DBMSs On A Modern Processor: Where Does Time Go?

Anastassia Ailamaki David J. DeWitt Mark D. Hill David A. Wood

University of Wisconsin-Madison
Computer Science Dept.
1210 W. Dayton St.
Madison, WI 53706
U.S.A.
{natassa,dewitt,markhill,david}@cs.wisc.edu

Abstract

Recent high-performance processors employ sophisticated techniques to overlap and simultaneously execute multiple computation and memory operations. Intuitively, these techniques should help database applications, which are becoming increasingly compute and memory bound. Unfortunately, recent studies report that faster processors do not improve database system performance to the same extent as scientific workloads. Recent work on database systems focusing on minimizing memory latencies, such as cache-conscious algorithms for sorting and data placement, is one step toward addressing this problem. However, to best design high performance DBMSs we must carefully evaluate and understand the processor and memory behavior of commercial DBMSs on today's hardware platforms.

In this paper we answer the question "Where does time go when a database system is executed on a modern computer platform?" We examine four commercial DBMSs running on an Intel Xeon and NT 4.0. We introduce a framework for analyzing query execution time on a DBMS running on a server with a modern processor and memory architecture. To focus on processor and memory interactions and exclude effects from the I/O subsystem, we use a memory resident database. Using simple queries we find that database developers should (a) optimize data placement for the second level of data cache, and not the first, (b) optimize instruction placement to reduce first-level instruction cache stalls, but (c) not expect the overall execution time to decrease significantly without addressing stalls related to subtle implementation issues (e.g., branch prediction).

1 Introduction

Today's database servers are systems with powerful processors that overlap and complete instructions and memory accesses out of program order. Due to the sophisticated techniques used for hiding I/O latency and the complexity of modern database applications, DBMSs are becoming compute and memory bound. Although researchers design and evaluate processors using programs much simpler than DBMSs (e.g., SPEC, LINPACK), one would hope that more complicated programs such as DBMSs would take full advantage of the architectural innovations. Unfortunately, recent studies on some commercial DBMSs have shown that their hardware behavior is suboptimal compared to scientific workloads.

Recently there has been a significant amount of effort toward improving the performance of database applications on today's processors. The work that focuses on optimizing the processor and memory utilization can be divided into two categories: evaluation studies and cache performance improvement techniques. The first category includes a handful of recent studies that identified the problem and motivated the community to study it further. Each of these studies presents results

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

from experiments with only a single DBMS running a TPC benchmark on a specific platform. The second category includes papers that propose (a) algorithmic improvements for better cache utilization when performing popular tasks in a DBMS, such as sorting, and (b) data placement techniques for minimizing cache related waiting time.

Although generally the results of these evaluation studies corroborate each other, there are no results showing the behavior of more than one commercial DBMS on the same hardware platform. Such results are important in order to identify general trends that hold true across database systems and determine what problems we must work on to make database systems run faster.

This is the first paper to analyze the execution time breakdown of four commercial DBMSs on the same hardware platform (a 6400 PII Xeon/MT Workstation running Windows NT v4.0). The workload consists of range selections and joins running on a memory resident database, in order to isolate basic operations and identify common trends across the DBMSs. We conclude that, even with these simple queries, almost half of the execution time is spent on stalls. Analysis of the components of the stall time provides more insight about the operation of the cache as the record size and the selectivity are varied. The simplicity of the queries helped to overcome the lack of access to the DBMS source code. The results show that:

- On the average, half the execution time is spent in stalls (implying database designers can improve DBMS performance significantly by attacking stalls).
- In all cases, 90% of the memory stalls are due to:
  - Second-level cache data misses, while first-level data stalls are not important (implying data placement should focus on the second-level cache),

  and
  - First-level instruction cache misses, while second-level instruction stalls are not important (implying instruction placement should focus on level one instruction caches).
- About 20% of the stalls are caused by subtle implementation details (e.g., branch mispredictions) (implying that there is no "silver bullet" for mitigating stalls).
- (A methodological result.) Using simple queries rather than full TPC-D workloads provides a methodological advantage, because results are simpler to analyze and yet are substantially similar to the results obtained using full benchmarks. To verify this, we implemented and ran the TPC-D benchmark on three of the four systems, and the results are substantially similar to the results obtained using simpler queries.

The rest of this paper is organized as follows: Section 2 presents a summary of recent database workload characterization studies and an overview of the cache performance improvements proposed. Section 3 describes the vendor-independent part of this study: an analytic framework for characterizing the breakdown of the execution time and the database workload. Section 4 describes the experimental setup. Section 5 presents our results. Section 6 concludes, and Section 7 discusses future directions.

2 Related Work

Much of the related research has focused on improving the query execution time, mainly by minimizing the stalls due to memory hierarchy when executing an isolated task. There are a variety of algorithms for fast sorting techniques [1][12][15] that propose optimal data placement into memory and sorting algorithms that minimize cache misses and overlap memory-related delays. In addition, several cache-conscious techniques such as blocking, data partitioning, loop fusion, and data clustering were evaluated [17] and found to improve join and aggregate queries. Each of these studies is targeted to a specific task and concentrate on ways to make it faster.

The first hardware evaluation of a relational DBMS running an on-line transaction processing (OLTP) workload [22] concentrated on multiprocessor system issues, such as assigning processes to different processors to avoid bandwidth bottlenecks. Contrasting scientific and commercial workloads [14] using TPC-A and TPC-C on another relational DBMS showed that commercial workloads exhibit large instruction footprints with distinctive branch behavior, typically not found in scientific workloads and that they benefit more from large first-level caches. Another study [21] showed that, although I/O can be a major bottleneck, the processor is stalled 50% of the time due to cache misses when running OLTP workloads.

In the past two years, several interesting studies evaluated database workloads, mostly on multiprocessor platforms. Most of these studies evaluate OLTP workloads [4][13][10], a few evaluate decision support (DSS) workloads [11] and there are some studies that use both [2][16]. All of the studies agree that the DBMS behavior depends upon the nature of the workload (DSS or OLTP), that DSS workloads benefit more from out-of-order processors with increased instruction-level parallelism than OLTP, and that memory stalls are a major bottleneck. Although the list of references presented here is not exhaustive, it is representative of the work done in evaluating database workloads. Each of these studies presents results from a single DBMS running a TPC benchmark on a single platform, which makes contrasting the DBMSs and identifying common characteristics difficult.

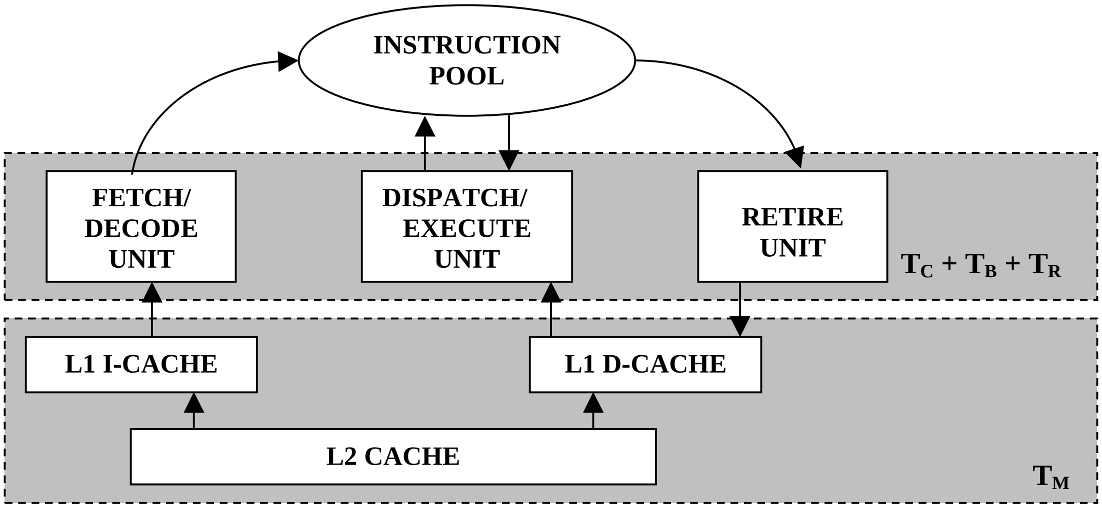


Figure 2.1: *Simplified block diagram of a processor operation*

3 Query execution on modern processors

In this section, we describe a framework that describes how major hardware components determine execution time. The framework analyzes the hardware behavior of the DBMS from the moment it receives a query until the moment it returns the results. Then, we describe a workload that allows us to focus on the basic operations of the DBMSs in order to identify the hardware components that cause execution bottlenecks.

3.1 Query execution time: a processor model

To determine where the time goes during execution of a query, we must understand how a processor works. The pipeline is the basic module that receives an instruction, executes it and stores its results into memory. The pipeline works in a number of sequential stages, each of which involves a number of functional components. An operation at one stage can overlap with operations at other stages.

Figure 3.1 shows a simplified diagram of the major pipeline stages of a processor similar to the Pentium II [6][8]. First, the FETCH/DECODE unit reads the user program instructions from the instruction cache (L1 I-cache), decodes them and puts them into an instruction pool. The DISPATCH/EXECUTE unit schedules execution of the instructions in the pool subject to data dependencies and resource availability, and temporarily stores their results. Finally, the RETIRE unit knows how and when to commit (retire) the temporary results into the data cache (L1 D-cache).

In some cases, an operation may not be able to complete immediately and delay (“stall”) the pipeline. The processor tries to cover the stall time by doing useful work, using the following techniques:

- Non-blocking caches: Caches do not block when servicing requests. For example, if a read request to one of the first-level caches fails (misses), the request is forwarded to the second-level cache (L2 cache), which is usually unified (used for both data and instructions). If the request misses in L2 as well, it is forwarded to main memory. During the time the retrieval is pending, the caches at both levels can process other requests.
- Out-of-order execution: If instruction X stalls, another instruction Y that follows X in the program can execute before X, provided that Y’s input operands do not depend on X’s results. The dispatch/execute unit contains multiple functional units to perform out-of-order execution of instructions.
- Speculative execution with branch prediction: Instead of waiting until a branch instruction’s predicate is resolved, an algorithm “guesses” the predicate and fetches the appropriate instruction stream. If the guess is correct, the execution continues normally; if it is wrong, the pipeline is flushed, the retire unit deletes the wrong results and the fetch/decode unit fetches the correct instruction stream. Branch mispredictions incur both computation overhead (time spent in computing the wrong instructions), and stall time.

Even with these techniques, the stalls cannot be fully overlapped with useful computation. Thus, the time to execute a query (T_Q) includes a useful computation time (T_C), a stall time because of memory stalls (T_M), a branch misprediction overhead (T_B), and resource-related stalls (T_R). The latter are due to execution resources not being available, such as functional units, buffer space in the instruction pool, or registers. As discussed above, some of the stall time can be overlapped (T_{OVL}). Thus, the following equation holds:

$$T_Q = T_C + T_M + T_B + T_R - T_{OVL}$$

$\mathbf{T_C}$			**computation time**
$\mathbf{T_M}$			**stall time related to memory hierarchy**
	T_{L1D}		stall time due to L1 D-cache misses (with hit in L2)
	T_{L1I}		stall time due to L1 I-cache misses (with hit in L2)
	T_{L2}	T_{L2D}	stall time due to L2 data misses
		T_{L2I}	stall time due to L2 instruction misses
	T_{DTLB}		stall time due to DTLB misses
	T_{ITLB}		stall time due to ITLB misses
$\mathbf{T_B}$			**branch misprediction penalty**
$\mathbf{T_R}$			**resource stall time**
	T_{FU}		stall time due to functional unit unavailability
	T_{DEP}		stall time due to dependencies among instructions
	T_{MISC}		stall time due to platform-specific characteristics

Table 3.1: *Execution time components*

Table 3.1 shows the time breakdown into smaller components. The DTLB and ITLB (Data or Instruction Translation Lookaside Buffer) are page table caches used for translation of data and instruction virtual addresses into physical ones. The next section briefly discusses the importance of each stall type and how easily it can be overlapped using the aforementioned techniques. A detailed discussion on hiding stall times can be found elsewhere [6].

3.2 Significance of the stall components

Previous work has focused on improving DBMS performance by reducing T_M, the memory hierarchy stall component. In order to be able to use the experimental results effectively, it is important to determine the contribution each of the different types of stalls makes to the overall execution time. Although out-of-order and speculative execution help hide some of the stalls, there are some stalls that are difficult to overlap, and thus are the most critical for performance.

It is possible to overlap T_{L1D} if the number of L1 D-cache misses is not too high. Then the processor can fetch and execute other instructions until the data is available from the second-level cache. The more L1 D-cache misses that occur, the more instructions the processor must execute to hide the stalls. Stalls related to L2 cache data misses can overlap with each other, when there are sufficient parallel requests to main memory. T_{DTLB} can be overlapped with useful computation as well, but a DTLB miss penalty depends on the page table implementation for each processor. Processors successfully use sophisticated techniques to overlap data stalls with useful computation.

Instruction-related cache stalls, on the other hand, are difficult to hide because they cause a serial bottleneck to the pipeline. If there are no instructions available, the processor must wait. Branch mispredictions also create serial bottlenecks; the processor again must wait until the correct instruction stream is fetched into the pipeline. The Xeon processor exploits spatial locality in the instruction stream with special instruction-prefetching hardware. Instruction prefetching effectively reduces the number of I-cache stalls, but occasionally it can increase the branch misprediction penalty.

Although related to instruction execution, T_R (the resource stall time) is easier to overlap than T_{ITLB} and instruction cache misses. The processor can hide T_{DEP} depending on the degree of instruction-level parallelism of the program, and can overlap T_{FU} with instructions that use functional units with less contention.

3.3 Database workload

The workload used in this study consists of single-table range selections and two table equijoins over a memory resident database, running a single command stream. Such a workload eliminates dynamic and random parameters, such as concurrency control among multiple transactions, and isolates basic operations, such as sequential access and index selection. In addition, it allows examination of the processor and memory behavior without I/O interference. Thus, it is possible to explain the behavior of the system with reasonable assumptions and identify common trends across different DBMSs.

The database contains one basic table, R, defined as follows:

create table R (a1 **integer not null**,
a2 **integer not null**,
a3 **integer not null**,
<rest of fields>)

In this definition, <rest of fields> stands for a list of integers that is not used by any of the queries. The relation is populated with 1.2 million 100-byte records. The values of the field a2 are uniformly distributed between 1 and 40,000. The experiments run three basic queries on R:

1. *Sequential range selection:*

select avg(a3)
from R
where a2 < Hi **and** a2 > Lo (1)

The purpose of this query is to study the behavior of the DBMS when it executes a sequential scan, and examine the effects of record size and query selectivity. Hi and Lo define the interval of the qualification attribute, a2. The reason for using an aggregate, as opposed to just selecting the rows, was twofold. First, it makes the DBMS return a minimal number of rows, so that the measurements are not affected by client/server communication overhead. Storing the results into a temporary relation would affect the measurements because of the extra insertion operations. Second, the average aggregate is a common operation in the TPC-D benchmark. The selectivity used was varied from 0% to 100%. Unless otherwise indicated, the query selectivity used is 10%.

2. *Indexed range selection:* The range selection (1) was resubmitted after constructing a non-clustered index on R.a2. The same variations on selectivity were used.

3. *Sequential join:* To examine the behavior when executing an equijoin with no indexes, the database schema was augmented by one more relation, S, defined the same way as R. The field a1 is a primary key in S. The query is as follows:

select avg(R.a3)
from R, S
where R.a2 = S.a1 (2)

There are 40,000 100-byte records in S, each of which joins with 30 records in R.

4 Experimental Setup

We used a 6400 PII Xeon/MT Workstation to conduct all of the experiments. We use the hardware counters of the Pentium II Xeon processor to run the experiments at full speed, to avoid any approximations that simulation would impose, and to conduct a comparative evaluation of the four DBMSs. This section describes the platform-specific hardware and software details, and presents the experimental methodology.

4.1 The hardware platform

The system contains one Pentium II Xeon processor running at 400 MHz, with 512 MB of main memory connected to the processor chip through a 100 MHz system bus. The Pentium II is a powerful server processor with an out-of-order engine and speculative instruction execution [23]. The X86 instruction set is composed by CISC instructions, and they are translated into up to three RISC instructions (μops) each at the decode phase of the pipeline.

Characteristic	**L1 (split)**	**L2**
Cache size	16KB Data 16KB Instruction	512KB
Cache line size	32 bytes	32 bytes
Associativity	4-way	4-way
Miss Penalty	4 cycles (w/ L2 hit)	Main memory
Non-blocking	Yes	Yes
Misses outstanding	4	4
Write Policy	L1-D: Write-back L1-I: Read-only	Write-back

Table 4.1: *Pentium II Xeon cache characteristics*

There are two levels of non-blocking cache in the system. There are separate first-level caches for instructions and data, whereas at the second level the cache is unified. The cache characteristics are summarized in Table 4.1.

4.2 The software

Experiments were conducted on four commercial DBMSs, the names of which cannot be disclosed here due to legal restrictions. Instead, we will refer to them as System A, System B, System C, and System D. They were installed on Windows NT 4.0 Service Pack 4.

The DBMSs were configured the same way in order to achieve as much consistency as possible. The buffer pool size was large enough to fit the datasets for all the queries. We used the NT performance-monitoring tool to ensure that there was no significant I/O activity during query execution, because the objective is to measure pure processor and memory performance. In addition, we wanted to avoid measuring the I/O subsystem of the OS. To define the schema and execute the queries, the exact same commands and datasets were used for all the DBMSs, with no vendor-specific SQL extensions.

4.3 Measurement tools and methodology

The Pentium II processor provides two counters for event measurement [8]. We used *emon*, a tool provided by Intel, to control these counters. Emon can set the counters to zero, assign event codes to them and read their values either after a pre-specified amount of time, or after a program has completed execution. For example, the following command measures the number of retired instructions during execution of the program *prog.exe*, at the user and the kernel level:

emon –C (INST_RETIRED:USER,
INST_RETIRED:SUP) prog.exe

Emon was used to measure 74 event types for the results presented in this report. We measured each event type in both user and kernel mode.

Stall time component			Description	Measurement method
T_C			computation time	Estimated minimum based on μops retired
T_M	T_{L1D}		L1 D-cache stalls	#misses * 4 cycles
	T_{L1I}		L1 I-cache stalls	actual stall time
	T_{L2}	T_{L2D}	L2 data stalls	#misses * measured memory latency
		T_{L2I}	L2 instruction stalls	#misses * measured memory latency
	T_{DTLB}		DTLB stalls	Not measured
	T_{ITLB}		ITLB stalls	#misses * 32 cycles
T_B			branch misprediction penalty	# branch mispredictions retired * 17 cycles
T_R	T_{FU}		functional unit stalls	actual stall time
	T_{DEP}		dependency stalls	actual stall time
	T_{ILD}		Instruction-length decoder stalls	actual stall time
T_{OVL}			overlap time	Not measured

Table 4.2: *Method of measuring each of the stall time components*

Before taking measurements for a query, the main memory and caches were warmed up with multiple runs of this query. In order to distribute and minimize the effects of the client/server startup overhead, the unit of execution consisted of 10 different queries on the same database, with the same selectivity. Each time emon executed one such unit, it measured a pair of events. In order to increase the confidence intervals, the experiments were repeated several times and the final sets of numbers exhibit a standard deviation of less than 5 percent. Finally, using a set of formulae[1], these numbers were transformed into meaningful performance metrics.

Using the counters, we measured each of the stall times described in Section 3.1 by measuring each of their individual components separately. The application of the framework to the experimental setup suffers the following caveats:

- We were not able to measure T_{DTLB}, because the event code is not available.
- The Pentium II event codes allow measuring the number of occurrences for each event type (e.g., number of L1 instruction cache misses) during query execution. In addition, we can measure the actual stall time due to certain event types (after any overlaps). For the rest, we multiplied the number of occurrences by an estimated penalty [18][19]. Table 4.2 shows a detailed list of stall time components and the way they were measured. Measurements of the memory subsystem strongly indicate that the workload is latency-bound, rather than bandwidth-bound (it rarely uses more than a third of the available memory bandwidth). In addition, past experience [18][19] with database applications has shown little queuing of requests in memory. Consequently, we expect the results that use penalty approximations to be fairly accurate.
- No contention conditions were taken into account.

T_{MISC} from Table 4.1 (stall time due to platform-specific characteristics) has been replaced with T_{ILD} (instruction-length decoder stalls) in Table 4.2. Instruction-length decoding is one stage in the process of translating X86 instructions into μops.

5 Results

We executed the workload described in Section 3 on four commercial database management systems. In this section, we first present an overview of the execution time breakdown and discuss some general trends. Then, we focus on each of the important stall time components and analyze it further to determine the implications from its behavior. Finally, we compare the time breakdown of our microbenchmarks against a TPC-D and a TPC-C workload. Since almost all of the experiments executed in user mode more than 85% of the time, all of the measurements shown in this section reflect user mode execution, unless stated otherwise.

5.1 Execution time breakdown

Figure 5.1 shows three graphs, each summarizing the average execution time breakdown for one of the queries. Each bar shows the contribution of the four components (T_C, T_M, T_B, and T_R) as a percentage of the total query execution time. The middle graph showing the indexed range selection only includes systems B, C and D, because System A did not use the index to execute this query. Although the workload is much simpler than TPC benchmarks [5], the computation time is usually less than half the execution time; thus, the processor spends most of the time stalled. Similar results have been presented for OLTP [21][10] and DSS [16] workloads, although none of the studies measured more than one DBMS. The high processor stall time indicates the importance of further

[1] Seckin Unlu and Andy Glew provided us with invaluable help in figuring out the correct formulae, and Kim Keeton shared with us the ones used in [10].

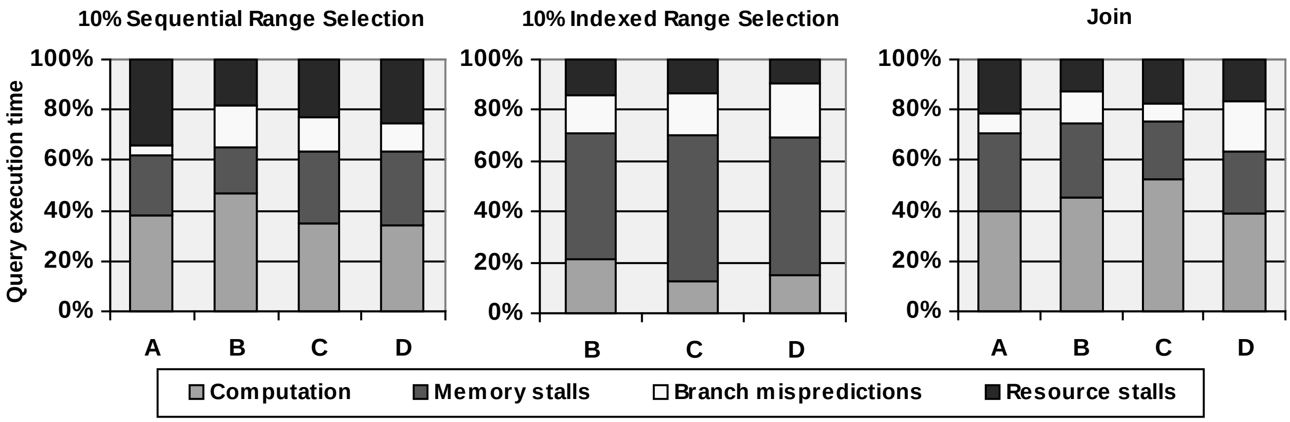


Figure 5.1: *Query execution time breakdown into the four time components.*

analyzing the query execution time. Even as processor clocks become faster, stall times are not expected to become much smaller because memory access times do not decrease as fast. Thus, the computation component will become an even smaller fraction of the overall execution time.

The memory stall time contribution varies more across different queries and less across different database systems. For example, Figure 5.1 shows that when System B executes the sequential range selection, it spends 20% of the time in memory stalls. When the same system executes the indexed range selection, the memory stall time contribution becomes 50%. Although the indexed range selection accesses fewer records, its memory stall component is larger than in the sequential selection, probably because the index traversal has less spatial locality than the sequential scan. The variation in T_M's contribution across DBMSs suggests different levels of platform-specific optimizations. However, as discussed in Section 5.2, analysis of the memory behavior yields that 90% of T_M is due to L1 I-cache and L2 data misses in all of the systems measured. Thus, despite the variation, there is common ground for research on improving memory stalls without necessarily having to analyze all of the DBMSs in detail.

Minimizing memory stalls has been a major focus of database research on performance improvement. Although in most cases the memory stall time (T_M) accounts for most of the overall stall time, the other two components are always significant. Even if the memory stall time is entirely hidden, the bottleneck will eventually shift to the other stalls. In systems B, C, and D, branch misprediction stalls account for 10-20% of the execution time, and the resource stall time contribution ranges from 15-30%. System A exhibits the smallest T_M and T_B of all the DBMSs in most queries; however, it has the highest percentage of resource stalls (20-40% of the execution time). This indicates that optimizing for two kinds of stalls may shift the bottleneck to the third kind. Research on improving DBMS performance should focus on minimizing all three kinds of stalls to effectively decrease the execution time.

5.2 Memory stalls

In order to optimize performance, a major target of database research has been to minimize the stall time due to memory hierarchy and disk I/O latencies [1][12][15][17]. Several techniques for cache-conscious data placement have been proposed [3] to reduce cache misses and miss penalties. Although these techniques are successful within the context in which they were proposed, a closer look at the execution time breakdown shows that there is significant room for improvement. This section discusses the significance of the memory stall components to the query execution time, according to the framework discussed in Section 3.2.

Figure 5.2 shows the breakdown of T_M into the following stall time components: T_{L1D} (L1 D-cache miss stalls), T_{L1I} (L1 I-cache miss stalls), T_{L2D} (L2 cache data miss stalls), T_{L2I} (L2 cache instruction miss stalls), and T_{ITLB} (ITLB miss stalls) for each of the four DBMSs. There is one graph for each type of query. Each graph shows the memory stall time breakdown for the four systems. The selectivity for range selections shown is set to 10% and the record size is kept constant at 100 bytes.

From Figure 5.2, it is clear that L1 D-cache stall time is insignificant. In reality its contribution is even lower, because our measurements for the L1 D-cache stalls do not take into account the overlap factor, i.e., they are upper bounds. An L1 D-cache miss that hits on the L2 cache incurs low latency, which can usually be overlapped with other computation. Throughout the experiments, the L1 D-cache miss rate (number of misses divided by the number of memory references) usually is around 2%, and never exceeds 4%. A study on Postgres95 [11] running TPC-D also reports low L1 D-cache miss rates. Further analysis indicates that during query execution the DBMS accesses private data structures more often than it accesses data in the relations. This

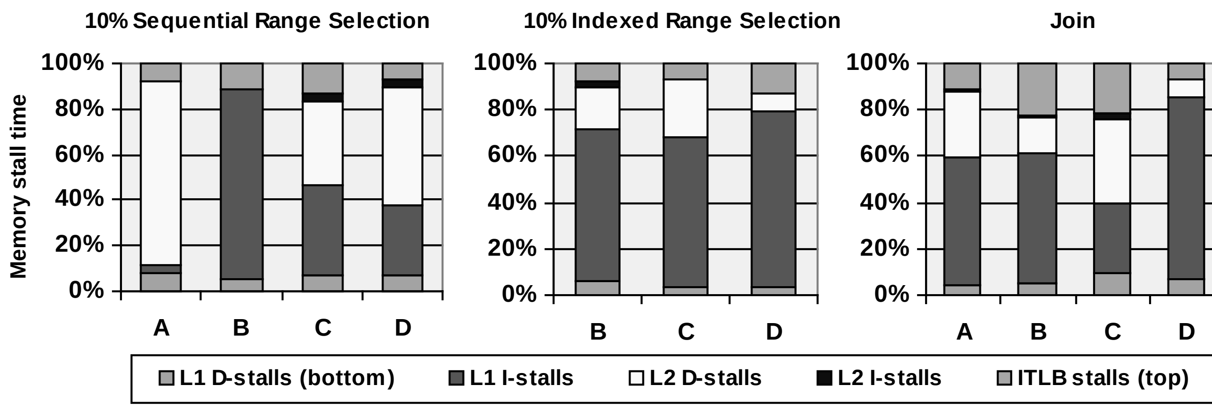


Figure 5.2: *Contributions of the five memory components to the memory stall time (T_M)*

often-accessed portion of data fits into the L1 D-cache, and the only misses are due to less often accessed data. The L1 D-cache is not a bottleneck for any of the commercial DBMSs we evaluated.

The stall time caused by L2 cache instruction misses (T_{L2I}) and ITLB misses (T_{ITLB}) is also insignificant in all the experiments. T_{L2I} contributes little to the overall execution time because the second-level cache misses are two to three orders of magnitude less than the first-level instruction cache misses. The low T_{ITLB} indicates that the systems use few instruction pages, and the ITLB is enough to store the translations for their addresses.

The rest of this section discusses the two major memory-related stall components, T_{L2D} and T_{L1I}.

5.2.1 Second-level cache data stalls

For all of the queries run across the four systems, T_{L2D} (the time spent on L2 data stalls) is one of the most significant components of the execution time. In three out of four DBMSs, the L2 cache data miss rate (number of data misses in L2 divided by number of data accesses in L2) is typically between 40% and 90%, therefore much higher than the L1 D-cache miss rate. The only exception is System B, which exhibits optimized data access performance at the second cache level as well. In the case of the sequential range query, System B exhibits far fewer L2 data misses per record than all the other systems (B has an L2 data miss rate of only 2%), consequently its T_{L2D} is insignificant.

The stall time due to L2 cache data misses directly relates to the position of the accessed data in the records and the record size. As the record size increases, T_{L2D} increases as well for all four systems (results are not shown graphically due to space restrictions). The two fields involved in the query, a2 and a3, are always in the beginning of each record, and records are stored sequentially. For larger record sizes, the fields a2 and a3 of two subsequent records are located further apart and the spatial locality of data in L2 decreases.

Second-level cache misses are much more expensive than the L1 D-cache misses, because the data has to be fetched from main memory. Generally, a memory latency of 60-70 cycles was observed. As discussed in Section 3.2, multiple L2 cache misses can overlap with each other. Since we measure an upper bound of T_{L2D} (number of misses times the main memory latency), this overlap is hard to estimate. However, the real T_{L2D} cannot be significantly lower than our estimation because memory latency, rather than bandwidth, bind the workload (most of the time the overall execution uses less than one third of the available memory bandwidth). As the gap between memory and processor speed increases, one expects data access to the L2 cache to become a major bottleneck for latency-bound workloads. The size of today's L2 caches has increased to 8 MB, and continues to increase, but larger caches usually incur longer latencies. The Pentium II Xeon on which the experiments were conducted can have an L2 cache up to 2 MB [23] (although the experiments were conducted with a 512-KB L2 cache).

5.2.2 First-level cache instruction stalls

Stall time due to misses at the first-level instruction cache (T_{L1I}) is a major memory stall component for three out of four DBMSs. The results in this study reflect the real I-cache stall time, with no approximations. Although the Xeon uses stream buffers for instruction prefetching, L1 I-misses are still a bottleneck, despite previous results [16] that show improvement of T_{L1I} when using stream buffers on a shared memory multiprocessor. As explained in Section 3.2, T_{L1I} is difficult to overlap, because L1 I-cache misses cause a serial bottleneck to the pipeline. The only case where T_{L1I} is insignificant (5%) is when System A executes the sequential range query. For that query, System A retires the lowest number of instructions per record of the four systems tested, as shown in Figure 5.3. For the other systems T_{L1I} accounts for between 4% and 40% of the total execution time, depending on the type of the query and the DBMS. For all DBMSs, the average contribution of T_{L1I} to the execution time is 20%.

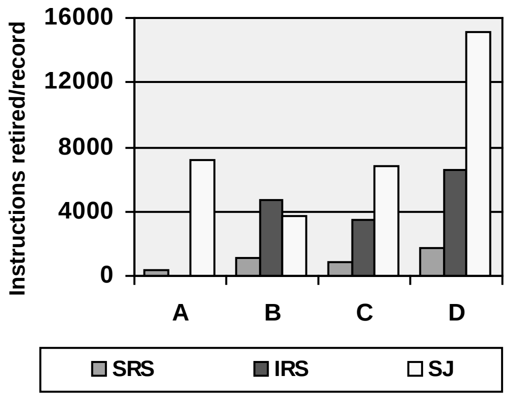


Figure 5.3: *Number of instructions retired per record for all four DBMSs. SRS: sequential selection (instructions/number of records in R), IRS: indexed selection (instructions/number of selected records), SJ: join (instructions/number of records in R)*

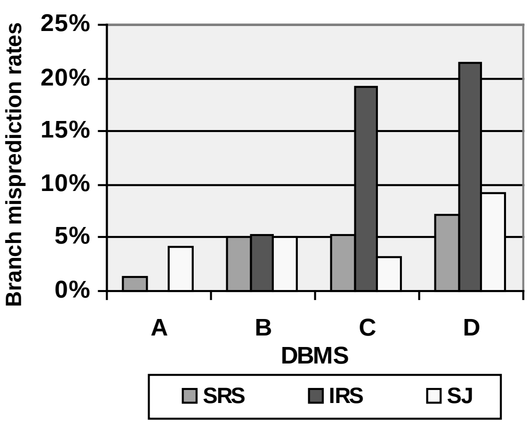
There are some techniques to reduce the I-cache stall time [6] and use the L1 I-cache more effectively. Unfortunately, the first-level cache size is not expected to increase at the same rate as the second-level cache size, because large L1 caches are not as fast and may slow down the processor clock. Some new processors use a larger (64-KB) L1 I-cache that is accessed through multiple pipeline stages, but the trade-off between size and latency still exists. Consequently, the DBMSs must improve spatial locality in the instruction stream. Possible techniques include storing together frequently accessed instructions while pushing instructions that are not used that often, like error-handling routines, to different locations.

An additional, somewhat surprising, observation was that increasing data record size increases L1 I-cache misses (and, of course, L1 D-cache misses). It is natural that larger data records would cause both more L1 and L2 data misses. Since the L2 cache is unified, the interference from more L2 data misses could cause more L2 instruction misses. But how do larger data records cause more L1 instruction misses? On certain machines, an explanation would be inclusion (i.e., an L1 cache may only contain blocks present in an L2 cache). Inclusion is often enforced by making L2 cache replacements force L1 cache replacements. Thus, increased L2 interference could lead to more L1 instruction misses. The Xeon processor, however, does not enforce inclusion. Another possible explanation is interference of the NT operating system [19]. NT interrupts the processor periodically for context switching, and upon each interrupt the contents of L1 I-cache are replaced with operating system code. As the DBMS resumes execution, it fetches its instructions back into the L1 I-cache. As the record size varies between 20 and 200 bytes, the execution time per record increases by a factor of 2.5 to 4, depending on the DBMS. Therefore, larger records incur more operating system interrupts and this could explain increased L1 I-cache misses. Finally, a third explanation is that larger records incur more frequent page boundary crossings. Upon each crossing the DBMS executes buffer pool management instructions. However, more experiments are needed to test these hypotheses.

5.3 Branch mispredictions

As was explained in Section 3.2, branch mispredictions have serious performance implications, because (a) they cause a serial bottleneck in the pipeline and (b) they cause instruction cache misses, which in turn incur additional stalls. Branch instructions account for 20% of the total instructions retired in all of the experiments.

Even with our simple workload, three out of the four DBMSs tested suffer significantly from branch misprediction stalls. Branch mispredictions depend upon



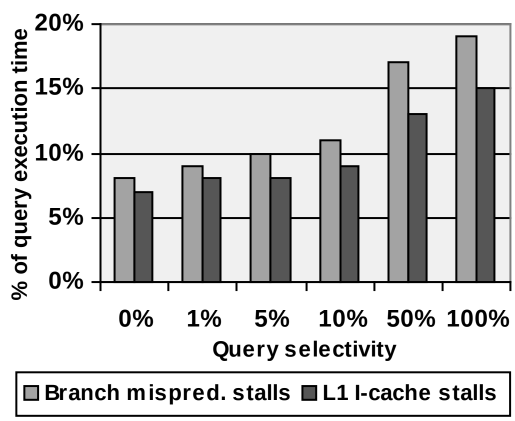


Figure 5.4: *Left: Branch misprediction rates. SRS: sequential selection, IRS: indexed selection, SJ: join. Right: System D running a sequential selection. T_B and T_{L1I} both increase as a function of an increase in the selectivity.*

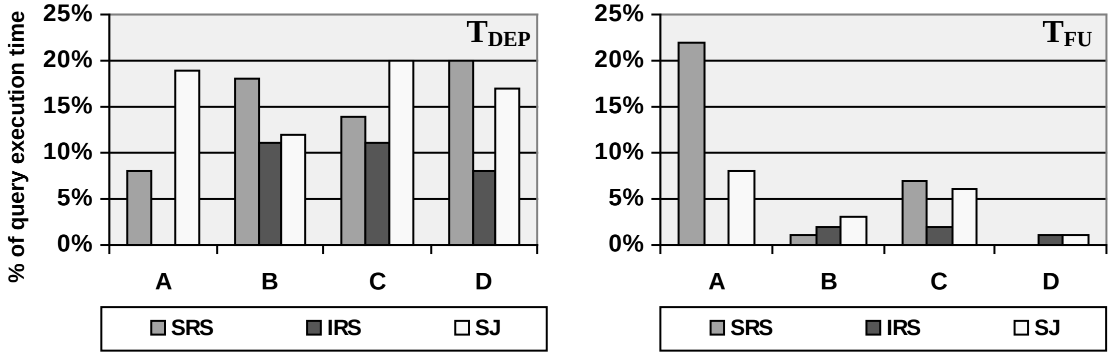


Figure 5.5: *T_{DEP} and T_{FU} contributions to the overall execution time for four DBMSs. SRS: sequential selection, IRS: indexed selection, SJ: join. System A did not use the index in the IRS, therefore this query is excluded from system A's results.*

how accurately the branch prediction algorithm predicts the instruction stream. The branch misprediction rate (number of mispredictions divided by the number of retired branch instructions) does not vary significantly with record size or selectivity in any of the systems. The average rates for all the systems are shown in the left graph of Figure 5.4.

The branch prediction algorithm uses a small buffer, called the Branch Target Buffer (BTB) to store the targets of the last branches executed. A hit in this buffer activates a branch prediction algorithm, which decides which will be the target of the branch based on previous history [20]. On a BTB miss, the prediction is static (backward branch is taken, forward is not taken). In all the experiments the BTB misses 50% of the time on the average (this corroborates previous results for TPC workloads [10]). Consequently, the sophisticated hardware that implements the branch prediction algorithm is only used half of the time. In addition, as the BTB miss rate increases, the branch misprediction rate increases as well. It was shown [7] that a larger BTB (up to 16K entries) improves the BTB miss rate for OLTP workloads.

As mentioned in Section 3.2, branch misprediction stalls are tightly connected to instruction stalls. For the Xeon this connection is tighter, because it uses instruction prefetching. In all of the experiments, T_{L1I} follows the behavior of T_B as a function of variations in the selectivity or record size. The right graph of Figure 5.4 illustrates this for System D running range selection queries with various selectivities. Processors should be able to efficiently execute even unoptimized instruction streams, so a different prediction mechanism could reduce branch misprediction stalls caused by database workloads.

5.4 Resource stalls

Resource-related stall time is the time during which the processor must wait for a resource to become available. Such resources include functional units in the execution stage, registers for handling dependencies between instructions, and other platform-dependent resources. The contribution of resource stalls to the overall execution time is fairly stable across the DBMSs. In all cases, resource stalls are dominated by dependency and/or functional unit stalls.

Figure 5.5 shows the contributions of T_{DEP} and T_{FU} for all systems and queries. Except for System A when executing range selection queries, dependency stalls are the most important resource stalls. Dependency stalls are caused by low instruction-level parallelism opportunity in the instruction pool, i.e., an instruction depends on the results of multiple other instructions that have not yet completed execution. The processor must wait for the dependencies to be resolved in order to continue. Functional unit availability stalls are caused by bursts of instructions that create contention in the execution unit. Memory references account for at least half of the instructions retired, so it is possible that one of the resources causing these stalls is a memory buffer. Resource stalls are an artifact of the lowest-level details of the hardware. The compiler can produce code that avoids resource contention and exploits instruction-level parallelism. This is difficult with the X86 instruction set, because each CISC instruction is internally translated into smaller instructions (μops). Thus, there is no easy way for the compiler to see the correlation across multiple X86 instructions and optimize the instruction stream at the processor execution level.

5.5 Comparison with DSS and OLTP

We executed a TPC-D workload against three out of four of the commercial DBMSs, namely A, B, and D. The workload includes the 17 TPC-D selection queries and a 100-MB database. The results shown represent averages from all the TPC-D queries for each system.

Figure 5.6 shows that the clock-per-instruction breakdown for the sequential range selection query (left) is similar to the breakdown of TPC-D queries (right). The clock-per-instruction (CPI) rate is also similar between the two workloads, ranging between 1.2 and 1.8. A closer look into the memory breakdown (Figure 5.7) shows that

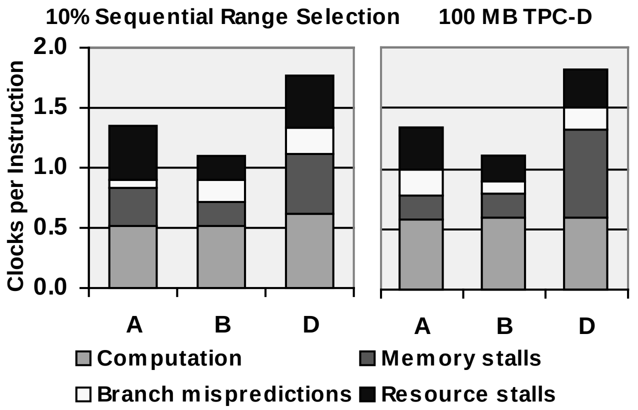


Figure 5.6: *Clocks-per-instruction (CPI) breakdown for A, B, and D running sequential range selection (left) and TPC-D queries (right).*

first-level instruction stalls dominate the TPC-D workload, indicating that complicated decision-support queries will benefit much from instruction cache optimizations.

TPC-C workloads exhibit different behavior than decision-support workloads, both in terms of clocks-per-instruction rates and execution time breakdown. We executed a 10-user, 1-warehouse TPC-C workload against all four DBMSs (results are not shown here due to space restrictions). CPI rates for TPC-C workloads range from 2.5 to 4.5, and 60%-80% of the time is spent in memory-related stalls. Resource stalls are significantly higher for TPC-C than for the other two workloads. The TPC-C memory stalls breakdown shows dominance of the L2 data and instruction stalls, which indicates that the size and architectural characteristics of the second-level cache are even more crucial for OLTP workloads.

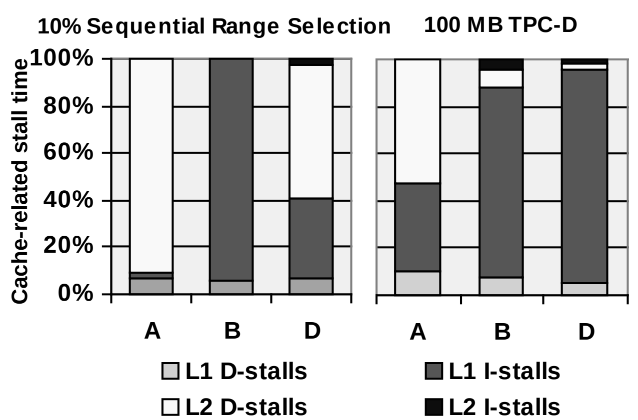


Figure 5.7: *Breakdown of cache-related stall time for A, B, and D, running the sequential range selection (left) and TPC-D queries (right).*

6 Conclusions

Despite the performance optimizations found in today's database systems, they are not able to take full advantage of many recent improvements in processor technology. All studies that have evaluated database workloads use complex TPC benchmarks and consider a single DBMS on a single platform. The variation of platforms and DBMSs and the complexity of the workloads make it difficult to thoroughly understand the hardware behavior from the point of view of the database.

Based on a simple query execution time framework, we analyzed the behavior of four commercial DBMSs running simple selection and join queries on a modern processor and memory architecture. The results from our experiments suggest that database developers should pay more attention to the data layout at the second level data cache, rather than the first, because L2 data stalls are a major component of the query execution time, whereas L1 D-cache stalls are insignificant. In addition, first-level instruction cache misses often dominate memory stalls, thus there should be more focus on optimizing the critical paths for the instruction cache. Performance improvements should address all of the stall components in order to effectively increase the percentage of execution time spent in useful computation. Using simple queries rather than full TPC workloads provides a methodological advantage, because the results are much simpler to analyze. We found that TPC-D execution time breakdown is similar to the breakdown of the simpler query, while TPC-C workloads incur more second-level cache and resource stalls.

7 Future Work

Although database applications are becoming increasingly compute and memory intensive, one must measure the I/O factor as well and determine its effects on the time breakdown. Our experiments did not include I/O, but we intend to study that in the near future.

In addition, we intend to compare the behavior of a prototype system with commercial DBMSs, using the same workloads. With a prototype DBMS we will verify the actual cause of major bottlenecks and evaluate techniques for improving DBMS performance.

8 Acknowledgements

We would like to thank NCR for funding this research through a graduate student fellowship, Intel and Microsoft for donating the hardware and the operating system on which we conducted the experiments for this study. This work is supported in part by the National Science Foundation (MIPS-9625558) and Wisconsin Romnes Fellowships. We would also like to thank Seckin Unlu and Andy Glew for their help with the Pentium II counters and microarchitecture, Kim Keeton for her collaboration on the formulae, Babak Falsafi for his invaluable feedback on the paper, and Miron Livny for his suggestions on how to design high-confidence experiments. Last but not least, we thank Jim Gray, Yannis Ioannidis, Hal Kossman, Paul Larson, Bruce Lindsay, Mikko Lipasti, Michael Parkes, and Don Slutz for their useful comments.

9 References

[1] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of 1997 ACM SIGMOD Conference*, May 1997.

[2] L.A. Barroso, K. Gharachorloo, and E.D. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3-14, June 1998.

[3] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of Programming Languages Design and Implementation '99 (PLDI)*, May 1999.

[4] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu. Evaluation of multithreaded uniprocessors for commercial application environments. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[5] J. Gray. *The benchmark handbook for transaction processing systems*. Morgan-Kaufmann Publishers, Inc., 2nd edition, 1993.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, Inc., 1996, 2ond edition.

[7] R. B. Hilgendorf and G. J. Heim. Evaluating branch prediction methods for an S390 processor using traces from commercial application workloads. Presented at *CAECW'98*, in conjunction with *HPCA-4*, February 1998.

[8] Intel Corporation. Pentium® II processor developer's manual. Intel Corporation, Order number 243502-001, October 1997.

[9] K. Keeton. Personal communication, December 1998.

[10] 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 *Proceedings of the 25th International Symposium on Computer Architecture*, pages 15-26, Barcelona, Spain, June 1998.

[11] P. Trancoso, J.L. Larriba-Pey, Z. Zhang, and J. Torellas. The memory performance of DSS commercial workloads in shared-memory multiprocessors. In *Proceedings of the HPCA conference, 1997*.

[12] P. Å. Larson, and G. Graefe. Memory management during run generation in external sorting. In *Proceedings of the 1998 ACM SIGMOD Conference*, June 1998.

[13] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 39-50, June 1998.

[14] A. M. G. Maynard, C. M. Donelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1994.

[15] C. Nyberg, T. Barklay, Z. Cvetatonic, J. Gray, and D. Lomet. Alphasort: A RISC Machine Sort. In *Proceedings of 1994 ACM SIGMOD Conference*, May 1994.

[16] P. Ranganathan, K. Gharachorloo, S. Adve, and L. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1998.

[17] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20^th^* VLDB Conference, Santiago, Chile, 1994.

[18] S. Unlu. Personal communication, September 1998.

[19] A. Glew. Personal communication, September 1998.

[20] T. Yeh and Y. Patt. Two-level adaptive training branch prediction. In *Proceedings of IEEE Micro-24*, pages 51-61, November 1991.

[21] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 285-298, December 1995.

[22] S. S. Thakkar and M. Sweiger. Performance of an OLTP Application on Symmetry Multiprocessor System. In *Proceedings of the International Symposium on Computer Architecture*, 1990.

[23] K. Diefendorff. Xeon Replaces PentiumPro. In *The Microprocessor Report* 12(9), July 1998.