cdot l_{\mathrm{L2}},$$

$$T_{\mathrm{Mem}}(s) = M_{\mathrm{L2}}(s) \cdot l_{\mathrm{Mem}},$$

where l_x ($x \in$ {L2,Mem}) denote the (cache) memory access latencies for each level, respectively. We measure the L2

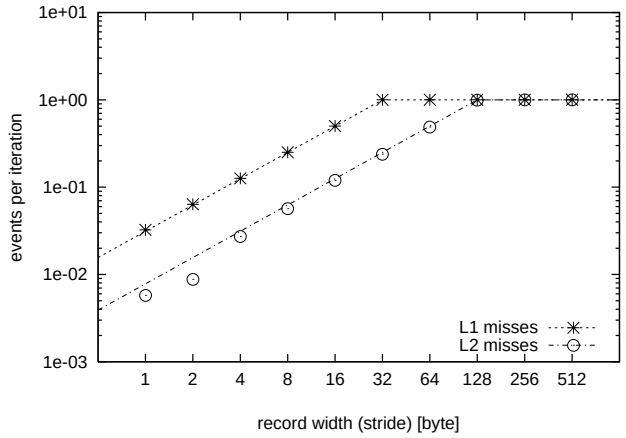

Fig. 4. Measured (*points*) and modeled (*lines*) cache misses

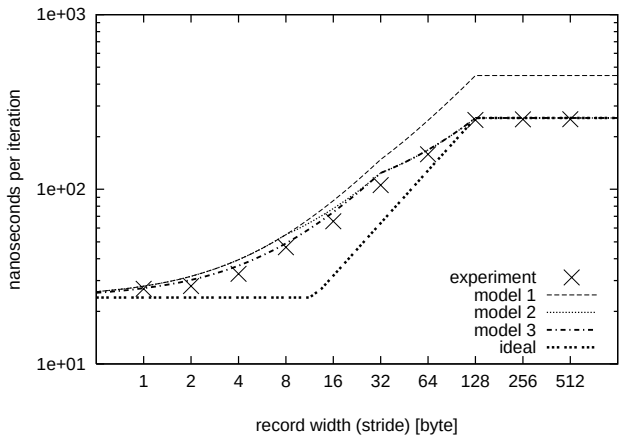

Fig. 5. Sequential scan: experiment and models

and memory latency with our hardware calibration tool presented in [MBK00].[1] Figure 5 shows the resulting curve as "model 1."

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

$$\begin{aligned} O(o,x,y) &= \max\{x,y\} + (1-o)\cdot\min\{x,y\} \\ &= x + y - o\cdot\min\{x,y\}. \end{aligned}$$

This overlap function forms a linear interpolation between the two extreme cases

- no overlap $(o = 0) \implies O(0,x,y) = x + y$,
- full overlap $(o = 1) \implies O(1,x,y) = \max\{x,y\}$.

Let o_1 and o_2 be the degrees of overlap for L2 access and main-memory access, respectively. Then, we get the total cost considering overlap by applying the overlap function twice:

$$\begin{aligned} T' &= O(o_1, T_{\mathrm{CPU}}, T_{\mathrm{L2}}), \\ T &= O(o_2, T', T_{\mathrm{Mem}}). \end{aligned}$$

The following consideration will help us to determine o_1 and o_2. In our experiments, we have a pure sequential memory access pattern. Up to a stride of 8 bytes, 4 subsequent memory references refer to the same 32-bytes L1 line, i.e., only one L1 line is loaded at a time, not allowing any overlap of pure calculation and memory access ($o_1 = o_2 = 0$). With strides between 8 and 32 bytes, o_1 linearly increases towards its maximum. The same holds for o_2 with strides between 32 and 128 bytes, as L2 lines contain 128 bytes on the R10000. Thus, we get ($i \in \{1,2\}$)

$$o_i(s) = \begin{cases} 0, & \text{if } s \le \frac{LS_{\mathrm{L}i}}{q} \\ \dfrac{s - \frac{LS_{\mathrm{L}i}}{q}}{LS_{\mathrm{L}i} - \frac{LS_{\mathrm{L}i}}{q}}, & \text{if } \dfrac{LS_{\mathrm{L}i}}{q} < s < LS_{\mathrm{L}i} \\ 1, & \text{if } s \ge LS_{\mathrm{L}i}. \end{cases}$$

However, we have to consider another limit: bandwidth. L2 bandwidth is large enough to allow $q = 4$ concurrent L1 loads within a single L2 latency period, reducing the effective latency to (1/q)th. Memory bandwidth, however, is limited to 555 MB/s (see [MBK99]). Hence, at least $l_{\min} = 220$ ns (55 cycles) are needed to load one L2 line (128 bytes).

Now, we can refine our model as follows:

$$\begin{aligned} T'(s) &= O\big(o_1(s), T_{\mathrm{CPU}}, T_{\mathrm{L2}}(s)\big) \\ T(s) &= O\big(o_2(s), T'(s), T_{\mathrm{Mem}}(s)\big) \end{aligned} \qquad \text{("model 2")}$$

with

$$T_{\mathrm{L2}}(s) = M_{\mathrm{L1}}(s)\cdot\left(1 - \frac{o_1(s)}{M_{\mathrm{L1}}(s)}\right)\cdot l_{\mathrm{L2}}$$

$$T_{\mathrm{Mem}}(s) = M_{\mathrm{L2}}(s)\cdot\left(1 - \frac{o_2(s)}{M_{\mathrm{L2}}(s)}\cdot(1 - l_{\min})\right)\cdot l_{\mathrm{Mem}}.$$

and T_{CPU}, M_{L1}, M_{L2}, l_{L2}, l_{Mem} as in "model 1". Figure 5 depicts the resulting curve as "model 2." This fits the experimental curve pretty well. The differences for small strides can be eliminated by setting $o_1 = 1$ for all strides, as "model 3" shows in the figure. This in turn means that the CPU itself loads several L1 lines concurrently, even if 4 subsequent memory references refer to the same L1 line. In this scenario, the "ideal" performance of

$$T(s) = \max\{T_{\mathrm{CPU}}, T_{\mathrm{L2}}(s), T_{\mathrm{Mem}}(s)\}, \qquad \text{("ideal")}$$

i.e., with $o_1 = o_2 = 1$, is not reached (see "ideal" in Fig. 5), because the whole memory bandwidth cannot be utilized automatically for smaller strides, i.e., when several memory references refer to a single L2 line.

2.3 Discussion

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

While all machines in Fig. 3 exhibit the same pattern of performance degradation with decreasing data locality, Fig.

[1] See also http://www.cwi.nl/∼monet/.

3 clearly shows that the penalty for poor memory cache usage has dramatically increased in the last six years. The CPU speed has improved by almost an order of magnitude, both through higher clock frequencies and through increased CPU-inherent parallelism. However, the memory access latencies have hardly changed. In fact, we must draw the sad conclusion that if no attention is paid in query processing to data locality all advances in CPU power are neutralized due to the memory access bottleneck caused by memory latency. The considerable growth of memory bandwidth – reflected in the growing cache line sizes [2] – does not solve the latency problem if data locality is low.

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

3 Architectural consequences

In the previous sections we have shown that it is less and less appropriate to think of the main memory of a computer system as "random access" memory. In this section, we analyze the consequences for both data structures and algorithms used in database systems.

3.1 Data structures

The default physical tuple representation is a consecutive byte sequence, which must always be accessed by the bottom operators in a query evaluation tree (typically selections or projections). In the case of sequential scan, we have seen that performance is strongly determined by the record width (the position on the x-axis of Fig. 3). This width quickly becomes too large, and hence performance decreases (e.g., an Item tuple, as shown in Fig. 6, occupies at least 80 bytes on relational systems). To achieve better performance, a smaller stride is needed, and for this purpose we recommend using *vertically decomposed* data structures.

Monet [BK95, BK99] uses the Decomposed Storage Model [CK85], storing each column of a relational table in a separate binary table, called a binary association table (BAT).

[2] In one memory fetch, the Origin2000 gets 128 bytes, whereas the Sun LX gets only 16; an improvement of factor 8.

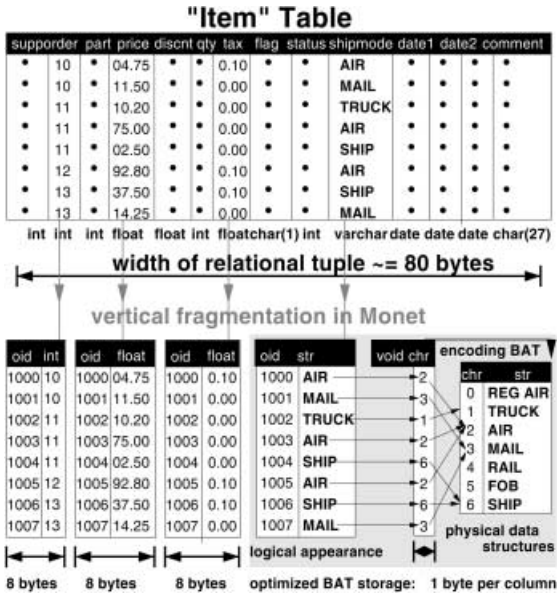

Fig. 6. Vertically decomposed storage in BATs

A BAT is represented in memory as an array of fixed-size two-field records [OID,value], or binary units (BUN). Their width is typically 8 bytes.

In the case of the Origin2000 machine, we deduce from Fig. 3 that a scan-selection on a table with stride 8 takes 10 CPU cycles per iteration, whereas with a stride of 1 it takes only 4 cycles. In other words, in a simple range-select, there is so little CPU work per tuple (4 cycles) that the memory access cost for a stride of 8 still weighs quite heavily (6 cycles). Therefore we have found it useful in Monet to apply two space optimizations that further reduce the per-tuple memory requirements in BATs:

Virtual OIDs: Generally, when decomposing a relational table, we get an identical system-generated column of OIDs in all decomposition BATs, which is *dense and ascending* (e.g., 1000, 1001, ..., 1007). In such BATs, Monet computes the OID values on the fly when they are accessed using positional lookup of the BUN and avoids allocating the 4-byte OID field. This is called a "virtual OID" or VOID column. Apart from reducing memory requirements by half, this optimization is also beneficial when joins or semi-joins are performed on OID columns.[3] When one of the join columns is a VOID, Monet uses positional lookup instead of, e.g., hash-lookup, effectively eliminating all join cost.

Byte encodings: Database columns often have a low domain cardinality. For such columns, Monet uses fixed-size encodings in 1- or 2-byte integer values. This simple technique was chosen because it does not require decoding effort when the values are used (e.g., a selection on a string "MAIL" can be re-mapped to a selection on a byte with value 3). A more complex scheme (e.g., using bit compression) might yield even more memory savings, but the decoding step required whenever values are accessed can

[3] The projection phase in query processing typically leads in Monet to additional "tuple-reconstruction" joins on OID columns, which are caused by the fact that tuples are decomposed into multiple BATs.

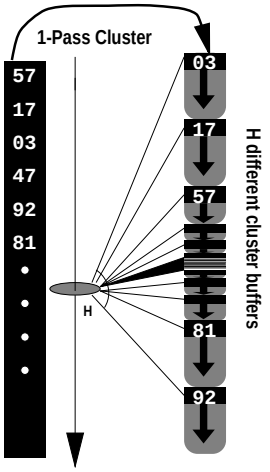

Fig. 7. Straightforward clustering algorithm

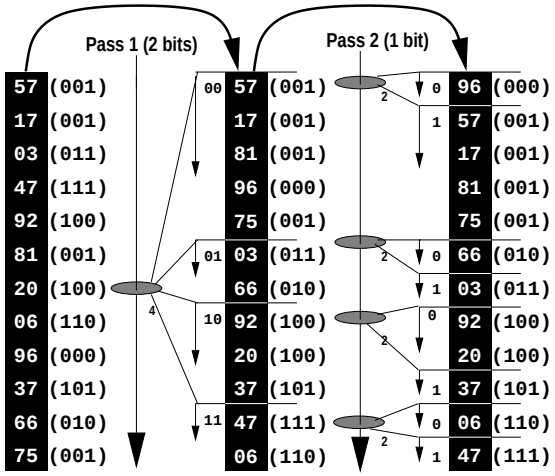

Fig. 8. 2-pass/3-bit Radix-cluster (lower bits given in parentheses)

quickly become counterproductive due to extra CPU effort. Even if decoding would only cost a handful of cycles for each tuple, this would more than double the amount of CPU effort in simple database operations, such as the range-select from our experiment.

Figure 6 shows that when applying both techniques; the storage needed for 1 BUN in the "shipmode" column is reduced from 8 bytes to just 1.

3.2 Query processing algorithms

We now briefly discuss the effect of the memory access bottleneck on the design of algorithms for common query processing operators.

Selections: If the selectivity is low, most data needs to be visited, and this is best done with a scan-select (it has optimal data locality). For higher selectivities, Lehman and Carey [LC86] concluded that the T-tree and bucket-chained hash-table were the best data structures for accelerating selections in main-memory databases. The work in [Ron98] reports, however, that a B-tree with a block-size equal to the cache line size is optimal. Our findings about the increased impact of cache misses indeed support this claim, since lookup using a hash-table or T-tree causes random memory access to the entire relation, a non-cache-friendly access pattern.

Grouping and aggregation: Two algorithms are often used here: sort/merge and hash-grouping. In sort/merge, the table is first sorted on the group-by attribute(s) followed by scanning. Hash-grouping scans the relation once, keeping a temporary hash-table where the group-by values are a key that give access to the aggregate totals. This number of groups is often limited, such that this hash-table fits the L2 cache, and probably also the L1 cache. This makes hash-grouping superior to sort/merge concerning main-memory access, as the sort step has random access behavior and is performed on the entire relation to be grouped, which probably does not fit any cache.

Equi-joins: Hash-join has long been the preferred main-memory join algorithm. It first builds a hash-table on the smaller relation (the inner relation). The outer relation is then scanned, and for each tuple a hash-lookup is performed to find the matching tuples. If this inner relation plus the hash-table does not fit in any memory cache, a performance problem occurs, due to the random access pattern. Merge-join is not a viable alternative as it requires sorting on both relations first, which would cause random access over even a larger memory region.

Consequently, we identify join as the most problematic operator; therefore we investigate possible alternatives that can get optimal performance out of a hierarchical memory system.

3.3 Clustered hash-join

Shatdahl 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*, which each fit the memory cache, performs better than a 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 Fig. 7. This constitutes a random access pattern that writes into H separate locations. If H exceeds the number of available cache lines (L1 or L2), *cache trashing* occurs; alternatively, if H exceeds the number of TLB entries, the number of TLB misses will explode. Both factors will severely degrade overall join performance.

As an improvement of this straightforward algorithm, we propose a clustering algorithm that has a cache-friendly memory access pattern, even for high values of H.

Radix algorithms The *radix-cluster* algorithm splits a relation into H clusters using multiple passes (see Fig. 8). Radix-clustering on the lower B bits of the integer hash-value of a column is done 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 as one cluster and is subdivided in $H_1 = 2^{B_1}$

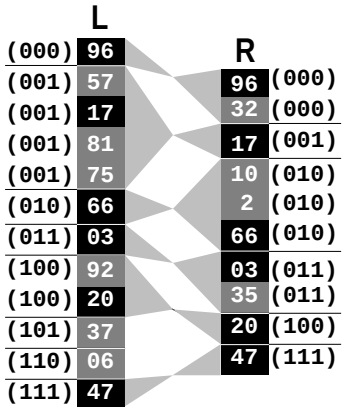

Fig. 9. Joining 3-bit radix-clustered inputs (black tuples hit)

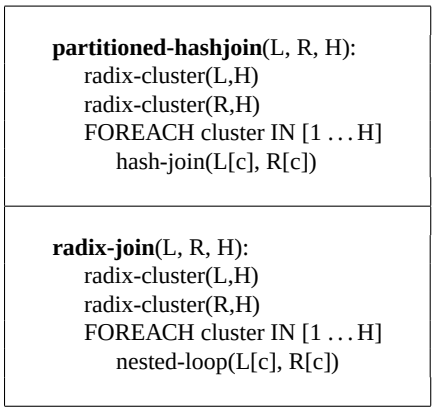

Fig. 10. Join algorithms employed

clusters. The next pass takes these clusters and subdivides each into $H_2 = 2^{B_2}$ new ones, yielding $H_1 \cdot H_2$ clusters in total, etc. Note that with $P = 1$, the radix-cluster behaves like the straightforward algorithm.

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, we avoid cache trashing altogether.

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 $C/2^B$ tuples. 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 *ordered* on radix bits (see parentheses next to the right-most column in Fig. 8). When using the 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 obtain the pairs of clusters that should be hash-joined with each other (see Figs. 9 and 10).

The alternative **radix-join** algorithm, also proposed here, makes use of the very fine clustering capabilities of radix-cluster. If the number of clusters H is high, the radix-clustering brings the potentially matching tuples near to each other. As chunk sizes are small, a simple nested loop is then sufficient to filter out the matching tuples (see Fig. 10). Radix-join is similar to hash-join in the sense that the number H should be tuned to be the relation cardinality C 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.

4 Quantitative assessment

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

1. the number of 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 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 parameters' settings are optimal for radix-clustering a relation on B bits, we turn our attention to the performance of the join algorithms with varying values of B. Finally, these two experiments are combined to gain insight in overall join performance.

4.1 Experimental setup

In our experiments, we use binary relations (BATs) of 8-bytes-wide tuples and varying cardinalities, consisting of uniformly distributed unique random numbers. In the join experiments, the join hit rate is one, and 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 as in [SKN94] we do not include it in our comparison.

The experiments were carried out with an Origin2000 machine on one 250 MHz MIPS R10000 processor. This system has 32 KB of L1 cache, consisting of 1024 lines of 32 bytes, 4 MB of L2 cache, consisting of 32 768 lines of 128 bytes, and sufficient main memory to hold all data structures. Further, this system uses a page size of 16 KB and has 64 TLB entries. We used the hardware event counters of the MIPS R10000 CPU [Sil97] to get exact data on the number of cycles, TLB misses, L1 misses and L2 misses during these experiments.[4] Using the data from the experiments, we formulate an analytical main-memory cost model that quantifies query cost in terms of these hardware events.

4.2 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.

[4] The Intel Pentium family, SUN UltraSparc, and DEC Alpha provide similar counters.

First, we conduct experiments with various numbers of bits and passes, distributing the bits evenly across the passes. The points in Fig. 11 depict the results for a BAT of 8M tuples – the remaining cardinalities ($15\,625 \leq C \leq 64$M) behave the same way. The vertical grid lines indicate where the number of clusters created is equal to the number of TLB entries and L1 and L2 cache lines – or a power of those – respectively. Up to 6 bits, using just one pass yields the best performance (in ms). Then, as the number of clusters to be filled concurrently exceeds the number of TLB entries (64), the number of TLB misses increases tremendously, decreasing the performance. With more than 6 bits, two passes perform better than one. The costs of an additional pass are more than compensated by having significantly less TLB misses in each pass using half the number of bits. Analogously, three passes should be used with more than 12 bits, and four passes with more than 18 bits. Thus, the number of clusters per pass is limited to at most the number of TLB entries. A second more moderate increase in TLB misses occurs when the number of clusters exceeds the number of L2 cache lines. Then, the additional L2 misses are caused by cache conflicts, forcing modified cache lines to be written back to memory before they are completely filled. These write-backs refer to pages whose addresses are no longer cached in the TLB, yielding an additional TLB miss per L2 miss.

Similarly, the number of L1 cache misses and L2 cache misses significantly increases whenever the number of clusters per pass exceeds the number of L1 cache lines (1024) and L2 cache lines (32 768), respectively. The impact of the additional L2 misses on the total performance is obvious for one pass (it does not occur with more than one pass, as then at most 13 bits are used per pass). The impact of the additional L1 misses on the total performance nearly completely vanishes due to the heavier penalty of TLB misses and L2 misses. Finally, we notice that the best-case execution time increases with the number of bits used.

The following model calculates the total execution costs for a radix-cluster depending on the number of passes, the number of bits, and the cardinality of the input relation ($C = |Re|$):

$$T_{\mathrm{c}}(P,B,C) = P \cdot \left(C \cdot w_{\mathrm{c}} + M_{\mathrm{L1,c}}\left(\frac{B}{P}, C\right) \cdot l_{\mathrm{L2}} + M_{\mathrm{L2,c}}\left(\frac{B}{P}, C\right) \cdot l_{\mathrm{Mem}} + M_{\mathrm{TLB,c}}\left(\frac{B}{P}, C\right) \cdot l_{\mathrm{TLB}} \right)$$

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

$$M_{\mathrm{Li,c}}(B_p, C) = 2 \cdot |Re|_{\mathrm{L}i} + \begin{cases} C \cdot \dfrac{H_p}{|Li|_{\mathrm{L}i}}, & \text{if } H_p \leq |Li|_{\mathrm{L}i} \\ C \cdot \left[1 + \log\left(\dfrac{H_p}{|Li|_{\mathrm{L}i}}\right)\right], & \text{if } H_p > |Li|_{\mathrm{L}i} \end{cases}$$

and

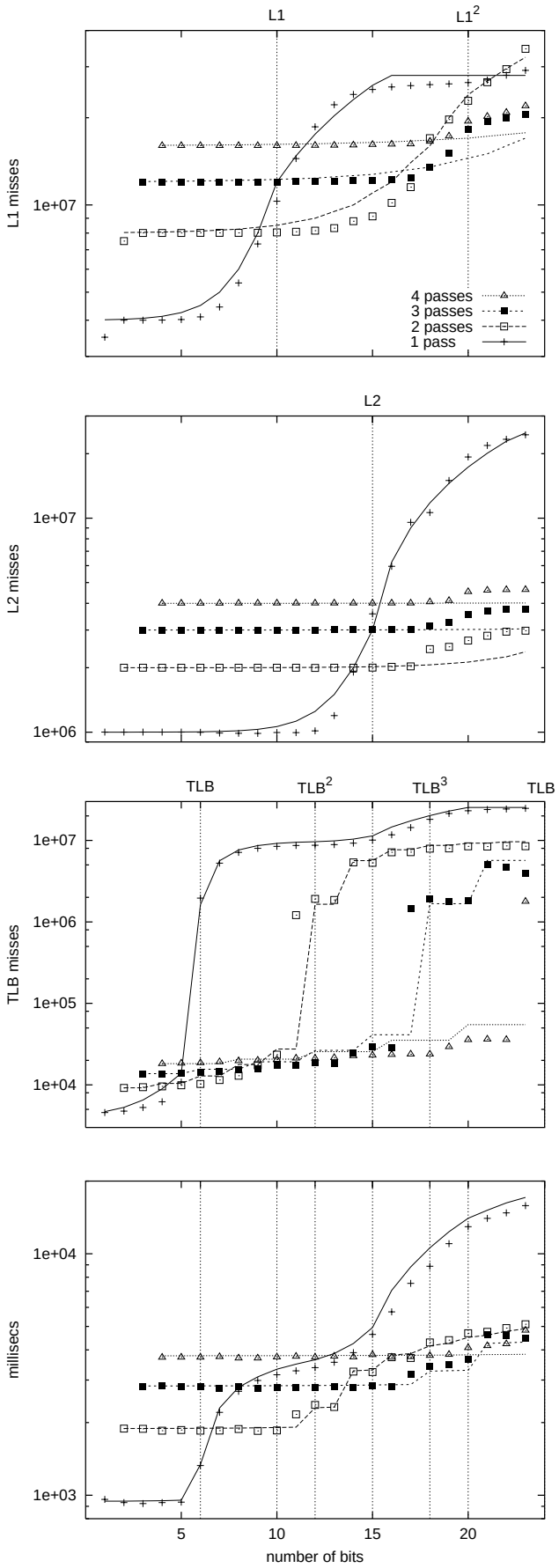

Fig. 11. Performance (points) and model (lines) of radix-cluster ($C = 8$M)

$$M_{TLB,\mathrm{c}}(B_p, C) =$$

$$2 \cdot |Re|_{\mathrm{Pg}} + \begin{cases} |Re|_{\mathrm{Pg}} \cdot \left(\frac{H_p}{|TLB|}\right), \text{ if } H_p \le |TLB| \\ C \cdot \left(1 - \frac{|TLB|}{H_p}\right), \text{ if } H_p > |TLB| \end{cases} + \begin{cases} C \cdot \left(\frac{H_p}{|L2|_{\mathrm{L2}}}\right), & \text{if } H_p \le |L2|_{\mathrm{L2}} \\ C \cdot \left[1 + \log\left(\frac{H_p}{|L2|_{\mathrm{L2}}}\right)\right], & \text{if } H_p > |L2|_{\mathrm{L2}} \end{cases}$$

$|Re|_{\mathrm{L}i}$ and $|Cl|_{\mathrm{L}i}$ denote the number of cache lines per relation and cluster, respectively, $|Re|_{\mathrm{Pg}}$ the number of pages per relation, $|Li|_{\mathrm{L}i}$ the total number of cache lines, both for the L1 ($i = 1$) and L2 ($i = 2$) caches, and $|TLB|$ the number of TLB entries.

w_{c} denotes the pure CPU costs per tuple. To calibrate w_{c}, we reduced the cardinality so that all data fits in L1, and pre-loaded the input relation. Thus, we avoided memory access completely. We measured $w_{\mathrm{c}} = 50\,\mathrm{ns}$ on the Origin2000 (250 MHz).

The first term of $M_{\mathrm{L}i,\mathrm{c}}$ equals the minimal number of Li misses per pass for fetching the input and storing the output. The second term counts the number of additional Li misses, when the number of clusters either approaches the number of available Li lines ($H_p \le |Li|_{\mathrm{L}i}$) or even exceeds this ($H_p > |Li|_{\mathrm{L}i}$). First, the probability that the requested cluster is not in the cache – due to address conflicts – increases until $H_p = |Li|_{\mathrm{L}i}$. Then, the cache capacity is exhausted, and a cache miss for each tuple to be assigned to a cluster is certain. But, with further increasing H_p, the number of cache misses also increases, as now also the cache lines of the input may be replaced before all tuples are processed. Thus, each input cache line has to be loaded more than once.

The first two terms of $M_{\mathrm{TLB},\mathrm{c}}$ are made up analogously. Additionally, using the same schema as $M_{\mathrm{L2},\mathrm{c}}$, the third term models the additional TLB misses that occur due to write-backs (see above) when the number of clusters either approaches the number of available L2 lines ($H_p \le |L2|_{\mathrm{L2}}$) or even exceeds this ($H_p > |L2|_{\mathrm{L2}}$).

The lines in Fig. 11 represent our model for a BAT of 8M tuples. The model shows to be very accurate[5]. Figures 12 and 13 confirm the accuracy of our model for various cardinalities. In Fig. 12, the optimal number of passes is chosen per event, showing that our model correctly predicts the behavior of each single event. Figure 13 uses the number of passes that achieves minimal execution time. Here, the graphs show that (on the Origin2000) the impact of TLB misses dominates the execution time. At most 64 clusters should be generated per pass, although the caches would allow 1024 (L1 cache) or even 32 768 (L2 cache).

The question remaining is how to distribute the number of radix bits over the passes. We conducted another number of experiments, using a fixed number of passes, but varying the number of radix bits per pass. The results showed that an even distribution of radix bits (i.e., $B_i \approx \frac{B}{P}$, $i \in \{1, \ldots, P\}$) achieves the best performance.

[5] On our Origin2000, we calibrated $l_{\mathrm{TLB}} = 228\,\mathrm{ns}$, $l_{\mathrm{L2}} = 24\,\mathrm{ns}$, and $l_{\mathrm{Mem}} = 400\,\mathrm{ns}$ [MBK99].

4.3 Isolated join performance

We now analyze the impact of the number of radix bits on the pure join performance, not including the clustering cost.

4.3.1 Radix-Join

The points in Fig. 14 depict the experimental results of radix-join (L1 and L2 cache misses, TLB misses, elapsed time) for different cardinalities. The lower graph (in ms) shows that the performance of radix-join improves with increasing number of radix bits. The upper graph confirms that only cluster sizes significantly smaller than L1 size are reasonable. Otherwise, the number of L1 cache misses explodes due to cache trashing. We limited the execution time of each single run to 15 min, thus using only cluster sizes significantly smaller than L2 size and TLB size (i.e., number of TLB entries $\times$ page size). That is why the number of L2 cache misses stay almost constant. The performance improvement continues until the mean cluster size is 1 tuple. At that point, radix-join has degenerated to sort/merge-join. The high cost of radix-join with a large cluster size is explained by the fact that it performs nested-loop join on each pair of matching clusters. Therefore, clusters need to be kept small; our results indicate that a cluster size of 8 tuples is optimal.

The following model calculates the total execution costs for a radix-join, depending on the number of bits and the cardinality[6]:

$$T_{\mathrm{r}}(B, C) = C \cdot \left\lceil \frac{C}{H} \right\rceil \cdot w_{\mathrm{r}} + C \cdot w'_{\mathrm{r}} + M_{\mathrm{L1,r}}(B, C) \cdot l_{\mathrm{L2}} + M_{\mathrm{L2,r}}(B, C) \cdot l_{\mathrm{Mem}} + M_{\mathrm{TLB,r}}(B, C) \cdot l_{\mathrm{TLB}},$$

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

$$M_{\mathrm{L}i,\mathrm{r}}(B, C) =$$

$$3 \cdot |Re|_{\mathrm{L}i} + \begin{cases} C \cdot \frac{||Cl||}{||Li||}, \text{ if } ||Cl|| \le ||Li|| \\ C \cdot |Cl|_{\mathrm{L}i}, \text{ if } ||Cl|| > ||Li|| \end{cases}$$

and

$$M_{TLB,r}(B, C) =$$

$$3 \cdot |Re|_{Pg} + \begin{cases} C \cdot \frac{||Cl||}{||TLB||}, \text{ if } ||Cl|| \le ||TLB|| \\ C \cdot |Cl|_{\mathrm{Pg}}, \quad \text{if } ||Cl|| > ||TLB||. \end{cases}$$

$|Re|_{\mathrm{Pg}}$, $|Re|_{\mathrm{L}i}$, and $|Li|_{\mathrm{L}i}$ are as above ($i \in \{1, 2\}$); $||Cl||$ and $|Cl|_{\mathrm{Pg}}$ denote the cluster size in byte and number of pages, respectively; $||Li||$ and $||TLB||$ denote (in byte) the size of both caches ($i \in \{1, 2\}$) and the memory range covered by $|TLB|$ pages, respectively.

[6] For simplicity of presentation, we assume the cardinalities of both operands and the result to be the same.

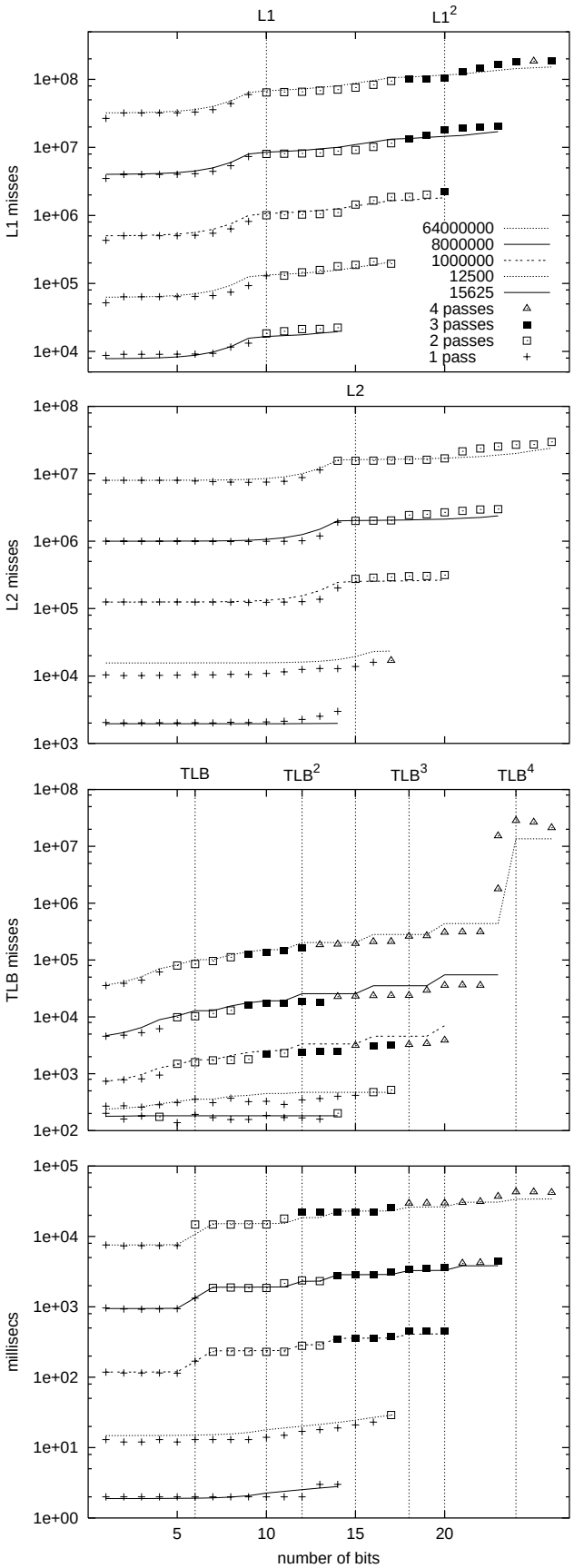

Fig. 12. Performance (*points*) and model (*lines*) of radix-cluster (optimal number of passes per event)

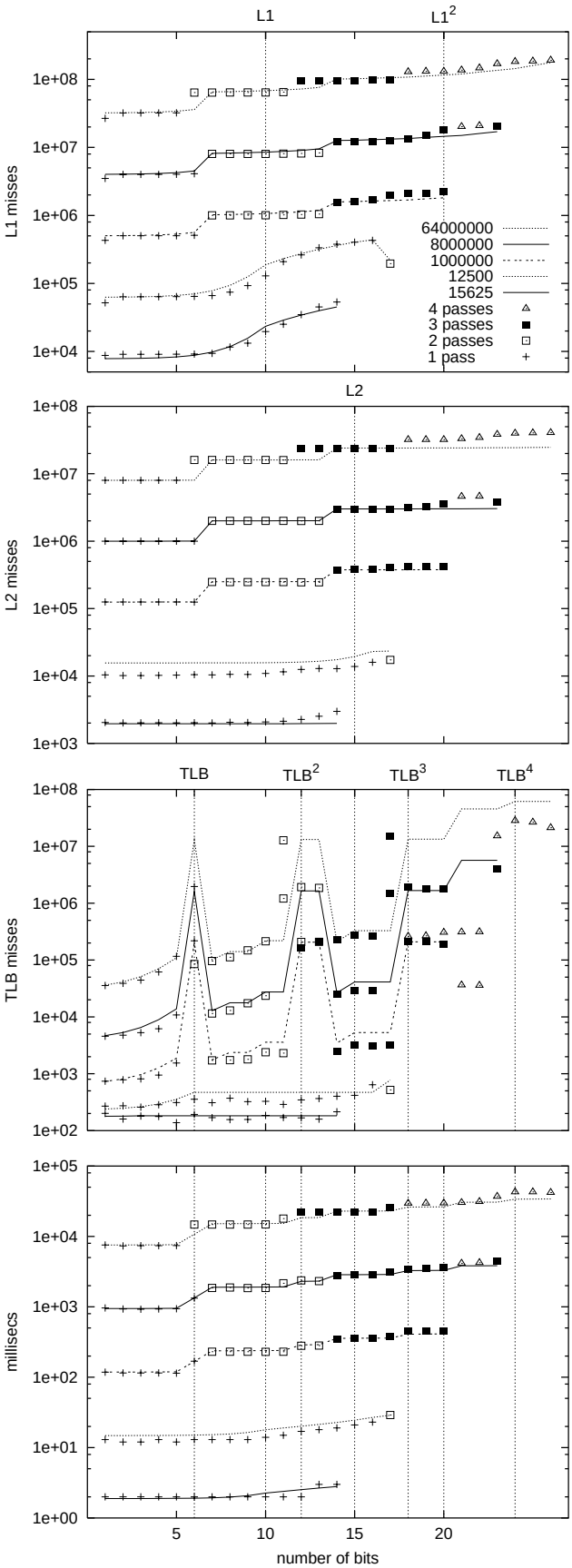

Fig. 13. Performance (*points*) and model (*lines*) of radix-cluster (number of passes for best performance)

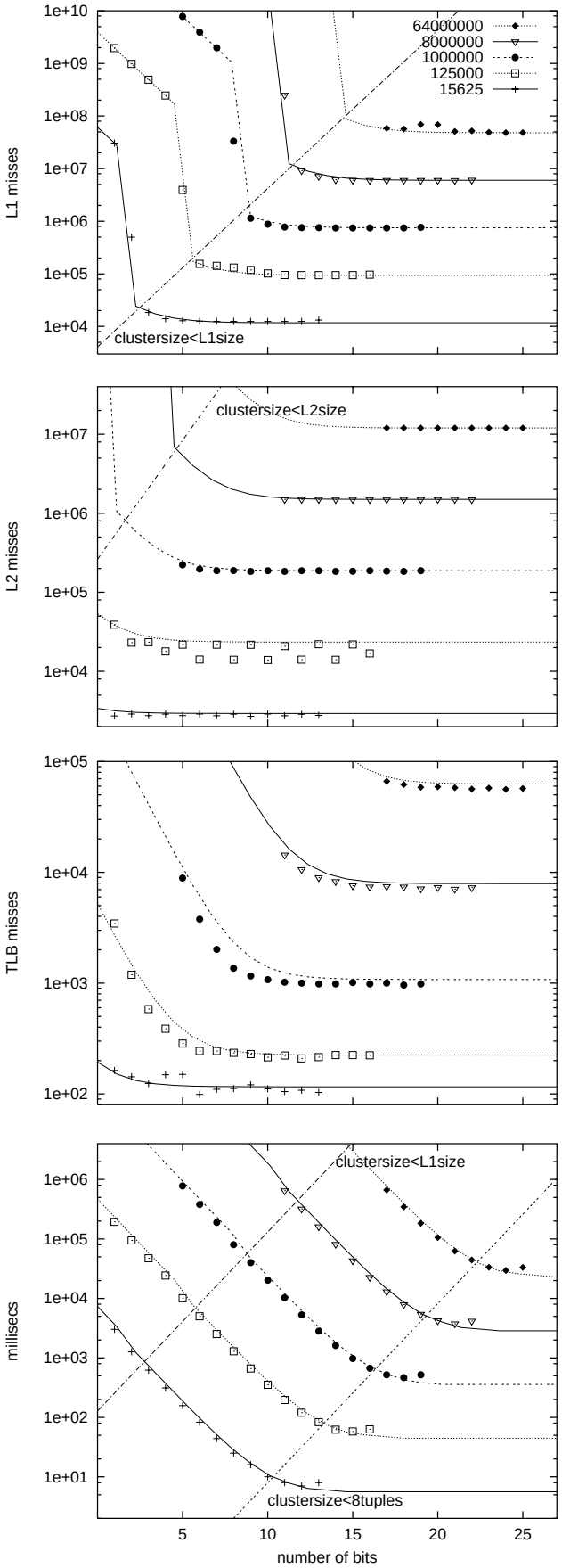

Fig. 14. Performance and model of radix-join

The first term of T_r calculates the costs for evaluating the join predicate – each tuple of the outer relation has to be checked against each tuple in the respective cluster; the cost per check is w_r. The second term represents the costs for creating the result, with w'_r denoting the costs per tuple.

The left term of $M_{\mathrm{L}i,\mathrm{r}}$ equals the minimal number of Li misses for fetching both operands and storing the result. The right term counts the number of additional Li misses during the inner loop, when the cluster size either approaches Li size ($||Cl|| \leq ||Li||$) or even exceeds this ($||Cl|| > ||Li||$). First, the probability that the requested tuple is not in the cache – due to capacity conflicts – increases with growing cluster size. Then, the cache capacity is exhausted, and a cache miss for each tuple to be joined is certain. With further increasing cluster size, the number of cache misses also increases, as now each iteration of the inner loop also causes a cache miss. $M_{\mathrm{TLB},\mathrm{r}}$ is made up analogously. The lines in Fig. 14 prove the accuracy of our model for different cardinalities ($w_\mathrm{r} = 24\,\mathrm{ns}$, $w'_\mathrm{r} = 240\,\mathrm{ns}$).

4.3.2 Partitioned hash-join

The partitioned hash-join also exhibits increased performance with increasing number of radix bits. Figure 15 shows that performance increase flattens after 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 decreases only slightly until the point that the inner cluster fits the L1 cache. Here, performance reaches its minimum. The fixed overhead by allocation of the hash-table structure causes performance to decrease when the cluster sizes get too small ($\lesssim 200$ tuples) and clusters get very numerous.

As for the radix-join, we also provide a cost model for the partitioned hash-join:

$$\begin{aligned} T_\mathrm{h}(B,C) = {} & C \cdot w_\mathrm{h} + H \cdot w'_\mathrm{h} + M_{\mathrm{L1,h}}(B,C) \cdot l_\mathrm{L2} \\ & + M_{\mathrm{L2,h}}(B,C) \cdot l_\mathrm{Mem} \\ & + M_{\mathrm{TLB,h}}(B,C) \cdot l_\mathrm{TLB}, \end{aligned}$$

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

$$M_{\mathrm{Li,h}}(B,C) =$$

$$3 \cdot |Re|_\mathrm{Li} + \begin{cases} C \cdot \dfrac{||Cl||}{||Li||}, & \text{if } ||Cl|| \leq ||Li|| \\ C \cdot 10 \cdot \left(1 - \dfrac{||Li||}{||Cl||}\right), & \text{if } ||Cl|| < ||Li|| \end{cases}$$

and

$$M_{\mathrm{TLB,h}}(B,C) =$$

$$3 \cdot |Re|_\mathrm{Pg} + \begin{cases} C \cdot \dfrac{||Cl||}{||TLB||}, & \text{if } ||Cl|| \leq ||TLB|| \\ C \cdot 10 \cdot \left(1 - \dfrac{||Li||}{||TLB||}\right), & \text{if } ||Cl|| > ||TLB||. \end{cases}$$

$|Re|_\mathrm{Li}$, $|Re|_\mathrm{Pg}$, $||Cl||$, $||Li||$, and $||TLB||$ are as above.

w_h represents the pure calculation costs per tuple, i.e., building the hash-table, doing the hash-lookup and creating

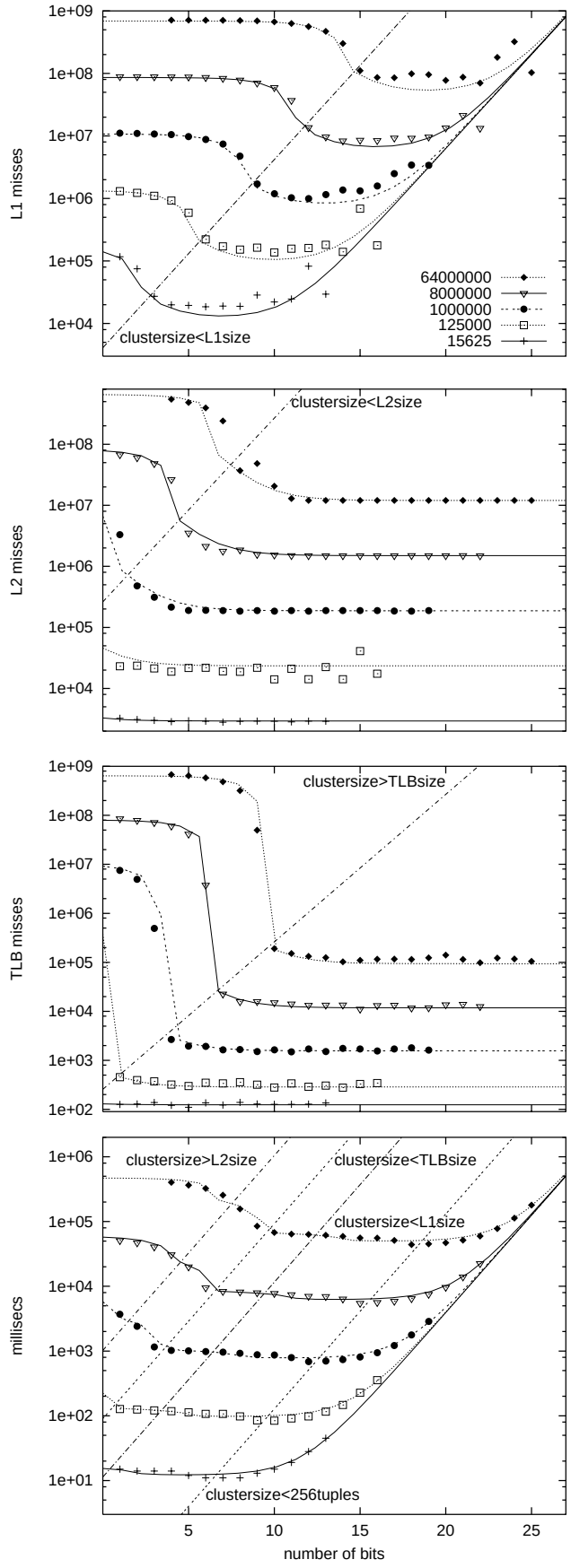

Fig. 15. Performance and model of partitioned hash-join

the result. w'_{h} represents the additional costs per cluster for creating and destroying the hash-table.

The left term of $M_{\mathrm{Li,h}}$ equals the minimal number of Li misses for fetching both operands and storing the result. The right term counts the number of additional Li misses, when the cluster size either approaches Li size or even exceeds this. As soon as the clusters become significantly larger than Li, each memory access yields a cache miss due to cache trashing: with a bucket-chain length of 4, up to 8 memory accesses per tuple are necessary while building the hash-table and performing the hash-lookup, and another 2 to access the actual tuple. When the cluster sizes get very small, hash-tables of a fixed minimal size (256 buckets) need to be allocated and destroyed at increasing frequency. This causes additional L1 misses, approximately 6 per cluster. Hence, the term $H \cdot 6$ has to be added to $M_{\mathrm{L1,h}}$. Again, the number of TLB misses is modeled analogously.

The lines in Fig. 15 represent our model for different cardinalities ($w_{\mathrm{h}} = 680\,\mathrm{ns}$, $w'_{\mathrm{h}} = 3600\,\mathrm{ns}$). The predictions are very accurate.

4.3.3 Improved partitioned hash-join

The model in the previous section shows that our original implementation of partitioned hash-join suffers from two parameter settings: the average hash-bucket size of 4 tuples and the minimal hash-table size of 256 buckets.

Hash-bucket size. Following the linked list within a hash-bucket (during hash-build and hash-probe) performs a random memory access pattern with very poor data locality. Hence, up to 8 additional cache and TLB misses per tuple occur with large clusters (see above). The only way to improve this situation is to avoid random memory access as much as possible by reducing the hash-bucket size. We modify our implementation to use *perfect hashing*, i.e., reducing the targeted hash-bucket size from 4 tuples to just 1 tuple.

Hash-table size. Another (minor) improvement we make is that we decrease the minimal hash-table size from 256 buckets to just 2 buckets.

Figure 16 compares our original version of the partitioned hash-join as presented in [BMK99] with the improved implementation as presented above. Reducing the hash-bucket size yields a significant reduction in TLB and cache misses for large clusters: from 8 to just 3 additional misses per tuple; in other words, almost a factor of 2 in the total number of misses. This in turn speeds up the execution time by almost a factor of 2 for large clusters. Even for small clusters, where no additional cache or TLB misses occur, the performance has (slightly) improved, as now less comparisons are necessary during hash-lookups. Additionally, reducing the minimal hash-table size avoids the increase in L1 cache misses with very small clusters. But this has hardly any impact on the execution time, as the CPU costs for creating and destroying a large number of tiny hash-tables dominate the performance.

Altogether, our results show that tuning the join phase of a partitioned hash-join appropriately – in addition to optimizing the clustering phase as we proposed in [BMK99] – achieves an additional performance improvement of up to a factor of 2.

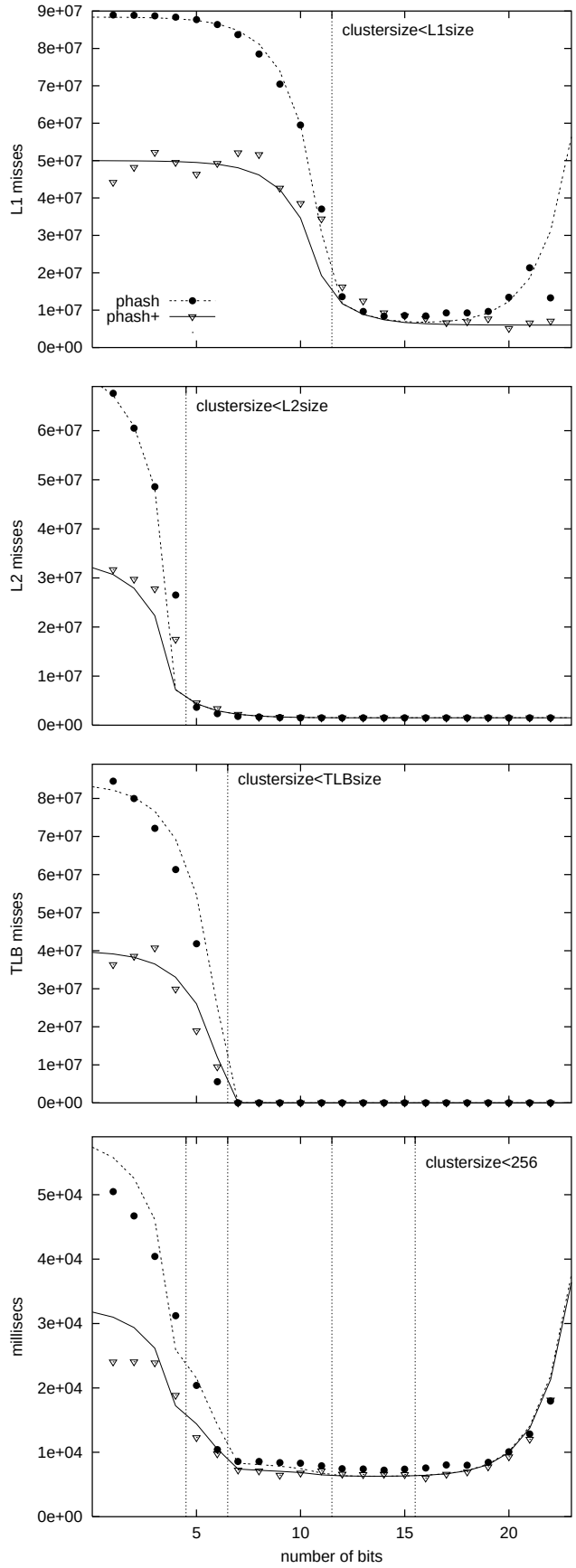

Fig. 16. Original vs. improved partitioned hash-join ($C = 8$M)

4.4 Overall 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 cost for both the partitioned hash-join and radix-join. Radix-cluster gets cheaper for less radix B bits, whereas both radix-join and partitioned hash-join get more expensive. Putting together the experimental data we obtained on both cluster and join performance, we determine the optimum number of B for the relation cardinality and the join algorithm.

It turns out that there are four possible strategies, which correspond to the diagonals in Figs. 15 and 14:

phash(+) L2: (improved) partitioned hash-join on $B = \log_2(C \cdot 12/||L2||)$ clustered bits, so the inner relation plus hash-table fits the L2 cache. This strategy was used in the work of Shatdahl et al. [SKN94] in their partitioned hash-join experiments.

phash(+) TLB: (improved) partitioned hash-join on $B = \log_2(C \cdot 12/||TLB||)$ clustered bits, so the 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: (improved) partitioned hash-join on $B = \log_2(C \cdot 12/||L1||)$ 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 trashing on this many clusters).

radix: radix-join on $B = \log_2(C/8)$ clustered bits. The radix-join has the most stable performance but has higher startup cost, as it needs to radix-cluster on significantly more bits than the other options. It is therefore only a winner with large cardinalities.

Figure 17 compares radix-join (thin lines) and original partitioned hash-join (thick lines) throughout the whole bit range, using the corresponding optimal number of passes for the radix-cluster (see Sect. 4.2). The diagonal lines mark the setting for B that belong to the four strategies. The optimal setting for original partitioned hash-join varies between phash TLB and phash L1. With bigger clusters, the join phase is too expensive; with smaller clusters, clustering becomes too expensive. The differences between phash TLB and phash L1 are very small; hence, for simplicity of presentation, we refer to the optimal setting as "phash TLB." Similarly, radix-join yields it best performance somewhere between 16 and 4 tuples per cluster. We refer to the optimal setting as "radix 8." In most cases, radix 8 outperforms phash TLB slightly. Figure 18 shows the respective comparison between radix-join and improved partitioned hash-join. Again, phash+ TLB is the optimal setting, but now phash+ TLB slightly outperforms radix 8.

Figure 19 compares the overall performance of all three join algorithms – radix, phash, and phash+ – for a cardinality of 8M tuples. The non-logarithmic scale on the y-axis clearly shows the large improvement in the performance of database algorithms (an equi-join, in this case) due to memory access optimization.

Finally, Table 2 compares our radix-cluster-based strategies to non-partitioned ("simple") hash-join and sort/merge-

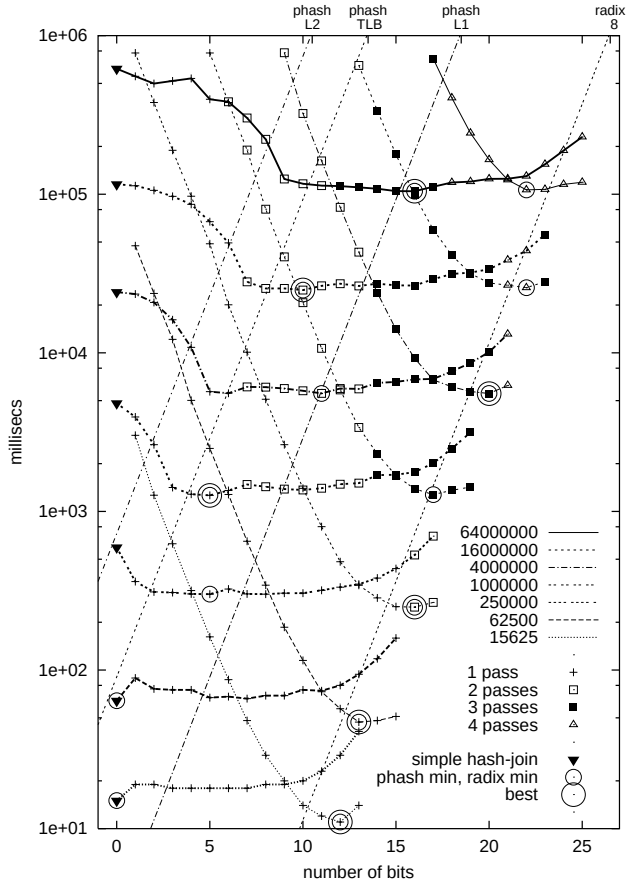

Fig. 17. Overall performance of radix-join (*thin lines*) vs. *original* partitioned hash-join (*thick lines*)

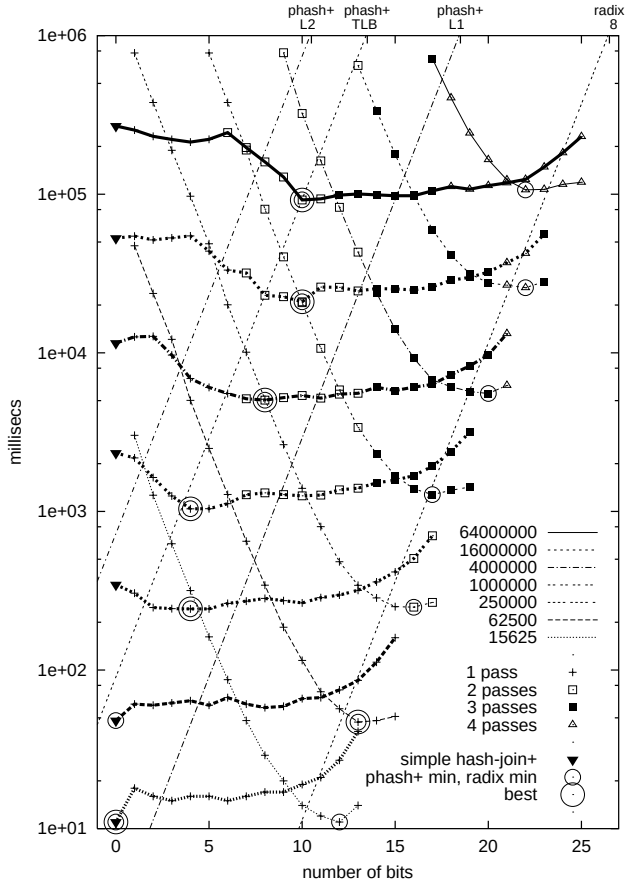

Fig. 18. Overall performance of radix-join (*thin lines*) vs. *improved* partitioned hash-join (*thick lines*)

join. Our sort/merge-join uses a quick-sort algorithm, which shows a reasonably good memory access pattern. For larger cardinalities, it runs up to twice as fast as simple hash-join. However, on smaller relations the that fit into the L2 cache, simple hash-join is about twice as fast as sort/merge-join. The original version of partitioned hash-join (phash TLB) and radix 8 show very similar performance, running up to almost 6 times faster than simple hash-join. The improved version of partitioned hash-join (phash+ TLB) performs about 10%–20% faster than radix 8 and phash TLB, being almost a factor of 7 faster than the simple hash-join. This clearly demonstrates that cache-conscious join algorithms perform significantly better than the "random-access" algorithms. Here, "cache conscious" does not only refer to L2 cache, but also to the L1 cache and especially the TLB. Further, Figs. 17 and 18 show that our radix algorithms improve join performance, both in the "phash(+) TLB / L1" strategies (cardinalities larger than 250 000 require at least two clustering passes) and with the radix-join itself.

5 Evaluation

In this research, we brought to light the severe impact of memory access on the performance of elementary database operations. Hardware trends indicate that this bottleneck has been present for quite some time; hence our expectation is that its impact will eventually become deeper than the I/O bottleneck. Database algorithms and data structures should therefore be designed and optimized for memory access from the outset. Sloppy implementation of the key algorithms or "features" at the innermost level of an operator tree (e.g., pointer swizzling/object table lookup) can lead to a performance disaster that ever faster CPUs will not be able to rescue.

Conversely, careful design can lead to an order of magnitude performance advancement. In our Monet system, under development since 1992, we have decreased the memory access stride using vertical decomposition; a choice that back in 1992 was mostly based on intuition. The work presented here now provides strong evidence that this feature is in fact the basis of good performance. Our simple-scan experiment demonstrates that decreasing the stride is crucial for optimizing usage of memory bandwidth.

Concerning query processing algorithms, we have formulated radix algorithms and demonstrated through experimentation that these algorithms form both an addition and an improvement to the work in [SKN94]. The modeling work done to show how these algorithms improve cache behavior during join processing represents an important improvement over previous work on main-memory cost models [LN96,WK90]. Rather than characterizing main-memory performance on the coarse level of a procedure call with "magical" costs factors obtained by profiling, our methodology mimics the memory access pattern of the algorithm to be modeled and then quantifies its cost by counting cache miss events and CPU cycles. We were helped in formulating these models through our usage of

Table 2. Algorithm comparison: absolute performance and relative improvement over simple hash-join and sort/merge-join

Cardinality	Simple hash-join	Sort/merge-join		phash TLB			radix 8			phash+ TLB		
	$\mathcal{A}$ (ms)	$\mathcal{B}$ (ms)	$\mathcal{A}/\mathcal{B}$	$\mathcal{C}$ (ms)	$\mathcal{A}/\mathcal{C}$	$\mathcal{B}/\mathcal{C}$	$\mathcal{D}$ (ms)	$\mathcal{A}/\mathcal{D}$	$\mathcal{B}/\mathcal{D}$	$\mathcal{E}$ (ms)	$\mathcal{A}/\mathcal{E}$	$\mathcal{B}/\mathcal{E}$
64000000	618527	400074	1.55	104467	5.92	3.83	106504	5.81	3.76	91922	6.73	4.35
32000000	273312	179150	1.53	49579	5.51	3.61	49104	5.57	3.65	43471	6.29	4.12
16000000	117519	55515	2.12	24892	4.72	2.23	25759	4.56	2.16	20972	5.60	2.65
8000000	52652	26639	1.98	11992	4.39	2.22	11547	4.56	2.31	10134	5.20	2.63
4000000	24170	12193	1.98	5552	4.35	2.20	5540	4.36	2.20	5033	4.80	2.42
2000000	11033	5855	1.88	2730	4.04	2.14	2743	4.02	2.13	2064	5.35	2.84
1000000	4849	2743	1.77	1264	3.84	2.17	1277	3.80	2.15	1038	4.67	2.64
500000	1877	1326	1.42	617	3.04	2.15	608	3.09	2.18	498	3.77	2.66
250000	597	613	0.97	301	1.98	2.04	249	2.40	2.46	243	2.46	2.52
125000	149	286	0.52	138	1.08	2.07	100	1.49	2.86	118	1.26	2.42
62500	66	137	0.48	64	1.03	2.14	47	1.40	2.91	48	1.38	2.85
31250	31	65	0.48	31	1.00	2.10	23	1.35	2.83	23	1.35	2.83
15625	15	31	0.48	15	1.00	2.07	11	1.36	2.82	11	1.36	2.82

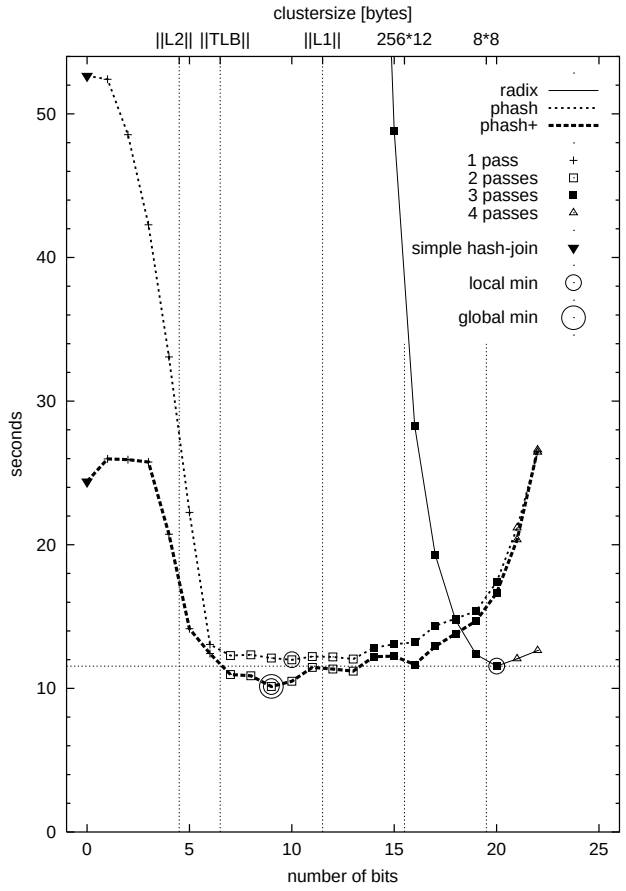

Fig. 19. Overall performance of radix-join, original and improved partitioned hash-join (C=8M)

hardware event counters present in modern CPUs. Our detailed cost models enabled us to identify a significant bottleneck in the implementation of the partitioned hash-join (following the bucket-chain during hash-lookups caused too many cache and TLB misses) and hence to improve the implementation using perfect hashing.

We think our findings are not only relevant to main-memory databases engineers. Vertical fragmentation and memory access cost 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 such as 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. We observed this same effect with the Drill Down Benchmark [BRK98], where a commercial database product managing disk-resident data was run with a large buffer pool. While executing almost exclusively memory-bound, this product was measured to be a factor 40 slower on this benchmark than the Monet system. After inclusion of cache-optimization techniques such as described in this paper, we have since been able to improve our own results on this benchmark with almost an extra order of magnitude. This clearly shows the importance of main-memory access optimization techniques.

In Monet, we use I/O by manipulating virtual memory mappings and hence treat disk-resident data as memory with a large granularity. 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 (Fig. 2). Algorithms that are tuned to run well on one level of the memory also exhibit high performance on the lower levels (e.g., radix-join has pure sequential access and consequently also runs well on virtual memory). As the major performance bottleneck is shifting from I/O to memory access, we therefore think that main-memory optimization of both data structures and algorithms – like those described in this paper – will be increasingly decisive in efficiently exploiting the power of custom hardware.

6 Conclusion and future work

It was shown that memory access cost is increasingly a bottleneck for database performance. We subsequently discussed the consequences of this finding on both the data structures and algorithms employed in database systems. We recommend using vertical fragmentation in order to better use scarce memory bandwidth. We introduced new radix algorithms for use in join processing, and we formulated detailed analytical cost models that explain why these algorithms make optimal use of hierarchical memory systems found in modern computer

hardware. Further, our cost models enabled us to identify the hash-bucket size as a performance bottleneck for hash-joins in the main memory. We showed that using perfect hashing solves the problem and achieves the best performance, up to twice as fast as our previous results [BMK99]. Finally, we placed our results in a broader context of database architecture, and made recommendations for future systems.

We only used one sample architecture (an SGI Origin2000) in this study; however, our on-going work is to investigate how both our optimization techniques and our cost models perform on other hardware platforms. In [MBK00], we present a *calibration tool* to automatically extract relevant characteristics of the (cache-) memory system (such as cache sizes, cache line sizes, and cache miss latencies) of any computer hardware. We show that feeding these parameters into the cost models presented here is sufficient to accurately predict the performance of database algorithms on different popular computer systems, such as a Sun Ultra workstation, an Intel PentiumIII PC, and an AMD Athlon PC. This in turn enables us to automatically tune our memory-conscious algorithms to their optimal settings on any hardware they run on.

References

[AvdBF+92] Apers P.M.G., van den Berg C.A., Flokstra J., Grefen P.W.P.J., Kersten M., Wilschut A.N. PRISMA/DB: A parallel main memory relational DBMS. IEEE Transactions on Knowledge and Data Engineering 4(6): 541–554, December 1992

[BK95] Boncz P., Kersten M. Monet: An impressionist sketch of an advanced database system. In: Proc. Basque Int. Workshop on Information Technology, San Sebastian, Spain, July 1995

[BK99] Boncz P., Kersten M. MIL primitives for querying a fragmented world. The VLDB Journal 8(2): 101–119, Oct 1999

[BMK99] Boncz P., Manegold S., Kersten M. Database architecture optimized for the new bottleneck: memory access. In: Proc. Int. Conf. on Very Large Data Bases, pp. 54–65, Edinburgh, Scotland, UK, September 1999

[BQK96] Boncz P., Quak W., Kersten M. Monet and its geographical extensions: a novel approach to high-performance GIS processing. In: Proc. Int. Conf. on Extending Database Technology, pp. 147–166, Avignon, France, June 1996

[BRK98] Boncz P., Rühl T., Kwakkel F. The drill down benchmark. In: Proc. Int. Conf. on Very Large Data Bases, pp. 628–632, New York, USA, June 1998

[BWK98] Boncz P., Wilschut A.N., Kersten M. Flattening an object algebra to provide performance. In: Proc IEEE Int. Conf. on Data Engineering, pp. 568–577, Orlando, Fla., USA, February 1998

[CK85] Copeland G.P., Khoshafian S. A decomposition storage model. In: Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 268–279, Austin, Tex., USA, May 1985

[Knu68] Knuth D.E. The art of computer programming, vol. 1. Addison-Wesley, Reading, Mass., USA, 1968

[LC86] Lehman T.J., Carey M.J. A study of index structures for main memory database management systems. In: Proc. Int. Conf. on Very Large Data Bases, pp. 294–303, Kyoto, Japan, August 1986

[LN96] Listgarten S., Neimat M.-A. Modelling costs for a MM-DBMS. In: Proc. In.t Workshop on Real-Time Databases, Issues and Applications, pp. 72–78, Newport Beach, Calif., USA, March 1996

[MBK99] Manegold S., Boncz P., Kersten M. Optimizing main-memory join on modern hardware. Technical Report INS-R9912, CWI, Amsterdam, The Netherlands, October 1999

[MBK00] Manegold S., Boncz P.A., Kersten M.L. What happens during a join? – Dissecting CPU and memory optimization effects. In: Proc. Int. Conf. on Very Large Data Bases, Cairo, Egypt, September 2000, pp. 339–350

[McC95] J.D. McCalpin. Memory bandwidth and machine balance in current high performance computers. IEEE Technical Committee on Computer Architecture Newsletter, December 1995

[MKW+98] McKee S., Klenke R., Wright K., Wulf W., Salinas M., Aylor J., Batson A. Smarter memory: improving bandwidth for streamed references. IEEE Computer 31(7): 54–63, July 1998

[Mow94] Mowry T.C. Tolerating latency through software-controlled data prefetching. PhD thesis, Stanford University, Computer Science Department, 1994

[NBC+94] Nyberg C., Barclay T., Cvetanovic Z., Gray J., Lomet D. AlphaSort: a RISC machine sort. In Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 233–242, Minneapolis, Minn., USA, May 1994

[Ram96] Rambus Technologies, Inc. Direct rambus technology disclosure, 1996. http://www.rambus.com/docs/drtechov.pdf

[Ron98] Ronström M. Design and Modeling of a parallel data server for telecom applications. PhD thesis, Linköping University, Sweden, 1998

[Sil97] Silicon Graphics, Inc., Mountain View, Calif. Performance Tuning and Optimization for Origin2000 and Onyx2, January 1997

[SKN94] Shatdal A., Kant C., Naughton J. Cache conscious algorithms for relational query processing. In: Proc. Int. Conf. on Very Large Data Bases, pp. 510–512, Santiago, Chile, September 1994

[SLD97] SLDRAM Inc. SyncLink DRAM Whitepaper, 1997. http://www.sldram.com/Documents/SLDRAMwhite970910.pdf

[Val87] Valduriez P. Join indices. ACM Transactions on Database Systems 12(2): 218–246, June 1987

[WK90] Whang K.-Y., Krishnamurthy R. Query Optimization in a Memory-Resident Domain Relational Calculus Database System. ACM Transactions on Database Systems 15(1): 67–95, March 1990