Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Optimizing Main-Memory Join On Modern Hardware

S. Manegold, P. Boncz, M.L. Kersten

Information Systems (INS)

INS-R9912 October 31, 1999

Report INS-R9912
ISSN 1386-3681

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Optimizing Main-Memory Join On Modern Hardware

Stefan Manegold[1] Peter Boncz[2] Martin Kersten[1]

[1] *CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands*
{*Stefan.Manegold,Martin.Kersten*}*@cwi.nl*

[2] *Data Distilleries, Kruislaan 419, 1098 VA Amsterdam, The Netherlands*
Peter.Boncz@ddi.nl

ABSTRACT

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. In this article, we show how these resources can be optimized for large joins and translate these insights into guidelines for future database architectures, encompassing data structures, algorithms, cost modeling, and implementation. In particular, we discuss how vertically fragmented data structures optimize cache performance on sequential data access. On the algorithmic side, we refine the partitioned hash-join with a new partitioning algorithm called radix-cluster, which is specifically designed to optimize memory access. The performance of this algorithm is quantified using a detailed analytical model that incorporates memory access costs in terms of a limited number of parameters, such as cache sizes and miss penalties. We also present a calibration tool that extracts such parameters automatically from any computer hardware. The accuracy of our models is proven by exhaustive experiments conducted with the Monet database system on three different hardware platforms. Finally, we investigate the effect of implementation techniques that optimize CPU resource usage. Our experiments show that large joins can be accelerated almost an order of magnitude on modern RISC hardware when both memory and CPU resources are optimized.

1991 ACM Computing Classification System: [H.2.4] Main-Memory Database Systems, Query Processing
Keywords and Phrases: main-memory databases, query processing, memory access optimization, decomposed storage model, join algorithms, implementation techniques
Note: Work carried out under project INS1.2 "Database Architecture".

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 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]. One 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 is getting slower at a dramatic rate, making it increasingly difficult to achieve high processor efficiencies. Another trend is the ever increasing number 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.

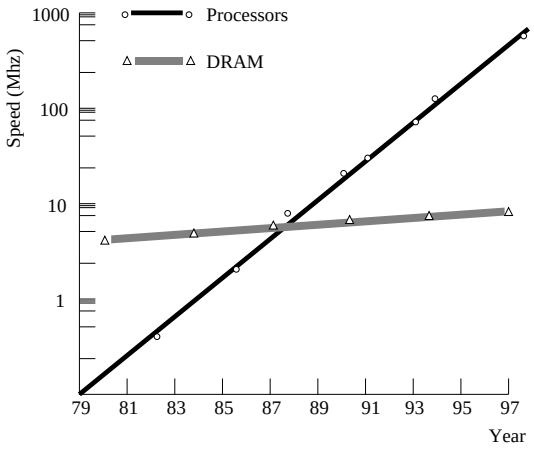


Figure 1: Trends in DRAM and CPU speed

In this article, 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 demonstrate that the performance of even simple database operations is severely constrained by memory access costs. For example, a simple in-memory table scan runs on Sun hardware from the year 1999 in roughly the same absolute time as on a Sun from 1992, now spending 95% of its cycles waiting for memory (see Section 2.2). 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. In this work, we 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. We also outline our *calibration tool* that extracts these cost parameters automatically from any computer system.

- *Memory optimization and efficient coding techniques boost each others effects.* CPU resource utilization can be optimized with 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 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. We use Monet as our experimentation platform.

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 50-100 is typical in 1999 (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 [BK99], developed by our own group since 1994 and commercially deployed in a Data Mining tool [KSHK97]. Monet is implemented using aggressive coding techniques for optimizing CPU resource utilization [BK99] that go much beyond the usual MMDBS implementation techniques [DKO+84]. For example, Monet is written in a macro language, from which C language implementations are generated. The macros implement a variety of techniques, by virtue of which the inner loops of performance-critical algorithms like join are free of overheads like database ADT calls, data movement, and loop condition management. These techniques were either pioneered by our group (e.g., logarithmic code expansion [Ker89]) or taken from the field of high performance computing [Sil97]. In this work, we will show that these techniques allow compilers to produce code that better exploits the parallel resources offered by modern CPUs.

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, ZLTI96, 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. 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.

1.2 Outline

In Section 2, we describe the aspects of memory and CPU technology found in custom hardware that are most relevant for the performance of main-memory query execution. We identify ongoing trends, and outline their consequences for database architecture. In addition, we describe our *calibration tool* which extracts the most important hardware characteristics like cache size, cache line size, and cache latency from any computer system, and provide results for our benchmark platforms (modern Sun, SGI, and Intel hardware).

In Section 3, 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, and we show how performance can exactly be predicted from hardware events like cache and TLB misses.

In Section 4, we present *radix-join* as an alternative to partitioned hash-join, and compare the performance of both algorithms.

In Section 5, 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.

2. Modern Hardware and DBMS Performance

First, we describe the technical details of modern hardware relevant for main-memory query performance, focusing on CPU and memory architectures. We perform experiments to illustrate how the balance between CPU and memory costs in query processing has shifted through time, and discuss a calibration tool that automatically extracts the hardware parameters most important for performance prediction from any computer system. We then look at what future hardware technology has in store, and identify a number of trends.

2.1 A Short Hardware Primer

While CPU clock frequency has been following Moore's law (doubling every three years), CPUs have additionally become faster through parallelism *within* the processor. Scalar CPUs separate different execution stages for instructions, e.g., allowing a computation stage of one instruction to be overlapped with the decoding stage of the next instruction. Such a *pipelined* design allows for inter-stage parallelism. Modern *superscalar* CPUs add intra-stage parallelism, as they have multiple copies of certain (pipelined) units that can be active simultaneously. Although CPUs are commonly classified as either RISC or CISC, modern CPUs combine successful features of both. Figure 2 shows a simplified schema that characterizes how modern CPUs work: instructions that need to be executed are loaded from memory by a fetch-and-decode unit. In order to speed up this process, multiple fetch-and-

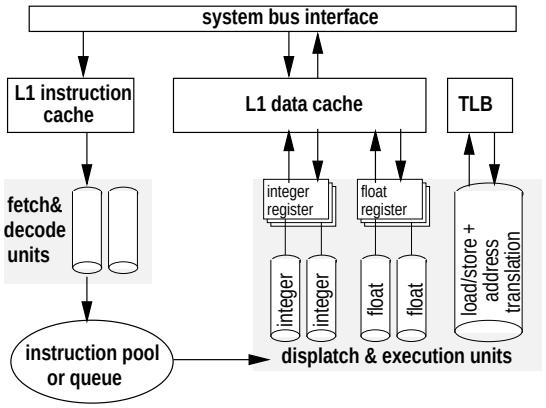


Figure 2: Modern Out-Of-Order CPU

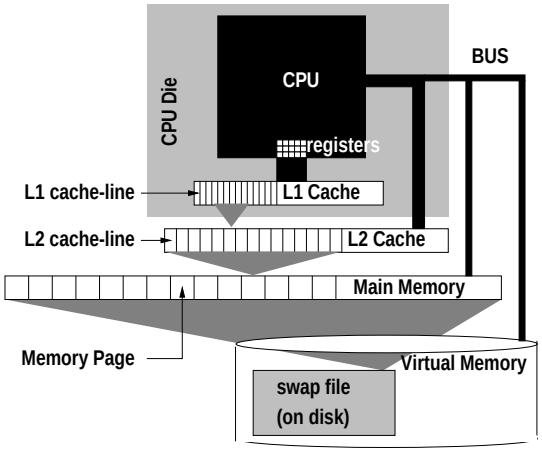


Figure 3: Hierarchical Memory System

decode units may be present (e.g., the PentiumIII has three, the R10000 two). Decoded instructions are placed in an instruction queue, from which they are executed by one of various functional units, which are sometimes specialized in integer-, floating-point, and load/store pipelines. The PentiumIII, for instance, has two such functional units, whereas the R10000 has even five. To exploit this parallel potential, modern CPUs rely on techniques like *branch prediction* to predict which instruction will be next before the previous has finished. Also, the modern cache memories are *non-blocking*, which means that a cache miss does not stall the CPU. Such a design allows the pipelines to be filled with multiple instructions that will probably have to be executed (a.k.a. *speculative execution*), betting on yet unknown outcomes of previous instructions. All this goes accompanied by the necessary logic to restore order in case of mispredicted branches. As this can cost a significant penalty, and as it is very important to fill all pipelines to obtain the performance potential of the CPU, much attention is paid in hardware design to efficient branch prediction. CPUs work with *prediction* tables that record statistics about branches taken in the past.

Modern computer architectures have a hierarchical memory system, as depicted in Figure 3, where access by the CPU to main memory, consisting of DRAM chips on the system board, is accelerated by various levels of cache memories. Introduction of these cache memories, that consist of fast but expensive SRAM chips, was necessary due to the fact that DRAM memory latency has progressed little through time, making its performance relative to the CPU become worse exponentially. First, one level of cache was added by placing SRAM chips on the motherboard. Then, as CPU clock-speeds kept increasing, the physical distance between these chips and the CPU became a problem, as it takes a minimum amount of time per distance to carry an electrical signal over a wire. As a result, modern CPUs have cache memories inside the processor chip. In this work, we assume one on-chip cache called L1, and a typically larger off-chip cache on the system board called L2. We identify three aspects that determine memory access costs:

latency Our exact definition of *memory latency* (l_{Mem}) is the time needed to transfer one byte from the main memory to the L2 cache. This occurs, when the piece of memory being accessed is in neither the L1 nor the L2 cache, so we speak of an *L2 miss*. It is important to note that during this time, all current hardware actually transfers multiple consecutive words to the memory subsystem, since each cache level has a smallest unit of transfer (called the *cache line*). During one memory fetch, modern hardware loads an entire cache line from main memory[1] in one go,

[1]To which cache line the memory is loaded, is determined from the memory address. An X-way associative cache allows to load a line in X different positions. If X>1, some *cache replacement* policy chooses one from the X candidates. Least Recently Used (LRU) is the most common replacement algorithm.

by reading data from many DRAM chips at the same time, transferring all bits in the cache line in parallel over a wide bus. Similarly, with *L2 latency* (l_{L2}) we mean the time it takes the CPU to access data that is in L2 but not in L1 (an *L1 miss*), and *L1 latency* (l_{L1}) is the time it takes the CPU to access data in L1. Each L2 miss is preceded by an L1 miss. Hence, the total latency to access data that is in neither cache is $l_{Mem} + l_{L2} + l_{L1}$. As L1 latency cannot be avoided, we assume in the remainder of this paper, that L1 latency is included in the pure CPU costs, and regard only memory latency and L2 latency as explicit memory access costs.

bandwidth We define *memory bandwidth* as the number of megabytes of main memory the CPU can access per second. On some architectures, there is a difference between read and write bandwidth, but this difference tends to be small. Bandwidth is usually maximized on a sequential access pattern, as only then all memory words in the cache lines are fully used. In conventional hardware, the memory bandwidth used to be simply the cache line size divided by the memory latency, but modern multiprocessor systems typically provide excess bandwidth capacity.

address translation The Translation Lookaside Buffer (TLB) is a common element in modern CPUs (see Figure 2). This buffer is used in the translation of logical virtual memory addresses used by application code to physical page addresses in the main memory of the computer. The TLB is a kind of cache that holds the translation for the most recently used pages (typically 64). If a logical address is found in the TLB, the translation has no additional costs. However, if a logical address is not cached in the TLB, a *TLB miss* occurs. The more pages an application uses (which is also dependent of the often configurable size of the memory pages), the higher the probability of TLB misses. A TLB miss can either be handled in hardware or in software, depending on the computer architecture. Hardware-handled TLB fetches the translation from a fixed memory structure, which is just filled by the operating system. Software-handled TLB leaves the translation method entirely to the operating system, but requires trapping to a routine in the operating system kernel on each TLB miss. Depending on the implementation and hardware architecture, TLB misses can therefore be more costly even than a main-memory access. Moreover, as address translation often requires accessing some memory structure, this can in turn trigger additional memory cache misses.

2.2 Experimental Quantification

We use a simple scan test to demonstrate the severe impact of memory access costs on the performance of elementary database operations. 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. We make sure that the buffer is in memory, but not in any of the memory caches, by first scanning it and then multiple times scanning some other buffer larger than the largest cache size. Our 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 in a simple aggregation (e.g., SELECT MAX(column) FROM table).

Figure 4 shows results of this experiment on a number of popular workstations of the past decade. The X-axis shows the different systems ordered by their age, and per system the different strides tested. The Y-axis shows the absolute elapsed time for the experiments. For each system, the graph is split up to show which part of the elapsed time is spent waiting for memory (upper), and which part with CPU processing (lower, gray-shaded).

All systems show the same basic behavior with best performance at stride 1, increasing to some maximum at a larger stride, after which performance stays constant. This is explained as follows: 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, and the memory access costs are negligible compared to the CPU costs. As the stride increases, the cache miss rates and thus the memory access costs also increase. The cache miss rates reach their maxima, as

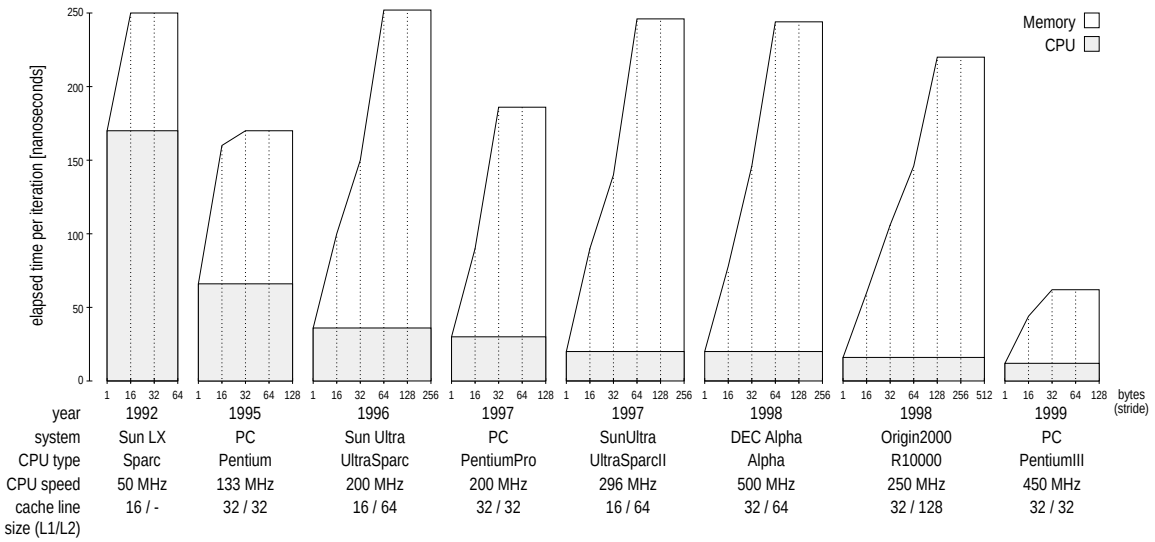


Figure 4: CPU and memory access costs per tuple in a simple table scan

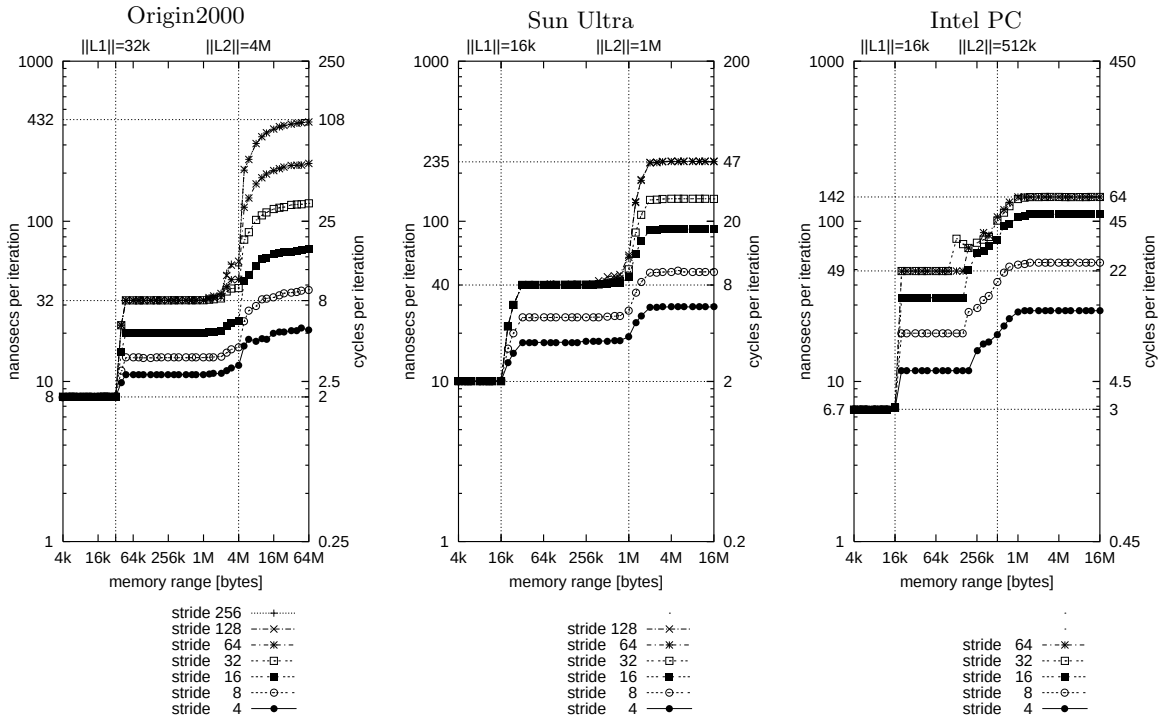
soon as the stride reaches the cache line size. Then, every memory read is a cache miss. Performance cannot become any worse and stays constant.

When comparing the Sun LX to the Origin2000, we see that CPU performance has increased 10-fold, of which a factor 5 can be attributed to faster clock frequency (from 50MHz to 250MHz), and a factor 2 to increased processor parallelism (the CPU costs have fallen from 160ns at 50MHz = 8 cycles to 16ns at 250MHz = 4 cycles). While this trend of exponentially increasing CPU performance is easily recognizable, the memory cost trend in Figure 4 shows a mixed picture, and has clearly not kept up with the advances in CPU power. Consequently, while our experiment was still largely CPU-bound on the Sun from 1992, it is dominated by memory access costs on the modern machines (even the PentiumIII with fast memory is 75% of the time waiting for memory). Note that the later machines from Sun, Silicon Graphics and DEC actually have memory access costs that in absolute numbers are even higher than on the Sun from 1992. This can be attributed to the complex memory subsystem that comes with SMP architectures, resulting in a high memory latency. These machines do provide a high memory bandwidth—thanks to the ever growing cache line sizes[2]—but this does not do any good in our experiment at large strides (when data locality is low).

This simple experiment also makes clear why database systems are quickly constrained by memory access, even on simple tasks like scanning, that seem to have an access pattern that is easy to cache (sequential). The default physical representation of a tuple is a consecutive byte sequence (a "record"), which must always be accessed by the bottom operators in a query evaluation tree (typically selections or projections). The record byte-width of typical relational table amounts to some hundreds of bytes. Figure 4 makes clear that such large strides lead to worst-case performance, such that the memory access bottleneck kills all CPU performance advances.

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

[2]In one cache miss, the Origin2000 fetches 128 bytes, whereas the Sun LX fetches only 16; an improvement of factor 8.



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

Figure 5: Calibration Tool: Cache sizes, line sizes, and latencies

therefore the key for reducing memory access costs (staying on the left side of the graphs in Figure 4). In Section 4, we will come back at specific implementation details of Monet.

2.3 Calibration Tool

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

Calibrating the Memory System Our calibrator is a simple C program, mainly a small loop that executes a million memory reads. By changing the stride and the size of the memory area, we force varying cache miss rates. Thus, we can calculate the latency for a cache miss by comparing the execution time without misses to the execution time with exactly one miss per iteration. This approach only works, if memory accesses are executed purely sequential, i.e., we have to make sure that neither two or more load instructions nor memory access and pure CPU work overlap. We use a simple pointer chasing mechanism to achieve this: the memory area we access is initialized such that each load returns the address for the subsequent load in the next iteration. Thus, super-scalar CPUs cannot benefit from their ability to hide memory access latency by speculative execution.

To measure the cache characteristics, we run our experiment several times, varying the stride and the array size. We make sure that the stride varies at least between 4 bytes and twice the maximal expected cache line size, and that the array size varies from half the minimal expected cache size to at least ten times the maximal expected cache size.

The leftmost plot in Figure 5 depicts the resulting execution time (in nanoseconds) per iteration

for different array sizes on an Origin2000 (MIPS R10000, 250 MHz = 4ns per cycle). Each curve represents a different stride. From this figure, we can derive the desired parameters as follows: Up to an array size of 32 KB, one iteration takes 8 nanoseconds (i.e., 2 cycles), independent on the stride. Here, no cache misses occur once the data is loaded, as the array completely fits in L1 cache. One of the two cycles accounts for executing the load instruction, the other one accounts for the latency to access data in L1. With array sizes between 32 KB and 4 MB, the array exceeds L1, but still fits in L2. Thus, L1 misses occur. The miss rate (i.e., the number of misses per iteration) depends on the stride (s) and the L1 cache line size (LS_{L1}). With $s < LS_{L1}$, $\frac{s}{LS_{L1}}$ L1 misses occur per iteration (or one L1 miss occurs every $\frac{LS_{L1}}{s}$ iterations). With $s \geq LS_{L1}$, each load causes an L1 miss. Figure 5 shows that the execution time increases with the stride, up to a stride of 32. Then, it stays constant. Hence, L1 line size is 32 byte. Further, L1 miss latency (i.e., L2 access latency) is 32ns − 8ns = 24ns, or 6 cycles. Similarly, when the array size exceeds L2 size (4 MB), L2 misses occur. Here, the L2 line size is 128 byte, and the L2 miss latency (memory access latency) is 432ns − 32ns = 400ns, or 100 cycles. Analogously, the middle and the rightmost plot in Figure 5 show the results for a Sun Ultra (Sun UltraSPARC, 200 MHz = 5ns per cycle) and an Intel PC (Intel PentiumIII, 450 MHz = 2.22ns per cycle).

The *sequential memory bandwidth* for our systems, listed in Table 1, is computed from the cache line sizes and the latencies as follows: $bw_{seq} = \frac{LS_{L2}}{l_{Mem} + \frac{LS_{L2}}{LS_{L1}} * l_{L2}}$. We will discuss *parallel memory bandwidth* in the next section.

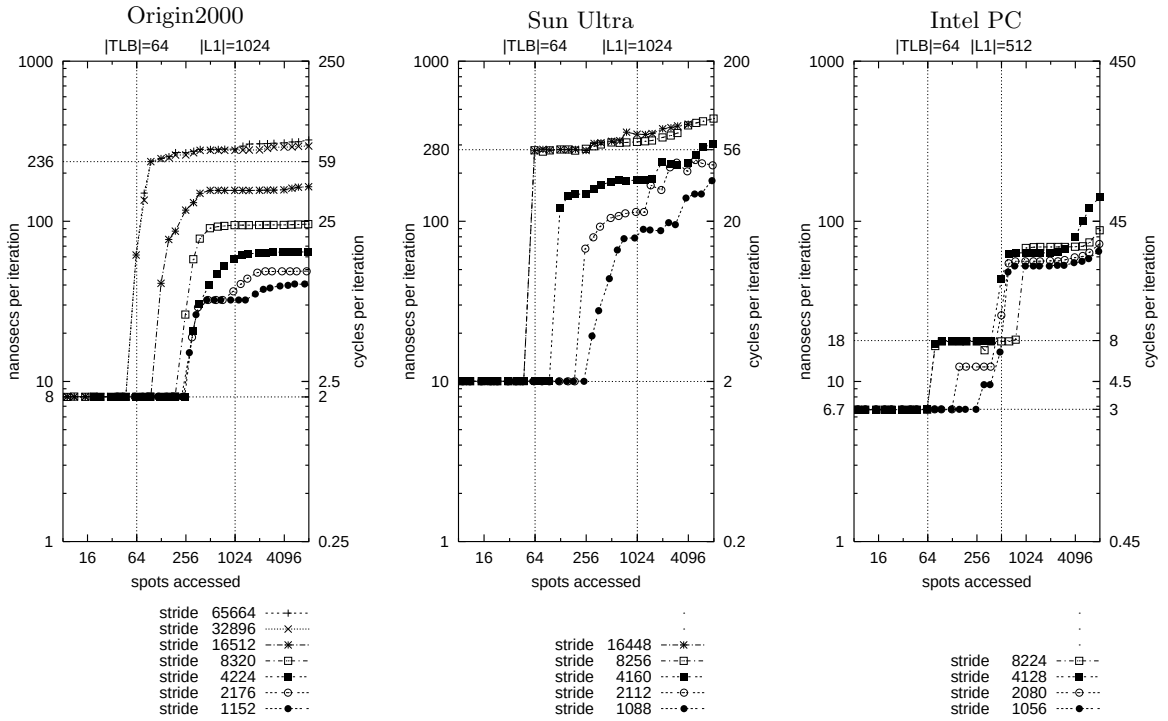
Calibrating the TLB We use a similar approach as above to measure *TLB miss costs*. The idea here is to force one TLB miss per iteration, but to avoid any cache misses. We force TLB misses by using a stride that is equal to or larger than the systems page size, and by choosing the array size such that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

Figure 6 shows the results for three machines. The X-axis now gives the number of spots accessed, i.e., array size divided by stride. Again, each curve represents a different stride. From the leftmost plot (Origin2000), e.g., we derive the following: Like above, we observe the base line of 8 nanoseconds (i.e., 2 cycles) per iteration. The smallest number of spots where the performance decreases due to TLB misses is 64, hence, there must be 64 TLB entries. The decrease at 64 spots occurs with strides of 32 KB or more, thus, the page size is 32 KB. Further, TLB miss latency is 236ns − 8ns = 228ns, or 57 cycles. In the rightmost plot, the second step at 512 spots is caused by L1 misses as L1 latency is 4 times higher than TLB latency on the PC. On the Origin2000 and on the Sun, L1 misses also occur with more than 1024 spots access, but their impact is negligible as TLB latency is almost 10 times higher than L1 latency on these machines.

Table 1 gathers the results for all three machines. The PC has the highest L2 latency, probably as its L2 cache is running at only half the CPU's clock speed, but it has the lowest memory latency. Its very low TLB latency is due to the hardware implementation of TLB management on the PC, which avoids the costs of trapping to the operating system on a TLB miss, that is necessary in the software controlled TLBs of the other systems. The Origin2000 has the highest memory latency, but due to its large cache lines, it achieves the best sequential memory bandwidth.

2.4 Parallel Memory Access

It is interesting to note that the calibrated latencies in Table 1 do not always confirm the suggested latencies in the sequential scan experiment from Figure 4. For the PentiumIII, the access costs per



(Vertical grid lines indicate derived number of TLB entries, horizontal grid lines indicate derived latencies.)

Figure 6: Calibration Tool: TLB entries and TLB miss costs

		SGI Origin2000	Sun Ultra	Intel PC
OS		IRIX64 6.5	Solaris 2.5.1	Linux 2.2.5
CPU		MIPS R10000	Sun UltraSparc	Intel PentiumIII
CPU speed		250 MHz	200 MHz	450 MHz
main-memory size		48 GB (4 GB local)	512 MB	512 MB
L1 cache size	$\|\|L1\|\|$	32 KB	16 KB	16 KB
L1 cache line size	LS_{L1}	32 bytes	16 bytes	32 bytes
L1 cache lines	$\|L1\|_{L1}$	1024	1024	512
L2 cache size	$\|\|L2\|\|$	4 MB	1 MB	512 KB
L2 cache line size	LS_{L2}	128 bytes	64 bytes	32 bytes
L2 lines	$\|L2\|_{L2}$	32,768	16,384	16,384
TLB entries	$\|TLB\|$	64	64	64
page size	$\|\|Pg\|\|$	32 KB	8 KB	4 KB
TLB size	$\|\|TLB\|\|$	2 MB	512 KB	256 KB
L1 miss latency	l_{L2}	24 ns = 6 cycles	30 ns = 6 cycles	42.2 ns = 19 cycles
L2 miss latency	l_{Mem}	400 ns = 100 cycles	195 ns = 39 cycles	93.3 ns = 42 cycles
TLB miss latency	l_{TLB}	228 ns = 57 cycles	270 ns = 54 cycles	11.1 ns = 5 cycles
seq. memory bandwidth	bw_{seq}	246 MB/s	193 MB/s	225 MB/s
par. memory bandwidth	bw_{par}	555 MB/s	244 MB/s	484 MB/s

Table 1: Calibrated Performance Characteristics

memory read of 52ns at a stride of 32 bytes, and 204ns at a stride of 128 bytes for the Origin2000, are considerably lower than their memory latencies (135ns resp. 424ns), whereas in the case of the Sun Ultra, the scan measurement at L2 line size almost coincides with the calibrated memory latency.

normal loop	multi-cursor	prefetch

for(**int** tot=i=0; i<N; i++) {
tot += buf[i];
}
	for(**int** tot_0 =tot_1 =i=0, C=N/2; i<C; i++) {
	tot_0+= buf[i];
	tot_1+= buf[i+C];
	} **int** tot = tot_0+ tot_1;
		for(**int** tot=i=0; i<N; i++) {
		#**prefetch** buf[i+32] **freq**=32
		tot += buf[i];
		}

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

Figure 7: Three ways to add a buffer of integers, and costs per addition on the Origin2000

The discrepancies are caused by *parallel memory access* that can occur on CPUs that feature both speculative execution and a non-blocking memory system. This allows a CPU to execute multiple memory load instructions in parallel, potentially enhancing memory bandwidth above the level of cache-line size divided by latency. Prerequisites for this technique are a bus system with excess transport capacity and a *non-blocking cache* system that allows multiple outstanding cache misses.

To answer the question what needs to be done by an application programmer to achieve these parallel memory loads, let us consider a simple programming loop that sums an array of integers. Figure 7 shows three implementations, where the leftmost column contains the standard approach that results in sequential memory loads into the buf[size] array. An R10000 processor can continue executing memory load instructions speculatively until four of them are stalled. In this loop, that will indeed happen if buf[i],buf[i+1],buf[i+2] and buf[i+3] are not in the (L2) cache. However, due to the fact that our loop accesses consecutive locations in the `buf` array, these four memory references request the same 128-byte L2 cache line. Consequently, no parallel memory access takes place. If we assume that this loop takes 2 cycles per iteration[3], we can calculate that 32 iterations cost 32*2 + 124 = 188 cycles (where 124 is the memory latency on our Origin2000); a total mean cost of 5.88 cycles per addition.

Parallel memory access can be enforced by having one loop that iterates two cursors through the buf[size] array (see the middle column of Figure 7). This causes 2 parallel 128 byte (=32 integer) L2 cache line fetches from memory per 32 iterations, for a total of 64 additions. On the R10000, the measured maximum memory bandwidth of the bus is 555MB/s, so fetching two 128-byte cache lines in parallel costs only 112 cycles (instead of 124 + 124). The mean cost per addition is hence 2 + 112/64 = 3.75 cycles.

It is important to note that parallel memory access is achieved only if the ability of the CPU to execute multiple instructions speculatively spans multiple memory references in the application code. In other words, the parallel effect disappears if there is too much CPU work between two memory fetches (more than 124 cycles on the R10000) or if the instructions are interdependent, causing a CPU stall before reaching the next memory reference. For database algorithms this means that random access operations like hashing will not profit from parallel memory access, as following a linked list (hash bucket chain) causes one iteration to depend on the previous; hence a memory miss will block execution. Only sequential algorithms with CPU processing costs less than the memory latency will profit, like in the simple scan experiment from Figure 4. This experiment reaches optimal parallel bandwidth when the stride is equal to this L2 cache line size. As each loop iteration then requests one subsequent cache line, modern CPUs will have multiple memory loads outstanding, executing them in parallel. Results are summarized at the bottom of Table 1, showing the parallel effect to be especially strong on the Origin2000 and the PentiumIII. In other words, if the memory access pattern is *not* sequential (like in equi-join), the memory access penalty paid on these systems is actually much higher than suggested by Figure 4, but determined by the latencies from Table 1.

[3]As each iteration of our loop consists of a memory load (buf[i]), an integer addition (of "total" with this value), an integer increment (of i), a comparison, and a branch, the R10000 manual suggests a total cost of minimally 6 cycles. However, due to the speculative execution in the R10000 processor, this is reduced to 2 cycles on the average.

2.5 Prefetched Memory Access

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

In order to provide the opportunity to still enforce memory prefetching in such situations, the MipsPRO compiler for the R10000 systems of Silicon Graphics allows passing of explicit prefetching hints by use of pragma's, as depicted in the rightmost column of Figure 7. This pragma tells the compiler to request the next cache line once in every 32 iterations. Such a prefetch-frequency is generated by the compiler by applying loop unrolling (it unrolls the loop 32 times and inserts one prefetch instruction). By hiding the memory prefetch behind 64 cycles of work, the mean cost per addition in this routine is reduced to $2 + ((124\text{-}64)/32) = 3.88$ cycles. Optimal performance is achieved in this case when prefetching two cache lines ahead every 32 iterations (#**prefetch** buf[i+64] **freq**=32). The 124 cycles of latency are then totally hidden behind 128 cycles of CPU work, and a new cache line is requested every 64 cycles. This setting effectively combines prefetching with parallel memory access (two cache lines in 128 cycles instead of 248), and reduces the mean cost per addition to the minimum 2 cycles; three times faster than the simple approach.

2.6 Future Hardware Features

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

A radical way around the high latencies mandated by off-CPU DRAM systems is presented in the proposal to integrate DRAM and CPU in a single chip called IRAM (Intelligent RAM) [PAC+97]. Powerful computer systems could then be built using many such chips. Finding a good model for programming such a highly parallel systems seems one of the biggest challenges of this approach. Another interesting proposal worth mentioning here has been "smarter memory" [MKW+98], which would allow the programmer to give a "cache-hint" by specifying the access pattern that is going to be used on a memory region in advance. This way, the programmer is no longer obliged to organize

his data structures around the size of a cache line. Instead, the cache adapts its behavior to the needs of the application. Such a configurable system is in some sense a protocol-driven bus system, so Rambus is a step in this direction. However, both configurable memory access and IRAM have not yet been implemented in custom hardware, let alone in OS and compiler tools that would be needed to program them usefully.

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

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

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

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

3. 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 8. 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[4] 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.

3.1 Radix-Cluster Algorithm

The *radix-cluster* algorithm divides a relation into H clusters using multiple passes (see Figure 9). 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 $H_1 * H_2$ clusters in total, etc.. Note that with $P = 1$, radix-cluster behaves like the straightforward algorithm.

[4] 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.

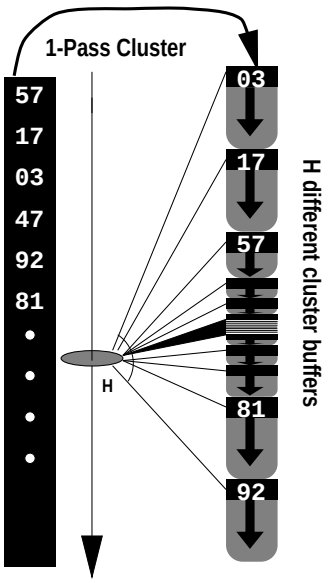


Figure 8: Straightforward clustering algorithm

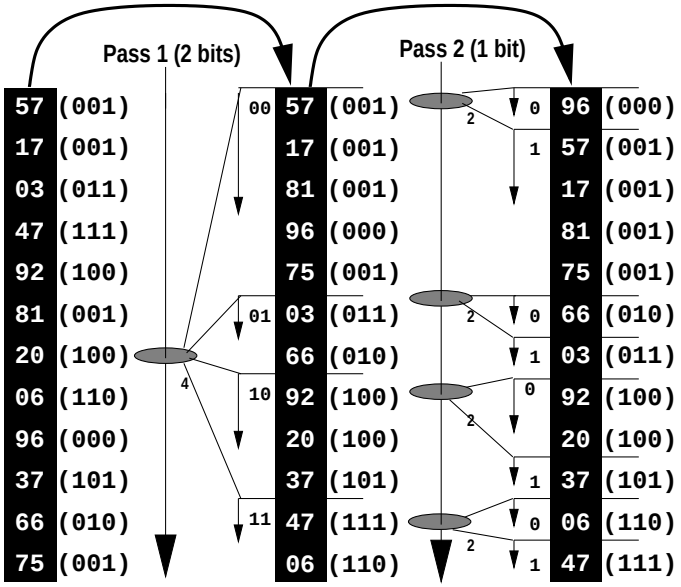


Figure 9: 2-pass/3-bit Radix Cluster (lower bits indicated between parentheses)

For ease of presentation, we did not use a hash function in the table of integer values displayed in Figure 9. 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 $C/2^B$ tuples (with C 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 *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.

3.2 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 parameter settings are optimal for radix-clustering a relation on B radix-bits, we turn our attention to the performance of the join

category	MIPS R10k	Sun UltraSPARC	Intel PentiumII
memory access	L1_data_misses * 6 cy L2_data_misses * 100 cy TLB_misses * 57 cy L1_inst_misses * 6 cy L2_inst_misses * 100 cy	DC_misses[5] * 6 cy EC_misses[6] * 39 cy M_{TLB} * 54 cy stall_IC_miss	cycles_while_DCU_miss_outstanding M_{TLB} * 5 cy cycles_instruction_fetch_pipe_is_stalled ITLB_misses * 32 cy[7]
CPU stalls	branches_mispredicted * 4 cy	stall_mispred stall_fpdep	taken_mispredicted_branches_retired * 17 cy[7] cycles_instruction_length_decoder_is_stalled cycles_during_resource_related_stalls[8] cycles_or_events_for_partial_stalls
integer divisions	$C * 2 * 35$ cy	$C * 2 * 60$ cy	cycles_divider_is_busy ($= C * 2 * 35$ cy)

Table 2: Hardware Counters used for Execution Time Breakdown

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.

Experimental Setup In our experiments, we use binary relations (BATs) of 8 bytes wide tuples and varying cardinalities (C), 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 2.3, an SGI Origin2000, a Sun Ultra, and an Intel PC (cf. Table 1).

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.

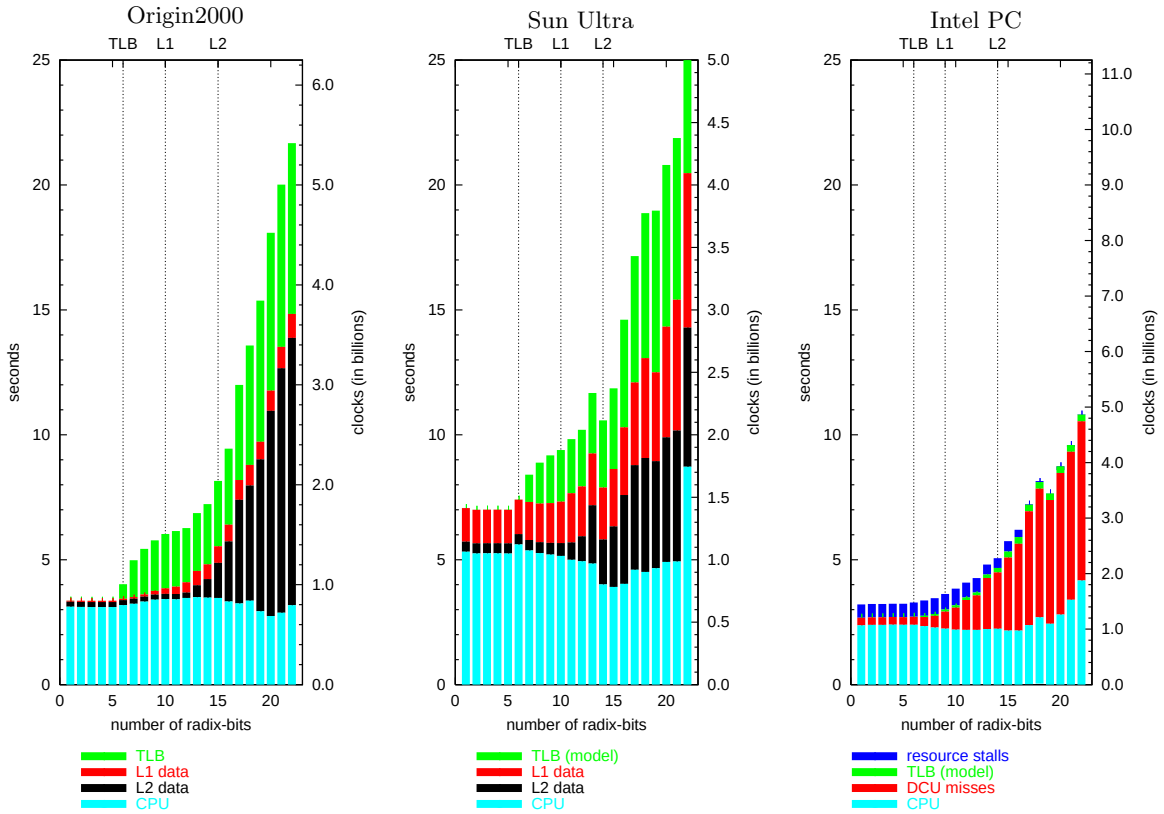
CPU stalls Beyond memory access, there are other events that make the CPU stall, like branch mispredictions or other so-called resource related stalls.

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 three architectures we investigate, provide different hardware counters [BZ98] that enable us to measure each of these cost factors accurately. Table 2 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 2.3). 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.

[5] DC_misses = DC_read - DC_read_hit + DC_write - DC_write_hit.
[6] EC_misses = EC_ref - EC_hit.
[7] Taken from [ADHW99].
[8] This counter originally includes "cycles_while_DCU_miss_outstanding". We use only the remaining part after subtracting "cycles_while_DCU_miss_outstanding", here.



(Vertical grid lines indicate, where the number of clusters created equals the number of TLB entries, L1, or L2 cache lines, respectively.)

Figure 10: Execution Time Breakdown of Radix-Cluster using one pass (Cardinality = 8M)

In our experiments, we found that in our algorithms, branch mispredictions and instruction cache misses do not play a role on either architecture. Thus, for simplicity of presentation, we omit them in our evaluation.

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 10 shows an execution time breakdown for 1-pass radix-cluster ($C = 8$M) on each architecture. The pure CPU costs are nearly constant across all numbers of radix-bits, taking about 3 seconds on the Origin, 2.5 seconds on the PC, and a about 5.5 seconds on the Sun. 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 PC however, the impact of TLB misses is hardly visible due to its very low TLB miss penalty. 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 PC, "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 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 PC, as the resource related

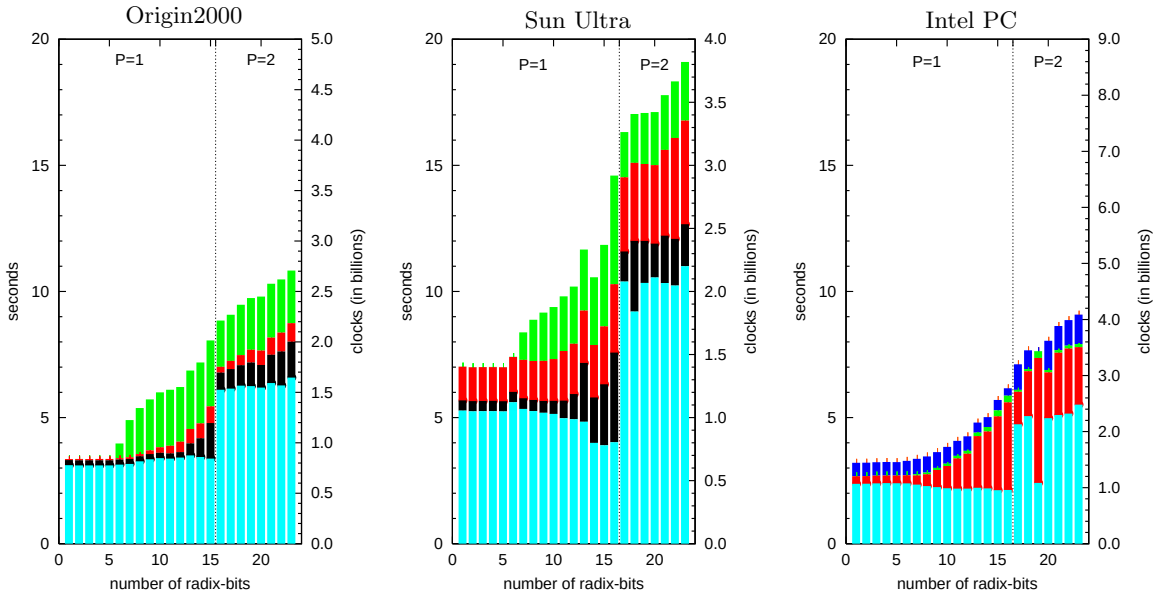


Figure 11: Execution Time Breakdown of Radix-Cluster using optimal number of passes ($C = 8$M)

```
#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;          ||  int idx = HASH(cur→v2)&M;
    memcpy(dst[idx], cur, sizeof(bun));      ||  *dst[idx] = *cur;
    if (++dst[idx]≥dst_end[idx]) REALLOC(dst[idx],dst_end[idx]);
  }
}
```

Figure 12: C language radix-cluster with annotated CPU optimizations (right)

stalls partly take over their part.

Figure 11 depicts the breakdown for radix-cluster using the optimal number of passes. 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 way to improve this situation is to reduce the CPU costs. Figure 12 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 $B = P * D$ bits is done by making subsequent calls to this function for pass $p = 1$ through $p = P$ with parameters $D_p = D$ and $R_p = (p - 1) * D$, 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

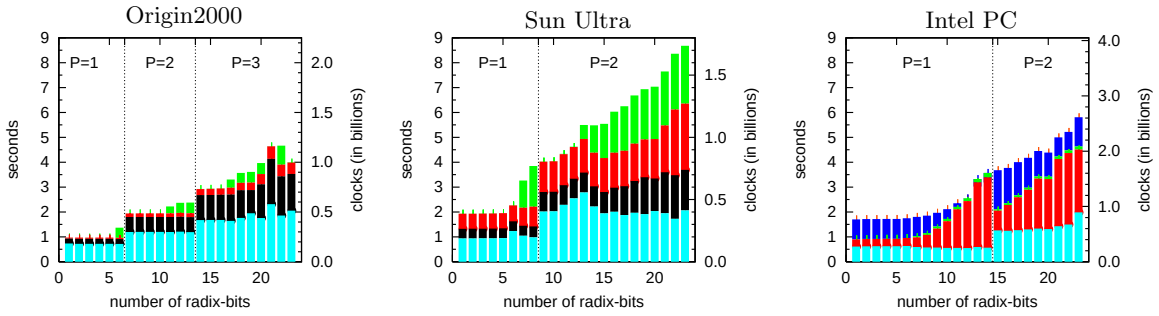


Figure 13: Execution Time Breakdown of optimized Radix-Cluster using optimal number of passes ($C = 8$M)

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 13 shows the execution time breakdown for the optimized multi-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 PC, 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 PC efficiently.

With this optimization, 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-bits, thus avoiding TLB thrashing nearly completely. On the PC, 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 estimate the performance of radix-cluster, and especially to predict the number of passes to be used for a certain number of radix-bits, we now provide an accurate cost model for radix-cluster. The cost model takes the number of passes, the number of radix-bits, and the cardinality as input and estimates the number of memory related events, i.e., L1 cache misses, L2 cache misses, and TLB misses. The overall execution time is calculated by scoring the events with their penalties and adding the pure CPU costs.

$$T_c(P,B,C) = P * \left(C * w_c + M_{L1,c}\left(\frac{B}{P},C\right) * l_{L2} + M_{L2,c}\left(\frac{B}{P},C\right) * l_{Mem} + M_{TLB,c}\left(\frac{B}{P},C\right) * l_{TLB} \right)$$

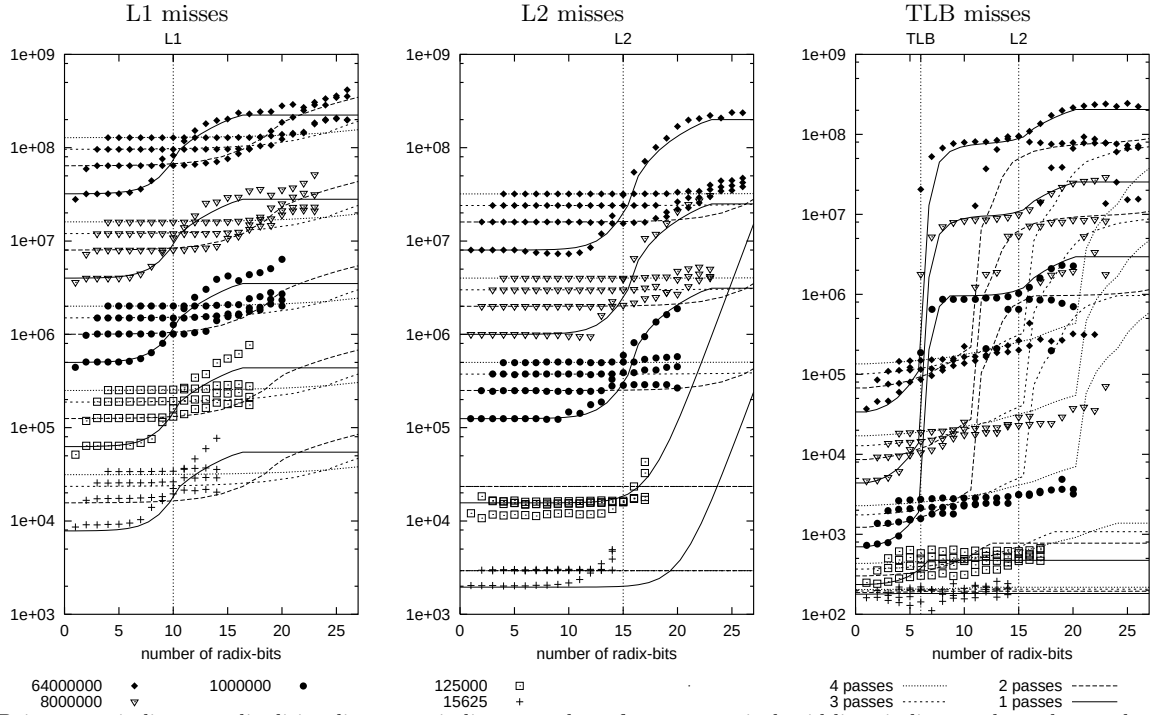
with

$$M_{Li,c}(B_p,C) = 2 * |Re|_{Li} + \begin{cases} C * \frac{H_p}{|Li|_{Li}} * \min\left\{1, \frac{|Re|_{Li}}{|Li|_{Li}}\right\}, & \text{if } \min\{H_p, |Re|_{Li}\} \leq |Li|_{Li} \\ C * \min\left\{3, 1 + \log\left(\frac{H_p}{|Li|_{Li}}\right)\right\}, & \text{else} \end{cases}$$

and

$$M_{TLB,c}(B_p,C) = 2 * |Re|_{Pg} + \begin{cases} |Re|_{Pg} * \left(\frac{\min\{H_p, |Re|_{Pg}\}}{|TLB|}\right), & \text{if } \min\{H_p, |Re|_{Pg}\} \leq |TLB| \\ C * \left(1 - \frac{|TLB|}{\min\{H_p, |Re|_{Pg}\}}\right), & \text{else} \end{cases}$$

$$(\text{if } |Re|_{Pg} > |TLB|) \quad + \begin{cases} C * \left(\frac{H_p}{|L2|_{L2}}\right), & \text{if } H_p \leq |L2|_{L2} \\ C * \min\left\{2, 1 + \log\left(\frac{H_p}{|L2|_{L2}}\right)\right\}, & \text{else} \end{cases}$$



(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.)

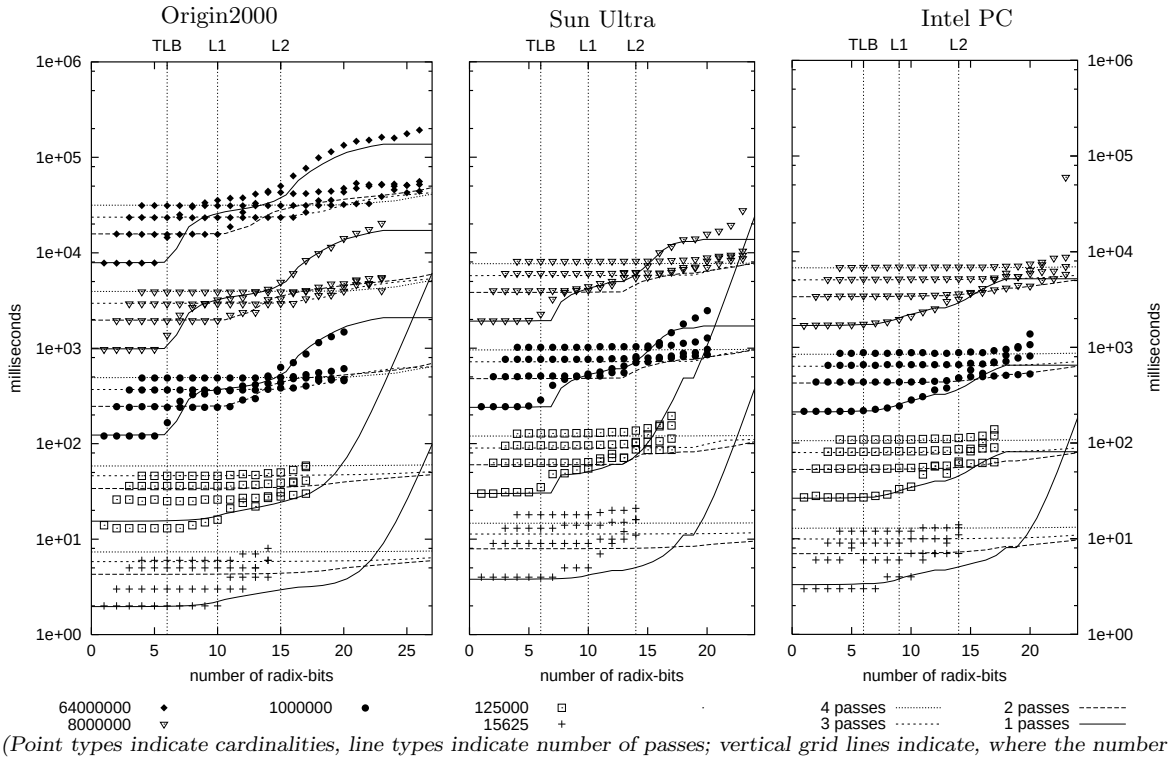
Figure 14: Measured (points) and Modeled (lines) Events of Radix-Cluster (Origin2000)

$|Re|_{Li}$ and $|Cl|_{Li}$ denote the number of cache lines per relation and cluster, respectively, $|Re|_{Pg}$ the number of pages per relation, $|Li|_{Li}$ 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 $w_c = 100$ns on the Origin2000, $w_c = 200$ns on the Sun, and $w_c = 180$ns on the PC (including resource stalls).

The first term of $M_{Li,c}$ 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 distinct Li lines accessed concurrently (i.e., x=min $\{H_p, |Re|_{Li}\}$)[9] either approaches the number of available Li lines ($x \leq |Li|_{Li}$) or even exceeds this. First, the probability that the requested cluster is not in the cache—due to address conflicts—increases until $H_p = |Li|_{Li}$. 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 $M_{TLB,c}$ are made up analogously. Additionally, using a similar schema as $M_{Li,c}$, the third term models—for relations that contain more pages than there are TLB entries—the additional TLB misses that occur when the number of clusters either approaches the number of available L2 lines ($H_p \leq |L2|_{L2}$) or even exceeds this.

Figure 14 compares our model (lines) with the experimental results (points) on the Origin2000 for different cardinalities. The model proves to be very accurate for the number of cache misses (both, L1 and L2) and TLB misses. The predicted elapsed time is also reasonably accurate on all architectures

[9] Using min $\{H_p, |Re|_{Li}\}$ instead of simply H_p takes into account, that smaller relations may completely fit in Li, i.e., with $H_p > |Li|_{Li} > |Re|_{Li}$, several (tiny) clusters share one cache line.



(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.)

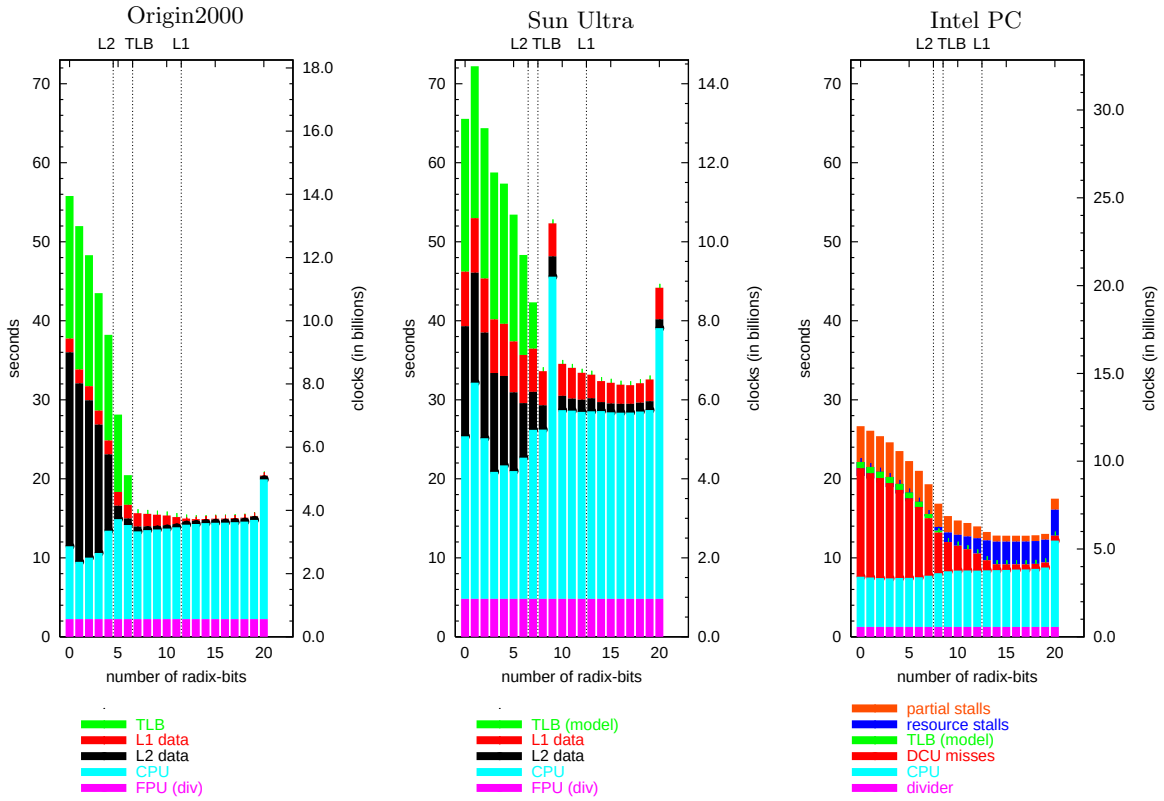
Figure 15: Measured (points) and Modeled (lines) Performance of Radix-Cluster

(cf. Figure 15). The plots clearly reflect the increase in cache and TLB misses and their impact on the execution time whenever the number of clusters per pass exceeds the respective limits.

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

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 16 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 PC 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.



(Vertical grid lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)

Figure 16: Execution Time Breakdown of Partitioned Hash-Join (Cardinality = 8M)

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 17 depicts the original implementation of our hash-join routine and the optimizations we applied.

Figure 18 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, and by factor 3 on the PC. The expensive divisions have vanished completely. Additionally, the dependency stalls on the PC have disappeared, but the functional unit stalls remain almost unchanged, now taking about half of 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.

As for the radix-cluster, we also provide a cost model for the partitioned hash-join. The model takes the number of radix-bits, the cardinality[10], and the (average) join hit rate as input.

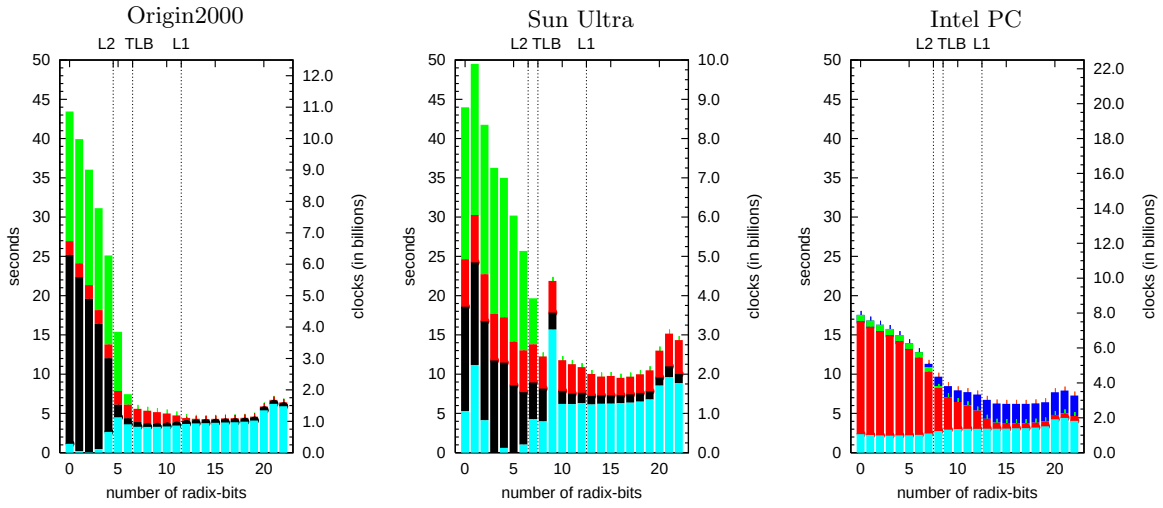
$$T_h(B,C,r) = C * w_h + M_{L1,h}(B,C,r) * l_{L2} + M_{L2,h}(B,C,r) * l_{Mem} + M_{TLB,h}(B,C,r) * l_{TLB}$$

with

[10]For simplicity of presentation, we assume the cardinalities of both input relations to be equal.

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

Figure 17: C language hash-join with annotated CPU optimizations (right)

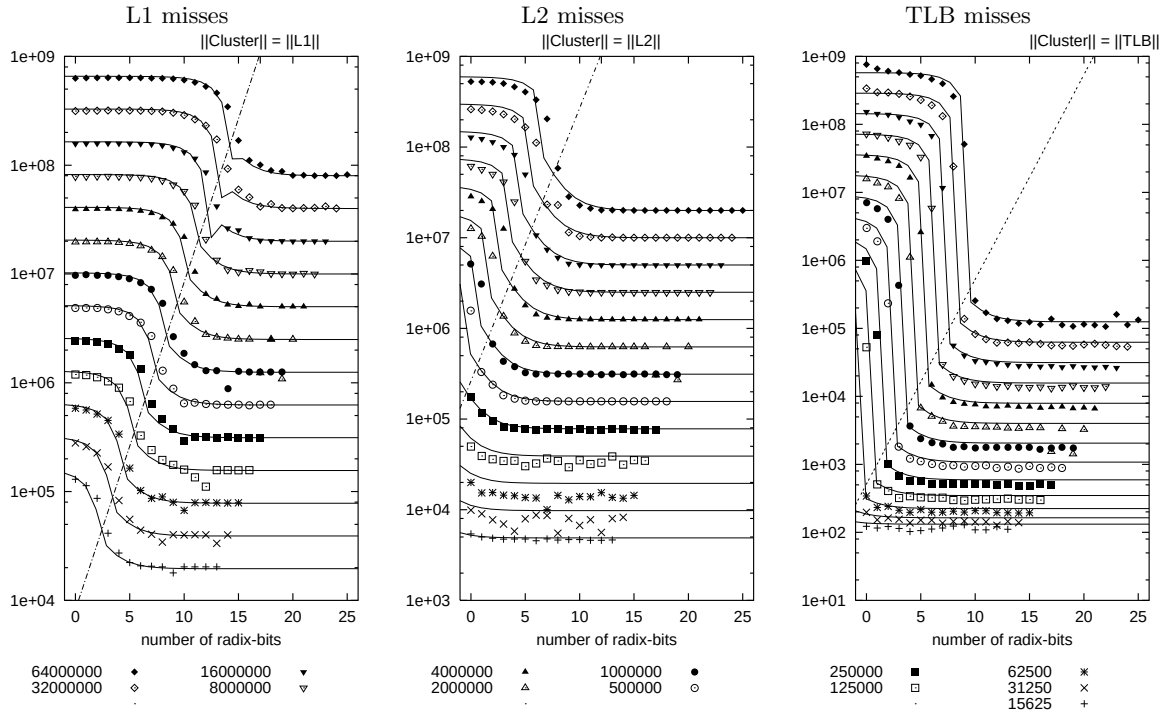


(Vertical grid lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)

Figure 18: Execution Time Breakdown of optimized Partitioned Hash-Join ($C = 8$M)

$$M_{Li,h}(B,C,r) = (2+r) * |Re|_{Li} + \begin{cases} C * \dfrac{||Cl||}{||Li||}, & \text{if } ||Cl|| \leq ||Li|| \\ C * (4+2r) * \left(1 - \dfrac{||Li||}{||Cl||}\right), & \text{else} \end{cases}$$

and



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

Figure 19: Measured (points) and Modeled (lines) Events of Partitioned Hash-Join (Origin2000)

$$M_{TLB,h}(B,C,r) = (2+r) * |Re|_{Pg} + \begin{cases} C * \dfrac{||Cl||}{||TLB||}, & \text{if } ||Cl|| \leq ||TLB|| \\ C * (4+2r) * \left(1 - \dfrac{||TLB||}{||Cl||}\right), & \text{else} \end{cases}$$

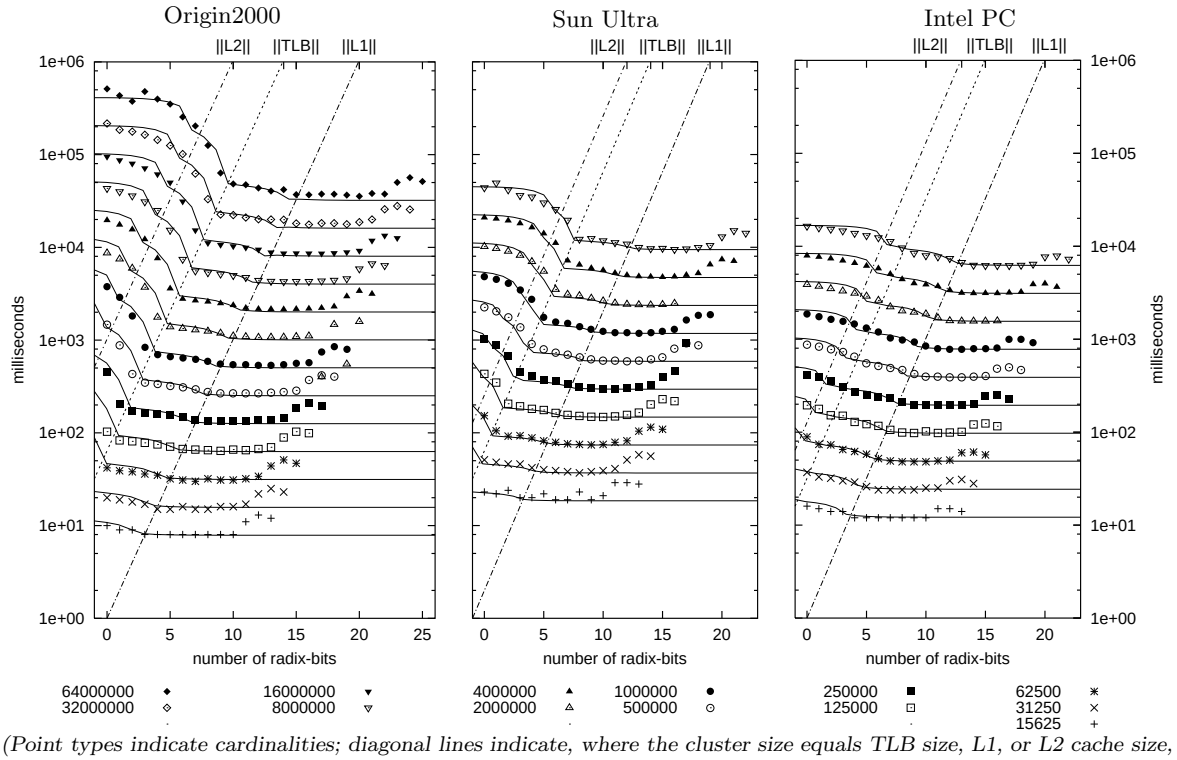
$|Re|_{Li}$, $|Re|_{Pg}$, and $|TLB|$ are as above. $||Cl||$, $||Li||$, and $||TLB||$ denote (in byte) the cluster size, the sizes of both caches ($i \in \{1,2\}$), and the memory range covered by $|TLB|$ pages, respectively.

w_h represents the pure CPU costs per tuple for building the hash-table, doing the hash lookup and creating the result. We calibrated $w_h = 600$ns on the Origin2000, $w_h = 1100$ns on the Sun, and $w_h = 711$ns on the PC (including resource stalls).

The first term of $M_{Li,h}$ equals the minimal number of Li misses for fetching both operands and storing the result. The second 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 get significantly larger than Li, each memory access yields a cache miss due to cache thrashing: 4 memory accesses per tuple for accessing the outer relation and the bucket array during hash build and hash probe, and 2 memory accesses per join hit to access the inner relation and the chain-lists. The number of TLB misses is modeled analogously.

Figures 19 and 20 confirm 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.

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-



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

Figure 20: Measured (points) and Modeled (lines) Performance of Partitioned Hash-Join

and join-performance, we determine the optimum number of B for relation cardinality.

It turns out that there are three possible strategies, which correspond to the diagonals in Figure 20:

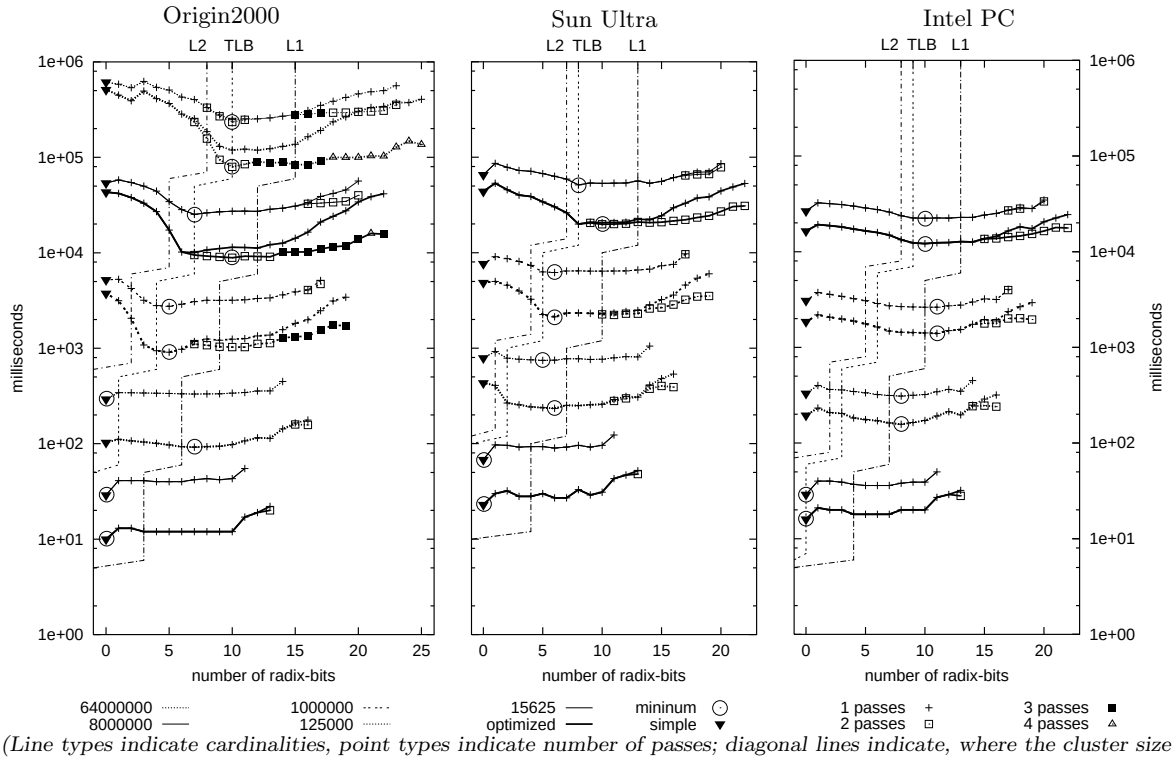
phash L2 partitioned hash-join on $B = log_2(C * 12/||L2||)$ clustered bits, so the inner relation plus hash-table fits the L2 cache. This strategy was used in the work of Shatdal et al. [SKN94] in their partitioned hash-join experiments.

phash TLB partitioned hash-join on $B = log_2(C * 12/||TLB||)$ 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 partitioned hash-join on $B = log_2(C * 12/||L1||)$ 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 thrashing on this many clusters).

Figure 21 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 PC, 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.

Further, these results 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



(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.)

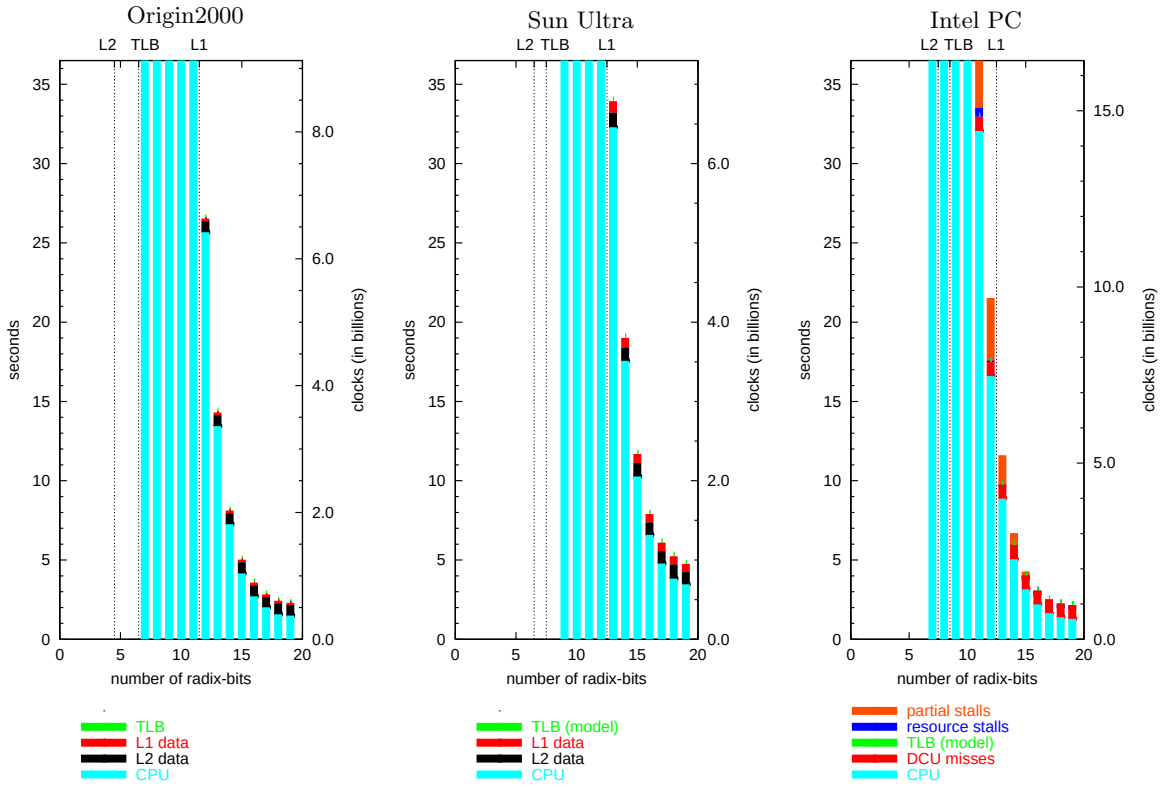
Figure 21: Overall Performance of Partitioned Hash-Join: non-optimized (thin lines) vs. optimized (thick lines) implementation

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 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 results are similar to those obtained on the Origin2000, although the absolute gains are somewhat smaller due to the fact that the Ultra CPU is so slow that trading memory for CPU less beneficial on this platform.

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 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.



(Vertical grid lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)

Figure 22: Execution Time Breakdown of Radix-Join (Cardinality = 1M)

4. Radix-Join

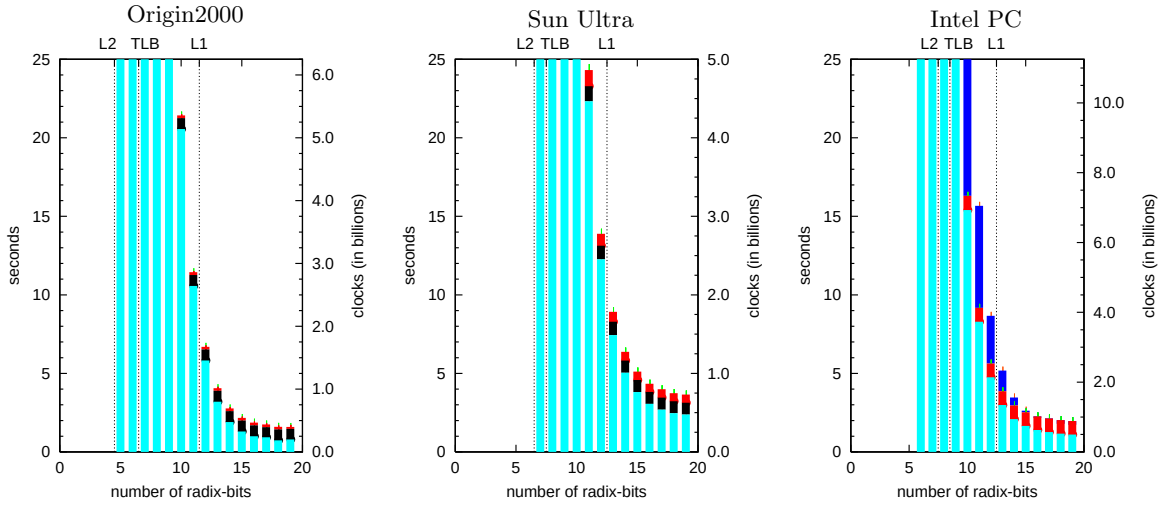
In this section, we present our *radix-join* algorithm as an altenative for the partitioned hash-join. Radix-join makes use of the very fine clustering capabilities of radix-cluster. If the number of clusters H is high, the radix-clustering has brought the potentially matching tuples near to each other. 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 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.

Isolated Radix-Join Performance Figure 22 shows the execution time breakdown for our radix-join algorithm (C = 1M). On all three architectures, 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. On the PC, partial resource stalls become clearly visible, as soon as the cluster size reaches and exceeds L1 cache size.

Like with radix-cluster and partitioned hash-join, we optimize our radix-join implementation (cf. Figure 23) in order to reduce the CPU costs. Figure 24 depicts the results for the optimized radix-join. For very small clusters, the optimizations yield an improvement of factor 1.1 (on the PC) to 1.4 (on the Origin2000). For larger clusters, we observe the same improvements as with partitioned hash-join: factor 4 on the RISC architectures and factor 3 on the PC. We also note that the dependency stalls

```
nested_loop(bun *dst, bun *end                                  /* start and end of result buffer */
    bun *outer, bun *outer_end, bun *inner, bun* inner_end,     /* inner and outer relations */
){
    for(bun *outer_cur=outer; outer_cur < outer_end; outer_cur++) {
        for(bun *inner_cur=inner; inner_cur < inner_end; outer_cur++) {
        if ((*compareFcn)(outer_cur→v2,inner_cur→v2)==0) {  ||  if ((outer_cur→v2 == inner_cur→v2)) {
            memcpy(&dst→v1,&outer_cur→v1, sizeof(int));     ||    dst→v1 = outer_cur→v1;
            memcpy(&dst→v2,&inner_cur→v1, sizeof(int));     ||    dst→v2 = inner_cur→v1;
            if (++dst≥end) REALLOC(dst,end);
        }
    }
}
```

Figure 23: C language nested-loop with annotated CPU optimizations (right)



(Vertical grid lines indicate, where the cluster size equals TLB size, L1, or L2 cache size, respectively.)

Figure 24: Execution Time Breakdown of optimized Radix-Join ($C = 1$M)

(partial stalls) on the PC are completely replaced by functional unit stalls (resource stalls).

The following model calculates the total execution costs for a radix-join, depending on the number of radix-bits, the cardinality and the (average) join hit rate.

$$T_r(B,C,r) = C * \left\lceil \frac{C}{H} \right\rceil * w_r + C * r * w'_r + M_{L1,r}(B,C,r) * l_{L2} + M_{L2,r}(B,C,r) * l_{Mem} + M_{TLB,r}(B,C,r) * l_{TLB}$$

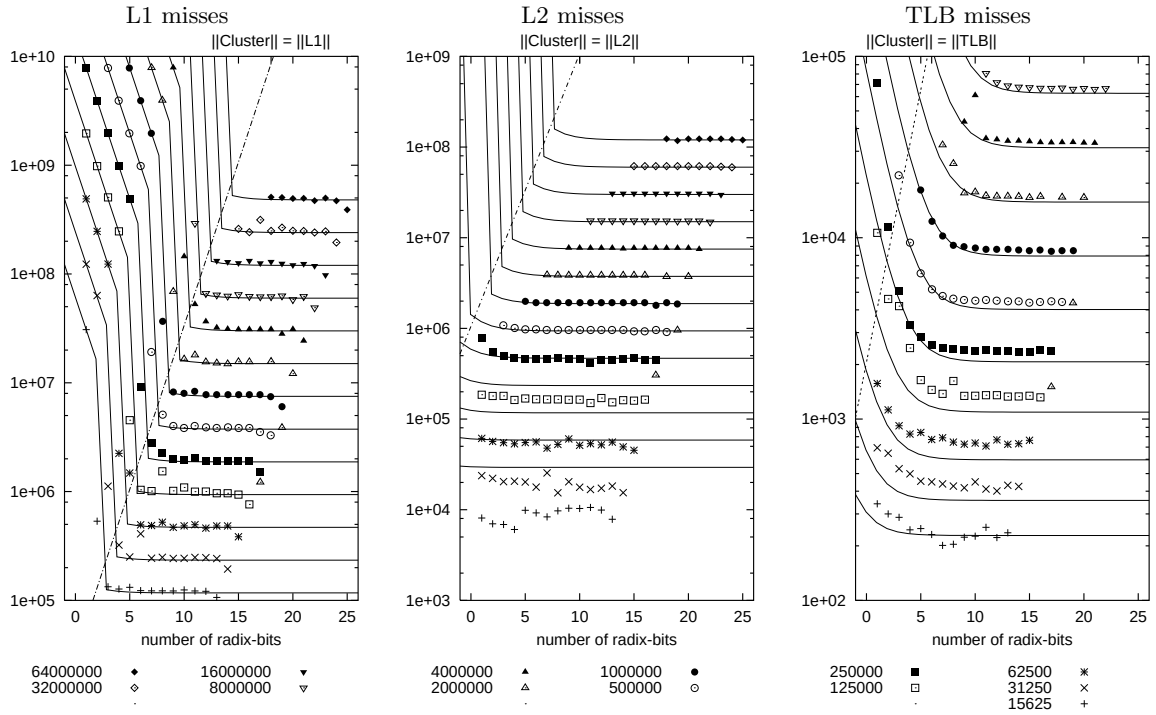
with

$$M_{Li,r}(B,C,r) = (2+r) * |Re|_{Li} + \begin{cases} C * \frac{|Cl|_{Li}}{|Li|_{Li}}, & \text{if } |Cl|_{Li} \leq |Li|_{Li} \\ C * |Cl|_{Li}, & \text{else} \end{cases}$$

and

$$M_{TLB,r}(B,C,r) = (2+r) * |Re|_{Pg} + C * \frac{||Cl||}{||TLB||}$$

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



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

Figure 25: Measured (points) and Modeled (lines) Events of Radix-Join (Origin2000)

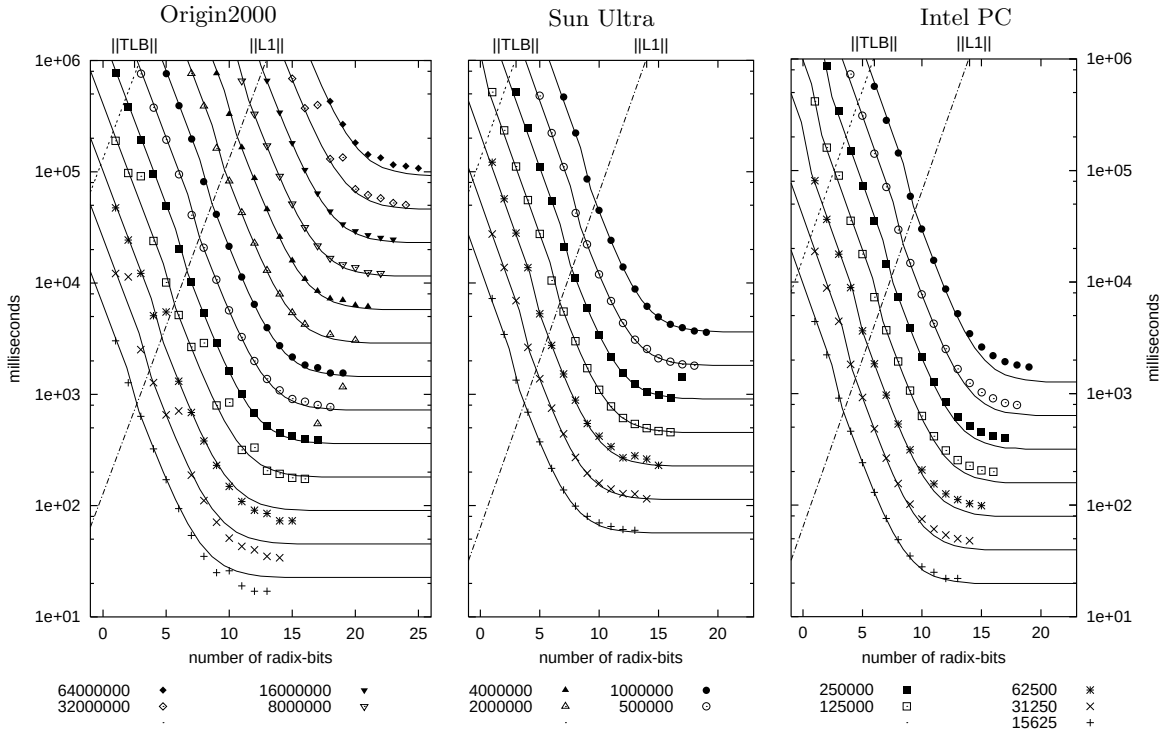
second term represents the costs for creating the result with w'_r denoting the costs per tuple. We calibrated $w_r = 20$ns and $w'_r = 320$ns on the Origin2000, $w_r = 40$ns and $w'_r = 1000$ns on the Sun, and $w_r = 20$ns and $w'_r = 250$ns on the PC.

The left term of $M_{Li,r}$ 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 number of Li lines per cluster either approaches the number of available Li lines ($|Cl|_{Li} \leq |Li|_{Li}$) or even exceeds this ($|Cl|_{Li} > |Li|_{Li}$). 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. $M_{TLB,r}$ is made up analogously.

Figures 25 and 26 confirm 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.

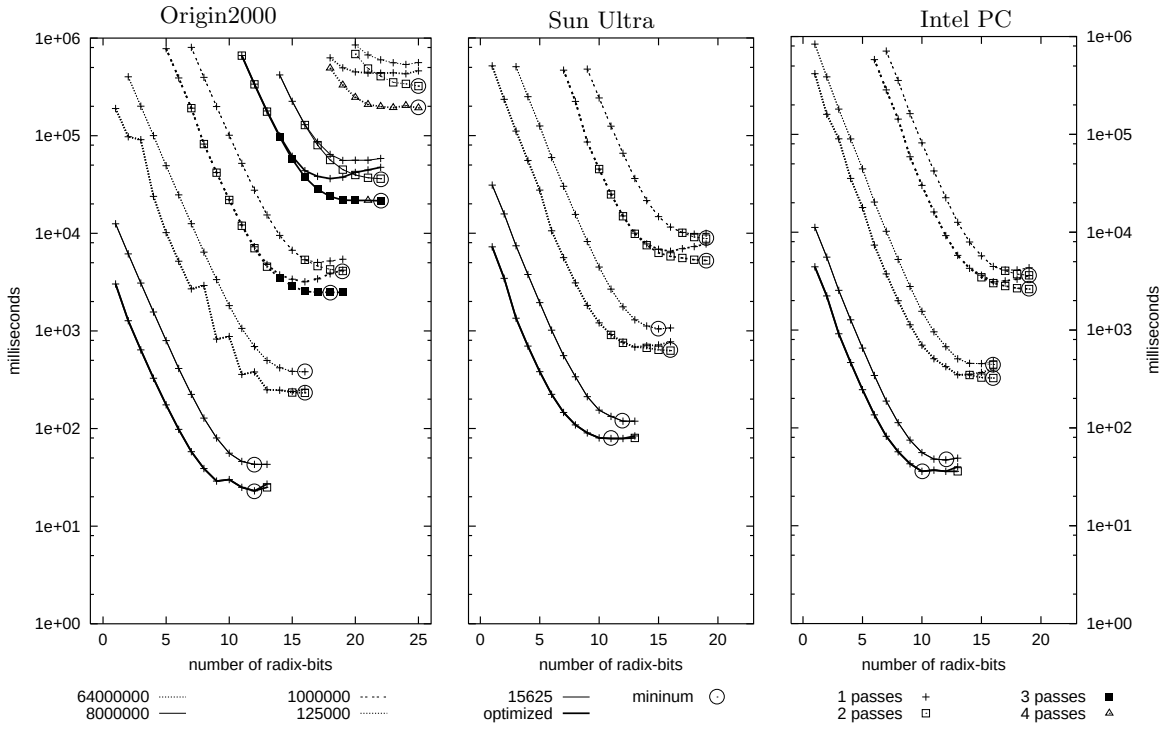
Overall Radix-Join Performance Figure 27 shows the overall performance of radix-join, i.e., including radix-cluster of both input relations. For larger clusters, the performance is dominated by the high CPU costs of radix-join. Only very small clusters with up to 8 tuples are reasonable, requiring multi-pass radix-clustering in most cases.

Partitioned Hash-Join vs. Radix-Join Finally, Figures 28 and 29 compare the overall performance of partitioned hash-join and radix-join. With the original non-optimized implementation, the optimal performance of partitioned hash-join (circled points on thick lines) is only slightly better than the optimal radix-join performance (circled points on thin lines). With code optimizations applied, the difference becomes more significant.



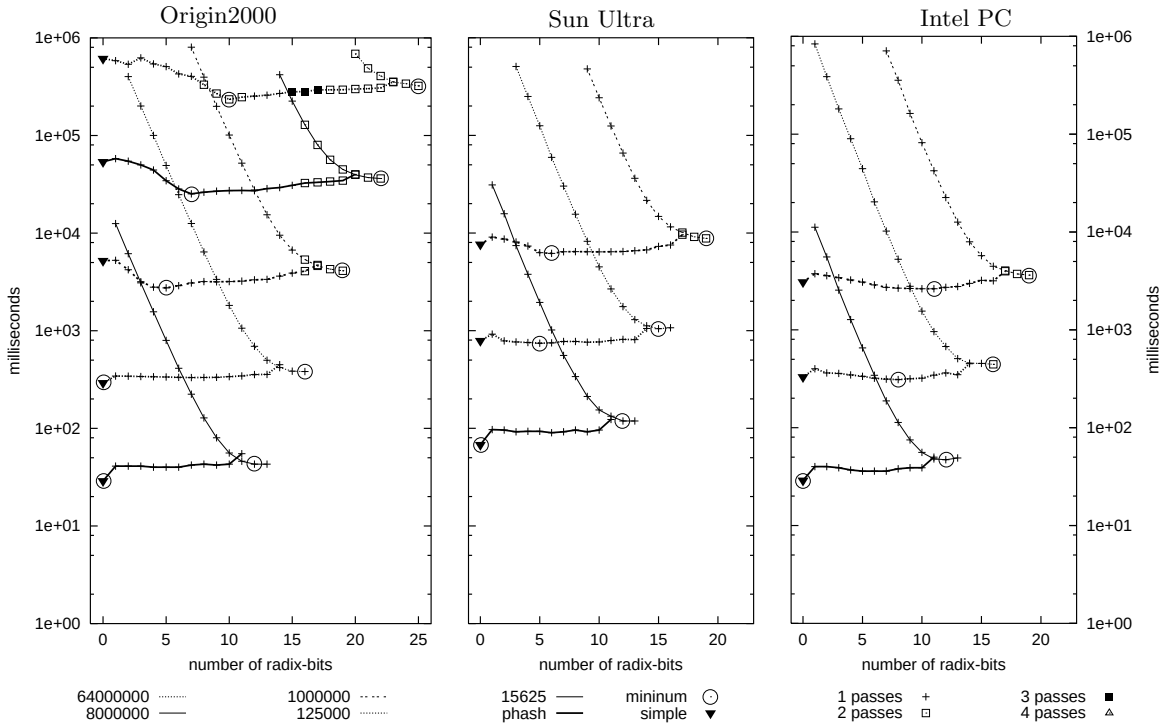
(Point types indicate cardinalities; diagonal lines indicate, where the cluster size equals TLB size or L1 cache size.)

Figure 26: Measured (points) and Modeled (lines) Performance of Radix-Join



(Line types indicate cardinalities, point types indicate number of passes.)

Figure 27: Overall Performance of Radix-Join: non-optimized (thin lines) vs. optimized (thick lines)



(Line types indicate cardinalities, point types indicate number of passes.)

Figure 28: Partitioned Hash-Join (thick lines) vs. Radix-Join (thin lines) (non-optimized)

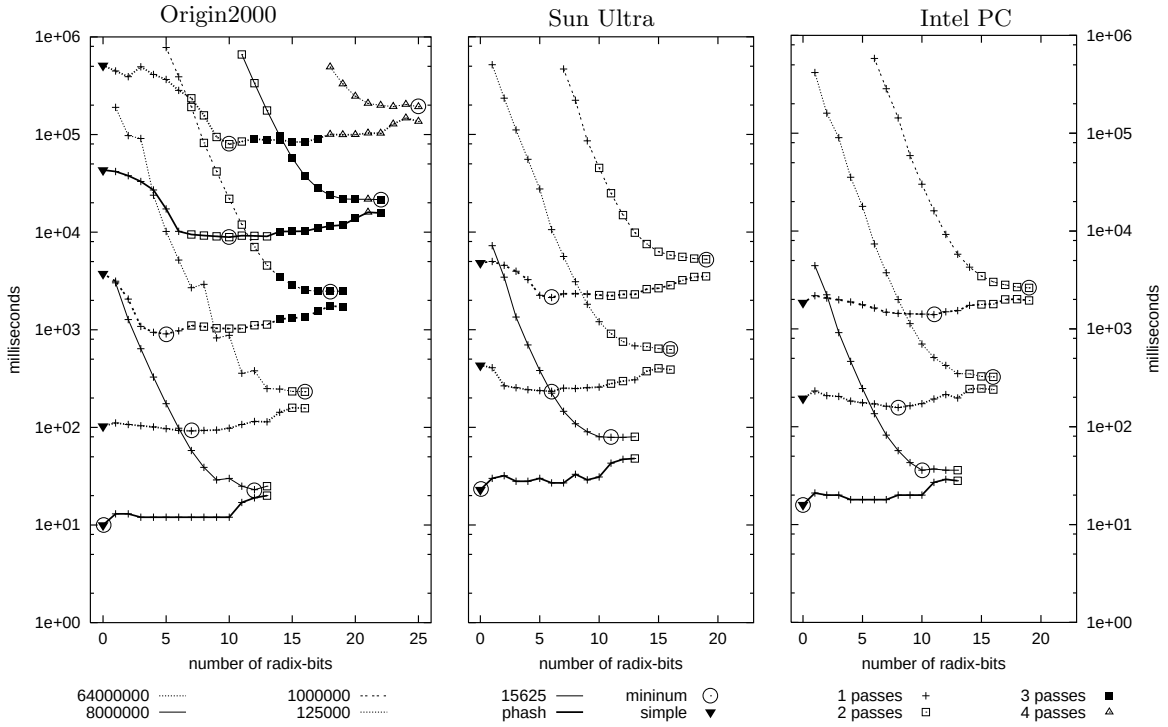


Figure 29: Partitioned Hash-Join (thick lines) vs. Radix-Join (thin lines) (optimized)

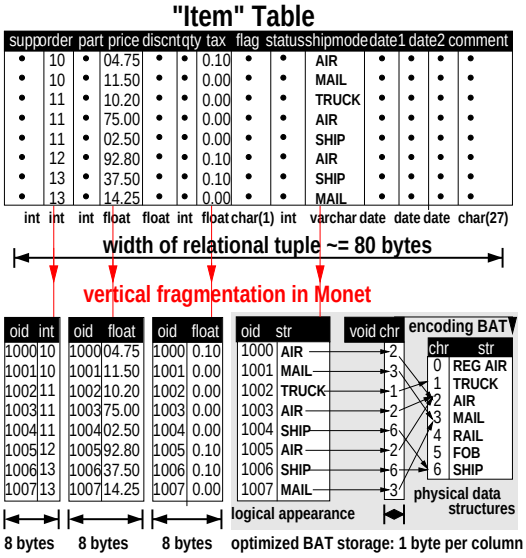


Figure 30: Vertical Decomposition in BATs

5. 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 2.6, 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.

5.1 Implications for Data Structures

In terms of data structures for query processing, we already noted from the simple scan experiment in Figure 4 that *full vertical table fragmentation* optimizes column-wise memory access to table data. This is particularly beneficial if the table is accessed in a sequential scan that reads a minority of all columns. Such table scans very often occur in both OLAP and Data Mining workloads. When record-oriented (i.e., non-fragmented) physical storage is used, such an access leads to data of the non-used columns being loaded into the cache lines, wasting memory bandwidth. In case of a vertically fragmented table, the table scan just needs to load the vertical fragments pertaining to the columns of interest. Reading those vertical fragments sequentially achieves a 100% hit rate on all cache levels, exploiting optimal bandwidth on any hardware, including parallel memory access.

There are various ways to incorporate vertical fragmentation in database technology. In Monet, which we designed for OLAP and Data Mining workloads, vertical fragmentation is the basic building block of all physical storage, as Monet fully fragments all relations into Binary Association Tables (BATs) (see Figure 30). Flat binary tables are a simple set-oriented physical representation, that is not tied to a particular logical data model, yet is sufficiently powerful to represent e.g. join indices [Val87]. Monet has successfully been used to store and query relational, object-oriented and network data structures, using this very simple data model and a small kernel of algebraic operations on it [BK99]. In Monet, we applied two additional optimizations that further reduce the per-tuple memory requirements in its BATs:

- *virtual-OIDs.* Generally, when decomposing a relational table, we get an identical system-generated column of OIDs in all decomposition BATs, which is *dense and ascending* (e.g., 1000,

1001, ... , 1007). In such BATs, Monet computes the OID-values on-the-fly when they are accessed using positional lookup of the BUN, and avoids allocating the 4-byte OID field. This is called a "virtual-OID" or VOID column. Apart from reducing memory requirements by half, this optimization is also beneficial when joins or semi-joins are performed on OID columns.[11] When one of the join columns is VOID, Monet uses positional lookup instead of e.g., hash-lookup; effectively eliminating all join costs.

- *byte-encodings.* Database columns often have a low domain cardinality. For such columns, Monet uses fixed-size encodings in 1- or 2-byte integer values. This simple technique was chosen because it does not require decoding effort when the values are used (e.g., a selection on a string "MAIL" can be re-mapped to a selection on a byte with value 3). A more complex scheme (e.g., using bit-compression) might yield even more memory savings, but the decoding-step required whenever values are accessed can quickly become counterproductive due to extra CPU effort. Even if decoding would just cost a handful of cycles per tuple, this would more than double the amount of CPU effort in simple database operations, like a simple aggregation from Section 2.2, which takes just 2 cycles of CPU work per tuple.

Figure 30 shows that when applying both techniques, the storage needed for 1 BUN in the "ship-mode" column is reduced from 8 bytes to just one. Reducing the stride from 8 to 1 byte significantly enhances performance in the scan experiment from Figure 4, eliminating all memory access costs.

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

5.2 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 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.

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

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 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. This was later replaced by a simple bit-wise transformation using bit-wise shifts, AND and XOR. 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.

5.3 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 quicksort on the partitions), and might well change the tradeoffs for other well-known main-memory algorithms (e.g., radix-sort has a highly cachable memory access pattern and is likely to outperform quicksort).

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.

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). Algorithms that are tuned to run well on one level of the memory, also exhibit good performance on the lower levels.

6. 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, and PC 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.

References

[ACM+98] D. August, D. Connors, S. Mahlke, J. Sias, K. Crozier, B. Cheng, P. Eaton, Q. Olaniran, and W. Hwu. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. In *Proc. of the Int'l Symp. on Computer Architecture*, pages 227–237, June 1998.

[ADHW99] A. G. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where does time go? In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 266–277, Edinburgh, Scotland, UK, September 1999.

[AP92] A. Analyti and S. Pramanik. Fast Search in Main Memory Databases. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 215–224, San Diego, CA, USA, June 1992.

[AvdBF+92] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. Kersten, and A. N. Wilschut. PRISMA/DB: A Parallel Main Memory Relational DBMS. *IEEE Trans. on Knowledge and Data Eng.*, 4(6):541–554, December 1992.

[Bat79] D. S. Batory. On Searching Transposed Files. *ACM Trans. on Database Systems*, 4(4):531–544, 1979.

[BGB98] L. A. Barroso, K. Gharachorloo, and E. D. Bugnion. Memory System Characterization of Commercial Workloads. In *Proc. of the Int'l Symp. on Computer Architecture*, Barcelona, Spain, June 1998.

[BK99] P. Boncz and M. Kersten. MIL Primitives For Querying a Fragmented World. *The VLDB Journal*, Y(X), to appear 1999.

[BMK99] P. Boncz, S. Manegold, and M. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 54–65, Edinburgh, Scotland, UK, September 1999.

[BQK96] P. Boncz, W. Quak, and M. Kersten. Monet and its Geographical Extensions: a Novel Approach to High-Performance GIS Processing. In *Proc. of the Intl. Conf. on Extending Database Technology*, pages 147–166, Avignon, France, June 1996.

[BWK98] P. Boncz, A. N. Wilschut, and M. Kersten. Flattening an Object Algebra to Provide Performance. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 568–577, Orlando, FL, USA, February 1998.

[BZ98] R. Berrendorf and H. Ziegler. PCL - The Performance Counter Library. Technical Report FZJ-ZAM-IB-9816, ZAM, Forschungzentrum Jülich, Germany, 1998.

[CDH+99] J. Clear, D. Dunn, B. Harvey, M. Heytens, P. Lohman, A. Mehta, M. Melton, H. Richardson, L. Rohrberg, A. Savasere, R. Wehrmeister, and M. Xu. NonStopSQL/MX. In *Proc. of the Int.'l Conference on Knowledge Discovery and Data Mining*, San Diego, CA, USA, August 1999.

[CK85] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 268–279, Austin, TX, USA, May 1985.

[Com98] Compaq Corp. Whitepaper. *Infocharger*, January 1998.

[DKO+84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 1–8, Boston, MA, USA, June 1984.

[Eic89] M. H. Eich. Main Memory Database Research Directions. In *Database Machines. 6th International Workshop*, pages 251–268, Deauville, France, June 1989.

[GMS92] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Trans. on Knowledge and Data Eng.*, 4(6):509–516, December 1992.

[Ker89] M. Kersten. Using Logarithmic Code-Expansion to Speedup Index Access and Maintenance. In *Proc. of the Int'l. Conf. on Foundation on Data Organization and Algorithms*, pages 228–232, Paris, France, October 1989.

[Knu68] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, MA, USA, 1968.

[KPH+98] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance Characterization of a quad Pentium Pro SMP using OLTP workloads. In *Proc. of the Int'l Symp. on Computer Architecture*, pages 15–26, Barcelona, Spain, June 1998.

[KSHK97] M. Kersten, A. P. J. M. Siebes, M. Holsheimer, and F. Kwakkel. Research and Business Challenges in Data Mining Technology. In *Proc. Datenbanken in Büro, Technik und Wissenschaft*, pages 1–16, Ulm, Germany, March 1997.

[LC86a] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 294–303, Kyoto, Japan, August 1986.

[LC86b] T. J. Lehman and M. J. Carey. Query Processing in Main Memory Database Systems. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 239–250, Washington, DC, USA, May 1986.

[LN96] S. Listgarten and M.-A. Neimat. Modelling Costs for a MM-DBMS. In *Proc. of the Int'l. Workshop on Real-Time Databases, Issues and Applications*, pages 72–78, Newport Beach, CA, USA, March 1996.

[MKW+98] S. McKee, R. Klenke, K. Wright, W. Wulf, M. Salinas, J. Aylor, and A. Batson. Smarter Memory: Improving Bandwidth for Streamed References. *IEEE Computer*, 31(7):54–63, July 1998.

[Mow94] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, Computer Science Department, 1994.

[NBC+94] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 233–242, Minneapolis, MN, USA, May 1994.

[PAC+97] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas,

and K. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, March 1997.

[Ram96] Rambus Technologies, Inc. *Direct Rambus Technology Disclosure*, 1996. http://www.rambus.com/docs/drtechov.pdf.

[Ron98] M. Ronström. *Design and Modeling of a Parallel Data Server for Telecom Applications.* PhD thesis, Linköping University, 1998.

[Sem97] Sematech. *National Roadmap For Semiconductor Technology: Technology Needs*, 1997. http://www.itrs.net/ntrs/publntrs.nsf.

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

[SKN94] A. Shatdal, C. Kant, and J. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 510–512, Santiago, Chile, September 1994.

[Syb96] Sybase Corp. Whitepaper. *Adaptive Server IQ*, July 1996.

[Tea99] Times Ten Team. In-memory data management for consumer transactions the times-ten approach. *ACM SIGMOD Record*, 28(2):528–529, June 1999.

[TLPZT97] P. Trancoso, J. L. Larriba-Pey, Z. Zhang, and J. Torellas. The Memory Performance of DSS Commericial Workloads in Shared-Memory Multiprocessors. In *Int'l. Symp. on High Performance Computer Architecture*, San Antonio, TX, USA, January 1997.

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

[Wil91] A. Wilschut. *Parallel Query Execution in a Main-Memory Database System.* PhD thesis, Universiteit Twente, 1991.

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

[Yea96] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[ZLTI96] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proc. of the Supercomputing '96 Conf.*, Pittsburgh, PA, USA, November 1996.