DBMSs on modern processors: Where does time go?

Anastassia G. Ailamaki David J. DeWitt Mark D. Hill David A. Wood
Department of Computer Science
University of Wisconsin - Madison
{natassa,dewitt,markhill,david}@cs.wisc.edu

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 they 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 executes 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 queues, 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, designed in a sophisticated way for fast computation and memory access. Due to the sophisticated techniques used for hiding I/O latency, DBMSs are becoming compute and memory bound. Although researchers design and evaluate processors using programs much simpler than DBMSs (SPEC, LINPACK), one would hope that more complicated programs such as DBMSs 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 that is on improving 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, which brought the problem to the surface, and motivated the community to further look into it. Each of these studies presents results 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, in their majority, the results of the 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 across database systems, and determine what problems we must work on to make the DBMSs run faster. In addition, to better understand where in the hardware the time goes during query execution, we need an analytic framework that models the work breakdown in terms of execution time.

This is the first paper to analyze, based on an intuitive framework, 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. The conclusion is that, even with these simple queries, almost half of the execution time is spent on stalls. Careful 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 source code for the DBMSs. 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 level two 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).
- Significant stalls (e.g, 20%) 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 workloads provides a methodological advantage, because results can be more simply analyzed and yet are substantially similar to the breakdowns with full benchmarks. To verify this, we implemented and ran the TPC-D benchmark on three out of four systems, and the results are substantially similar to the breakdowns with the simple queries.

The rest of this paper is organized as follows: Section 2 presents a summary of the 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 the breakdown of the execution time and the database workload. Section 4 describes the experimental setup, i.e., the hardware and software used to conduct the experiments. Section 5 presents our results. The conclusions and future directions are contained in Section 6.

2 Related work

Much of the database 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. All of the above studies are targeted to a specific task and concentrate on ways to make it faster.

The first hardware evaluation of a relational DBMS running an 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 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 workload, that DSS workloads benefit more than OLTP from out-of-order processors with increased instruction-level parallelism, and that the 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 it difficult to contrast the DBMSs and identify detailed common behavior.

3 Query execution on modern processors

To better understand what happens during execution of a query, we designed a framework to analyze the hardware behavior of the DBMS from the moment it receives a query until the moment it returns the results. However, the operation of today's processors is too complex to be described in detail with an accurate performance model. In addition, complex queries produce different query plans on different DBMSs, because each product is optimized for a certain type of queries.

In this section, we first introduce a framework that describes the role of the major hardware components in terms of execution time. Then, we describe a workload that allows us to focus on the basic operations of the DBMSs and 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. The operation at one stage can overlap with the operation of any of the other stages.

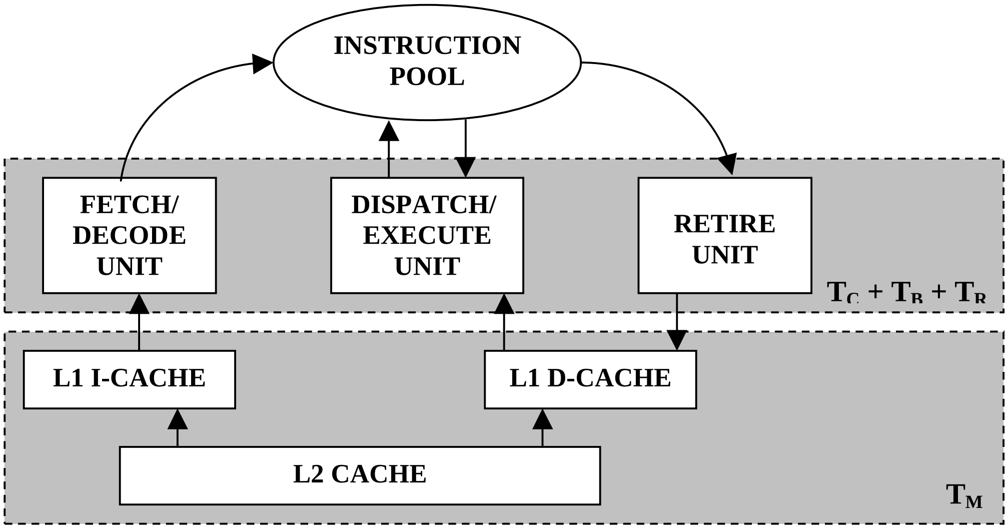

Figure 3.1: *Simplified block diagram of a processor operation*

Figure 3.1 shows a simplified diagram of the major stages of a processor pipeline [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 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 retrieval from the L1 I-cache or the L1 D-cache fails, the request is forwarded to the second-level cache (L2-cache), which is usually unified (used for both data and instructions). If the request fails in L2, it is forwarded to main memory. During the time the retrieval is pending, both caches can process other requests.
- *Out-of-order execution*: If instruction X stalls, an instruction Y which, in the program comes after X, can execute before X if its 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 target address is resolved, an algorithm “guesses” the target 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 misprediction incurs both computation overhead, for computing the wrong instructions, and stall time.

$\mathbf{T_C}$			computation time
$\mathbf{T_M}$	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
$\mathbf{T_R}$	T_{FU}		stall time due to functional unit unavailability
	T_{DEP}		stall time due to dependencies among instructions
	T_{MISC}		platform-specific resource stall time

Table 3.1: *Execution time components*

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) and a stall time because of memory stalls (T_M), branch misprediction overhead (T_B), and resource-related stalls (T_R). The latter are due to unavailability of execution resources, 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:

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

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

3.2 Significance of the stall components

Most of the research on improving DBMS performance aims at reducing T_M, the memory hierarchy stall component. In order to be able to use the experimental results effectively, it is important to determine how significant each of the different types of stalls is to the overall execution time. Although out-of-order and speculative execution help hide some of the stalls, this is not the case for all of them. Some stalls are impossible to overlap, and thus they 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, with parallel requests to main memory. T_{DTLB} misses can be hidden by useful computation as well, but their penalty depends on the page table implementation for each processor. Processors are successfully using sophisticated techniques to overlap data stalls with useful computation.

Instruction-related cache stalls, on the other hand, are impossible to hide because they cause a serial bottleneck. If there are no instructions available, the processor can do nothing but wait. The same serial bottleneck occurs on a branch misprediction; the processor again must wait until the correct instruction stream is loaded into the pipeline. Some processors exploit spatial locality in the instruction stream by using instruction-prefetching hardware. Instruction prefetching effectively reduces the number of I-cache stalls, but 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. T_{DEP} can be hidden depending on the degree of instruction-level parallelism of the program, and T_{FU} can be overlapped by instructions that use functional units with less contention.

3.3 Database workload

There is a large set of workloads to choose from and run on a DBMS to study their performance. The workload used in this study consists of single-table range selections and two table equijoins over a memory resident database, running in single-user mode. 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 studying 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, *Person*, defined as follows:

```
create table Person (pkey      integer not null,
                     age       integer not null,
                     salary    integer not null,
                     <rest of fields> )
```

where <rest of fields> stands for a list of integer fields insignificant to the queries. The relation is populated with 100MB of data, with uniform distribution on the values of the field *age*. The experiments use three basic queries on *Person*:

1. *Sequential range selection*:

   ```
   select avg(salary) from Person where age < Hi and age > 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 the record size and query selectivity. *Hi* and *Lo* define the interval of the qualification attribute, *age*. 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 query selectivity used ranged from 0% to 100%, and was submitted against variations of *Person* with different row sizes (20, 100, and 200 bytes).

2. *Indexed range selection*: The range selection above (1) was resubmitted after constructing a non-clustered index on *Person.age*. The same variations on selectivity and record size were used.

3. *Sequential join*:

   ```
   select avg(Person.salary) from Person, Student where Person.age = Student.age
   ```

   To examine the behavior when executing an equijoin with no indexes, the database schema was augmented by one more relation, *Student,* defined the same way as *Person*. The record size of *Student* is 100 bytes and it was populated with 10 MB of data. The query was submitted on variations of *Person* with different record sizes (20, 100, and 200 bytes).

4 Experimental Setup

We used a 6400 PII Xeon/MT Workstation to conduct all of the experiments. As mentioned in Section 2, many studies use simulation to carry out experiments. Simulation is necessary in order to evaluate architectural alternatives and future designs, but is also very slow. This section describes the hardware and software used, and presents the experimentation methodology.

We use the hardware counters of the Pentium II Xeon processor to run the experiments at full speed, to avoid any approximations the simulation would impose, and to conduct a comparative evaluation of the four DBMSs.

4.1 The hardware platform

The system contains one Pentium II Xeon processor running at 400 MHz, and a 512-MB 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 execution of instructions. 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.

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.

Characteristic	L1 Instruction	L1 Data	L2 (Unified)
Size	16KB	16KB	512KB
Associativity	4-way	4-way	4-way
Miss Penalty	4 cycles (w/ L2 hit)	4 cycles (w/ L2 hit)	Main memory latency
Non-blocking	Yes	Yes	Yes
Misses outstanding	4	4	4
Write Policy	Write-back	Write-back	Write-back

Table 4.1: *Pentium II Xeon cache characteristics*

4.2 The software

Experiments were conducted on four of the most popular 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. I/O effects were excluded from this study, because the objective is to measure pure processor and memory performance. Also, 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. Emon was used to measure 74 events for the data presented in this report. We measured each event in both user and kernel mode.

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 relation, 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.

Stall time component			Description	Measurement method
$\mathbf{T_C}$			computation time	Estimated minimum based on μops retired
$\mathbf{T_M}$	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
$\mathbf{T_B}$			branch misprediction penalty	# branch mispredictions retired * 17 cycles
$\mathbf{T_R}$	T_{FU}		functional unit stalls	actual stall time
	T_{DEP}		dependency stalls	actual stall time
	T_{ILD}		I-length decoder stalls	actual stall time
$\mathbf{T_{OVL}}$			overlap time	Not measured

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

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 deficiencies:

- We were not able to measure T_{DTLB}, because the event code is not available.
- For some of the events, we measured the actual stall time (after any overlaps), while for some others we measured the number of events that occurred during the query execution and then we multiplied by an estimated penalty [18][19]. Measurements of the memory subsystem showed that the workload is latency-bound, rather than bandwidth-bound, because it rarely uses more than a third of the available memory bandwidth. Thus, the results that use penalty approximations are fairly accurate, because the

[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].

probability of queueing of requests in memory is minimal. Table 4.2 shows a detailed list of stall time components and the way they were measured.

- No contention conditions were taken into account.

The Xeon has one more type of stall, T_{ILD}, for stalls related to the instruction length decoder during translation of X86 instructions to μops.

5 Results

We executed the workload presented 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. 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 otherwise stated.

5.1 Execution time breakdown

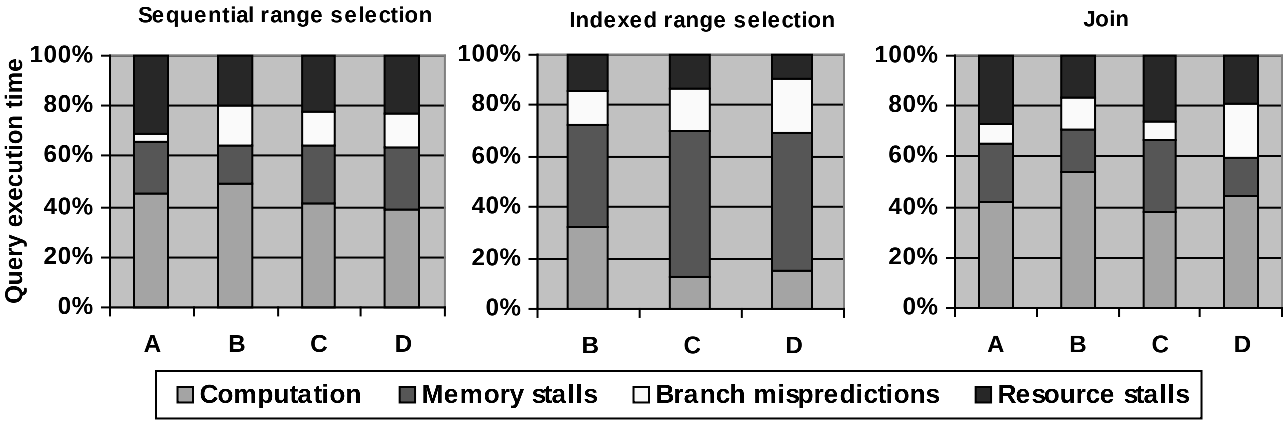

Figure 5.1: *Average query execution time breakdown into the four time components.*

Figure 5.1 shows three graphs, each summarizing the average execution time breakdown for each 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 analyzing the query execution time. Even as processor clock rates increase, stall times will not decrease, thus the computation component will be a much smaller fraction of the execution time.

Memory stall time contribution varies the most, across both different queries and different database systems. Memory stalls vary across queries because they are highly dependent on the workload. The sequential range selection and join queries cause fewer memory stalls than the indexed range selection. A plausible explanation is that the use of the index increases the data and instruction footprint of the execution. The memory component exhibits the highest variation across the DBMSs as well, because some systems optimize for memory stalls on the specific platform, while others don't. However, the analysis of the memory behavior yields that L2 D-cache stalls and L1 I-cache stalls account for 90% of T_M 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.

Minimizing memory stalls has been the major focus of database research on performance improvement. Although in most cases the memory stall time (T_M) accounts for the 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 least 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 is expected, since optimizing for two kinds of stalls shifts 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, the main target of database researchers in the past has been to minimize the stall time due to latencies caused by the memory hierarchy [1][12][15][17]. Several techniques for

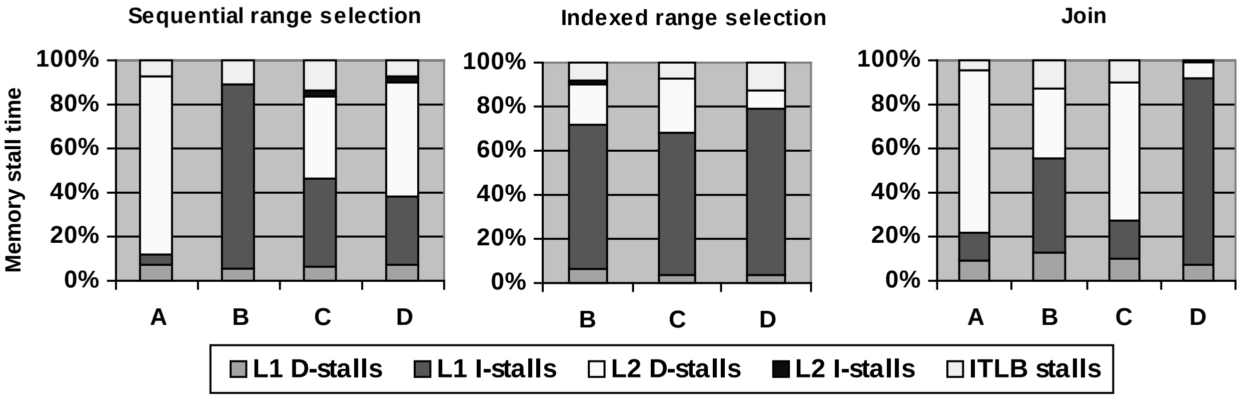

Figure 5.2: *Contributions of the five memory components to the memory stall time*

cache-conscious data placement have been proposed [3] to reduce cache misses and miss penalty. Although these techniques are successful within the context in which they were proposed, a closer look to 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} (L1 D-cache 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 obvious 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. 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%. The L1 D-cache miss rates are low because, as the DBMS executes the query, it accesses private data structures more often than it accesses the data in the relations [11]. This often-accessed portion of data fits into the L1 D-cache, and the only misses are due to less often accessed data. Thus, the L1 D-cache is not a performance bottleneck for any of the DBMSs we evaluated.

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 (40% to 90%) is typically much higher than the L1 D-cache miss rate. The only exception is System B, which optimizes data layout and access for the second cache level as well. In the case of the sequential range query, System B exhibits an L2 data miss rate of only 2%, and this results in an insignificant T_{L2D}.

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 less than our estimation because the workload is bound by memory latency, rather than bandwidth (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 would expect data access to the L2 cache to become a major bottleneck for latency-bound workloads. Fortunately, the size of today's L2 caches has increased to 4MB, and continues to increase. The Pentium II Xeon on which the experiments were conducted can have an L2 cache up to 2 MB (although the experiments were conducted with a 512 KB L2 cache).

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 impossible to hide, because L1 I-cache misses cause a serial bottleneck to the pipeline. System A is the only DBMS for which T_{L1I} is insignificant. System A has the smallest instruction footprint of the four systems for the queries executed, and exhibits optimized instruction cache behavior that minimizes T_{L1I} to at most 5% of the overall execution time. The rest of the systems do not optimize for the instruction cache. Depending on the type of the query and the DBMS, T_{L1I} accounts for 4% up to 40% of the total execution time, while for all the DBMSs, the average contribution of T_{L1I} to the execution time is 20%. 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 will not increase in the future 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 which is accessed through more than one pipeline stages, but the trade-off between size and speed still exists. Consequently, the DBMSs must be optimized to use the L1 I-cache as effectively as possible.

It may be possible to reduce the L1 I-cache stalls by optimizing the data placement at the second level cache. In most processors, caches obey the inclusion rule: the information stored at the L1 cache at any time is a subset of the information stored in the L2 cache. Systems that observe inclusion maintain consistency by only searching the lowest memory hierarchy level; if information is consistent there, it will also be consistent at the higher levels. On the other hand, whenever new data invalidates old information in the L2 cache, the same information is invalidated in the L1 cache. This information may be data or instructions because the L2 cache is unified. Thus, more invalidated information in the L2 cache is more likely to invalidate instructions in the L1 I-cache, causing more stalls in order to bring them back in. We verified this hypothesis by running the sequential range selection with variable record sizes. The two fields involved in the query, *salary* and *age*, are always in the beginning of the records. Records are stored sequentially, and as the record size increases, the *salary* and *age* fields of subsequent tuples are further apart. This results in invalidating more L2 cache lines, some of which contain instructions, and T_{L1I} increases. Consequently, one way of reducing T_{L1I} would be to use processors with caches that do not observe the inclusion rule, and hence do not cause undesired instruction invalidations. On processors that use inclusion like the Xeon, T_{L1I}

can still be reduced by partitioning the L2 cache into a section for exclusive instruction use and another for exclusive data use, hence avoiding interference between instructions and data.

The stall time caused by L2 cache instruction misses (T_{L2I}) and ITLB misses (T_{ITLB}) is insignificant in all the experiments. T_{L2I} is low because the second-level cache misses are very few. The low T_{ITLB} indicates that the systems use few instruction pages, and the ITLB is enough to store the translations for their addresses.

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.

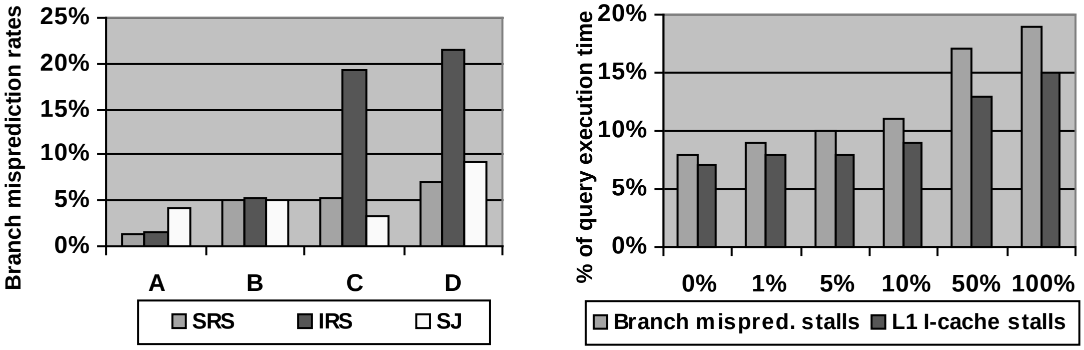

Figure 5.3: *Left: Branch misprediction rates. SRS: sequential selection, IRS: indexed selection, SJ: join. Right: System D running a sequential selection. T_B and T_{L1I} both increase as a function of an increase in the selectivity.*

Even with our simple workload, three out of four DBMSs suffer from branch misprediction stalls. Branch mispredictions depend upon 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 5.3.

The branch misprediction 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 (forward branch is taken, backward 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 5.3 illustrates this for System D running range selection queries with various selectivities. Database developers can help reduce the instruction cache stalls by writing the critical sections (e.g., the inner loops) in a language that does not perform as many jumps as object oriented languages do. Then, the compiler can produce code that is more predictable. On the other hand, processors should be able to efficiently execute even unoptimized instruction streams, so a different prediction mechanism could reduce these stalls.

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.

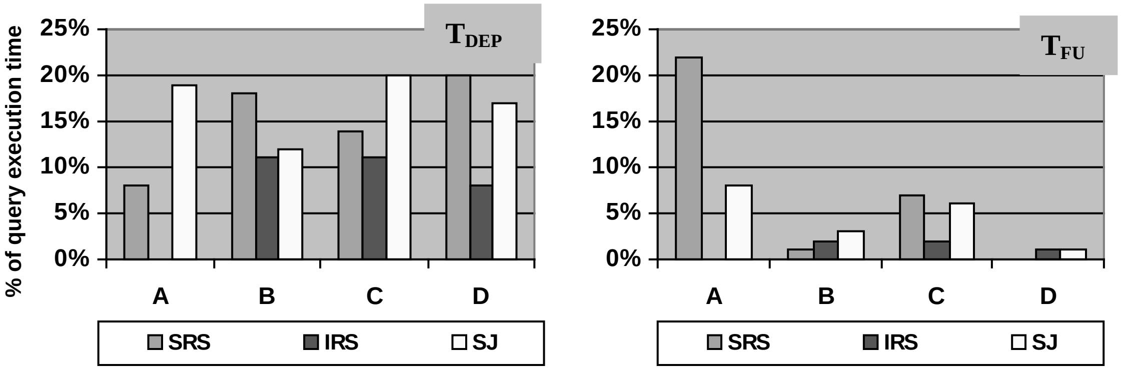

Figure 5.4: *T_{DEP} and T_{FU} contributions to the overall execution time for four DBMSs. SRS: sequential selection, IRS: indexed selection, SJ: join. System A did not use the index in the IRS, therefore this query is excluded from system A's results.*

Figure 5.4 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 correlations across multiple X86 instructions and optimize the instruction stream at the processor execution level.

6 Conclusions

Despite the performance optimizations found in today's database systems, they are not able to take advantage of many of the 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 impossible 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. Using simple queries rather than full TPC workloads provides a methodological advantage, because results can be more simply analyzed. We implemented and ran the TPC-D benchmark on Systems A, B, and D, and the results are substantially similar to the breakdowns with the simple queries. 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, memory stalls are often dominated by first-level instruction cache misses, thus there should be more focus on optimizing the critical paths for the instruction cache. Finally, performance improvements should address all of the stall components in order to effectively increase the percentage of execution time spent in useful computation.

7 Acknowledgements

We would like to thank Intel and Microsoft for donating the hardware and the operating system on which we conducted the experiments for this study. We would also like to thank Seckin Unlu and Andy Glew for their help with the Pentium II counters and microarchitecture, Babak Falsafi for his valuable feedback on the paper, and Miron Livny for his suggestions on how to design high-confidence experiments.

8 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 20th* 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.