Cache Optimization for Memory-Resident Decision Support Commercial Workloads[1]

Pedro Trancoso[2] and **Josep Torrellas**

Department of Computer Science
University of Illinois at Urbana-Champaign, IL 61801
trancoso@intercol.edu torrellas@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

Abstract

Dramatic increases in the main-memory size of computers is allowing some applications to shift their main data storage area from disk to main memory and, as a result, increase their performance. This trend is at work in some databases, resulting in what is called memory-resident databases. However, because of the increasing gap between processor and main memory speed, in these systems, effective use of the cache hierarchy is crucial to high performance. Unfortunately, there has been relatively little work on building cache-friendly database systems.

In this paper, we present several cache-oriented optimizations to enable effective exploitation of caches in memory-resident decision support databases. The main optimization involves developing a query optimizer that includes the cost of cache misses in its cost metrics. The other optimizations are sophisticated data blocking and software prefetching. These optimizations require no custom-designed hardware support and are effective for the more complicated TPC-D queries. In a simple database, these queries run about 13% faster with the cache-oriented optimizer and blocking, and a total of 31% faster if, in addition, we add prefetching. The effectiveness of these optimizations is stable across a range of cache sizes, cache line sizes, and miss penalties.

1 Introduction

Dramatic advances in semiconductor technology continue to increase the integration level of DRAM memory. According to the Semiconductor Industry Association Technology Roadmap [12], in three years from now, memory density will have increased by a factor of 4 to allow the design of 4-Gbit DRAM chips, while the memory price will have decreased by a factor close to 3. In addition, by that time, most microprocessors will be using 64 bits, which will enable huge address spaces. These trends combined will enable many applications to shift their main data storage area from disk to main memory with performance benefits.

One class of workloads that is likely to be affected by these trends is databases. Indeed, although database sizes are also increasing, it is likely that an increasing number of databases will be able to keep large chunks of their working set resident in memory.

Currently, most database systems have the built-in assumption that the data resides on disk. These are called Disk-Resident Database (DRDB) systems. While a large main memory improves the performance of DRDB systems, a study by Garcia-Molina *et al* [3] suggested that the disk-resident assumption inhibits higher performance. It is necessary to redesign the database to fully exploit main-memory residence. The resulting systems are called Memory-Resident Database (MRDB) systems. MRDB systems have been studied by several researchers [5, 21] and have even been successfully implemented in a commercial system called TimesTen Performance Software [14].

The MRDB systems proposed assume a uniform memory access latency. Current high-performance machines, however, include a deep cache hierarchy that makes the latency of an access to the memory system highly variable. Furthermore, it is widely-known that effective exploitation of the cache hierarchy is often key to application performance. The work of Trancoso *et al* [17] and Barroso *et al* [1] shows that this is the case even for current DRDB systems when the machine has enough disk and I/O hardware not to be I/O-bound.

Surprisingly, little attention has been devoted to devising cache-friendly database algorithms. To our knowledge, work has been largely limited to the new sorting algorithm by Nyberg *et al* [9] and the data blocking optimizations for joins and aggregates by Shatdal *et al* [13]. These contributions all focus on optimizing a single operation, even though database queries are typically composed of multiple operations that are highly time-interleaved.

In this paper, we present several cache-oriented optimizations to enable effective exploitation of caches in memory-resident decision-support databases. Traditional, DRDB systems should also benefit if they are not completely I/O-bound. The main optimization involves developing a new query optimizer that includes the cost of cache misses in its cost metrics. The other optimizations are sophisticated data blocking and software prefetching. These optimizations require no custom-designed hardware support and are effective for the more complicated TPC-D queries. In a simple database, these queries run about 13% faster with the cache-oriented optimizer and blocking, and a total of 31% faster if, in addition, we add prefetching.

This paper is organized as follows: Section 2 introduces background concepts; Section 3 presents our proposed cache-oriented optimizations; Section 4 describes the setup for the experiments; and Section 5 presents the results of the experiments.

2 Background Concepts

2.1 Database Basics

A database management system (DBMS) is a system that controls the storage of data and the execution of requests to it [2]. Each request to a DBMS is called a *query*. In a relational system, the data is stored in *tables*. Each table is composed of records or *tuples*, and each tuple includes different *attributes* of certain types. Queries are usually written in

[1]This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, ASC-9612099, and MIP-9619351, DARPA Contract DABT63-95-C-0097, NASA Contract NAG-1-613, and gifts from Intel and IBM. Pedro Trancoso was also supported by the Portuguese government under scholarship JNICT PRAXIS XXI/BD/5877/95.

[2]Now at Intercollege Limassol, Cyprus.

Structured Query Language (SQL). In a DBMS, the execution of a query involves several steps (Figure 1-(a)). First, when the query is submitted to a DBMS, it is parsed in order to check for correctness. Second, the query is decomposed into a set of *abstract operations* (Section 2.2). Each of these operations can be implemented by a set of different algorithms. The third step is to *optimize* the query execution by choosing both the order of execution of the operations and the algorithm for each operation. The output of this step is a *query plan* or *query tree* (for example, Figure 1-(b)). These three steps are often known as *query compilation*. The last step of the query execution is to traverse the query plan and execute the algorithm in each of the nodes. At the end, the results of the execution are passed to the application.

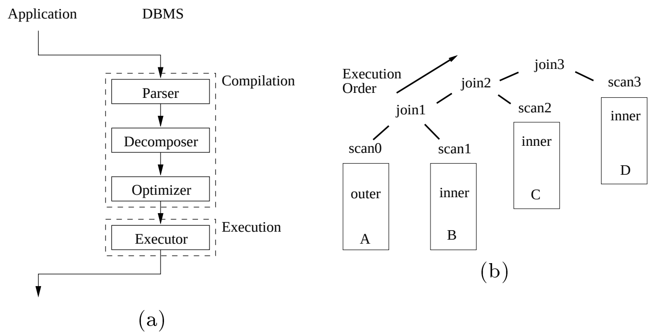


Figure 1: Steps in the execution of a query (a) and example of a query plan (b).

2.2 Abstract Operations and Algorithms

Some of the most common abstract operations are *scan* (or *select*), *join*, *sort*, *group* and *aggregate*. In this work, only *pipelined* operations are considered. This means that, once an operation generates a result tuple, the flow of control passes to the operation at the next higher level of the query tree, so that the tuple can be processed immediately. This is done to reduce the time taken by the query to generate its first result, and to reduce the size of temporary storage between different operations.

The scan operation retrieves tuples from a table. If the tuple satisfies the *scan condition*, it is selected; otherwise it is discarded. There are two implementations of this operation: *Sequential Scan* and *Index Scan*. The first one simply traverses all the tuples in the table sequentially, selecting those that satisfy the condition. The second one uses a special data structure, often tree-based, called the *index* to directly access only those tuples that satisfy the scan condition.

The join operation joins together information from two tables into a new, result table. A result tuple is formed out of two tuples, one from each input table, when their join attributes satisfy the *join condition*. The most common algorithms for join are: *Nested Loop Join*, *Index Nested Loop Join*, and *Hash Join*. The *Nested Loop Join* (for example, *join2* in Figure 1-(b)) starts its execution by requesting a tuple from the lower level operation (in our example, *join1*). Then, the algorithm performs a sequential scan on the inner table (in our example, table *C*), attempting to find a matching tuple, namely one with a matching value in the join attribute and that satisfies the scan condition. If a match is found, a result tuple is formed and is returned to the operation at the higher level (in our example, *join3*). This process is repeated for all output tuples from the lower level operation (*join1*). A variation of this algorithm is the *Index Nested Loop Join*. In this case, an index structure is used to scan the inner table for a tuple with the matching value in the join attribute. The *Hash Join* algorithm starts by building a hash table out of the tuples in its inner table (build phase). Then, the algorithm requests tuples from the operation at the lower level and uses them to probe the hash table to find matching tuples (probe phase).

Finally, the *sort* operation sorts tuples, the *group* operation groups together tuples with a common characteristic, and the *aggregate* operation accumulates over all the tuples [2].

3 Cache-Oriented Optimizations

To help decision support queries exploit cache hierarchies, we design several cache-oriented optimizations. The first two optimizations, namely *data blocking* and *software prefetching*, target the algorithms for the abstract operations, while the third one, *cache-aware optimizer*, targets the query optimizer.

3.1 Data Blocking

Data blocking consists of reorganizing the order of execution of loop iterations so that the data is accessed in highly-reused tiles or blocks [4]. Blocking has been applied to scientific codes to improve the use of the caches [7]. In DRDB systems, it has been used to exploit the main memory better and, therefore, reduce the number of disk accesses. In addition, Shatdal *et al* [13] have recently applied blocking for the cache to single operations in DRDB systems. For example, they applied blocking to a single nested loop join or a single hash join. Unfortunately, as indicated before, database queries are executed in a pipelined manner. Blocking one operation may speed that operation up, but it also inhibits pipelining, delaying the generation of the first query result.

We extend Shatdal *et al*'s work and block pipelined query trees. Our algorithm works as follows. Each operation in the tree takes as input a block of tuples coming from the immediately lower level of the tree. The operation generates some output tuples. If the aggregate size of the output tuples is larger than a block, control is passed to the operation immediately higher in the tree to process that block. Otherwise, another input block is processed and its output is added to the output of the previous block. The process is repeated until a full output block is generated. At that point, the output block is passed to the next level. Overall, therefore, operations pass blocks of tuples to each other.

The use of such coarse-grain pipelining between operations tends to delay the generation of the first query result. However, by operating on blocks of data, we reuse cached data better, suffer fewer cache misses and, as a result, execute the overall query faster.

The data reuse can occur in three ways, depending on whether the data is accessed sequentially, through indices, or in a hash table. When the tuples are accessed sequentially and there are repeated accesses to the same datum, we block the algorithm. An example is the *Nested Loop Join*, where each tuple in one table is checked against every tuple in another table. Figure 2 shows the pseudo-code for the *Blocked Nested Loop Join* algorithm. In the algorithm, we work on one input block of size at least S_i at a time. The block is received from the operation in the immediately lower level of the tree via the *GetBlockFromLower()* function. The block is kept in the cache and reused as we read in each element of table B. Once we have a block of output of size at least S_j, it is passed to the next level of the tree with *PassBlockToHigher()*. The sizes of the blocks will be discussed in Section 3.3.

The result of blocking is a significant miss reduction. Suppose that we are joining two large tables A and B. If l_a, l_b,

```
while( GetBlockFromLower(block_i) )
  /* sizeof(*block_i) is >= Si */
  while( tuple_b = scan(B) )
    while( tuple_i from block_i )
      if( match( tuple_i, tuple_b ) )
        tuple_j = project( tuple_i, tuple_b )
        add( tuple_j, block_j )
  if( sizeof(*block_j) >= Sj )
    PassBlockToHigher(block_j)
```

Figure 2: *Blocked Nested Loop Join* algorithm.

and S_a are the number of cache lines in A, B, and an input block from A respectively, the number of cache misses decreases from $l_b * l_a + l_a$ to $l_b * l_a / S_a + l_a$.

When the data is accessed through indices, tuples are unlikely to be reused. However, we are likely to reuse parts of the index structure. Specifically, given a tree-based index organization, we are likely to reuse the nodes close to the root of the tree. This occurs in the *Index Scan* and *Index Nested Loop Join*. Consequently, we use blocking to keep such reused data in the cache.

Blocking in these two algorithms is accomplished by simply invoking the operation several times in a row for different tuples. Some of the index nodes will remain in the cache across invocations. For example, the *Blocked Index Nested Loop Join* is shown in Figure 3. As in the blocked nested loop join, the algorithm starts by getting a block of input tuples from the lower level of the tree. These tuples, however, will not be reused. For each tuple, the index structure will be accessed to find the matching tuple in table B. Every time that we select a new tuple in block_i, we access the index structure, possibly reusing some of its nodes. The size of the input block, therefore, determines how many times the indices are reused. Finally, the algorithm passes control to the next level of the tree when an output block has been generated.

```
while( GetBlockFromLower(block_i) )
  while( tuple_i from block_i )
    tuple_b = index_scan( B, tuple_i )
    if( match( tuple_i, tuple_b ) )
      tuple_j = project( tuple_i, tuple_b )
      add( tuple_j, block_j )
  if( sizeof(*block_j) >= Sj )
    PassBlockToHigher(block_j)
```

Figure 3: *Blocked Index Nested Loop Join* algorithm.

Finally, in a *Blocked Hash Join*, a block of tuples received from the lower level of the query tree are probed into a hashed form of table B. The algorithm is similar to the *Blocked Index Nested Loop Join*, where the index structure is now the hash table structure. However, since the hash table is dimensioned to minimize hash conflicts, the data reuse in the hash structure is minimal.

3.2 Software Prefetching

With *software prefetching*, data is requested before it is actually needed. The goal is to hide the latency of the access to the slower levels of the storage hierarchy. If prefetching is successful, by the time the data is needed, the data will be present in the faster levels of the storage hierarchy.

Many researchers have examined software prefetching techniques and algorithms to prefetch data into caches. These algorithms usually work at the cache-line grain size, and have been usually applied to scientific and engineering applications (for example [8, 10]). In DBMS systems, prefetching is typically used to bring data from the disk into memory. Prefetching is performed to hide the latency of disk accesses and is typically done at the disk-block grain size.

Our approach to improve the use of caches in DBMS systems is to use software prefetching to prefetch cache lines from memory to the cache. As in Section 3.1, we consider three types of access patterns, namely sequential, through indices, and through a hash table.

Sequential accesses to tuples are amenable to prefetching if the accesses happen in rapid time succession. This occurs in the three blocked algorithms of Section 3.1 when they read the block of tuples coming from the lower level of the tree. It also occurs in the *Blocked Nested Loop Join* when it accesses the tuples of the table on the right (table B). Finally, such accesses also occur when a *Sequential Scan* is executed on the left table of the lowest join in the tree.

Prefetching tuples accessed sequentially is easy because the access is predictable. We simply insert a prefetch statement in the loop that, in iteration i, prefetches the tuple to be used in iteration *i+PrefOffset*. As indicated in [8], *PrefOffset* is the ratio between the latency of a memory access and the latency of the miss-free execution of one iteration of the loop. Appropriate prologues and epilogues are added to the loop [8].

If these algorithms are not blocked, then prefetching is less effective because an operation returns control to the next level up in the query tree as soon as one output tuple is found. At that point, all the in-flight prefetches are wasted. Consequently, we modify the sequential accesses in non-blocked algorithms so that, once an output tuple is found, the algorithm continues accessing and checking the next set of tuples for *PrefOffset* iterations. This is done to use up all the in-flight prefetches for these iterations. During these *PrefOffset* iterations, however, no new prefetches are issued. The core code for such a prefetching code is shown in Figure 4-(a). This algorithm is used when the *Nested Loop Join* accesses the tuples of the table on the right (table B) and when a *Sequential Scan* is executed on the left table of the lowest join in the tree. Note that this algorithm needs an output buffer because it may actually generate more than one output tuple before calling the the operation in the next higher level of the tree.

```
...
for( i = 0; i < size(A) - PrefOffset; i++ )
  PREFETCH( A[ i + PrefOffset ] )
  if( matching( A[ i ] ) )
    /* Match! */
    tuple_x = project( A[ i ] )
    add( tuple_x, block_x )
    /* block_x is a buffer */
    for( j = i + 1; j < i + 1 + PrefOffset; j++ )
      if( matching( A[ j ] ) )
        tuple_x = project( A[ j ] )
        add( tuple_x, block_x )
    return( block_x )
...
```

(a)

```
min = 0; max = index_size;
value = attribute( tuple_in );
PREFETCH( index[ (max-min) / 2 ].value );
while( min < max )
  mid = ( max - min ) / 2;
  PREFETCH( index[ (mid-min) / 2 ].value );
  PREFETCH( index[ (max-mid) / 2 ].value );
  if( value < index[ mid ].value )
    max = mid;
  else if( value > index[ mid ].value )
    min = mid;
  else
    tuple_out = project( index[ mid ].tuple );
    return( tuple_out );
```

(b)

Figure 4: Two examples of software prefetching algorithms: prefetching for non-blocked sequential accesses (a) and prefetching for blocked or non-blocked index accesses (b).

For tuples accessed through indices, prefetching is harder. Consider, for example, a tree-based index structure. We can prefetch index nodes before we need them. Unfortunately, for a given tuple search, it is unknown what path within the index structure we will follow. Therefore, we must prefetch for several possible paths. In this environment, it is important to compute the addresses of the index nodes with low overhead. One way to do this, is to use the organization proposed by Whang and Krishnamurthy [21], where the addresses are found with shift operations.

Figure 4-(b) shows the pseudo-code for an algorithm that prefetches indices one iteration in advance for an index structure organized as Whang and Krishnamurthy's. The prefetching follows two possible paths. We use this algorithm for the right table (table B) of the *Index Nested Loop Join*. We also use it for an *Index Scan* on the left table of the lowest join in the tree. This algorithm is applied to both non-blocked and blocked algorithms. In blocked algorithms, the algorithm in Figure 4-(b) is simply called several times in a row for the same operation.

Finally, prefetching for *Hash Join* is very similar to prefetching for the *Index Nested Loop Join*, where the hash table structure is similar to the index structure. Unfortunately, it is hard to prefetch an efficient, conflict-free hash table early enough. Consequently, we do not use prefetching for *Hash Join* or *Blocked Hash Join*.

3.3 Cache-Aware Optimizer

The third optimization proposed is to improve the selection of the algorithms for the abstract operations in the query tree, and the order of such operations in the tree. For this, we propose a cache-aware query optimizer.

In a typical cost-driven optimizer, there is a *cost model* for each combination of algorithm and operation supported by the system. This cost model is an analytical function that, given a set of inputs like the size of the data tables, returns a value that represents the cost of that algorithm/operation combination according to a certain metric. The metric is usually the predicted execution time. Given a query tree, with a certain ordering of operations and using a certain algorithm for each operation, the optimizer adds up all the costs to produce a cost for the whole query tree. As a result, we can classify the query trees according to their estimated cost.

For a DRDB system, the cost model includes two factors: the number of I/O operations (*IO*) and the amount of CPU activity (*CPU*). Because of the large I/O access latency, the number of I/O operations is the dominant factor. Examples of optimizers that use such cost models are those in System R [11] and Postgres [15]. The latter is used as the baseline for this study. For a MRDB system, the cost models also include the number of I/O operations (*IO*) and the amount of CPU activity (*CPU*). However, the cost due to CPU activity is clearly the dominant factor of the cost. The number of I/O operations can be largely neglected, since it accounts for just the initial loading of the data into memory, and is independent of the algorithm used. One system that has an optimizer with such characteristics is OBE [21].

In modern architectures with cache memories, however, the cost models described above for MRDB systems are not enough. Such models ignore memory access costs. It is well known that, in modern architectures, stall time due to memory accesses may account for a large fraction of the execution time. Furthermore, memory accesses do not all have the same cost. Cache hits have a much smaller cost than cache misses. Therefore, we propose a cache-aware cost model that differs from a plain MRDB cost model in that it includes an extra factor, namely the cost of memory accesses due to cache misses (*MISS*).

Our overall cost model has a *MISS* and a *CPU* component, but no *IO* component. The components are computed as follows. We compute the *MISS* component for a given algorithm by analyzing its memory access patterns and predicting the number of misses. The actual computation of the *MISS* component for the different cases is presented in Section 3.3.1 for non-blocked algorithms and in Section 3.3.2 for blocked algorithms. The *CPU* component is computed as in OBE [21], by profiling the execution of the algorithm to determine the number of instructions executed. Finally, we give a different weight to each of the two factors, for a final cost model of the form $w_{MISS} * MISS + w_{CPU} * CPU$. In this formula, w_{MISS} is set to the average latency of a cache miss, while w_{CPU} is set to the average CPI of the processor with a perfect memory system. These two parameters are set according to architecture specifications.

It could be argued that, conceptually, there is little difference between our cache-aware query optimizer, which uses *CPU* and *MISS* cost components, and a traditional DRDB query optimizer, which uses *CPU* and *IO* cost components. After all, the cache-aware query optimizer tries to keep a subset of the memory data in the cache, while the DRDB query optimizer tries to keep a subset of the disk data in the memory.

There are, however, significant differences between the two systems. The three major differences are that the ratio between disk and memory access time is much higher than the ratio between memory and cache access time; that the granularity of the data retrieved from disk (disk block) is much coarser than the one of the data retrieved from memory (memory line); and that the cache space is managed in hardware, while the memory space is managed in software.

The first difference implies that, due to the smaller latency ratio, the cache-oriented system may after all tolerate some cache misses for certain algorithms. The second difference implies that, for a disk-resident system to obtain good performance, spatial locality must be exploited well, while this is less necessary in a system with caches. Finally, the third difference implies that, because the cache-oriented system does not have full software control of what resides in the cache at any time, there is the need to be more careful with the use of the cache space in the cache-oriented system. Overall, these three differences imply that our cache-aware query optimizer and a typical DRDB query optimizer will often choose different optimal trees for a given query.

In the following two sections, we give a high-level overview of how we estimate the number of cache read misses in each algorithm. A more detailed discussion is presented in [16]. In our analysis, we assume that no table can completely fit in cache. In addition, we only show the number of misses involved in completing the whole query. To get only a fraction of the query results, non-blocked algorithms suffer on average the same fraction of total misses; for blocked algorithms, the fraction of total misses is on average a bit higher than the fraction of results because the finest grain size in these algorithms is a block of tuples [16].

3.3.1 MISS Models for Non-Blocked Algorithms

The first five rows in Table 1 give a high-level notion of the estimated read misses for each non-blocking algorithm. The analysis is made easier by the fact that these algorithms hardly exploit any cached data reuse across significant time periods. For the *Sequential Scan*, the misses are decomposed into those suffered while reading the selection attribute for each scanned tuple (*read_input*), and those suffered while forming an output tuple every time we detect a match (*copy_output*). The latter consist of reading the tuple attribute that will be used as join attribute in the next join of the query tree, and reading a pointer to the tuple itself. These two pieces of information will be passed, for each matching tu-

Algorithm	Number of Read Misses
Sequential Scan	$read\_input + copy\_output$
Index Scan	$read\_index + read\_input\_element + copy\_output$
Nested Loop Join	$read\_outer\_input + outer\_size \times sequential\_scan + copy\_output$
Index Nested Loop Join	$read\_outer\_input + outer\_size \times index\_scan + copy\_output$
Hash Join	$build\_hash + read\_outer\_input + outer\_size \times probe\_hash + copy\_output$
Blocked Index Scan	$read\_index^* + read\_input\_element + copy\_output + cross + self$
Blocked Nested Loop Join	$read\_outer\_input + nblocks \times sequential\_scan + copy\_output + cross + self$
Blocked Index Nested Loop Join	$read\_outer\_input + outer\_size \times blocked\_index\_scan + copy\_output$

Table 1: Modeling the number of read misses for the pipelined non-blocked and blocked algorithms. The sequential scan and hash join algorithms do not have any data reuse and, therefore, do not save any misses over their respective non-blocked algorithms.

ple, to the next level of the query tree. More details can be found in [16].

For the *Index Scan*, the misses are decomposed into those recorded while traversing the index (*read_index*), reading the selection attribute of the tuples found in the leaf (*read_input_element*), and forming the output tuples for them (*copy_output*).

In the *Nested Loop Join*, we receive a total of *outer_size* tuples from the left input to the join in the query tree (Figure 1-(b)). Reading these tuples, one at a time, involves *read_outer_input* misses. For each of these tuples, we perform a sequential scan on the inner table of the join, recording a grand total of $outer\_size \times sequential\_scan$ misses. Within the *sequential_scan* term, the *copy_output* misses correspond to selecting the tuples that match the selection condition. Among these, some tuples may additionally match the join condition. For each of those, we generate an output tuple. This operation involves reading a pointer to each of the two tuples to be joined, and reading the tuple attribute that will be used as join attribute in the next join. The combined misses for this last step are the extra *copy_output*.

The misses for *Index Nested Loop Join* are found like those of *Nested Loop Join* except that we incorporate the *index_scan* misses instead of the *sequential_scan* ones.

Finally, the misses in the *Hash Join* are the same as in the *Index Nested Loop Join* except that, instead of recording *index_scan* misses every time we access the inner table of the join, we now record *probe_hash* misses every time we probe an element in the hash table for the inner table of the join. In addition, we record *build_hash* misses to build the hash table in the first place [16].

3.3.2 MISS Models for Blocked Algorithms

To estimate the number of misses in the blocked algorithms, we use a model similar to that of Lam *et al* [7]. In a blocked operation, there are three types of misses: those necessary for the execution of the operation (*Basic*), those induced on the reused block by interference with other data (*Cross*) and those induced on the reused block by self interference (*Self*). *Basic* misses include start-up misses plus misses on reused, non-block data that is displaced from the cache due to the large size of the tables or indices.

The estimated misses for the three major blocked algorithms are shown in the last three rows in Table 1. Consider the *Blocked Nested Loop Join* first. The *Basic* misses include several components. First, we miss while reading in all the tuples coming from the left input of the join operation (*read_outer_input*). These tuples are read in *nblocks* blocks. Every read is likely to cause a miss because the lower left operation in the query tree has likely flushed the cache. Second, for each of the *nblocks* blocks that we read in, we record *sequential_scan* misses while scanning the inner input table trying to find matches. Finally, the *Basic* misses also include the ones recorded when generating the output block of tuples (*copy_output*).

The *cross* misses are induced on the blocks by accesses to the inner input table of the join and to the output tuples. These accesses may displace data from the reused block. Since we do not know the exact mapping of all these data structures, we can only estimate how many of these accesses cause a *cross* miss. We assume that these accesses can touch any location in the cache with uniform probability. Knowing the fraction of the cache that is utilized by the reused block, we can estimate how many misses these accesses cause [16]. Finally, the *self* misses are zero because the reused block is smaller than the cache.

The complete formula for the number of misses for the *Blocked Nested Loop Join* ends up being a function of the size of the reused input block. Consequently, we choose the block size that minimizes the number of misses [16].

The misses for one scan in a *Blocked Index Scan* are somewhat similar to the ones for the *Index Scan* of Table 1 in that they contain the *read_index*, *read_input_element*, and *copy_output* components. However, because we will call this operation repeatly when we operate on a block, the index structure will end up being reused in the cache (Section 3.1). As a result, we have fewer *read_index* misses, which we represent as $read\_index^*$ [16]. However, we also suffer *cross* and *self* misses on the reused index structure, which we compute in a similar way as in the *Blocked Nested Loop Join* [16].

Finally, the *Blocked Index Nested Loop Join* records *read_outer_input* misses trying to read *outer_size* tuples from the left input of the join in the query tree. These tuples are operated upon in blocks. For each tuple, however, we record *blocked_index_scan* misses trying to index-scan the right input table. These misses assume some reuse of the index tree. Within the *blocked_index_scan* misses, the *cross* misses have to be recomputed because the cross interference is different now [16]. Finally, we record *copy_output* misses to generate the output tuples of the join.

4 Experimental Setup

In this section, we describe the setup for our experiments: the database system we built, the query plans and application evaluated, and the platforms used.

4.1 SimpleDB

We developed *SimpleDB*, which includes a subset of the subsystems that are found in a real database system. The three essential subsystems for the experiments in this work are: a *data storage*, a *query optimizer*, and a *query executor*. A real database system would include at least three more subsystems: *concurrency control*, *recovery*, and *security and au-*

thorization. However, due to the characteristics of the decision support workload studied, these last three subsystems are largely unnecessary. The decision support queries do not need to be performed on live data. For example, the queries are executed during the day and the database is updated during a short period at night. Because all queries are read-only, there is no need to use any locking to allow concurrent access. Furthermore, keeping a log of the operations is also unnecessary because there is no need for recovery, since the data is not modified. Overall, *SimpleDB* is an optimized decision support query engine that is able to correctly evaluate the queries without the overheads that would be found in an off-the-shelf, general-purpose DBMS. Such overheads are a consequence of asserting the ACID properties (Atomicity, Consistency, Isolation, and Durability) [2] for every query, even when they are not absolutely necessary, like in our case. Figure 5 shows a block diagram of *SimpleDB*.

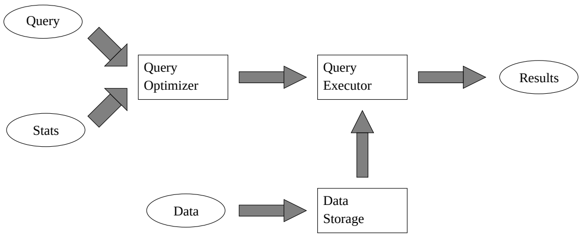


Figure 5: The *SimpleDB* system.

The input to the *data storage* module is the set of database data tables; its output is a collection of data structures allocated in the heap and loaded with the database data. The input to the *query optimizer* is the SQL query and statistics information on the data and the system. The output of this module is a query plan, which is the input to the *query executor*. The output of the executor is the result table.

4.2 Query Plans and Application

The *SimpleDB* optimizer uses different cost models and algorithms to generate, for each query, five potentially different query plans: *pg*, *pgx*, *cpu*, *m* and *cm*. The *pg* plan is the one that would be generated by a traditional DRDB system. The optimizer uses the cost models that are found in the Postgres database [15]. Note that, although the optimizer assumes that the system is disk-resident, we define a memory buffer larger than the total database data size. As a result, the optimizer does not need to perform any inefficient data structure transformation that would put the system at a performance disadvantage relative to the other plans. No algorithms blocked for caches are allowed.

The *pgx* plan is obtained by using the *pg* plan and exchanging the original non-blocked algorithms for the algorithms blocked for caches. Obviously, this is only possible for the algorithms that have a blocked version. This optimization is better than the one proposed by Shatdal *et al* [13] because, as indicated in Section 3.1, our blocking works in a pipelined environment.

The *cpu* plan would be generated by a traditional MRDB system. It uses a cost function that includes only the *CPU* component. The optimizer uses only the non-blocked algorithms because it is not aware of the existence of caches.

The next two plans are representative of our proposed systems. The *m* plan is generated by a naive cache-aware optimizer that uses only the *MISS* component in its cost function. It assumes that misses are the most costly operations and neglects the cost of the CPU computation. Finally, the *cm* plan is generated by a cache-aware optimizer that uses both the *CPU* and the *MISS* components in its cost function. For this optimizer, we use w_{CPU} equal to 1 and w_{MISS} equal to 100. This somewhat arbitrary choice means that we are assuming that the observed latency of a cache miss is 100 times higher than the average CPI with a perfect memory system. Both the *m* and the *cm* plans can use our algorithms blocked for caches if they are found to have a lower cost.

Table 2 summarizes the cost functions and algorithms for the different plans. All these plans can be used with or without software prefetching. Initially, we will not apply prefetching. We will apply prefetching only in Section 5.3.

Plan	**System**	**Cost Function**	**Algorithms**
pg	Disk resident	Postgres	Non blk only
pgx	Disk resident	Postgres plus extensions	Blk and non blk
cpu	Mem resident	*CPU*	Non blk only
m	Naive cache	*MISS*	Blk and non blk
cm	Advanced cache	$w_{CPU} \times CPU + w_{MISS} \times MISS$	Blk and non blk

Table 2: Different approaches to generate query plans.

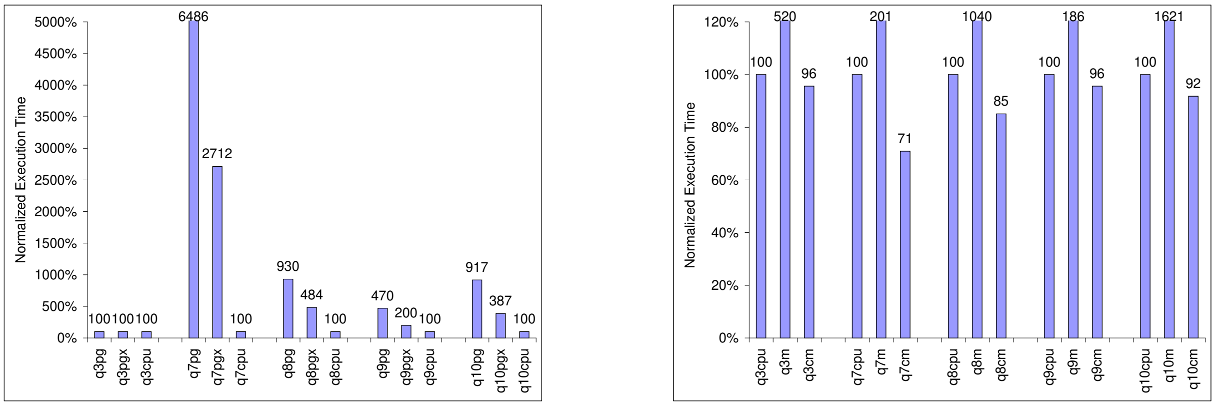
The database application for these experiments is composed of five queries from the TPC-D benchmark [18]: *Q3*, *Q7*, *Q8*, *Q9* and *Q10*. These queries are selected because they are complex: each of them includes 2-7 joins. Complex queries have a larger potential for improvement because they can be expressed with many different query trees and, therefore, offer more choices to the optimizer. At the time when our experiments were conducted, the TPC-H decision support benchmark [19] was unavailable. We expect the TPC-H queries, which are more complex, to be a better showcase for our techniques.

We generate the database data with the standard data generation tool *dbgen*, which is provided by the TPC organization. We use data sizes smaller than the standard 1 Gbyte because we have limited resources. For the experiments, we use a machine configured with 128 Mbytes of main memory. As a result, since we focus on main-memory database systems, we set the database data set size to 100 Mbytes.

4.3 Architectural Platforms

We use two uniprocessor architectural platforms to evaluate our proposed optimizations. The first one, used in Section 5.1, is a 1-processor SGI Origin 200 configured with 32-Kbyte primary data and instruction caches, a 1-Mbyte unified secondary cache, and a 128-Mbyte main memory. The R10000 processor in this machine includes several hardware performance counters [22] that we use to measure execution cycles and cache misses.

The second platform is an architecture simulated in software that is somewhat similar to the Origin 200 and is easy to modify to model different configurations. We use a MINT-based [20] execution-driven simulation environment [6]. The architecture modeled has an aggressive dynamic superscalar processor that can fetch and retire up to 4 MIPS instructions every cycle. The processor has an instruction window of 32 entries, 32 additional registers for renaming, and can perform multiple branch predictions, even when there are pending unresolved branches. All instructions take 1 cycle to complete except multiply and divide, which take 3 and 15 cycles, respectively. A perfect instruction cache is assumed. There are two levels of data cache: a 32-Kbyte 2-way set-associative primary cache and a 1-Mbyte direct-mapped secondary cache. Both caches are non blocking, support up to 8 outstanding loads and 16 outstanding stores, and enable full load bypassing. The line size for both caches is 64 bytes. The no-contention



(a) (b)

Figure 6: Execution time of the different query plans on the Origin 200 with a 100 Mbyte database data set: existing schemes (a) and new schemes (b). The data is presented on a per-query basis and is normalized to *cpu*.

round-trip access latencies from the processor to caches and memory are 1 cycle for the primary cache, 9 cycles for the secondary cache, and 76 cycles for the main memory. We use release memory consistency. The memory hierarchy parameters of this *baseline* architecture will be changed in the different experiments.

5 Experimental Results

5.1 Real Architecture

We start our evaluation by running the queries on the Origin 200. For each query, we ran the 5 potentially different plans generated by the optimizer (Figure 6). Because of the different orders of magnitude in the results, Figure 6 has two charts: chart (a) corresponds to the existing approaches, namely *pg*, *pgx* and *cpu*, while chart (b) corresponds to the *cpu* and the new approaches *m* and *cm*. The bars measure the total time taken by each query plan to execute, normalized to the time taken by the *cpu* plan. The data is presented on a per-query basis. Table 3 shows the absolute execution time of the *cpu* plan and its miss rate in the primary data cache.

Query	Q3	Q7	Q8	Q9	Q10
Execution Time (Seconds)	1.3	1.2	1.0	81.8	6.4
L1 Data Miss Rate (%)	3.2	2.2	1.9	3.0	4.1

Table 3: Execution time and miss rate in the 32-Kbyte primary data cache for *cpu*.

Figure 6-(a) shows that, for all the queries except *Q3*, the Postgres plan (*pg*), even when enhanced with the blocked algorithms (*pgx*), performs much worse than the *cpu* plan. The different behavior of *Q3* is due to the fact that it is a simple query with only two join operations. Therefore, the number of different plans is small and, consequently, plans are unlikely to differ much. For the other queries, a close analysis of the Postgres code shows that the reason for the poor performance of its plans is a hard-coded condition. This condition asserts that a *Hash Join* algorithm can be used only if the inner input table is smaller than the outer input table. This DRDB condition, which tries to avoid costly data spills to disk, ends up disabling good plans. Garcia-Molina and Salem [3] also report this problem in other DRDB systems. Overall, we conclude that the plans generated for a memory-resident system deliver better performance.

Figure 6-(b) shows that, among memory-resident plans, the *m* plan is always the worst. Clearly, therefore, optimizing for the minimum number of cache misses does not lead to the best performance. This is because it often involves using algorithms that, even though they cause few misses, they add many extra instructions. For example, in *Q7*, there is a 100% reduction in the cache misses from *cpu* to *m*, and a 100% increase in the execution time. This effect occurs because some *Blocked Nested Loop Joins* are used instead of *Hash Joins*. In this case, the *Blocked Nested Loop Joins* reduce the number of misses at the cost of many more instructions. For the other queries, the increase in execution time is even larger.

The best performance is achieved when considering both the *CPU* and the *MISS* cost, as in the *cm* plan. Relative to *cpu*, *cm* runs *Q7* 29% faster, *Q8* 15% faster, and *Q10* 8% faster. For the rest of the queries, the improvements are smaller. In general, these execution time reductions may seem large compared to the relatively small miss rates recorded in the primary data cache for the *cpu* plan as shown in Table 3. However, under a wide-issue superscalar processor, even a small miss rate reduction can have a large impact on the execution time. In addition, due to the regular access patterns, many of the accesses missing in the primary cache also miss in the secondary cache, which makes them more costly. Finally, it is also possible that the use of the *cm* cost function happens to trigger the choice of a set of algorithms with lower real computation cost than the ones chosen by *cpu*.

Overall, we conclude that, for the particular database and the complicated queries tested, the cache-aware optimizer hat we propose produces query plans (*cm*) that are faster than the traditional memory-resident ones (*cpu*). One of the queries now runs 29% faster, while the 5 complicated queries in TPC-D run an average of 12% faster. Although this is not an order-of-magnitude improvement, it only requires simple software changes.

5.2 Simulated Architectures

We now use simulations to evaluate our optimizations under a broad set of architectural configurations. Unfortunately, query *Q9* has too long an execution time for us to simulate. Its execution time is approximately 80 times longer than most of the other queries. Consequently, we can only simulate *Q3*, *Q7*, *Q8* and *Q10*.

Although the purpose of the baseline simulated architecture is not to replicate the real one, since both architectures are

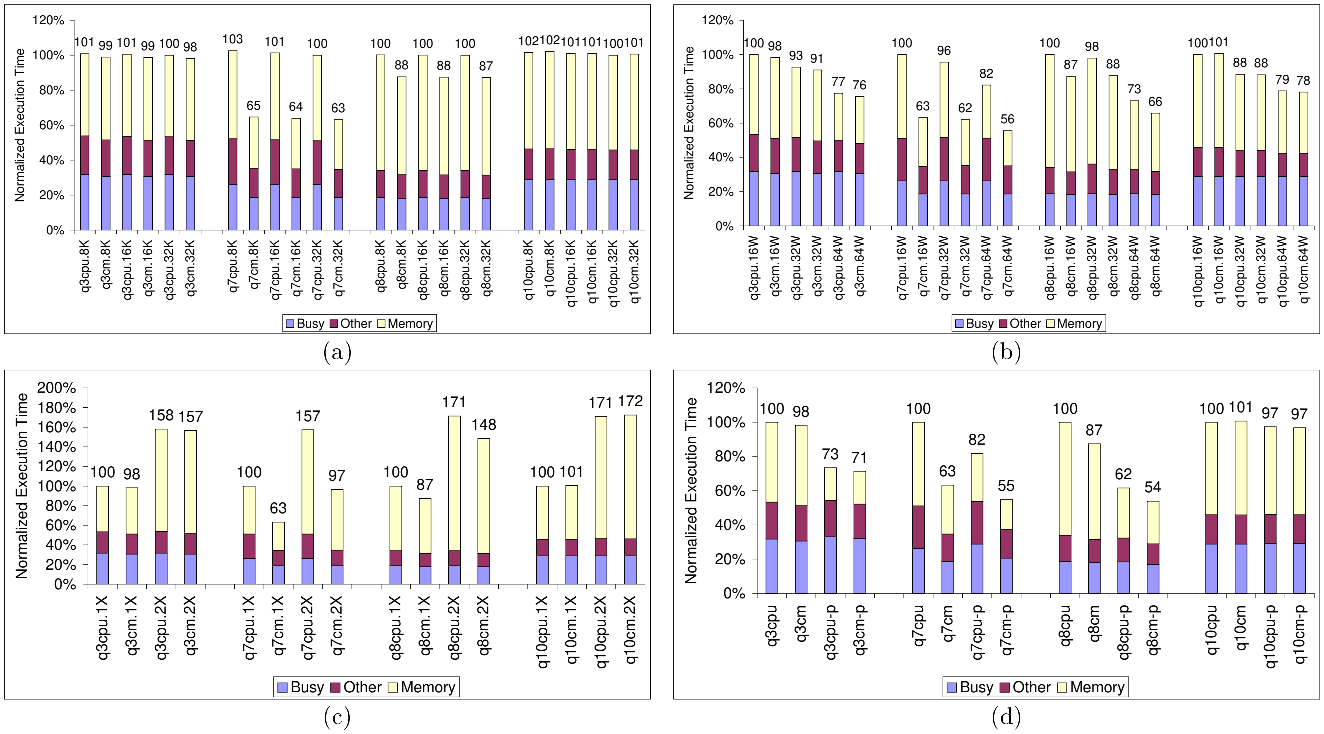


Figure 7: Normalized execution time for different cache sizes (a), cache line sizes (b), and cache miss penalties (c). Chart (d) shows the impact of the software prefetching optimization on the execution time.

similar, we compare the speed in which they run the queries. Because the architectures are not exactly the same, instead of comparing absolute values, we compare how much faster the *cm* plan is over the *cpu* plan in each architecture. Table 4 shows the results. Overall, although one query shows a sign inversion, we feel that the simulator generates qualitatively similar improvements as the real hardware when it is configured to imitate this system.

Query	Real	Simulated
Q3	4%	2%
Q7	29%	37%
Q8	15%	13%
Q10	8%	-1%

Table 4: Speed improvement of *cm* over *cpu* for the four queries in both the real and the simulated architecture.

5.2.1 Cache Size

The first parameter that we vary is the size of the primary data cache. We study 8-, 16-, and 32-Kbyte caches. Figure 7-(a) shows the relative execution times of the *cpu* and *cm* plans. For each query, we present 6 bars, corresponding to all combinations of cache size and query plan. Execution time is divided into instruction execution (*Busy*), memory stall time (*Memory*), and slots wasted to other reasons, including structural hazards, data dependencies, and mispredicted branches (*Other*). In each query, the bars are normalized to *cpu* with a 32-Kbyte cache.

Figure 7-(a) shows that there is little variation as we change the cache size between 8 and 32 Kbytes. This implies that the data reuse captured by 32-Kbyte caches is already largely captured by 8-Kbyte caches. As a result, for these configurations, the improvement of *cm* over *cpu* is largely constant across cache sizes.

5.2.2 Cache Line Size

The second parameter that we vary is the cache line size. We examine 16-, 32-, and 64-word lines for both data caches. Figure 7-(b) shows the execution time for the different cache line sizes and query plans. The figure is organized as Figure 7-(a). In each query, the bars are normalized to *cpu* with 16-word lines.

The figure shows that, in both plans, as we increase the line size, the execution time of the queries decreases. This implies that the algorithms used by these query plans exploit spatial locality. However, once again, the relative improvement of *cm* over *cpu* changes little with the cache line changes examined. Perhaps the only change occurs in *Q7*, where the improvement of *cm* over *cpu* slowly decreases for the longer cache lines. This may be due to the large speedups delivered by the *cm* plan in the first place, which makes it hard to deliver further speedups with long lines.

5.2.3 Cache Miss Penalty

Finally, we change the cache miss penalty. We compare the baseline configuration (1X) to one with a double cache miss penalty (2X). For the latter, we double the w_{MISS} in the *cm* cost model. The execution time results are organized as in the previous figures and presented in Figure 7-(c). In each query, the bars are normalized to *cpu* with baseline latency. The figure shows that, while the execution time increases significantly for the 2X configuration, the relative improvement of *cm* over *cpu* stays remarkably constant. Once again, our cache-oriented optimizations are stable across different architectural configurations.

Overall, the results presented in the last few sections suggest that, for these queries, the proposed cache-aware plan (*cm*) provides an improvement over the state-of-the-art memory-resident plan (*cpu*) that is fairly robust across changes in cache size, cache line size, and cache miss penalty.

5.3 Software Prefetching

Finally, we apply the software prefetching optimization discussed in Section 3.2 to the *cpu* and *cm* plans for the baseline architecture. The results are depicted in Figure 7-(d) in the usual manner. For each query and plan, the figure shows the execution time without prefetching (*cpu* and *cm*) and with prefetching (*cpu-p* and *cm-p*). In each query, the bars are normalized to *cpu* without prefetching. Due to space limitations in this paper, we only present a very summarized discussion of the results [16].

Figure 7-(d) shows that most of the queries examined run significantly faster with software prefetching. Prefetching reduces the *Memory* time with minimal increases in the *Busy* time. Relative to the non-prefetching *cpu* plan, the queries run 13% faster under the non-prefetching *cm* plan, and 31% faster under the prefetching *cm* plan. Prefetching also speeds up *cpu*. The relative improvement of *cm* over *cpu* stays relatively unchanged with the addition of prefetching, at least for the configurations examined.

6 Conclusions

Dramatic increases in the main-memory size of computers is allowing some databases to shift their main data storage area from disk to main memory. The resulting systems are called Memory-Resident Database (MRDB) systems. While their performance is higher, these systems can still waste a lot of time in the cache hierarchy. In this paper, we have presented and evaluated several cache-oriented optimizations to enable effective exploitation of caches in memory-resident decision-support databases. Traditional, disk-resident databases should also benefit to some extent.

The main optimization presented involved developing a query optimizer that included the cost of cache misses in its cost metrics. Based on the cost of cache misses and instruction execution, the optimizer chooses the ordering and algorithm implementation of the query's abstract operations. The other optimizations were sophisticated data blocking and software prefetching. These optimizations required no custom-designed hardware support and were effective for the more complicated TPC-D queries. In a simple database, these complicated queries ran about 13% faster with the cache-aware optimizer and blocking, and a total of 31% faster if, in addition, we added prefetching. The effectiveness of these optimizations was stable across a range of cache sizes, cache line sizes, and miss penalties. We expect the optimizations to be more effective on the TPC-H queries because of the higher complexity of the latter.

Acknowledgments

We would like to thank Sharad Mehrotra for the long discussions, Daniel Reed for his suggestions, and the referees and the graduate students from the I-ACOMA group for their feedback.

References

[1] L.A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[2] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc., second edition, 1994.

[3] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, pages 509–516, December 1992.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan-Kaufmann Publishers, San Francisco, CA, 1996.

[5] H.V. Jagadish, D. Lieuwen, R. Rastogi, and A. Silberschatz. Dalí: A High Performance Main Memory Storage Manager. In *Proceedings of the 20th VLDB Conference*, pages 48–59, September 1994.

[6] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1998.

[7] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[8] T.C. Mowry, M.S. Lam, and A. Gupta. Design and Evaluation of Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[9] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *Proceedings of 1994 ACM-SIGMOD International Conference on Management of Data*, pages 233–242, May 1994.

[10] V. Santhanam, E. Gornish, and W.-C. Hsu. Data Prefetching on the HP PA-8000. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 264–273, June 1997.

[11] M.M. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access Path Selection in a Relational DBMS. In *Proceedings of 1979 ACM-SIGMOD International Conference on Management of Data*, June 1979.

[12] Semiconductor Industry Association. The National Technology Roadmap for Semiconductors, 1998. http://notes.sematech.org/ ntrs/PublNTRS.nsf.

[13] A. Shatdal, C. Kant, and J.F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th VLDB Conference*, pages 510–521, September 1994.

[14] TimesTen Performance Software. Main-Memory Data Management, August 1997. http://www.timesten.com/ customers/wp.html.

[15] M. Stonebraker and G. Kemnitz. The POSTGRES Next Generation Database Management System. *Communications of the ACM*, October 1991.

[16] P. Trancoso. Optimizing Memory-Resident Decision Support System Workloads for Cache Memories. PhD Thesis, University of Illinois at Urbana-Champaign, July 1998.

[17] P. Trancoso, J.-L. Larriba-Pey, Z. Zhang, and J. Torrellas. The Memory Performance of DSS Commercial Workloads in Shared-Memory Multiprocessors. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture (HPCA-3)*, pages 250–260, February 1997.

[18] Transaction Processing Performance Council. *TPC Benchmark D (Decision Support) Standard Specification Revision 1.2.3*, June 1997.

[19] Transaction Processing Performance Council. *TPC Benchmark H (Decision Support) Standard Specification Revision 1.0.0*, February 1999.

[20] J.E. Veenstra and R.J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 201–207, January 1994.

[21] K.-Y. Whang and R. Krishnamurthy. Query Optimization in a Memory-Resident Domain Relational Calculus Database. *ACM Transactions on Database Systems*, pages 67–95, March 1990.

[22] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings of Supercomputing'96*, November 1996.