What happens during a Join? Dissecting CPU and Memory Optimization Effects

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

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

Abstract

Performance of modern hardware increasingly depends on proper utilization of both the memory cache hierarchy and parallel execution possibilities in todays super-scalar CPUs. Recent database research has demonstrated that database system performance severely suffers from poor utilization of these resources. In previous work, we presented join algorithms that strongly accelerate large equi-join by tuning the memory access pattern to match the characteristics of the memory cache subsystem in the benchmark hardware.

In order to make such algorithms applicable in database systems that run on a wide variety of platforms, we now present a calibration tool that automatically extracts the relevant parameters about the memory subsystem from any hardware. Exhaustive experiments with join-queries demonstrate how a database system equipped with this calibrator can automatically tune memory-conscious database algorithms to their optimal settings.

Once memory access is optimized, CPU resource usage becomes crucial for database performance. We demonstrate how CPU resource usage can be improved by using appropriate implementation techniques. Join experiments with the Monet database system on various hardware platforms confirm that combining memory and CPU optimization can lead to almost an order of magnitude of performance improvement on modern processors.

Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000.

1 Introduction

As database technology becomes more pervasive, DBMS software is being deployed on an ever wider variety of hardware, that ranges from high-end servers to workstations, PCs, notebooks, and in the near future, portable devices like web pads, palm pilots and even mobile phones. In the previous VLDB conference, we described experiments on an SGI Origin2000 server platform that showed how severely DBMS performance can be impacted by hardware-specific factors. We established through cost modeling and experimentation that a commonly used DBMS algorithm like hash-join runs factors slower than algorithms that are optimally tuned for the specific cache memory subsystem of the benchmark hardware [5]. Several other studies into the behavior of modern hardware on a variety of DBMS query loads corroborate this result, as all report utilization levels on modern super-scalar CPUs that are just a small fraction of their true potential. The sobering truth is that a modern CPU serving a DBMS is typically "stalled" for most of its time (i.e., non-working, waiting for something) [1, 2, 8, 12]. This percentage of CPU under-utilization in DBMS performance is still rising, due to continuing developments in commodity computer hardware. The left table in Figure 1 shows hardware characteristics of a number of popular workstations and PCs of the past decade. The right-hand plot in exponential scale reveals the trend that CPU performance and memory bandwidth[1] have increased with 50% each year (a.k.a. Moore's law), while memory latency has stayed roughly equal. This lack of progress in memory latency means that from the perspective of the CPU, memory access gets more expensive each year at an exponential rate. Therefore, optimal use of the memory caches has become crucial for obtaining good performance, and that is exactly the goal of the cache-conscious DBMS join algorithms we described.

Memory access, however, is not the only cause of under-utilization of modern CPUs. Other factors that are increasingly important have to do with the interaction between CPU and detailed characteristics of ap-

[1] We use the STREAM/copy benchmark [10] for characterizing memory bandwidth.

year	computer model	processor			memory	
		type	MHz	#par. units	STREAM/copy (bandwidth)	latency (ns)
1989	Sun 3/60	68020	20	1	6.5	
1990	Sun 3/80	68030	20	1	4.9	
1991	Sun 4/280	Sparc	17	1	9.6	160
1992	Sun ss10/31	superSparc I	33	3	42.9	
1993	Sun ss10/41	superSparc I	40	3	48.0	
1994	Sun ss20/71	superSparc II	75	3	62.5	870
1995	Sun Ultra1 170	ultraSparc I	167	5	225.2	225
1996	Sun Ultra2 2200	ultraSparc II	200	5	228.5	225
1996	SGI Power Chall.	R10000	195	5	172.7	610
1997	SGI Origin 2000	R10000	250	5	332.0	424
1998	SGI Origin 2000	R12000	300	5	336.0	404
1992	Intel PC	80486	66	1	33.3	
1993	Intel PC	Pentium	60	2	47.1	161
1994	Intel PC	Pentium	90	2	46.4	161
1995	Intel PC	Pentium	100	2	85.1	161
1996	Intel PC	Pentium	133	2	84.4	161
1996	Intel PC	PentiumPro	200	5	140.0	203
1997	Intel PC	PentiumII	300	5	188.2	145
1998	Intel PC	PentiumII	350	5	279.3	145
1998	Intel PC	PentiumII	400	5	304.0	145
1999	Intel PC	PentiumIII	600	5	379.2	135
2000	Intel PC	PentiumIII	733	5	441.9	135
1999	AMD PC	Athlon	500	9	373.5	217
2000	AMD PC	Athlon	800	9	387.9	217

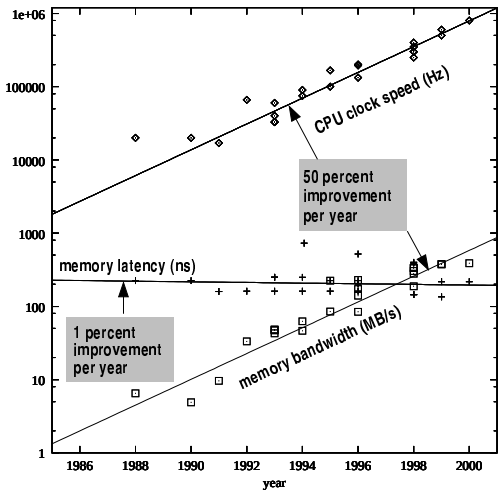

Figure 1: Trends in DRAM and CPU speed

plication program code, like the degree of dependence between instructions. This is explained by another trend in modern CPUs, which is that CPUs get more powerful not only through ever higher clock speeds, but also due to increasing parallelism *inside* the processor. Figure 1 shows that whereas the 80486 and SPARC based systems from the early 1990s still could execute at maximum 1 CPU instruction per clock cycle, an AMD Athlon from 1999 can (in theory) reach a maximum of 9 instructions per clock cycle. For this to happen in practice, aggressive compilers are needed, as well as application code whose inner program loops contain a sufficient substance of independent statements. Only then, the compiler is able to produce code that keeps the parallel units of the CPU busy. Currently, this tends to be the case only in certain scientific computation programs, not in DBMS software.

These issues may seem to drive a DBMS architect into contradictory directions. On the one hand, DBMS technology should be hardware-optimized in order to exploit the cache hierarchy and CPU resources, which could be tackled with all kinds of system-specific optimizations, but on the other hand, that same DBMS technology should be able to run on a very broad spectrum of hardware platforms that each have widely different characteristics. This paper presents important contributions that help to solve this problem.

Road-Map In Section 2, we first briefly explain the basic concepts of modern hardware that determine memory and CPU performance. Then, we recapitulate our partitioned hash-join algorithm [5] that improves join performance by trading extra partitioning CPU work for a strong reduction in memory cache misses. Finally, we summarize the relevant characteristics of our Monet system [4], which we use as experimentation platform.

In Section 3, we present our *calibration tool*, which extracts the most important hardware characteristics like cache line size, number of cache lines, and cache latency from any computer system. This generic tool can be used by any DBMS system that employs cache-conscious algorithms in order to automatically derive the right tuning parameters.[2]

We then describe large main-memory join experiments performed on four different hardware platforms (SGI, Sun, PentiumIII and Athlon) in Section 4. Studying in isolation the two phases of our partitioned hash-join algorithm (the radix-cluster phase and the hash phase), we dissect our experimental results by establishing a link between hot-spots in our DBMS implementation code and detailed split-ups into several CPU and memory cost components. Here, we show how additional factors of improvement, on top of the earlier described gains by memory cost reduction can be gained on all platforms using certain DBMS implementation techniques.

In Section 5, we combine the isolated measurements of the radix-cluster phase and the hash-join phase into full join results. Here, we achieve successful cross-platform validation of the cost models formulated in our earlier work on the SGI architecture, by filling in the hardware parameters derived by our calibrator program for the other hardware platforms into our cost formulas, and comparing the performance predicted by the models with our actual measurements. Finally, we conclude the paper by summarizing our main findings in Section 6.

2 Background

We now describe the technical details of modern hardware relevant for main-memory query performance, introduce our memory-conscious partitioned hash-join algorithm, and describe the Monet system used for experimentation.

[2]The software is freely available for download from `http://www.cwi.nl/~monet` and we encourage DBMS designers to incorporate it into their products.

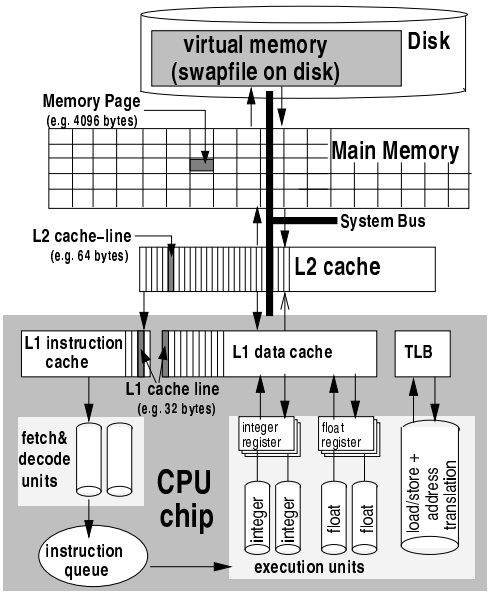

Figure 2: Modern CPU/Memory Computer Architecture

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-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 2. The main memory on the system board consists of DRAM chips. To narrow the exponentially growing performance gap between CPU speed and memory latency (cf., Figure 1), cache memories have been introduced, consisting of fast but expensive SRAM chips. Cache memories are organized in multiple cascading levels, where the faster and smaller caches are closest to the CPU. Caches consist of *cache lines*, typically 16 to 128 bytes long, which represent the smallest unit of transfer between adjacent cache levels. We assume a typical system with a small on-chip cache called *L1*, and a larger off-chip cache on the system board called *L2*. Our observations and results can be generalized to an arbitrary number of cache levels in a straightforward way.

We identify three aspects that determine memory access costs:

latency Latency is the time span that passes after issuing a data access until the requested data is available in the CPU. In hierarchical memory systems, the latency increases with the distance from the CPU. Accessing data that is already available in the L1 cache causes *L1 latency* (l_{L1}), which is typically rather small (1 or 2 CPU cycles). In case the requested data in not found in L1, an *L1 miss* occurs, additionally delaying the data access by *L2 latency* (l_{L2}) for accessing the L2 cache. Analogously, if the data is not yet available in L2, an *L2 miss* occurs, further delaying the access by *memory latency* (l_{Mem}) to finally load the data from main memory. Hence, the total latency to access data that is in neither cache is $l_{Mem} + l_{L2} + l_{L1}$. As L1 accesses 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 *Memory bandwidth* is a metric for the data volume (in megabytes) that can be transfered between CPU and main memory per second. 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 For data access, logical virtual memory addresses used by application code have to be translated to physical page addresses in the main memory of the computer. In modern CPUs, a Translation Lookaside Buffer (TLB) is used as a cache for physical page addresses, holding the translation for the most recently used pages (typically 64). If a logical address is found in the TLB, the translation has no additional costs. Otherwise, a *TLB miss* occurs. The more pages an application uses (which also depends on the often configurable size of the memory pages), the higher the probability of TLB misses. The actual *TLB miss latency* (l_{TLB}) depends on whether a system handles a TLB miss in hardware or in software.

2.2 Partitioned Hash-Join

The *radix-cluster* algorithm presented in [5] forms a basis for the experiments in this paper. In the following, we briefly recall the principle ideas.

The radix-cluster algorithm divides a relation into H clusters using multiple passes (Figure 3 shows relations R and L both being clustered into 8 clusters using two passes). 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 a straightforward clustering algorithm.

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

Note that a radix-clustered relation is in fact *ordered* on radix-bits (in Figure 3, after radix-clustering relation L, 96 is the first value, as it has radix-bits 000, then come 57,17,81,75, which all have radix-bits 001, etc.). 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.

2.3 Monet

We implemented the algorithms described above in Monet, a database kernel developed at CWI, targeted

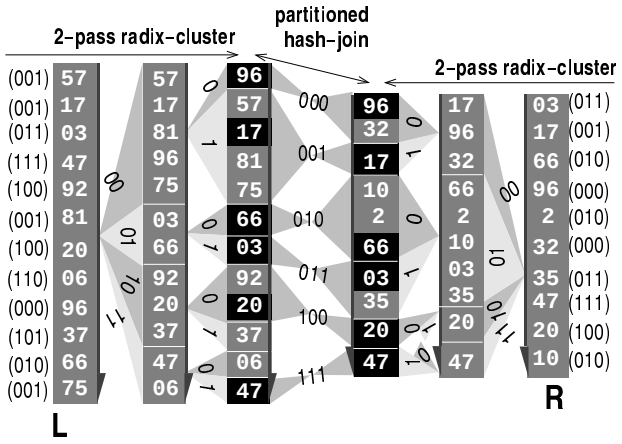

Figure 3: Partitioned Hash-join; black tuples hit (lowest 3-bits of values between parenthesis)

at achieving high performance on *query-intensive* workloads, such as created by OLAP or data mining applications. It uses the Decomposed Storage Model (DSM) [7], storing each column of a relational table in a separate binary table, called a Binary Association Table (BAT). A BAT is represented in memory as an array of fixed-size two-field records [OID,value], or Binary UNits (BUN). The OIDs in the left column are unique per original relational tuple, i.e., they link all BUNs that make up an original relational tuple. The major advantage of the DSM is that it minimizes I/O and memory access costs for column-wise data access, which occurs frequently in OLAP and data mining workloads [6]. The BAT data structure is maintained as a dense memory array, without wasted space for unused slots, both in order to speed up data access (e.g., not having to check for free slots) and because all data in the array is used, which optimizes memory cache utilization on sequential access.

Most commercial relational DBMSs were designed in a time when OLTP was the dominant DBMS application, hence their storage structures, buffer management infrastructure, and core query processing algorithms remain optimized towards OLTP. In the architecture of Monet, we took great care that systems facilities that are only needed by OLTP queries do not slow down the performance of query-intensive applications. We shortly discuss two such facilities in more detail: buffer management and lock management.

Buffer management in Monet is done on the coarse level of a BAT (it is entirely loaded or not at all), hence the query operators always have direct access to the entire relation in memory. The first reason for this strategy is to eliminate buffer management as a source of overhead inside the query processing algorithms, which would result if each operator must continuously make calls to the buffer manager asking for more tuples, typically followed by copying of tuple data into the query operator. The second reason is that all-or-nothing I/O is much more efficient nowadays than ran-

dom I/O (similarly to memory, I/O bandwidth follows Moore's law, I/O latency does not).

In Monet, we chose to implement explicit transaction facilities, which provide the building blocks for ACID transaction systems, instead of implicitly building in transaction management into the buffer management. Monet applications use the explicit locking primitives to implement a transaction protocol. In OLAP and data mining, a simple transaction protocol with a very coarse level of locking is typically sufficient (a read/write lock on the database or table level). We can safely assume that all applications adhere to this, as Monet clients are front-end programs (e.g., an SQL interpreter, or a data mining tool) rather than end-users. The important distinction from other systems is hence that Monet separates lock management from its query services, eliminating all locking overhead inside the query operators.

As a result, a sequential scan over a BAT comes down to a very simple loop over a memory array with fixed-length records, which makes Monet's query operator implementations look very much like scientific programs doing matrix computations. Such code is highly suitable for optimization by aggressive compiler techniques, and does not suffer from interference with other parts of the system, making it feasible to understand, e.g., what happens during a join? An in-depth discussion of the design and implementation of Monet can be found in [4].

3 Calibration Tool

To achieve their best performance, memory-conscious database algorithms need to be tuned to the characteristics of the very computer system they run on. Preferably, this task should be done by the database system automatically at installation time. For this to be feasible, two requirements have to be fulfilled. On the one hand, the database system has to be provided with appropriate analytical performance models for the algorithms that are to be tuned. In [5], we demonstrate how to create analytical performance models for memory-conscious database algorithms like our radix-cluster algorithm. On the other hand, characteristic parameters of the memory system, including memory sizes, cache sizes, cache line sizes, and access latencies need to be known. In the following, we describe a powerful *calibration tool* to measure the (cache) memory characteristics of an arbitrary machine on the fly.

3.1 Calibrating the Memory System

The idea underlying our calibrator tool is to have a micro benchmark whose performance only depends on the frequency of cache misses that occur. Our calibrator is a simple C program, mainly a small loop that executes a million memory reads. By changing the *stride* (i.e., the offset between two subsequent memory accesses) and the size of the memory area, we force varying cache miss rates. In principle, the occurrence of cache misses is determined by the array size. Array sizes that fit into the L1 cache do not generate any cache misses once the data is loaded into the cache. Analogously, arrays that exceed the L1 cache size, but still fit into L2, will cause L1 misses but no L2 misses. Finally, arrays larger than L2 cause both L1 and L2 misses. The frequency of cache misses depends on the access stride and the cache line size. With strides equal to or larger than the cache line size, a cache miss occurs with every iteration. With strides smaller than the cache line size, a cache miss occurs only every n iterations (on average), where n is the ratio cache_line_size/stride. Thus, we can calculate the latency for a cache miss by comparing the execution time without misses to the execution time with exactly one miss per iteration. This approach only works, if memory accesses are executed purely sequential, i.e., we have to ensure that neither two or more load instructions nor memory access and pure CPU work can overlap. We use a simple pointer chasing mechanism to achieve this: the memory area we access is initialized such that each load returns the address for the subsequent load in the next iteration. Thus, superscalar 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.

Figure 4 depicts the resulting execution time (in nanoseconds) per iteration for different array sizes on four different machines (see Table 1 for details). Each curve represents a different stride. All curves show two steps, indicating the existence of two cache levels and their sizes. Matching curves mean, that the cache miss frequency has reached its maximum (one miss per iteration), i.e., that the respective stride is equal to (or larger than) the cache line size.

3.2 Calibrating the TLB

We use a similar approach as above to measure *TLB miss costs*. The idea here is to force one TLB miss per iteration, but to avoid any cache misses. We force TLB misses by using a stride that is 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.

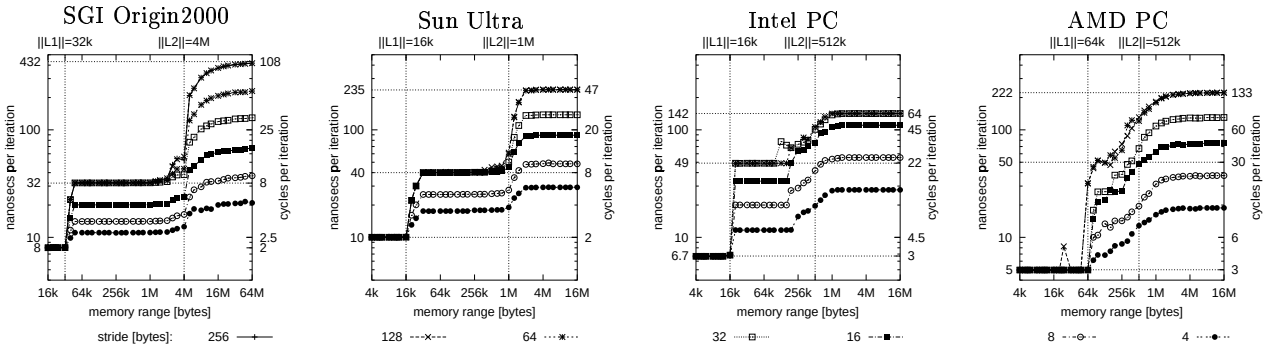

Figure 4: Cache sizes (vertical grid lines), line sizes, and miss latencies (horizontal grid lines)

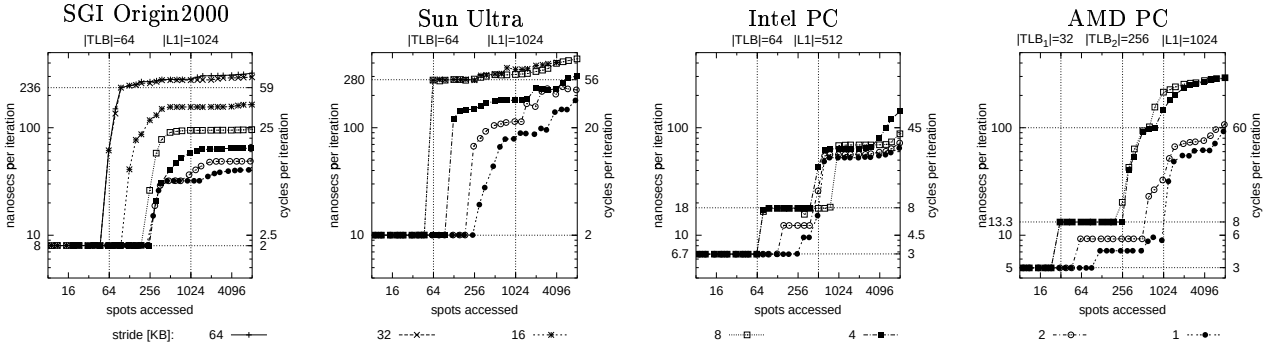

Figure 5: TLB entries (vertical grid lines), page sizes, and TLB miss costs (horizontal grid lines)

Figure 5 shows the results for four machines. The X-axis now gives the number of spots accessed, i.e., array size divided by stride. Again, each curve represents a different stride. For the SGI and the Sun, the curves depict a single distinctive step, indicating a single TLB with 64 entries. The impact of L1 misses when more than 1024 spots are accessed is hardly visible as L1 miss penalty is small compared to TLB miss penalty. On the Intel PC, the first step relates to the 64-entry TLB and the second step relates to L1 misses, which are more expensive than TLB misses on the Intel PC. On the AMD PC, there are two TLBs with 32 and 256 entries, respectively. The third step in the curves again relates to L1 misses. The page sizes can be derived just like the cache line sizes before.

Table 1 gathers the results for all four machines. The PCs have the highest L2 access latencies, probably as their L2 caches are running at only half the CPUs' clock speed. Main-memory access, however, is faster on the PCs than it is on the SGI and the Sun. The TLB miss latency of the PentiumIII and the Athlon (TLB_1) are very low, as their TLB management is implemented in hardware. This avoids the costs of trapping to the operating system on a TLB miss, that is necessary in the software controlled TLBs of the other systems. The TLB_2 miss latency on the Athlon is comparable to that on the R10000 and the UltraSPARC.

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

	SGI Origin2000	Sun Ultra	Intel PC	AMD PC
OS	IRIX64 6.5	Solaris 2.5.1	Linux 2.2.12	Linux 2.2.12
CPU	R10000	UltraSPARC	PentiumIII	Athlon
CPU speed	250 MHz	200 MHz	450 MHz	600 MHz
memory size (local)	64 GB (4 GB)	512 MB	512 MB	384 MB
L1 size	32 KB	16 KB	16 KB	64 KB
L1 line size	32 bytes	16 bytes	32 bytes	64 bytes
L2 size	4 MB	1 MB	512 KB	512 KB
L2 line size	128 bytes	64 bytes	32 bytes	64 bytes
TLB entries	64	64	64	32
TLB_2 entries	-	-	-	256
page size	32 KB	8 KB	4 KB	4 KB
L1 miss latency	24 ns 6 cycles	30 ns 6 cycles	42 ns 19 cycles	45 ns 27 cycles
L2 miss latency	400 ns 100 cycles	195 ns 39 cycles	93 ns 42 cycles	172 ns 103 cycles
TLB miss latency	228 ns 57 cycles	270 ns 54 cycles	11 ns 5 cycles	8 ns 5 cycles
TLB_2 miss latency	- -	- -	- -	87 ns 52 cycles

Table 1: Calibrated Performance Characteristics

4 Dissecting and Optimizing CPU Utilization

Recent database research demonstrates, that current commercial database systems are not able to exploit the performance potentials of modern CPUs like parallel execution pipelines and speculative execution adequately. Studies on several DBMS products on a variety of workloads [1, 2, 8, 12] consistently show that modern CPUs stall most of the execution time. Lacking access to the source code and insight in implementation details, these studies could not satisfactory answer the question, why the CPUs stall so severely when performing database tasks, nor could they provide any solution for this problem.

In this section, we use the Monet DBMS to analyze the main-memory performance behavior of hash-join algorithms on several modern hardware platform in detail. We demonstrate that once memory access is optimized, CPU utilization becomes crucial. While our original implementations show a similarly poor behavior as described in the previous studies, we present implementation techniques to optimize the CPU utilization significantly. Although we use a specific DBMS as experimentation platform, the observations we make and the improvements we suggest are relevant for any DBMS on any architecture.

4.1 Surgical Instruments

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 described in Section 2.1, 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 so-called resource-related stalls.
- *divisions.* We treat integer divisions separately, as they play a significant role in our hash-join.
- *real CPU.* This is the time the CPU is indeed busy executing the algorithms.

We use the four architectures discussed in Section 3 for our investigation. The respective CPUs provide different hardware counters [3] 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 3). Measuring data TLB misses is not possible on the UltraSPARC and the PentiumIII. We use our analytical models instead [5]. None of the architectures provides a counter for the pure CPU activity. Hence, we subtract the cycles spent on memory access, CPU stalls, and integer division from the overall number of cycles and assume the rest to be pure CPU costs.

In current commercial DBMS, branch mispredictions and instruction cache misses play a significant role [1]. In our experiments, however, we found that in our algorithms, branch mispredictions, instruction TLB misses, and instruction cache misses do not play a role on any tested architecture. The reason is that, in contrast to most commercial DBMSs, Monet's code base is designed for efficient main-memory processing. Monet uses a very large grain size for buffer management in its operators (an entire BAT), processing therefore exhibits much code locality during execution, and hence avoids instruction cache misses and branch mispredictions. Thus, for simplicity of presentation, we omit these events in our evaluation.

category	R10000	UltraSPARC	PentiumIII	Athlon
memory access	L1_data_misses L2_data_misses TLB_misses L1_inst_misses L2_inst_misses	DC_misses[3] EC_misses[4] M_{TLB} stall_IC_miss	DCU_miss_outstanding M_{TLB} IFU_mem_stall ITLB_miss	DC_refills_(L2) DC_refills_(sys) L1_DTLB_misses L2_DTLB_misses IC_misses L1_ITLB_misses L2_ITLB_misses
CPU stalls	branch_mispred	stall_mispred stall_fpdep	br_miss_pred ILD_stalled resource_stalls	branch_mispred
divisions	$C * 2 * 35$cy	$C * 2 * 60$cy	cycles_div_busy	$C * 2 * 40$cy

Table 2: Hardware Counters used for Execution Time Breakdown

4.2 Operating Theatre

In our experiments, we use binary relations (BATs) of 8 bytes wide tuples 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 [13]). Subsequent tuple reconstruction is cheap in Monet, and equal for all algorithms, so just like in [11] we do not include it in our comparison. The experiments were carried out on the machines presented in Section 3, an SGI Origin2000, a Sun Ultra, an Intel PC, and an AMD PC (cf. Table 1).

We varied the cardinalities of the relations between 15,000 and 64M tuples, but due to space limits, we only present the results for one cardinality ($C = 8$M). The effects we discuss occur with all relation sizes. For the complete results, we refer the reader to [9].

4.3 Radix Cluster

Original Implementation Figure 6 shows an execution time breakdown for 1-pass radix-cluster on each architecture. The pure CPU costs are nearly constant across all numbers of radix-bits. Memory and TLB costs are low with small numbers of radix-bits, but grow significantly with rising numbers of radix-bits. Only on the Intel PC, TLB thrashing is hardly visible due to its very low TLB miss penalty. The figures clearly reflect the impact of TLB thrashing and cache thrashing on the execution time on all architectures. This confirms that the observations we made in [5] on only one system also hold for other platforms.

Figure 7 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—and thus the memory costs—low,

[3] = DC_read - DC_read_hit + DC_write - DC_write_hit.
[4] = EC_ref - EC_hit.

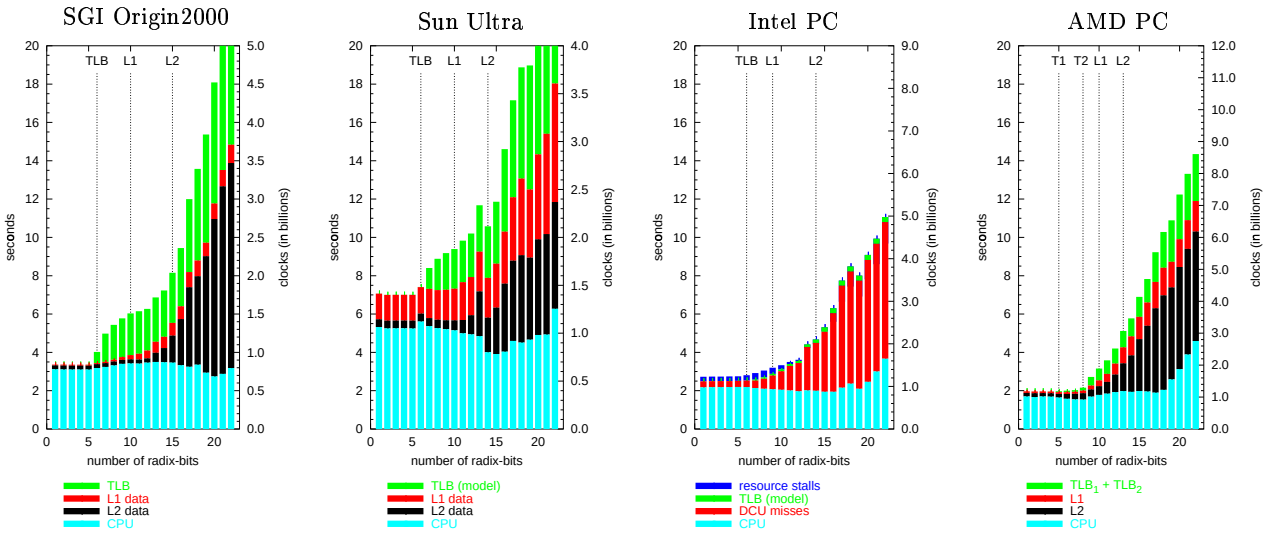

Figure 6: Execution Time Breakdown of Radix-Cluster using one pass

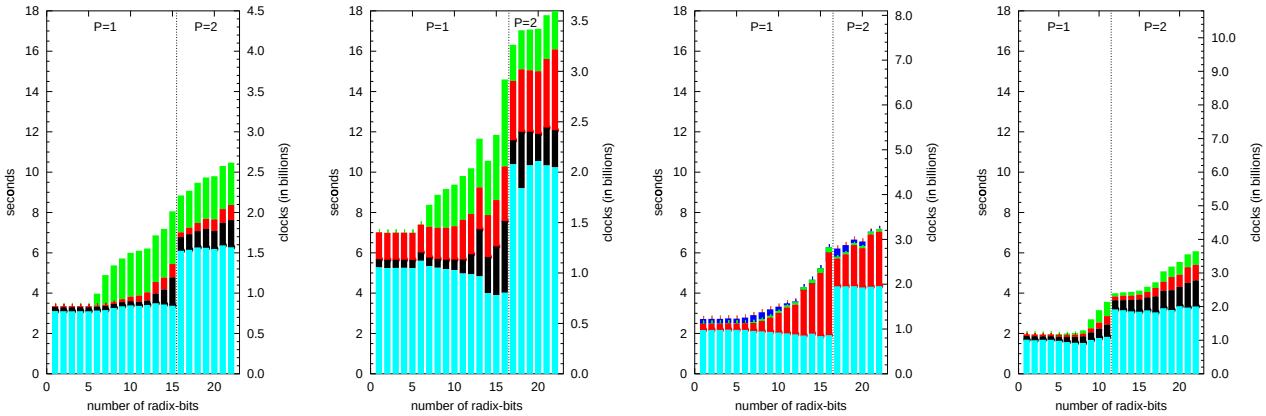

Figure 7: Execution Time Breakdown of Radix-Cluster using multiple passes

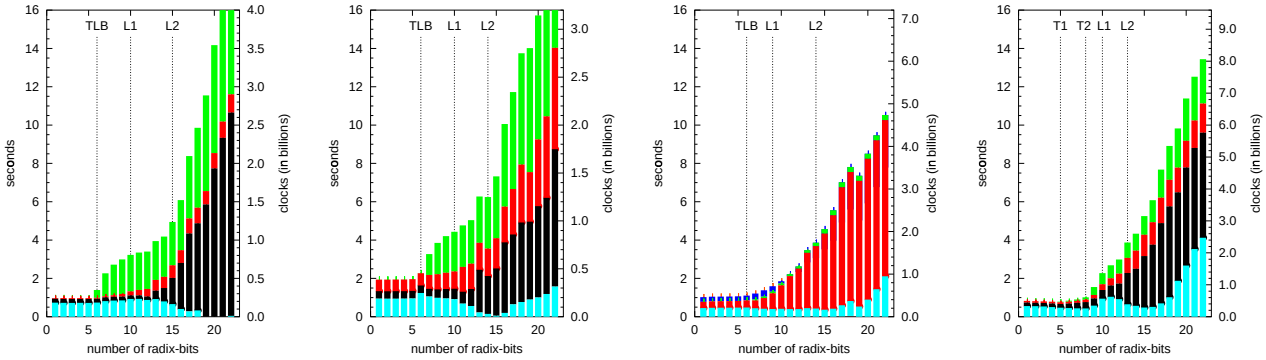

Figure 8: Execution Time Breakdown of optimized Radix-Cluster using one pass

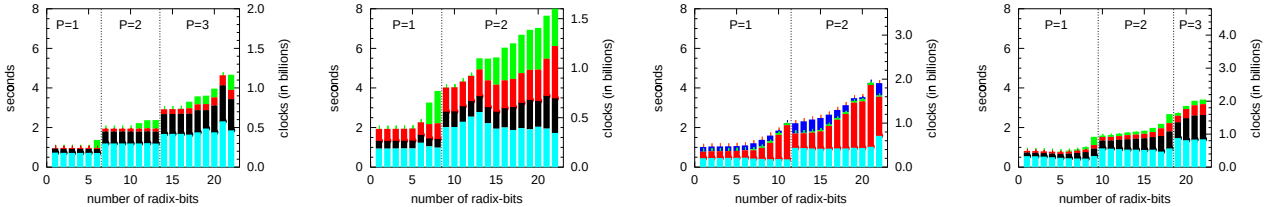

Figure 9: Execution Time Breakdown of optimized Radix-Cluster using multiple passes

at the expense of increased CPU costs. Obviously, the CPU costs are too high to avoid the TLB costs by using two passes from 7 radix-bits onward. Only with more than 15 radix-bits—i.e., when the memory costs exceed the CPU costs—two passes win over one pass. Due to the Athlon's high clock speed, two passes outperform one pass already from 11 radix-bits onward.

Optimized Implementation The only way to improve this situation is to reduce the CPU costs, i.e., to optimize the implementation of radix-cluster. Figure 10 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 techniques. We replace the generic ADT-like implementation by a specialized one for each data type.[5] Thus, we can inline the hash function and replace the `memcpy` by a simple assignment, saving two function calls per iteration.

Figure 8 shows the execution time breakdown for the optimized single-pass radix-cluster. The pure CPU costs have reduced significantly, by factor 4 on the Origin and the Intel PC, by factor 5 on the Sun, and by factor 3.5 on the AMD PC. Replacing function calls has two effects. First, CPU cycles, otherwise needed to copy the parameters to/from the stack and to perform the call itself, are saved. Second, the CPUs can benefit more from their internal parallel capabilities using speculative execution, as the code has become simpler and parallelization options more predictable.

[5] 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 [4] for a detailed discussion of this subject.

```
#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 (clusters) */
  bun *rel, bun *rel_end,           /* input relation */
  int R, int D                      /* radix and cluster bits */
){
 int idx, M = (2^D - 1) ≪ R;
 for(bun*cur=rel; cur<rel_end; cur++) {
  idx = (*hashFcn)(cur→v2)&M;     || 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 10: C language radix-cluster with annotated CPU optimizations (*right*)

With this optimization, multi-pass radix-cluster is feasible already with smaller numbers of radix-bits (cf. Figure 9). 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. Analogously, the algorithm creates at most 512 clusters per pass on the AMD PC, avoiding L1 thrashing, which is expensive due to the rather high L1 miss penalty on the Athlon.

4.4 Isolated Join Performance

Original Implementation Partitioned hash-join exhibits increased performance with increasing number of radix-bits. Figure 12 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 rising number of radix-bits. The smaller the clusters are, the less TLB and cache thrashing occurs. These results confirm that our previous observations hold for all platforms. We point out that division operations significantly contribute to the pure CPU costs on all architectures.

Optimized Implementation Like with radix-cluster, once the memory access is optimized, the execution of partitioned hash-join is dominated by CPU costs. Hence, we apply the same optimizations as above. We inline the hash-function calls during hash build and hash probe as well as the compare-function

```
hash_join(
   bun *dst, bun *dst_end          /* result buffer */
   bun *outer, bun *outer_end,     /* outer relation */
   bun *inner, bun* inner_end,     /* inner relation */
   int R                           /* radix bits */
){
   /* build hash table on inner */
   int pos=0, S=inner_end-inner, H=log2(S), N=2^H;
   int M=(N-1)≪R;
   /* hash bucket array and chain-lists */
   int next[S], bucket[N] = { -1 };
   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≥dst_end) REALLOC(dst, dst_end);
         }
      }
   }
}
```

Figure 11: C language hash-join with annotated CPU optimizations (*slanted*)

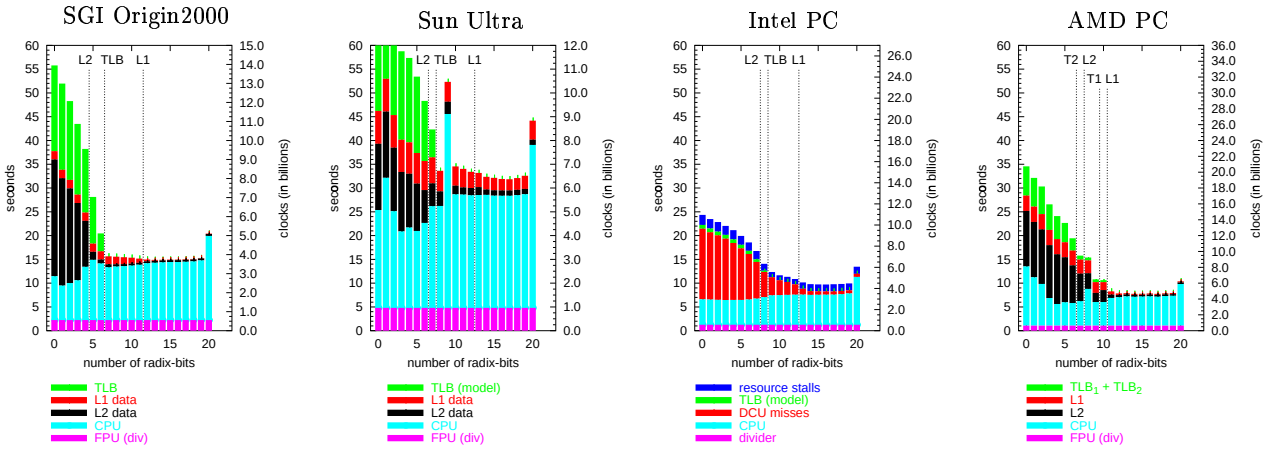

Figure 12: Execution Time Breakdown of Partitioned Hash-Join

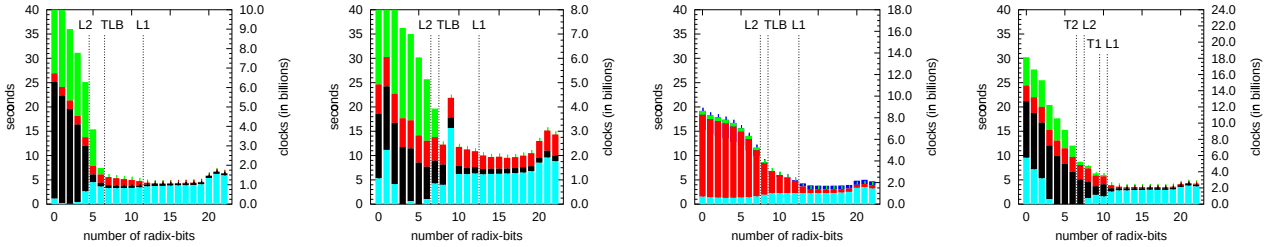

Figure 13: Execution Time Breakdown of optimized Partitioned Hash-Join

call during hash probe and replace two `memcpy` calls by simple assignments, saving five function calls per iteration. Further, we replace the modulo division ("%") for calculating the hash index by a bit operation ("&"). Figure 11 depicts the original implementation of our hash-join routine and the optimizations we apply.

Figure 13 shows the execution time breakdown for the optimized partitioned hash-join. For the same reasons as with radix-cluster, the CPU costs are reduced by almost a factor 4 on the Origin and the Sun, and by factor 3 on the PCs. The expensive divisions have vanished completely. Additionally, the stalls on the Intel PC have almost disappeared, as well.

It is interesting to note that the 450 MHz PentiumIII and the 600 MHz Athlon outperform the 250 MHz R10000 on non-optimized code, but on CPU optimized code, where the RISC chip executes without any overhead, the R10000 becomes as fast as the PCs.

5 Cross-Platform Validation

Now we turn our attention to the overall join performance, combining both phases. First, we will show that our cost model presented in [5] applies on all architectures. Then, we compare the gains due to CPU and memory optimization on the different platforms.

5.1 Validating Cost Models

In [5], we present an accurate cost model to estimate the performance of our partitioned hash-join algorithm on the Origin2000. The question remaining is, whether this model can be used to estimate the partitioned hash-join performance on other architectures as well.

The cost model mimics the memory access pattern of the algorithm and estimates the number of cache and TLB misses. To reflect platform specifics, we parameterize the model by the machine-specific memory characteristics provided by our calibration tool. Further, we calibrate the pure CPU costs using an in-cache experiment. Due to space limits, we omit the detailed cost formulae, here. The reader is referred to [9].

Figure 14 shows the overall performance for the original and the CPU-optimized versions of our algorithms, using 1-pass and multi-pass clustering on all architectures. The points represent the measured results and the lines represent our model. The model shows to be reasonably accurate on all platforms, correctly reflecting the impact of memory access and implementation techniques on the execution time. We point out that the model accurately predicts the optimal number of passes for clustering and the optimal cluster sizes. Hence, it qualifies for being used to tune memory-conscious algorithms automatically.

The results presented confirm, that the hardware parameters extracted by our calibrator provide sufficient information to capture platform specific memory access behavior. This observation is relevant not only for database cost modeling, but also for database simulators. Further, the results show that calibrating pure CPU costs with an in-cache setup is a reasonable way to capture the impact of implementation techniques and CPU characteristics in cost models.

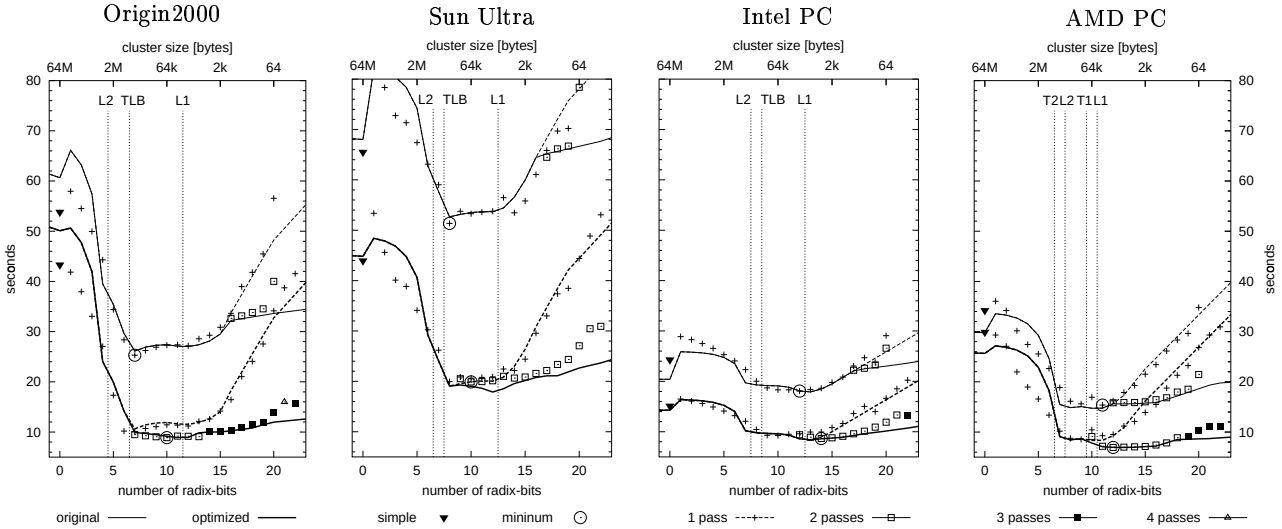

Figure 14: Measured (points) and Modeled (lines) Overall Join Performance

	SGI Origin2000					Sun Ultra					Intel PC					AMD PC				
	def		opt		rel.	def		opt		rel.	def		opt		rel.	def		opt		rel.
C	shj	phj	shj	phj	gain	shj	phj	shj	phj	gain	shj	phj	shj	phj	gain	shj	phj	shj	phj	gain
250k	0.8	0.6	0.4	0.2	3.97	1.7	1.5	1.0	0.4	3.50	0.6	0.4	0.3	0.2	3.01	0.6	0.3	0.4	0.1	3.96
500k	2.2	1.3	1.4	0.4	5.02	3.6	3.0	2.2	1.0	3.62	1.3	1.0	0.8	0.4	2.96	1.5	0.7	1.1	0.3	4.30
1M	5.2	2.7	3.7	0.9	5.74	7.6	6.1	4.8	2.1	3.59	2.7	2.0	1.6	0.9	2.81	3.3	1.5	2.6	0.7	4.59
2M	11.4	5.5	8.6	1.8	6.13	15.7	12.8	10.1	4.5	3.44	5.7	4.2	3.5	2.0	2.77	7.5	3.2	6.3	1.6	4.65
4M	25.0	11.6	19.5	4.1	6.02	32.2	26.2	20.8	9.3	3.45	11.7	8.8	7.2	4.2	2.79	16.1	7.0	13.9	3.4	4.64
8M	53.8	25.2	43.3	8.8	6.05	65.5	51.3	43.9	19.8	3.29	24.3	18.0	15.1	8.6	2.80	34.2	15.3	29.9	6.9	4.94
16M	119.6	53.4	95.1	18.1	6.60															
32M	265.4	113.3	216.7	38.1	6.96															
64M	614.2	234.6	511.4	79.5	7.71															

C: cardinality
gain: overall gain (def_shj/opt_phj)
def: default implementation
opt: CPU-optimized implementation
shj: simple hash-join (non memory-optimized)
phj: phash TLB/L1 (memory-optimized)

Table 3: Overall Join Performance (in seconds) without and with CPU and/or Memory Optimization

5.2 Overall Join Performance

From Figure 14, we derive that cluster sizes just below TLB size achieve the best performance on the RISC architectures. The PCs require even smaller clusters, fitting into the L1 cache. We refer to these settings as *phash TLB/L1*. In all cases, multi-pass radix-clustering is essential to reach the optimal performance.

Table 3 lists the absolute performance of simple hash-join and phash TLB/L1 both without and with CPU optimization applied. The numbers show that CPU and memory optimization support each other and *boost* their effects. The gain of CPU optimization for phash TLB/L1 is bigger than that for simple hash-join ((def_phj−opt_phj) > (def_shj−opt_shj)), and the gain of memory optimization for the CPU-optimized implementation is bigger than that for the non-optimized implementation ((opt_shj−opt_phj) > (def_shj−def_phj)). 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-costs are reduced, reducing the impact of memory access costs even more.

Finally, the "gain"-column in Table 3 shows, that the Origin2000 achieves the best overall performance improvement: factor 6 for 8M tuples and up to almost factor 8 for larger relations. Second is the AMD PC with factor 5, followed by the Sun (factor 3.3) and the Intel PC (factor 2.8).

6 Conclusion

The research presented here shows how the results earlier obtained on one specific platform [5] can be generalized to other hardware, and how cache-conscious query optimization can be generalized and incorporated into existing DBMS technology. A key element for achieving this is the calibrator program we provide, that automatically discovers what the memory subsystem of a computer looks like and derives important cost model parameters like cache line size, numbers of cache lines, and latencies. Combining the param-

eters derived by the calibrator on a number of new platforms (we additionally tested Sun, Intel and AMD hardware) with the detailed main-memory cost models provided in [5], we were able to successfully predict performance. Hence we conclude that generic optimization of main-memory access costs is both feasible and desirable, as correctly tuned cache-conscious algorithms greatly enhance DBMS performance.

We performed exhaustive experiments on these hardware platforms, in which we dissected the performance of our partitioned hash-join by establishing a clear link between the hot-spots in our code and detailed performance results, split-up into various CPU and memory cost components. This analysis showed that performance can be significantly enhanced even after all memory access has been eliminated. The trend of increasing parallelism inside modern superscalar CPUs makes it ever more crucial for application code that the inner loops of the query processing algorithms contain sufficient (independent) work to keep the parallel units of the CPU busy. We find that performance can be increased by another factor three or four by eliminating all function calls from the inner loops of our algorithms. Interestingly, the memory- and code-optimization seem to boost each other: code-optimization without memory-optimization is much less effective than combined and vice versa. The overall effect of combining both optimizations can yield a performance increase of a factor eight.

Our experimentation platform is the Monet system, developed by our group to support high- performance OLAP and data mining. In previous experiments on the DD Benchmark, we found that Monet was 36 times faster on a data mining query load than a commercial DBMS product that also ran fully memory/CPU bound [6]. The insights gained in this research now tell us that the near 100% CPU utilization achieved by Monet on such tasks makes the crucial difference. The requirements for achieving such high performance lead straight to the core architectural decisions made for a DBMS, hence it will not be easy to repeat these results in already existing DBMS products. We therefore expect Monet to stay in a class of its own for some time to come. Still, we hope that DBMS engineers will pick up the lessons learned and incorporate techniques described here in future DBMS software.

References

[1] 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.

[2] 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.

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

[4] P. Boncz and M. Kersten. MIL Primitives For Querying a Fragmented World. *The VLDB Journal*, 8(2), October 1999.

[5] 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.

[6] P. Boncz, T. Rühl, and F. Kwakkel. The Drill Down Benchmark. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 628–632, New York, NY, USA, June 1998.

[7] 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.

[8] 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.

[9] S. Manegold, P. Boncz, and M. Kersten. Optimizing Main-Memory Join On Modern Hardware. Technical Report INS-R9912, CWI, Amsterdam, The Netherlands, October 1999.

[10] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Technical Committee on Computer Architecture newsletter*, December 1995.

[11] 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.

[12] 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.

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