The Memory Performance of DSS Commercial Workloads in Shared-Memory Multiprocessors[1]

Pedro Trancoso, **Josep-L. Larriba-Pey***, **Zheng Zhang**, and **Josep Torrellas**

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign, IL 61801
trancoso,zzhang,torrella@csrd.uiuc.edu
http://www.csrd.uiuc.edu/iacoma/

*Computer Architecture Department
Universitat Politècnica de Catalunya, Barcelona, Spain
larri@ac.upc.es

Abstract

Although cache-coherent shared-memory multiprocessors are often used to run commercial workloads, little work has been done to characterize how well these machines support such workloads. In particular, we do not have much insight into the demands of commercial workloads on the memory subsystem of these machines. In this paper, we analyze in detail the memory access patterns of several queries that are representative of Decision Support System (DSS) databases.

Our analysis shows that the memory use of queries differs largely depending on how the queries access the database data, namely via indices or by sequentially scanning the records. The former queries, which we call *Index* queries, suffer most of their shared-data misses on indices and on lock-related metadata structures. The latter queries, which we call *Sequential* queries, suffer most of their shared-data misses on the database records as they are scanned. An analysis of the data locality in the queries shows that both *Index* and *Sequential* queries exhibit spatial locality and, therefore, can benefit from relatively long cache lines. Interestingly, shared data is reused very little inside queries. However, there is data reuse across *Sequential* queries. Finally, we show that the performance of *Sequential* queries can be improved moderately with data prefetching.

1 Introduction

Cache-coherent shared-memory multiprocessors are becoming a cheap source of easy-to-program computing power. One promising use of such machines is in commercial workloads, widely used in applications like large wholesale suppliers or airline ticket reservation systems. Indeed, recently announced shared-memory multiprocessors like the Sequent STiNG machine [5] specifically target the commercial market.

However, while vendors usually present performance results for commercial workloads running on these machines, there is no solid understanding of why the workloads perform the way they do. In particular, there is very little understanding of the memory behavior of these workloads. This issue is important because, in shared-memory multiprocessors, the performance of an application is often determined by how well it exploits the memory hierarchy. Furthermore, with the continuous reduction in the price of memory, it may soon be feasible for medium-sized databases to completely reside in memory during execution on a shared-memory multiprocessor. Therefore, how well the memory hierarchy is exploited will directly determine the performance of the workload.

Databases have several characteristics that are likely to affect how they use the memory hierarchy. Specifically, they have complex locking schemes, directly manage the blocks of data read into memory from the I/O devices, and use complex data structures to manage database data efficiently. However, an analysis of how these characteristics affect memory use is not trivial. It usually involves monitoring the addresses referenced by the processors as well as other events. Furthermore, the information extracted from the address traces needs to be combined with an analysis of the database source code to determine the operations executed and the data structures accessed when the address references were issued. While this is not a problem for scientific workloads like Splash 2 [13], where the sources are publicly available, it is usually hard for commercial workloads. Obtaining the source code of a reasonably-tuned database management system (DBMS) is difficult because it is usually proprietary.

For these reasons, there is relatively little previous work in this area. In addition, most of the work has addressed the performance of these workloads from a high-level point of view. For example, DeWitt and Gray [3] studied parallel database systems and indicated that a shared-nothing architecture seems to be more cost-effective than a shared-memory architecture. Thakkar and Sweiger [9] looked at the performance of On-Line Transaction Processing (OLTP) workloads running on a Sequent cache-coherent shared-memory multiprocessor and highlighted the importance of process scheduling and the I/O capability of the machine. Maynard *et al* [6] contrasted the cache performance of technical and commercial workloads and concluded that the latter is often worse. Eickemeyer *et al* [4] showed that a significant performance improvement can be obtained for OLTP workloads when a multithreaded processor is used. Finally, other studies that have involved database workloads include the work by Cvetanovic and Bhandarkar [1] on a DEC Alpha AXP system, Torrellas *et al* [10] on an SGI multiprocessor, and Rosenblum *et al* [7] on a simulated SGI multiprocessor. In general, these studies agree on the relatively worse memory performance of commercial workloads. However, they do not give us the insight of what the actual memory access patterns are like.

In this paper, we analyze in detail the memory access pat-

[1] This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP 94-57436 and RIA MIP 93-08098, ARPA Contract No. DABT63-95-C-0097, NASA Contract No. NAG-1-613, and Intel Corporation. Pedro Trancoso was also supported by the Portuguese government under scholarship JNICT PRAXIS XXI/BD/5877/95. Josep-L. Larriba-Pey was supported by the Ministry of Education and Science of Spain under contract TIC-0429/95 and by Generalitat de Catalunya under grant contract 1995BEAI400095.

terns of three queries that are representative of Decision Support Systems (DSS) workloads. DSS databases store large quatities of data. Queries to these databases are usually read-only and extract useful information in order to aid decision making. We use three queries from the standard *TPC-D* benchmark [11] and simulate the memory hierarchy of a 4-processor cache-coherent NUMA running a memory-resident database. We use a modified version of Postgres95 [8, 14], a public-domain database from the University of California at Berkeley.

Our analysis shows that the memory use of queries differs largely depending on how the queries access the database data, namely via indices or by sequentially scanning the records. The former queries, which we call *Index* queries, suffer most of their shared-data misses on the indices and on lock-related metadata structures. The latter queries, which we call *Sequential* queries, suffer most of their shared-data misses on the database records as they are scanned. An analysis of the data locality in the queries shows that both *Index* and *Sequential* queries exhibit spatial locality and, therefore, can benefit from relatively long cache lines. In addition, we find that shared data is reused very little inside queries. However, there is data reuse across *Sequential* queries. Finally, we find that the performance of *Sequential* queries can be improved moderately with data prefetching.

This paper is organized as follows: Section 2 describes some background material on query processing and the workload used; Section 3 presents the three queries that we trace; Section 4 describes Postgres95 and defines the methodology used for our analysis; Section 5 performs the analysis of the memory performance; and Section 6 evaluates the impact of data prefetching.

2 Query Processing and TPC-D

In this section, we introduce some background material on query processing and then describe the TPC-D DSS workload that we use.

2.1 Query Processing

2.1.1 Query Operations

In the relational database model, data is stored in tables, also called relations. These tables are composed of records called tuples. Each tuple contains fields called attributes. A database query is composed of different basic operations. Typical operations are *Select*, *Join*, *Sort*, *Group* and *Aggregate*. Each of these operations consumes the data in one or two tables of tuples and generates one table of tuples as a result. Each of these operations can be implemented using different algorithms.

A select operation takes one table and generates another one that has all the tuples of the input table that satisfy a given condition on a tuple attribute or a set of attributes. This operation can be implemented with two algorithms: *Index Scan* select or *Sequential Scan* select. The first one uses an index data structure to access only the tuples in the input table that satisfy the condition. The second one is used when there is no index structure on the attributes that are checked. Therefore, all the tuples in the input table have to be visited.

A join operation takes two tables and produces one result table. A join selects a pair of tuples, one from each table, that have one or more attributes in common and that satisfy a given condition. The result table contains the chosen pairs of tuples without replicating the common attributes. A join can be implemented using different algorithms. The best known algorithms are the *Nested Loop*, *Merge*, and *Hash* join. The nested loop join simply uses a doubly nested loop to try to match each tuple of one table to all the tuples of the other table. The merge join orders both input tables and then tries to match the two ordered streams of tuples. Finally, the hash join builds a hash table using one of the input tables and then probes it for each of the tuples of the other input table.

The sort operation orders the tuples of a table based on the value of one attribute. The group operation generates a table that has one tuple for each group of tuples in the input table that have a common value in the grouping attribute. Finally, the aggregate operation generates a table where one or more attributes of each tuple in the input table are modified by an operation. This operation may be an arithmetic operation. More information on these operations and the relational database model can be found in [2].

2.1.2 Query Execution

Queries can be written in several database languages. For example, Figure 1-(a) shows an example of a query written in SQL that will be analyzed later. A query that is submitted to a database system undergoes three different steps, namely *parsing*, *optimization* and *execution*. The parsing step checks for the correctness of the query syntax and semantics.

The optimization step rewrites the query into a *Query Plan Tree* that contains the basic operations described in Section 2.1.1 in some order that minimizes the execution time. An example of such a tree is shown in Figure 1-(b). The shape of the query plan trees depends on the database system that generates them. The database system that we use in our experiments generates left-deep trees. The tree is built based on heuristics and cost analysis of different possible implementation alternatives. Usually, the optimization step is performed at compile time to save time during the execution. However, some optimizations may be a function of certain parameters that are only known at runtime. In those cases, the optimization step is completed at runtime.

Finally, in the execution step, the system performs the query operations according to the query plan tree. Each node in the tree corresponds to one of the basic operations described previously. The leaves of the tree correspond to sequential or index scans of the tables. Each child of a node represents the flow of a stream of data to the node. The execution of a left-deep tree is a depth-first recursive descend of the tree that scans the different tables, transfers the data to the topmost node of the tree and, in the process, performs the basic operations required by the query. To avoid the use of very large temporary tables, the results are passed tuple-by-tuple between the nodes in a pipelined manner. This approach is possible for any node that does not require the whole input data before it can perform its operation. However, in the sort nodes, we need temporary tables to store the whole input data. Example executions of query plan trees are discussed in detail in Section 3.

2.2 DSS Workloads and TPC-D

One of the most important classes of commercial workloads is DSS databases. These databases often store large quantities of information that is queried to make a decision. Typically, DSS queries are complex and access the data in a read-mostly manner. A well-known benchmark that simulates a DSS system is TPC-D [11]. In this section, we first describe TPC-D briefly and then examine its queries in detail.

2.2.1 TPC-D

TPC-D simulates an application for a wholesale supplier that manages, sells and distributes a product worldwide. The data in TPC-D is organized in several tables. The most important of the TPC-D relations are *lineitem*, *order*, *part*, *customer*, and *supplier*. The simulated company buys parts (stored in table *part*) from suppliers (stored in table *supplier*) and sells them to customers (stored in table *customer*). Each time an order is placed by a customer, it is added to the *order* table and the ordered parts are added to a list of ordered items (table *lineitem*). Any attribute of the tuples in these tables can potentially be accessed via indices.

2.2.2 TPC-D Queries

TPC-D has 17 read-only queries (Q1 to Q17) and 2 update queries (UF1 and UF2). Most of the queries are large and complex, and perform different operations on the database tables. Table 1 lists the operations performed by the read-only TPC-D queries when the query plan trees are generated by the database system used in this paper. The database system is Postgres95 and is described in Section 4.1. The select operations can be implemented by the sequential scan (SS) or the index scan (IS) algorithms. The join operations can be implemented by the nested loop (NL), merge (M), or hash (H) join algorithms. In the table, we have grouped the queries based on how the select operation is implemented. Such implementation, of course, is a function of the set of indices that we added. From the table, we see that some queries implement sequential scan selects only, while others implement index scan selects only, and others implement both.

Table 1: Operations in the read-only TPC-D queries.

Query	Select		Join			Sort	Group	Aggr.
	SS	IS	NL	M	H			
Q1, Q4	√					√	√	√
Q6	√							√
Q15	√					√	√	
Q16	√				√	√	√	√
Q2		√	√			√		
Q3, Q5, Q10, Q11		√	√			√	√	√
Q8		√	√					
Q7, Q9	√	√	√		√			
Q12	√	√		√		√	√	
Q13	√	√	√			√	√	√
Q14, Q17	√	√	√					√

From the queries in the table, we chose three representative ones that we will examine in detail in the rest of the paper. Our choice is based on the fact that, as we will see, the type of select algorithm used in the query largely determines the memory access patterns of the query. For this reason, we chose one query from each of the three groups in the table: Q3, Q6 and Q12. We do not examine any of the two queries that write data. This is because the locking support in the Postgres95 database is not as fine-grained as in some of the tuned commercial databases (Section 4.1.1). Update queries are much more demanding on the locking algorithm. In addition, the update queries are not as complex as the read-only ones.

3 Memory Access Patterns of TPC-D Queries

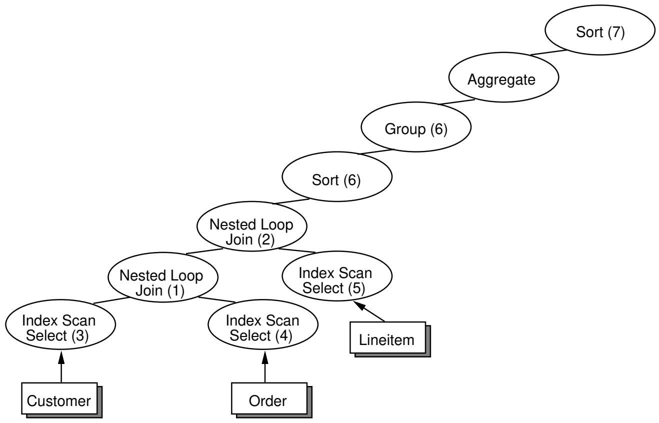
To understand the performance of the memory hierarchy under TPC-D Queries Q3, Q6 and Q12, we devote this section to an analysis of the memory access patterns of these queries. This analysis will be used to explain the simulation results in Section 5. The database data is stored in shared-memory buffers as will be described in Section 4.1.1. The queries are coded in the limited form of SQL supported by the database system that we use. In all cases, we coded the queries so that they have the same memory access patterns as if the queries were coded in a system that supported a full SQL implementation. Sometimes, this forced us to make small changes to the code. Consequently, the SQL programs that we use to code the queries do not compute exactly what the Transaction Processing Performance Council proposes. Their memory access patterns, however, are those of a system with full SQL implementation. In the following, we consider each query in turn.

3.1 TPC-D Query Q3

Q3 retrieves the unshipped orders of customers within a specific market segment and dates. For example, a possible query could retrieve all the orders from market segment "automobile" that have an order date prior to "February 3, 1995" and by date "March 19, 1995" had not yet been shipped. The SQL code that we use for query Q3 and the query plan tree generated by Postgres95 are shown in Figure 1.

```
SELECT
  lineitem.orderkey,
  SUM(lineitem.extendedprice) AS revenue1,       ] Aggregate
  SUM(lineitem.discount) AS revenue2,
  order.orderdate,
  order.shippriority
FROM
  customer, order, lineitem
WHERE
  customer.custkey = order.custkey AND       (1) ] Join
  lineitem.orderkey = order.orderkey AND     (2)
  customer.mktsegment = "segment" AND        (3) ] Select
  order.orderdate < "date1" AND              (4)
  lineitem.shipdate > "date2"                (5)
GROUP BY
  lineitem.orderkey,      ] (6) Sort and Group
  order.orderdate,
  order.shippriority
ORDER BY
  revenue1 USING >,       ] (7) Sort
  order.orderdate;
```

(a)



(b)

Figure 1: Query Q3. Chart (a) shows the SQL code, while Chart (b) shows the query plan tree. The numbers inside the nodes of the tree correspond to statement numbers in the SQL code.

The execution proceeds as follows. First, Q3 traverses table *customer* to select those customers that belong to market segment *"segment"*. This is shown in clause (3) of Figure 1-(a) and is represented by the leftmost leaf of the tree in Figure 1-(b). Note that the table is accessed via indices and, therefore, only the tuples that match are ever accessed. Each time that a matching tuple is found, it is sent to the Nested Loop Join (1) node of the tree. This node passes the *customer.custkey* attribute of the tuple to the Index Scan Select (4) node of the tree. At this point, the Index Scan Select (4) node searches the *order* table to find orders that belong to the same customer (clause (1) in Figure 1-(a)) and were placed previous to a certain date *"date1"* (clause (4) in Figure 1-(a)). Again, table *order* is accessed via indices. Every time one of these tuples is found, it is passed to the Nested Loop Join (1) node where it is joined with the *customer* tuple.

The resulting tuple is passed to the Nested Loop Join (2) node. The *order.orderkey* attribute of the tuple is passed to the Index Scan Select (5) node. There, the *lineitem* table is accessed via indices to find all the lineitems with the same orderkey (clause (2) in Figure 1-(a)) that have not been shipped by date *"date2"* (clause (5) in Figure 1-(a)). For each tuple that matches, the Nested Loop Join (2) node performs the join.

When the three tables have been completely searched and all the necessary joins have been performed, the selected tuples are sorted in the Sort (6) node. Then, in the Group (6) node, the selected tuples are grouped by attributes *lineitem.orderkey*, *order.orderdate*, and *order.shippriority* (clause (6) in Figure 1-(a)). Finally, the Aggregate and Sort (7) nodes perform the aggregate and remaining sort operations specified in the query.

Most of the memory accesses to shared data in this query are issued by the index scan operations. These operations access two major shared data structures, namely the data tables and the indices. If we focus first on the data tables, we see that there is practically no temporal or spatial locality at the tuple level. Indeed, a given tuple is not accessed more than once in the same query. Furthermore, two consecutive tuples are not necessarily related in anything and, therefore, are not necessarily accessed at similar times. All this is clear if we consider the tuples accessed in the query. The Index Scan Select (3) node reads the *customer* tuples whose *mktsegment* attribute is *"segment"*. The Index Scan Select (4) node in turn reads the *order* tuples whose *custkey* is equal to the *custkey* of one of the selected *customer* tuples. Finally, the Index Scan Select (5) node reads the *lineitem* tuples whose *orderkey* is equal to the *orderkey* of one of the selected *customer-order* tuple pairs.

Accesses within a tuple, however, have spatial locality. This is because, for a given tuple, several attributes are read. For example, both the *mktsegment* and *custkey* attributes of a given *customer* tuple are read. However, no significant temporal locality is present within a tuple. This is because the database is optimized so that the same attribute does not have to be read twice in this query.

Accesses to the index data structures have both temporal and spatial locality. For example, consider the index tree for the *custkey* attribute of the *order* table. There is temporal locality because the top levels of the index tree are re-read every time a new customer is considered. There is spatial locality because consecutive locations of the index b-tree are read when the query searches for the orders of a given customer.

Finally, we consider data reuse across queries. The index data structures are clearly reused across queries. However, the data tables are not likely to be reused. This is because each query accesses its own set of tuples. Tuple reuse is only possible if the two queries have clauses with the same or similar attributes, which force them to access some common tuples.

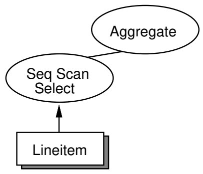
The select nodes in Figure 1 read shared data and make copies of the selected tuples into private data. The rest of the nodes work on this private data. Private data accesses have some spatial locality because, sometimes, several attributes of the same tuple are read close in time. They also have some temporal locality because the same private storage is reused for all the selected tuples.

3.2 TPC-D Query Q6

Q6 quantifies the revenue increase that would have resulted from eliminating discounts in a given percentage range during a given year. For example, a possible query could be to compute the difference in revenue between "February 3, 1994" and "February 3, 1995" for the range of products discounted by 15%. The SQL code that we use for query Q6 and the query plan tree generated by Postgres95 are shown in Figure 2.

```
SELECT
  SUM(lineitem.extendedprice) AS revenue1,     ] Aggregate
  SUM(lineitem.discount) AS revenue2           ]
FROM
  lineitem
WHERE
  lineitem.shipdate >= "date" AND              ] Select
  lineitem.shipdate < "date" + "1 year" AND    ]
  lineitem.discount > "discount" – 0.01 AND    ]
  lineitem.discount < "discount" + 0.01 AND    ]
  lineitem.quantity < "quantity";              ]
```

(a)



(b)

Figure 2: Query Q6. Chart (a) shows the SQL code, while Chart (b) shows the query plan tree.

This query is very simple. The query traverses the *lineitem* table sequentially, visiting all the tuples. The tuples that satisfy the select clauses shown in Figure 2-(a) are passed up to the Aggregate node. In the Aggregate node, two arithmetic operations are performed. Clearly, nearly all of the shared memory accesses in the query are issued by the sequential scan operation. They are directed to the *lineitem* table. For each tuple, the query reads the attributes to be checked in the select conditions, namely *lineitem.shipdate*, *lineitem.discount*, and *lineitem.quantity*. If the clauses are satisfied, the query also reads the attributes needed in the Aggregate node, namely *lineitem.extendedprice* and *lineitem.discount*. There is no access to indices.

There is abundant spatial locality in these accesses. Indeed, the query reads several attributes of the same tuple and, in addition, it reads consecutive tuples. There is, however, no reuse of a tuple within a query and, therefore, there is no temporal locality.

There is reuse and, therefore, temporal locality, across queries. This is because every time that Q6 is executed, it reads the whole *lineitem* table. Unfortunately, *lineitem* is large (approximately 70% of the total database data). In our experiments, it takes about 12 Mbytes.

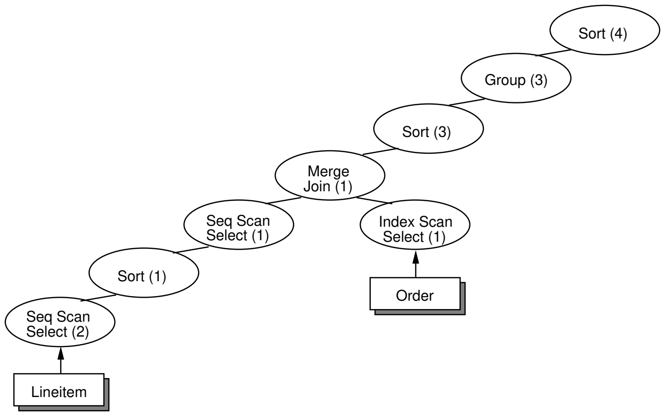
For this query, the locality and reuse of private data is the same as in Q3. There is no additional spatial locality across tuples because all the selected tuples reuse the same storage.

3.3 TPC-D Query Q12

Q12 determines whether selecting less expensive modes of shipping is negatively affecting critical-priority orders by delivering orders after the committed date. For example, a possible query would be to retrieve the orders that have been delivered after the committed date for a one year interval starting on "February 3, 1994" using the "regular air" and "air" shipping modes. The SQL code that we use for query Q12 and the query plan tree generated by Postgres95 are shown in Figure 3.

```
SELECT
  lineitem.shipmode,
  order.orderpriority
FROM
  order, lineitem
WHERE
  order.orderkey = lineitem.orderkey AND                  ] (1) Select and Join
  (lineitem.shipmode = "shipmode1" OR                     ] (2) Select
   lineitem.shipmode = "shipmode2") AND
  lineitem.commitdate < lineitem.receiptdate AND
  lineitem.shipdate > lineitem.commitdate AND
  (lineitem.receiptdate >= "date" AND
   lineitem.receiptdate < "date" + "1 year")
GROUP BY
  lineitem.shipmode           ] (3) Group and Sort
ORDER BY
  lineitem.shipmode           ] (4) Sort
```

(a)



(b)

Figure 3: Query Q12. Chart (a) shows the SQL code, while Chart (b) shows the query plan tree.

The execution proceeds as follows. Table *lineitem* is traversed sequentially by the Sequential Scan Select (2) node. Each tuple is selected with clauses (2) in Figure 3-(a). Each tuple that satisfies these clauses is passed to node Sort (1). There, a temporary table is formed and its tuples are sorted on attribute *lineitem.orderkey*. The sorting is necessary because the Merge Join (1) node requires the input tables to be sorted. Next, the Sequential Scan Select (1) node reads the sorted tuples one by one and passes them to the Merge Join (1) node. The latter node passes attribute *lineitem.orderkey* to the Index Scan Select (1) node, which selects the tuples in the *order* table that have the same *orderkey*. The selected tuples are joined one by one in the Merge Join (1) node and then sorted and grouped in the next few nodes.

The memory access patterns in this query are a combination of those in queries Q3 and Q6. Indeed, the locality and reuse patterns of the accesses to the *lineitem* table are similar to those of the accesses in Q6. In addition, the locality and reuse patterns of the accesses to the *order* table are similar to those of the accesses in Q3. Likewise, the locality and reuse of private data is the same as in Q3 and Q6.

3.4 Summary

The analysis in this section suggests that there are two clear types of access patterns. They result from the way database tables are accessed, namely sequentially or via indices. If the majority of the accesses in a query are sequential, we call the query *Sequential* query, while if the majority of the accesses are via indices, we call the query *Index* query.

In both types of queries, data accesses within a tuple are likely to have spatial locality. However, in *Sequential* queries, there is the additional effect of spatial locality across tuples and, if the cache space in the node is large enough, data reuse across queries. In *Index* queries, index structures have temporal and spatial locality within a query and, in addition, are reused across queries.

4 Experimental Setup

To validate our analysis and get a deeper insight into how databases exercise memory hierarchies, we run Q3, Q6, and Q12 on a real database and simulate the resulting memory accesses. In this section, we describe the experimental setup and in the next one, we discuss the results obtained. Our experimental setup is based on the Postgres95 database interfaced to an execution-driven simulator of a multiprocessor memory system. In this section, we examine Postgres95, the issues involved in scaling down the system, and finally the simulation system.

4.1 Postgres95

Postgres95 is a public-domain, client-server share-everything database developed at the University of California-Berkeley [8, 14]. It is a reduced and revised version of the Postgres database [8]. Postgres has led to commercial products, now being commercialized by Informix. Postgres95 runs on numerous platforms, is fairly popular, and is considerably well-tuned for a system developed in academia.

We chose Postgres95 for at least two practical reasons. First of all, we have its source code. This capability, not easily attainable for commercial databases like Oracle or Informix, is practically a requirement for a study of this type. Secondly, although Postgres95 was developed to run on a uniprocessor machine, it supports multiple concurrent transactions where processes communicate via shared memory. This, as we will see below, makes it possible to emulate a multiprocessor database system fairly well. To understand the behavior of Postgres95 better, we now examine its main data structures and then discuss how we emulate a multiprocessor.

4.1.1 Postgres95 Data Structures

The major parts of Postgres95 are shown in Figure 4. All the boxes shown represent *software* data structures. At the top of the figure, we show 4 processes with their own private *software* caches. These caches store data that is rarely modified, for example the system catalog. The Shared Memory Module is shown in the lower portion of the figure. In this space, we have the Invalidation Cache, which interacts with the private caches to maintain their contents consistent. We also have the Lock Management Module with its two hash tables (Lock Hash and Xid Hash), which determine if a lock can be acquired or not. The access to these data structures is protected by a lock called LockMgrLock. Finally, we have the Buffer Cache Module. This module contains the actual application data processed by the database. It also contains the indices. This module manages the pages of application data and indices similarly to how the operating system manages the pages of other applications. This module has three components. The Buffer Blocks are 8-Kbyte pages of memory that hold application data or indices. The Buffer Descriptors are control structures for the buffer blocks. Finally, the Buffer Lookup Hash is a hash table that is used to find buffer descriptors. The access to the Buffer Blocks is protected by a lock called BufMgrLock.

Postgres95 supports two types of locks, namely those that protect database data and those that protect the data structures of Postgres95. We call them *Datalocks* and *Metalocks* respectively. While metalocks are simple spinlocks, datalocks are more complex, since they are multi-type and multi-level. For example, they can be of type read or write and of level relation, page or tuple. The purpose of having all these types

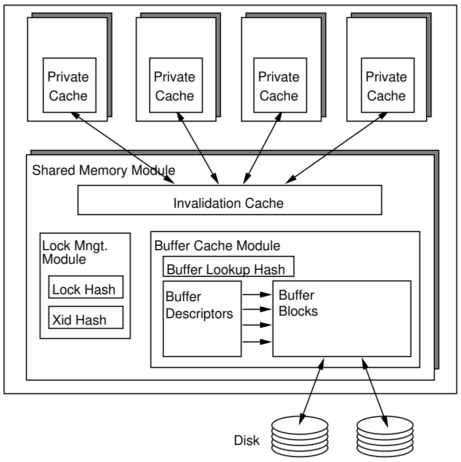


Figure 4: Block diagram of the major parts of Postgres95.

and levels is to offer higher concurrency: lock types allow multiple locking of the same data item as long as there are no conflicts, while lock levels provide different locking granularities. Currently, of all the levels, only the relation level is fully implemented in Postgres95. This fact clearly limits the level of concurrency in write-intensive queries. Fortunately, the queries that we examine are unaffected by this limitation because they are read-only.

4.1.2 Multiprocessor Emulation of Postgres95

Although Postgres95 was developed to run on a uniprocessor machine, it is relatively easy to make it run on a simulated multiprocessor. This is because Postgres95 supports the concurrent execution of transactions issued by different processes. In the case of a uniprocessor, these processes can interleave their use of the CPU. Clearly, this can be extended to a simulated multiprocessor system by having each process run on a different simulated processor. The parallel programming model for query execution is inter-query parallelism. This means that each simulated processor runs a different query or stream of queries.

Postgres95 was developed to use the client-server model. Therefore, its front-end communicates via sockets to the back-end. To make the system more efficient, we modified Postgres95 into a single executable that contains both the front-end and the back-end.

4.2 Scaling Down the System

To evaluate a complex system like a complete Postgres95 database running the standard TPC-D data set requires substantial simulation time. To keep the cost of our simulations affordable, we scale down the database. Specifically, we use the database population generator program distributed with the TPC-D code to populate the database. Then, we scale down the data set size 100 times. The result is a database of about 20 Mbytes of data that our simulations can manage. This change, however, prompts us to make two more corrections.

Since the database is small, the first correction is to reduce the size of the memory hierarchy of the machine simulated. We model a machine with slightly over 20 Mbytes of main memory, 128-Kbyte secondary caches, and 4-Kbyte primary caches. All these small caches overflow, as the full-sized ones would in a real system. The memory, however, is large enough to keep the whole database. This is because we want to study a memory-resident database.

The second correction has to do with misses on private data. As we described in Section 3, the tuples processed by Postgres95 may be copied from the shared address space to the private one. Large chunks of private heap space are sometimes allocated for tables of tuples after the join operations. Postgres95 operates on these tables as it performs sort or group operations. Consequently, private references to the heap are an integral part of Postgres95. Furthermore, it can be argued that the misses on private heap data structures should somehow scale up and down with the size of the database. However, the misses on the other private data, namely stack and static variables, should not scale with the size of the database. If we simply reduce the cache sizes, misses on private stack and static variables will swell disproportionally to their true weight. Consequently, the second correction that we do is to assume the all accesses to private stack and static variables hit in the cache.

Finally, it could be argued that misses on shared Postgres95 metadata also need a similar correction. Such metadata includes locks, hash tables, or buffer headers. Experimental data, however, shows that this is not the case. These data structures have a tiny footprint and, as we will see, most their misses are caused by coherence activity. Smaller caches do not change these misses.

4.3 Workloads and Architecture

In our experiments, we run one query of the same type on each node. Although the queries are of the same type, each of them has different parameters, chosen according to the TPC-D specifications. We record statistics for the complete execution stage of the queries, from start to finish. We do not run any warm-up query before collecting the statistics because, depending on the type of query, there could be data reuse. Without any instrumentation, the queries that we examine take 6-10 seconds to run on a 150 MHz workstation. With the tracing and simulation code enabled, the queries are slowed down by a factor of 1500. Since queries do not have intra-query data reuse, the traces do not have any transient period that we might want to discard.

The simulation setup is as follows. The object codes of the query and Postgres95 are linked together to produce an executable. Then, the executable is fed to *Mint* [12], an execution-driven multiprocessor simulation package. Mint performs an interleaved execution of all processes, correctly modeling all the aspects of the shared-memory and synchronization activity. In addition, Mint generates events on-the-fly to a back-end architecture simulator. The setup of the experiments is outlined in Figure 5.

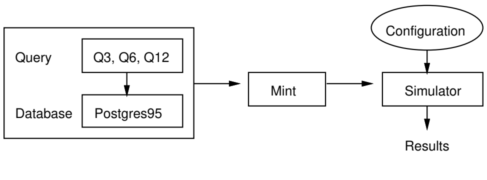


Figure 5: Experimental setup.

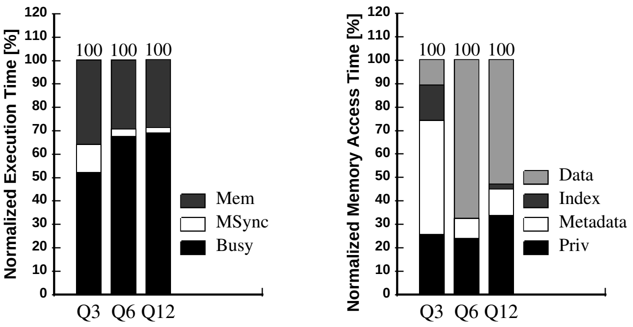
The simulated architecture is a 4-processor directory-based cache-coherent NUMA shared-memory multiprocessor. Each node of the architecture includes a simulated off-the-shelf 500 MHz processor with a 16-entry write buffer, a 4-Kbyte on-chip primary cache with 32-byte lines, and a 128-Kbyte off-chip secondary cache with 64-byte lines. The primary cache is direct-mapped, while the secondary cache is 2-way set-associative. Processors stall on read misses and on write buffer overflow. For simplicity, we model an interconnection network where a message traveling from one node to another takes a fixed 100 cycles. All contention in the system is modeled, except in the network, where the simulator assumes a constant delay. Overall, on a primary cache miss, the round-trip latency time for a request satisfied by the secondary cache, local memory, and remote node in a 2-hop or 3-hop transaction is 16, 80, 249, and 351 cycles respectively. Consequently, a 2-hop remote transaction takes under 500 ns. This architecture, which we call *baseline*, is modified in the course of our experiments. We change the sizes of the caches and cache lines. In all cases, however, the size of the line in the primary cache is half the size of the line in the secondary cache. When we mention only one line size we refer to the line size of the secondary cache.

5 Evaluation

We now present the results of the simulations. We start by studying the overall memory behavior of the queries in Section 5.1. Then, in Section 5.2, we examine the locality of the memory accesses in detail.

5.1 Overall Memory Behavior

A breakdown of the execution time of the queries for the *baseline* architecture is shown in Figure 6-(a). The bars are normalized and then broken down into three categories, namely *Mem*, *MSync*, and *Busy*. *Mem* is the processor stall time due to memory accesses not satisfied by the primary cache. It includes the effect of read misses and write buffer overflow. *MSync* is the time spent synchronizing in metalocks. The time spent synchronizing in datalocks is negligible. This is because the queries are read-only and, therefore, there is no contention for locks that protect application data. Finally, *Busy* includes the rest of the processor cycles.

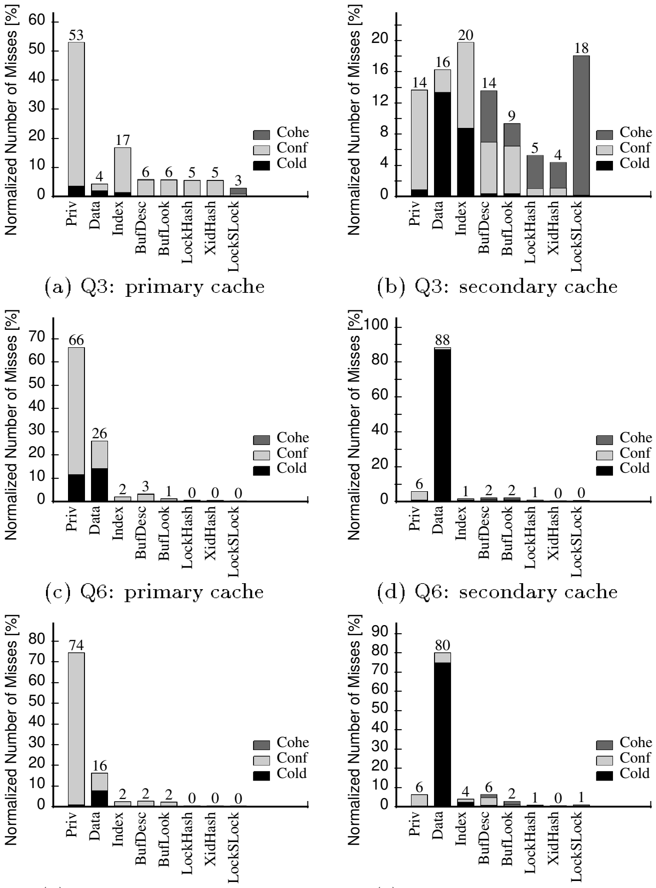


(a) (b)

Figure 6: Execution time (Chart (a)) and stall time due to memory accesses (Chart (b)) for Q3, Q6 and Q12.

Figure 6-(a) shows that *Busy* accounts for 50-70% of the execution time, while *Mem* accounts for 30-35%. Since this paper focuses on the impact of memory accesses, we concentrate on the *Mem* time. Figure 6-(b) normalizes the *Mem* time and then decomposes it based on the data structures that cause the memory stall. There are four main types of data structures, namely database data (*Data*), database indices (*Index*), database control variables (*Metadata*), and private data structures (*Priv*). From the figure we can observe that, while the portion due to private data does not change much across queries, the contribution of the rest of the data structures shows two different behaviors. First, in Q3, nearly all of it is due to metadata and indices. This is because Q3 accesses all the data via indices. It makes use of the control structures and indices to read only the necessary database tuples. This is the typical behavior of what we have called *Index* queries in Section 3.4. The second behavior is exhibited by both Q6 and Q12. In these two queries, the contribution of the shared data structures is dominated by the accesses to the database data. This pattern is typical of queries that read large tables sequentially, without using indices. We have called these queries *Sequential* queries in Section 3.4. Note that while Q12 has both a sequential and an index scan select, the former dominates.

To gain further insight into the stall time due to memory accesses, we classify the read misses in the primary and secondary caches according to the data structures that cause them. The classification is shown in Figure 7. The data structures that suffer a significant number of misses are: private data (*Priv*), database data (*Data*), database indices (*Index*), and several metadata structures, including buffer descriptors (*BufDesc*), the buffer lookup hash table (*BufLook*), the Lock and Xid hash tables (*LockHash* and *XidHash* respectively), and the LockMgrLock spinlock (*LockSLock*). In the figure, each bar is divided into three different types of misses, namely cold (*Cold*), conflict (*Conf*), and coherence (*Cohe*). In each chart, the sum of all the bars is 100.



(a) Q3: primary cache (b) Q3: secondary cache

(c) Q6: primary cache (d) Q6: secondary cache

(e) Q12: primary cache (f) Q12: secondary cache

Figure 7: Normalized number of misses in the primary and secondary caches classified according to the data structures missed on.

The leftmost charts of Figure 7 show that most of the misses in the primary cache are due to accesses to private data. Most of these misses are of the conflict type. The reason is that there are about 5 times more accesses to private data than to shared data. While the size of private data is relatively small, it is large enough to overflow the primary caches. The rightmost charts of Figure 7 show that the misses in the secondary cache are a function of the type of query. While Q3 has a mix of misses from metadata, indices, database data and private data, most of the misses in Q6 and Q12 occur on database data. In Q3, many of the metadata misses occur on *LockSLock*. As explained in Section 4.1.1, this data structure is the lock that controls the access to the Lock Management Module, which manages the multi-level multi-type data locking. This lock is necessary to fully support concurrency of data accesses in Postgres95 and is continuously accessed by all processors. The figure also shows that metadata misses are usually due to coherence activity and, to a lesser extent, cache conflicts. Data misses, on the other hand, are largely due to startup effects. This is due to the large amount of database data that is accessed and the little reuse present. Index misses, on the other hand, are present only in the *Index* query Q3. Indices are large, read-mostly data structures. As a result, index misses are due to start-up effects and cache conflicts. Overall, therefore, for the secondary cache, we see that *Sequential* queries suffer mostly cold misses, while *Index* queries suffer a combination of coherence, conflict, and cold misses.

The absolute data miss rate of the caches is as follows. In the primary cache, the miss rates are 5.5%, 3.4%, and 4.8% for Q3, Q6, and Q12 respectively. In the secondary cache, the global miss rates are 0.8%, 0.6%, and 0.5% for Q3, Q6, and Q12 respectively.

5.2 Locality of Accesses

The number of misses on the different data structures directly depends on the locality of the memory access patterns. In this section we analyze the spatial and temporal locality of the data accessed in the queries.

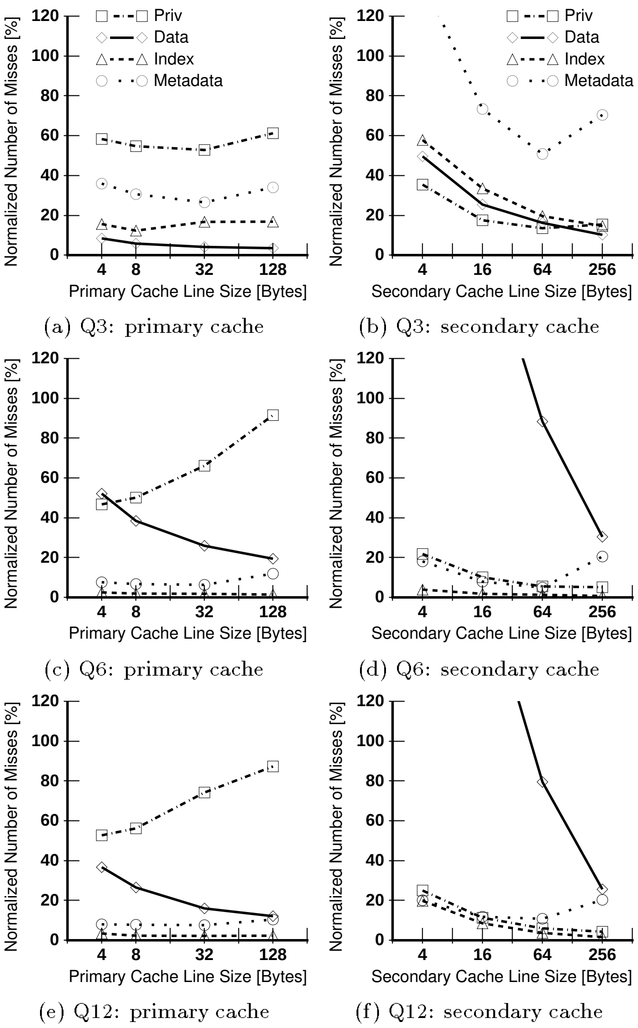
5.2.1 Spatial Locality

In Section 3 we indicated that accesses to database data tend to have good spatial locality. When a tuple is accessed, it is likely that several of its attributes will be referenced. Furthermore, in *Sequential* queries, there is also very good spatial locality across tuples. Accesses to index structures also tend to have good spatial locality because parts of the index data structure are traversed sequentially. Finally, metadata is unlikely to have spatial locality because of the diversity and small size of its data structures.

To validate these hypotheses, we measure the variation of the number of misses with the line size of the caches. Figure 8 shows, for each query, the number of misses in the primary cache (leftmost charts) and secondary cache (rightmost charts) for different cache line sizes. All the other parameters correspond to the *baseline* architecture. The charts are normalized to 100 for the *baseline* configuration, which has 32-byte primary cache lines and 64-byte secondary cache lines. In each chart, the misses are decomposed into those suffered on *Priv*, *Data*, *Index*, and *Metadata* data structures.

From the leftmost charts we see that the number of misses on private data in the primary cache increases with the line size. The reason is the poor locality of the accesses to heap data. By increasing the line size while maintaining the cache size, we are reducing the number of lines in the cache and, therefore, inducing more conflict misses. The database data, however, benefits from longer lines because it has good spatial locality. This is true in the primary caches and, especially, in the secondary caches (rightmost charts). In the two *Sequential* queries (Q6 and Q12), the misses on database data decrease spectacularly as the line size increases. For Q3, the misses on both database data and indices also decrease with the line size. This shows that indices have good spatial locality too. The behavior of the metadata, however, is irregular. In all queries, its misses decrease until 64 bytes and then increase. This indicates that the spatial locality of metadata is lower than the other data structures. Finally, the misses on private data decrease with the line size. Private accesses, therefore, also have some spatial locality. However, their locality cannot be captured by the small primary cache.

The impact of changing the line size on the execution time of the queries is shown in Figure 9. For each query, the bars in the figure are normalized to the bars for the *baseline* configuration. Each bar is divided like in Figure 6-(a) except that we split the *Mem* time into the contribution of the accesses to shared data structures (*SMem*) and the accesses to private data structures (*PMem*). The figure shows that, irrespective of the query, two trends occur as we increase the size of the cache line. On the one hand, *PMem* tends to increase after 16-byte secondary cache lines. This is because private data has poor primary cache performance. Longer lines cause more misses. On the other hand, *SMem* decreases as we increase the cache line. This is due to the good spatial locality of database data and indices. With longer lines, each miss takes longer to satisfy, but there are many fewer misses. When the two trends are combined, we see that the minimum for the total execution time is obtained for 64-byte secondary cache lines. Overall, therefore, we conclude that relatively long cache lines like those with 64 bytes perform well for these DSS queries.



(a) Q3: primary cache (b) Q3: secondary cache

(c) Q6: primary cache (d) Q6: secondary cache

(e) Q12: primary cache (f) Q12: secondary cache

Figure 8: Number of misses on the different data structures for several cache line sizes. In each chart, the number of misses is normalized to the *baseline* configuration (32-byte primary cache lines and 64-byte secondary cache lines.)

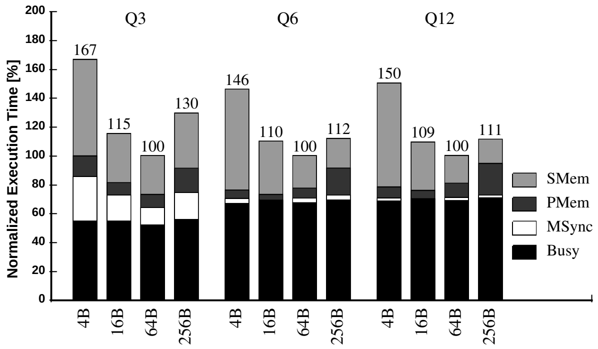


Figure 9: Execution time for different cache line sizes. Each bar is identified by the size of the secondary cache line.

5.2.2 Temporal Locality

Temporal locality can be exploited within a query or across queries. In this section, we consider each case in turn. For this part of the study, we use a fixed line size of 64 bytes for secondary caches.

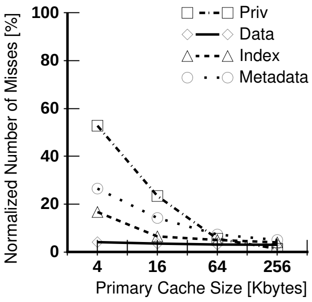
Intra-Query Temporal Locality

In Section 3 we indicated that database data is not reused within a query. Consequently, there is no temporal locality and, as a result, large caches should not make a difference in the miss rate of the database data. In reality, a close look at the traces reveals that attributes are often accessed a second time immediately after they are first accessed. The reason is that a given attribute is first read in a scan select to see if it satisfies a certain condition. If the tuple it belongs to satisfies all the conditions, then the attribute is then read again and copied to private storage. This reuse, however, occurs immediately and, whether or not it causes a miss cannot be affected by the cache size. Therefore, we do not consider it. The index data structure, instead, is often reused within a query for *Index* queries. Specifically, the top level nodes of its b-tree are traversed very frequently. Finally, it is hard to determine the reuse of metadata given its complex structure.

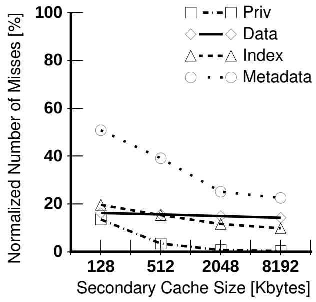
To validate our observations, we measure the variation of the number of misses with the cache size. Figure 10 shows, for each query, the number of misses in the primary cache (leftmost charts) and secondary cache (rightmost charts) as the cache sizes change from 4-Kbyte primary and 128-Kbyte secondary caches to 256-Kbyte primary and 8-Mbyte secondary caches. The charts are normalized to 100 for the *baseline* configuration, which has 4-Kbyte primary caches and 128-Kbyte secondary caches. In each chart, the misses are decomposed into those suffered on *Priv*, *Data*, *Index*, and *Metadata* data structures.

The most obvious trend in the primary cache charts is that misses on private data decrease significantly. This is consistent with the data in Section 5.2.1, which showed that private data suffers many misses in the primary cache and few in the secondary one. Consequently, private data is reused. The Q3 chart also shows that metadata and, as expected, indices have some temporal locality for *Index* queries. Focusing now on the secondary cache charts, it is clear that database data has no temporal locality. In all three queries, the curve for database data is flat. The Q3 chart shows that, again, metadata and indices have temporal locality for *Index* queries. These results are consistent with the discussion of Section 3.4.

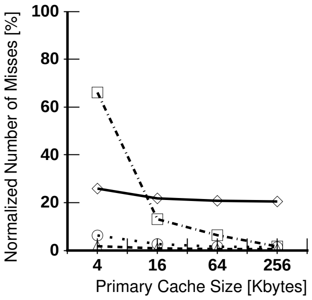
The impact of changing the cache size on the execution time of the queries is shown in Figure 11. For each query, the bars in the figure are normalized to the bars for the *baseline* configuration. Each bar is divided like in Figure 9. The figure shows that, as the caches increase in size, the queries run faster. Most of the speedup, however, comes from the reduction of misses on private data (*PMem* category). In the Q3 *Index* query, the temporal locality of indices and metadata also helps reduce the execution time (*SMem* category). For the other queries, however, the speedups are small because database data has no temporal locality within queries.



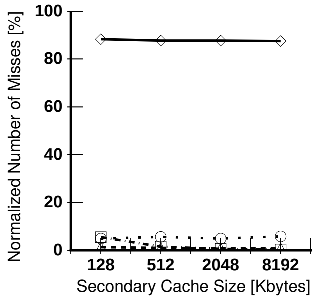
(a) Q3: primary cache



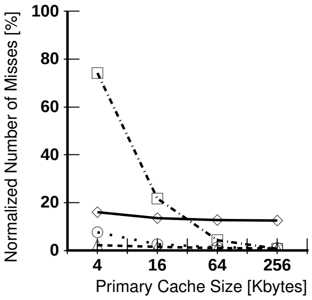
(b) Q3: secondary cache



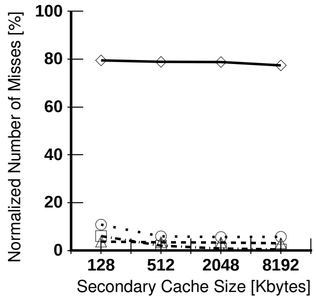
(c) Q6: primary cache



(d) Q6: secondary cache



(e) Q12: primary cache



(f) Q12: secondary cache

Figure 10: Number of misses on the different data structures for several cache sizes. In each chart, the number of misses is normalized to the *baseline* configuration (4-Kbyte primary caches and 128-Kbyte secondary caches).

Inter-Query Temporal Locality

Finally, we evaluate data reuse across queries. In our experiments, we focus only on queries Q3 and Q12. This is because Q6 behaves like Q12. We measure and compare the number of misses in Q3 when the query runs with cold-started caches, when it runs right after another Q3 query, and when it runs right after a Q12 query. We do the same thing for Q12. In our simulations, we use a 1-Mbyte primary cache and a 32-Mbyte secondary cache. We use these very large caches to identify the upper bound on the data reuse. In a real system, of course, the data reuse when two queries are run in sequence will be smaller.

Figure 12-(a) shows the number of misses in the secondary cache for an execution of Q3. We show data for our three setups, namely one where the caches are not warmed up (left-

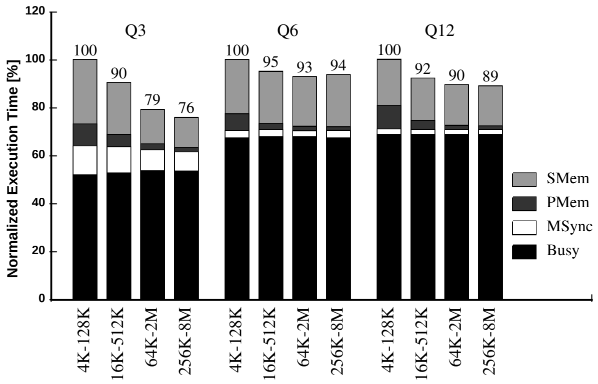


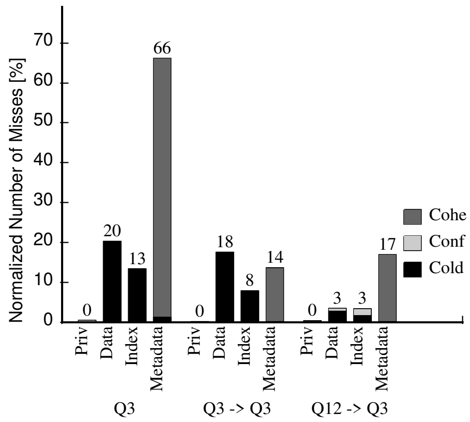
Figure 11: Execution time for different cache sizes.

most 4 bars), one where the caches are warmed up with another execution of Q3 using different parameters (central 4 bars), and one where the caches are warmed up with an execution of Q12 (rightmost 4 bars). For each setup, we classify the misses based on the data structures that cause them. The number of misses is normalized to 100 for the leftmost setup.

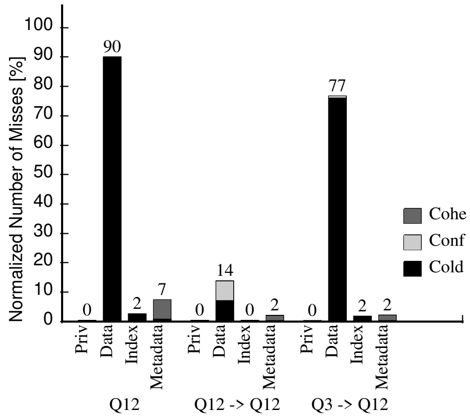
When the caches are not warmed up, most of the misses are distributed between metadata, indices, and database data. This is consistent with measurements for similar cache configurations shown in Figure 10-(b). If the cache is warmed up with another Q3, the number of misses on indices decreases. This is because indices are reused across *Index* queries as well as within *Index* queries. Database data, instead, is reused little. Two *Index* queries are not likely to access many common tuples. We note that the large reduction in metadata misses can only be attributed to random timing effects in the execution of the workload. This is because the misses supposedly saved by the warm cache cannot be of the coherence type. Coherence misses are largely unaffected by the initial cache state. Finally, if the cache is warmed up with Q12, database data and indices are reused across queries. Indeed, since Q12 is a *Sequential* query, it accesses all the tuples in a table (table *lineitem*). Some of these tuples are reused by Q3. In addition, Q12 accesses a second table via indices (table *order*). These indices can also be reused by Q3.

Figure 12-(b) shows the number of misses in the secondary cache for an execution of Q12. As in the previous chart, we consider a setup where the caches are not warmed up (leftmost 4 bars), one where the caches are warmed up with another execution of Q12 using different parameters (central 4 bars), and one where the caches are warmed up with an execution of Q3 (rightmost 4 bars). The bars are organized as in Figure 12-(a). When the caches are not warmed up, nearly all of the misses occur on database data. This is consistent with measurements for similar cache configurations shown in Figure 10-(f). If the cache is warmed up with another Q12, most of the misses on database data disappear. The reason is that the *lineitem* tuples accessed sequentially by the first *Sequential* query are reused by the second *Sequential* query. Therefore, there is much temporal locality across *Sequential* queries. Finally, if the cache is warmed up with Q3, only a few misses on the database data disappear. This is because the Q3 *Index* query accesses only a few of the *lineitem* tuples. The Q12 query that follows Q3 can only reuse those.

Overall, therefore, we conclude that if two *Sequential* queries accessing the same table are run consecutively, there is much reuse of data, namely the entire table. In other cases, there is some reuse of database indices or data, but the magnitude of the savings is substantially smaller. Note that, for reuse to occur, the two *Sequential* queries involved do not have to be of the same type. In fact, reuse occurs across the 5 TPC-D queries that read the *lineitem* table using the Sequential Scan algorithm. The amount of reuse, of course, is limited by the size of the table being scanned and by the size of the caches. For some tables, it is clear that very large caches might be needed to capture the whole reuse.



(a) Misses in Q3



(b) Misses in Q12

Figure 12: Breakdown of the number of misses in the secondary cache for Q3 and Q12.

6 Data Prefetching Optimization

In Section 5.2.1 we concluded that accesses to database data have good spatial locality, especially in *Sequential* queries. Consequently, the use of longer cache lines would be beneficial. However, we also observed that, as cache lines increase in size, misses on private data and metadata also go up (Charts (c) to (f) in Figure 8). One way to avoid this overhead and still capture the benefits of long lines for database data is to use sequential prefetching for database data only. In this section, we evaluate the impact of a very simple form of prefetching. For each access to database data, the hardware issues prefetches for the next 4 primary cache lines. The prefetched lines are loaded into the primary cache.

This optimization is trying to reduce the *Data* time in Figure 6-(b). It is clear from the figure that it can only have a modest impact on the total execution time of Q6 and Q12. Furthermore, it can barely change the total execution time of Q3. Nevertheless, we apply it to Q3 too. The execution time with and without prefetching is shown in Figure 13. For each query, the figure shows the execution time for the *baseline* architecture (*Base* bars) and the *baseline* architecture plus prefetching (*Opt* bars). The results show that, for Q6 and Q12, the gains are a modest 5-6%. This is because it can be shown that prefetching eliminates only about one third of the *Data* time in Figure 6-(b). In addition, it causes primary cache contention and extra misses that disrupt accesses to private data. As a result, *PMem* increases slightly in Figure 13. For Q3, this disruption causes a slowdown to the query. Overall, therefore, we suggest using this technique for *Sequential* queries only and, if the *Busy* time is so high as in Figure 6-(a), expect modest gains.

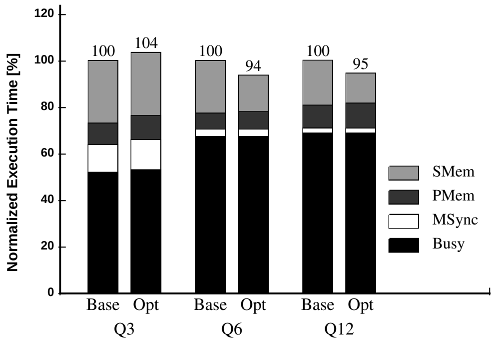


Figure 13: Impact of simple prefetching support for database data on the execution time of the queries.

7 Summary

Although cache-coherent shared-memory multiprocessors are often used to run commercial workloads, we did not have much insight into how these workloads use the memory subsystem of these machines. In this paper, we have addressed this problem for representative DSS queries running on a multiprocessor emulation of Postgres95. We have found that, from a memory performance point of view, queries differ largely depending on whether they access the database data via indices or by sequentially scanning the tuples. The former queries, which we call *Index* queries, suffer most of their shared-data misses on indices and lock-related metadata. The latter queries, which we call *Sequential* queries, suffer most of their shared-data misses on the database tuples. We have found that both *Index* and *Sequential* queries can exploit spatial locality and, therefore, can benefit from relatively long cache lines. We have also found that shared data has little temporal locality inside queries. Private data, however, has some temporal locality. In addition, there is temporal locality across *Sequential* queries. Finally, we have found that the performance of *Sequential* queries can be improved moderately with data prefetching.

Overall, this work is a first step towards understanding the memory performance of databases. The work remaining is huge. It is necessary to address other issues, including more complex queries that involve nested queries, other types of queries that contain frequent writes, and intra-query parallelism.

Acknowledgments

We thank Jolly Chen, Andrew Yu, and Paul Aoki for their help with Postgres95. We thank the Transaction Processing Performance Council for giving us the source code of TPC-D and dbgen. We also thank Sharad Mehrotra, the referees, and the graduate students in the I-ACOMA group for their feedback. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] Z. Cvetanovic and D. Bhandarkar. Characterization of Alpha AXP Performance Using TP and SPEC Workloads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 60–70, April 1994.

[2] C. J. Date. *An Introduction to Database Systems.* The Systems Programming Series. Addison Wesley, sixth edition, 1995.

[3] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[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*, pages 203–212, May 1996.

[5] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, May 1996.

[6] A. M. G. Maynard, C. M. Donnelly, 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*, pages 145–156, October 1994.

[7] 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 Systems Principles*, December 1995.

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

[9] S. S. Thakkar and M. Sweiger. Performance of an OLTP Application on Symmetry Multiprocessor System. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 228–238, May 1990.

[10] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162–174, October 1992.

[11] Transaction Processing Performance Council. *TPC Benchmark D (Decision Support) Standard Specification Revision 1.1*, December 1995.

[12] 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.

[13] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, July 1995.

[14] A. Yu and J. Chen. *The POSTGRES95 User Manual.* Computer Science Div., Dept. of EECS, University of California at Berkeley, July 1995.