A Window-Based Approach to Retrieving Memory-Resident Data for Query Execution

Jayaprakash Pisharath, Alok Choudhary
Dept. of Electrical & Computer Engineering
Northwestern University
Evanston IL - 60208 USA
{ jay, choudhar }@ece.northwestern.edu

Mahmut Kandemir
Dept. of Computer Science & Engineering
Pennsylvania State University
University Park PA - 16802 USA
kandermir@cse.psu.edu

Abstract

Memory-resident databases are gaining popularity. In this paper, we present a data-centric approach to optimize multiple queries issued to a memory-resident database. Our approach involves a bottom-up methodology that reuses each block of data in a table to optimize several queries. We define *data window* as a block of the data residing in the memory brought to the cache. The optimization named *data windowing*, schedules database queries efficiently by reorganizing queries and its operations based on a *data window*. Each data window brought into the cache is maximally reused by queries before evicting it from the cache to accommodate another data window. Experimental results show that data windowing yields up to 75% improvement in the cache performance for typical and heavy-duty memory-resident workloads.

1. Introduction

The rapid improvement in memory technology over the past decade has enabled larger memories and also the integration of memories with other components [6]. Memory accesses times have outnumbered the disk access times. Owing to the drastic increase in memory sizes and to various disk bottlenecks, databases are being shifted from disks to main memory to improve the system performance [9]. Such systems wherein the data primarily resides in the memory are called in-memory (also called memory-resident) database systems [8]. Currently, BerkeleyDB, TimesTen, DataBlitz, Birdstep RDM and Monet are a few widely used memory-resident databases. Traditional DBMS techniques that assume the presence of database on the disk are no longer applicable for these systems. New techniques that efficiently exploit the locality of data are needed. In fact, it is known that in a database system, the data cache misses can form as high as 90% of the memory stalls [1]. Consequently, optimizing database queries to improve their data access patterns can be extremely useful in practice. This can be achieved by maximizing the reuse of the data that resides in the data cache.

With data reuse as the primary goal, we propose a technique called *data windowing* that maximally utilizes every block of data brought into the cache by rearranging queries without disturbing their data consistency and dependency ordering. Queries (operations on tables) are rearranged based on the data in a table and not based on the queries themselves (multiquery control-flow optimization). We implement the data windowing strategy in a memory-resident database and evaluate it using TPC-H benchmark and some typical memory-resident workload as seen in hand-held devices. The results show that our technique significantly reduces the cache misses, and hence, effectively improves the query execution times by at least 21%. When similar queries are executed, there is up to 75% improvement in the performance.

The rest of the paper is organized as follows. Section 2 discusses the related work in query optimziation for memory databases. Our data windowing strategy is elaborated Section 3. In Section 4 and Section 5, we present the experimental setup and the performance results respectively. Section 6 summarizes our work.

2. Related Work

Query optimizations have been proposed exclusively for memory-resident databases. In [4], Boncz et al. considered a set of basic operations on memory databases and profiled them based on the main memory access costs. A query optimization based on multi-level hashing is also proposed for queries involving hash-join operation. Ross takes a bottom-up approach to optimize queries for memory-resident databases [13]. In his work, query optimizations are first proposed by extensively studying the system, and then a cost model is used to study the effect of these optimized queries on a memory-resident database. Researchers have exhaustively proposed multi-query optimizations. A survey of some of the multi-query optimizations specific to memory databases has been done by Choenni et al. [5].

Researchers have proposed query optimizations specifically targeting modern cache architectures in a move to make database systems cache friendly. A cache-friendly sorting algorithm targeting RISC machines is proposed by Nyberg et al. in [10]. Shatdal et al. discuss techniques to improve the cache performance by reorganizing the order of execution of operations in a query based on highly-reused data blocks in the query [15]. In [17], Trancoso et al. combine data blocking and software prefetching to achieve improvements in cache performance. Ailamaki et al. designed a new layout called Partition Attributes

Across (PAX) [2]. This scheme targets improvements in cache performance by extracting the best of slotted and decomposed storage schemes.

Our approach is unique in the sense that our technique optimizes operations based on the underlying data. We reorder operations to suit the data brought from a table, and do not emphasize the control or the flow of queries. In other words, we do not base our optimizations purely on data dependencies and control-flow of queries. Nor is the table layout changed. In contrast, we aim to reuse every bit of data read from the underlying tables. We elaborate our approach further in the following section.

3. Data Windowing

A database usually consists of multiple tables. Each table has multiple columns (fields). A query views the table as a collection of data blocks. Depending on the implementation, these blocks can either be file blocks, database page blocks or just user-defined blocks. Based on this, we define *data window* as a block of data from the table being accessed by a query. Memory databases usually provide two-dimensional tables. In this case, *data window* is the bounding rectangle that includes a subset of rows and columns accessed by a query. For instance, a query accessing a table having 100 entries (rows) and 7 fields (columns) might just require to access say the first four fields. In such a case, one can divide the table into blocks such that each data window has 10 entries and 4 fields. Hence, there would be 10 data windows, each consisting of 10 entries.

We propose an optimization strategy called *data windowing* that works on *data windows*. A query uses a set of *data windows* from each table that it accesses. We begin by presenting a scheme for a single table case and then extend it to the multiple table case. *Data windowing* consists of the following steps.

1. Consider a single table T. The table T is divided into N data windows (disjoint sets) such that each window ($i \in N$) fits into the cache.
2. For each data window *i* in N, identify a set of queries Q to be performed when *i* is loaded to the processor. The goal is to reuse the data in a data window to the maximum possible extent.
3. The number of queries to look at, and the order in which the selected queries are executed is not fixed. That is, the order of accessing the data in a data window is not pre-determined. One way to do it is to just follow the actual execution order as done without this optimization. Given a data window, there is lot of room usually for reusing the data when the execution order is rearranged. A *query queue* contains the queries that can be handled at a given instance (maximum look-ahead of multiple queries).
4. The queries are scheduled based on the window accesses. For each window that is accessed, instances of multiple queries that require data from the particular window are scheduled sequentially. That is, queries in the *query queue* are shuffled in a safe way to reuse the data that is being processed at a given instance. Thus, a *schedule queue* is built based on the data accessed by queries in the *query queue.*

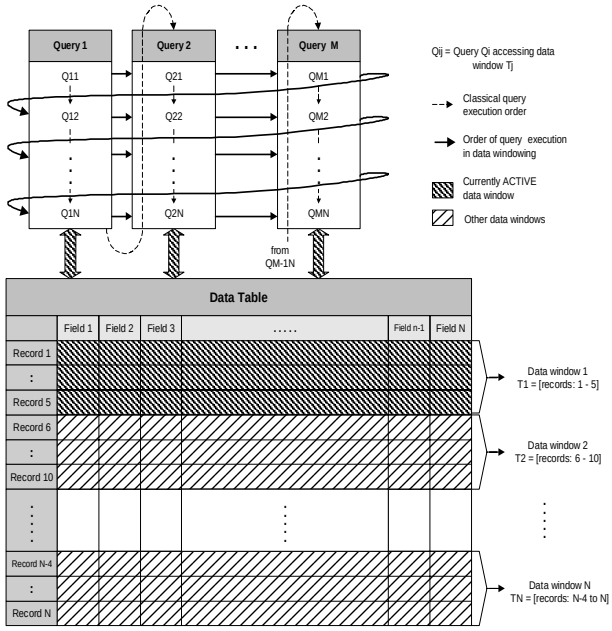

Figure 1. *Data windowing* applied to M queries that work on data from a single table with N data windows

5. The first query finishes executing the particular code section that requires data from the window and passes control to the section of the second query that requires the same set of data. The control goes from one query to another until all queries in the *schedule queue* are executed..
6. Now control is shifted to the next data window of the table. Steps 2 to 5 are repeated for this data window. The algorithm continues until all data windows are completed.

As an example, consider a single table T and three queries, Q1, Q2, and Q3. The table T is first divided into data windows (say, T1, T2, T3, and T4) such that each Ti fits in the cache. Let Qij refer to the part of Qi that access Tj. A classical query execution would be in this order: Q11, Q12, Q13, Q14, Q21, Q22, Q23, Q24, Q31, Q32, Q33, Q34. Our approach, on the other hand, executes this: Q11, Q21, Q31, Q12, Q22, Q32, Q13, Q23, Q33, Q14, Q24, Q34. That is, we restructure query parts (Qij) around the data and make use of the best cache locality. A graphical schematic of this example when applied for M queries and N windows is shown in Figure 1.

An extension to the algorithm is the presence of multiple tables. When multiple tables are accessed by a query, each table has its set of data windows. The queries are scheduled based on multiple data windows from multiple tables. For example, suppose there are two tables T1 and T2, and three queries Q1, Q2, and Q3. Let Q1 and Q3 make single scans over the tables, Q2 make a join between T1 and T2. Let each table be divided into two windows: (T11, T12) and (T21, T22). Then we have the following potential data sets for join:

S1 = T11 x T21

Table 1. The two classes of queries considered for our experiments.

Source	Query	Description	Table Sizes
TPC - H	Q6	Simple query	PART - 1000 entries CUSTOMER - 1000 entries ORDERS - 1000 entries LINEITEM - 1000 entries
	Q3	Complex query involving JOIN	
	Q4	Complex query involving NEST	
	Q17	Complex query involving JOIN and NEST	
Queries targeting a simple organizer	P1	Simple name and address lookup	ADDRESSBOOK - 200 entries FRIENDS - 100 entries COLLEAGUES - 50 entries
	P2	Lookup in directory of friends	
	P3	Lookup in directory of colleagues and friends	

S2 = T11 x T22
S3 = T12 x T22
S4 = T12 x T21

For *data windowing*, we consider each Si in turn, and execute the query portions that access it.

The main goal of our technique is to exploit spatial locality in database queries. Since data windowing changes cursor control based on the data windows, care should be taken to ensure that the data consistency of the database and data dependencies in the original query execution plan does not change. When queries are reordered, they might corrupt the database or just follow a wrong execution order violating the dependencies. To avoid this, we use existing standard techniques that check query execution violations and inter-query dependencies [11,12]. One can opt to use other simpler or even custom mechanisms for this process.

4. Setup

4.1. Queries

To evaluate our scheme for memory-resident databases, we considered two classes of queries. The first class is a subset of queries from the Transaction Processing Council (TPC-H) benchmark [16]. Operations in TPC-H (decision support benchmark) have good spatial locality with abundant data intensive operations. These queries were considered to rigorously test the database and our technique. The top part of Table 1 elaborates the TPC-H queries we used and the corresponding database parameters. The selected operations represent a good mix and could be used to build a variety of complicated queries.

Memory-resident databases run queries that are different from the typical database queries as seen in TPC-H. The second set of queries that we consider are representative of classical handheld devices. The typical operations that are performed on an organizer were imitated on our setup (we name the queries P1, P2, P3). The first query P1 involves a simple address lookup using a 'NAME' as input. The SQL query for P1 is described as follows:

```
SELECT
    a_name,
    a_address,
    a_city,
    a_office_phone,
    a_home_phone,
    a_mobile_phone.
    a_email,
    a_web,
    a_specialnotes
FROM
    addressbook
WHERE
    a_name = '[NAME]';
```

The recent organizers (Palm Organizers available at www.palm.com, Handspring Organizers available at www.handspring.com) provide an ordered view of the underlying addressbook database. For instance, organizers provide the creation of folders. A "friends" folder can be a collection of personnel with a tag set as "friend" in the addressbook. We defined folder as a restrained/customized view of the same database (address book). Intuitively, query P2 strives to do a lookup of friends living in a particular city. A person interested in visiting a city can run this query before he/she leaves for that place. The view and hence the query P2 is defined as follows:

```
create view friends as
 SELECT
    a_name,
    a_address,
    a_city,
    a_home_phone,
    a_mobile_phone
  FROM
    addressbook
  WHERE
    a_tag = '[FRIEND]'
  GROUP BY
    a_name;
```

P2:

```
SELECT
    a_address,
    a_home_phone,
    a_mobile_phone
FROM
    friends
WHERE
    a_city = '[CITY]'
GROUP BY
    a_name;
```

Query P3 combines views (folders). For this we defined a new folder called "colleagues". P3 aims to find friends and/or colleagues whose names start with an 'a', living in a particular 'CITY'. Since P3 is very similar to P2 with some extra fields, we do not present the SQL for P3. The intermediate tables and results of queries are also stored in the memory. Table 1 also shows the parameters used for the database tables in our experiments.

4.2. Execution Scenarios

Since simultaneous processing of multiple queries is needed to validate our approach, we considered a combination of queries, which we term as *scenarios*. Among the queries considered in Section 4.1, there can

Table 2. Scenarios for organizer queries

Type	Legend	Combination
Two query combinations	S11	P1 + P1
	S12	P1 + P2
	S13	P1 + P3
	S22	P2 + P2
	S23	P2 + P3
	S33	P3 + P3
Three query combinations	S111	P1 + P1 + P1
	S123	P1 + P2 + P3

Table 3. Scenarios for TPC-H queries

Type	Legend	Combination
Two query combinations	T11	Q6 + Q6
	T12	Q6 + Q3
	T13	Q6 + Q4
	T14	Q6 + Q17
	T22	Q3 + Q3
	T23	Q3 + Q4
	T24	Q3 + Q17
	T33	Q4 + Q4
	T34	Q4 + Q17
	T44	Q17 + Q17
Three query combinations	T222	Q3 + Q3 + Q3
	T123	Q6 + Q3 + Q4
Four query combinations	T1111	Q6 + Q6 + Q6 + Q6
	T1234	Q6 + Q3 + Q4 + Q17

be multiple combinations of queries that arrive sequentially, and that which are optimizable using our technique. The various combination (scenarios) of organizer queries and their naming schemes are shown in Table 2. For instance, S12 indicates that P1 is sequentially processed along with P2. The combination scenarios for TPC-H queries are shown in Table 3. The combinations shown in the above tables are the prominent ones and the behavior of other combinations are very similar to these, hence, they are not included in this paper.

4.3. Simulation Environment

The scenarios of Section 4.2 were implemented on a custom memory database. To be consistent with implementation techniques, our query execution plan is based on the execution plan generated by Berkeley DB [3] running the same set of scenarios. The *data windowing* technique explained in Section 3 was implemented in the query optimizer of the database.

We used the cache simulator from Sun Microsystems, Shade [14], for our analysis. The scenarios were executed using our database running atop the Shade simulator. We captured the access pattern and hence, the total cache references and misses (inclusive of L1 and L2). L2 cache had a significant effect on the performance and was considerably sensitive to the technique. Additionally, researchers have shown that second-level data cache stalls constitute a significant part of the execution time bottlenecks in databases [1]. Hence, we present our results based on the performance of L2 cache in the following sections. Table 4 shows the default parameters and settings used for our database and the Shade simulator. Before performing our experiments, the database is first populated. The scenarios considered in our experiments include queries that perform just read operations. A read operation, in turn, triggers a traversal of the database for seeking the record.

Table 4. Default parameters used in experiments

Parameter		Value
L1 I-cache	Size	16 KB
	Associativity	2-way
	Block Size	32 bytes
L1 D-cache	Size	16 KB
	Associativity	2-way
	Block Size	32 bytes
L2 cache (unified)	Size	256 KB
	Associativity	4-way
	Block Size	128 bytes

4.4. Indexing Schemes

Indexing schemes reduce the record-lookup times of databases by optimizing the traversal. We modified our database to implement extendible hashing [12] and T-tree indexing schemes [7]. Since our scenarios involve pure read operations, our approach integrates swiftly at the hash lookup and tree traversal level of operations. The following elaborates the methodology used. The underlying physical data is first sorted (done to make the lookup easier and to lay emphasis on data windowing without losing focus onto other issues. In the actual case, data need not be sorted). This facilitates a clustered scheme [12] on the tree indexes and a easier hash table build process. We then build the index table. Index entries are pointers to the actual data (since ours is a memory-resident database) in an effort to save space. In the case of T-trees, the entries are automatically balanced during the population operation.

We build the indexes completely before performing our query operations. This is to prevent the disturbing of indices during operations. There are two indexes for organizer-class queries: name (a_name) based and tag (a_tag) based. Tag defines the relationship to the person in the addressbook. This is to make the lookup faster. Without a tag-based indexing, the algorithm has to traverse all the records in case of a tag lookup. Multiple indexing is anyway cheap in a memory DB (as these indexes are just pointers to memory locations a.k.a. records in DB). In case of TPC-H tables, entries are indexed using customer name for CUSTOMER table, order date for ORDERS, and ship date for LINEITEM. These are the primary lookup fields in our queries. After doing the Join operation, we do not do indexing of the intermediate table. This is to avoid the complexities involved in rebuilding the indexes, and to avoid duplicate index removals. This might result in the lookup of all table entries, but the performance does not suffer much, as seen from the following section.

5. Evaluation of Data Windowing Strategy

5.1. Cache Performance

We first present the results for organizer-class queries and then consider the scenarios from TPC-H queries.

Figure 2 shows the L2 cache performance of organizer-class scenarios with hash indexing and Figure 3 shows that of T-tree indexing. In these figures, *ref* refers to L2 cache references, *miss* refers to L2 cache misses. The legends beginning with D (meaning Data-Window size) represent the number of L2 misses after applying the *data windowing* technique. The number following D indicates the number of entries (number of rows) considered in our technique. As an illustration, D5 indicates the scenario when 5 rows of a table are processed at a time. For the S12 scenario, query P1 reads 5 rows and then finishes processing the data. Before moving on with further rows, P1 transfers control to P2 and P2 reuses the same 5 rows to processes the query. D10 represents 10 rows at a time, D20 represents 20 rows and so on. We present the results only up to D10 since the benefits saturate beyond these values.

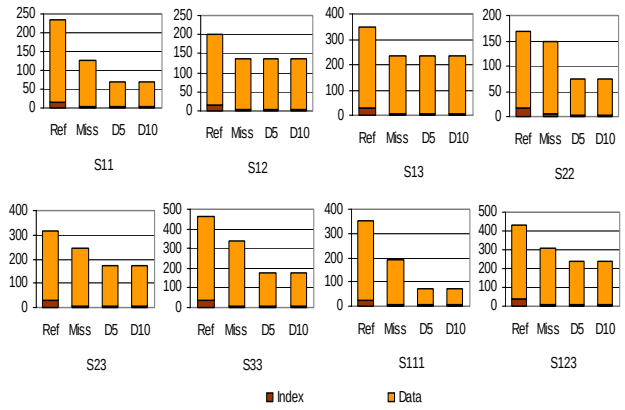

Figure 2. Cache misses for organizer scenarios with hash based indexing scheme

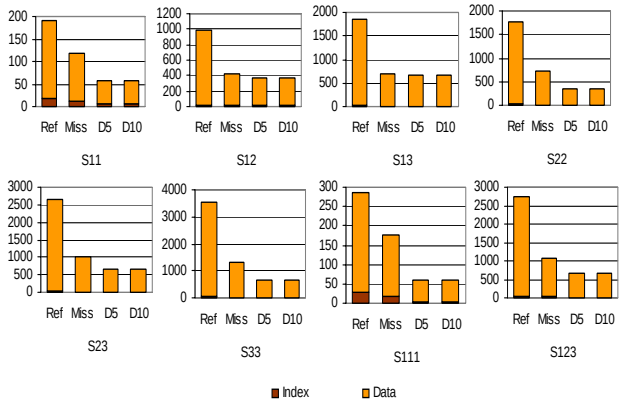

Figure 3. Cache misses for organizer queries with T-tree indexing scheme

The misses that arise from both the index and data references are captured in the graphs of Figure 2 and Figure 3. The L2 cache misses on the hashed indexes and the underlying data reduce by an average of 28% and 33% respectively. In the case of S111, the cached data is reused to the maximum extent to give a 50% reduction in index misses and a 64% reduction in data misses. Looking at the total number of references and the misses, hash based indexing is better than T-tree based indexing for organizer queries. This is attributed to the simple nature of the organizer database. But note that the benefits obtained from our technique are reflected more in T-tree indexing scheme. This is due to the larger number of references to the data in this scheme.

The performance of TPC-H scenarios with hash based and T-tree indexing is shown in Figure 4 and Figure 5. Our technique contributes an average 40% reduction in cache misses for hash indexing and 60% reduction for T-tree indexing. The index misses also reduce by similar percentages. Due to larger table sizes (and also increased number of tables) in the case of TPC-H queries, both T-tree and hash schemes perform at par as against the organizer case. This validates the fact that hash based indexing is good for smaller tables and T-tree indexing is good for larger tables.

The value set for the window size D impacts the cache misses. If the queries are complex and the D values are very small, the cache misses increase. The ability to reuse data decreases if the cache is disturbed a lot, which can happen if the size of D (number of rows considered) is very small. For simple queries, high values of D prove to be either useless or at times, worsen the performance. Hence, a small D size for simple queries and a larger D size for complex queries proves to be a good estimate (results not presented here due to space constraints).

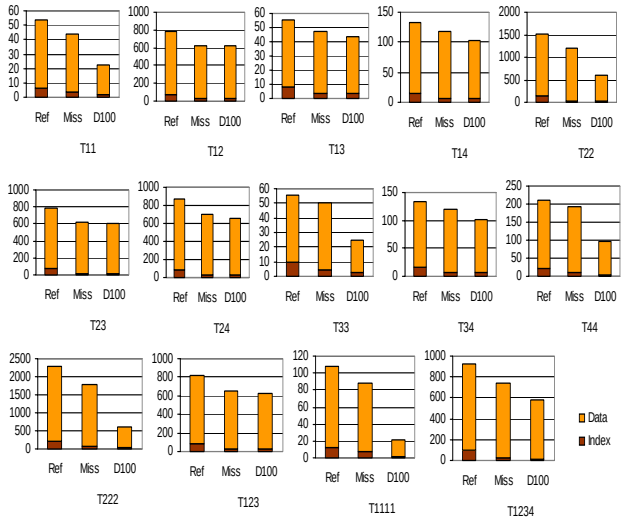

Figure 4. Cache misses for TPC-H scenarios with hash based indexing scheme

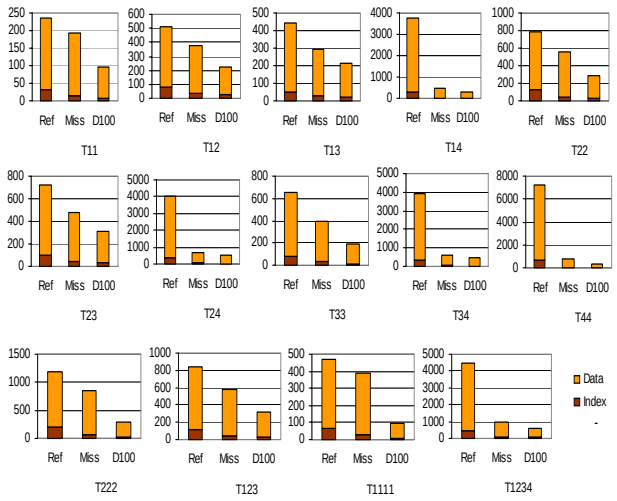

Figure 5. Cache misses for TPC-H scenarios with T-tree indexing scheme

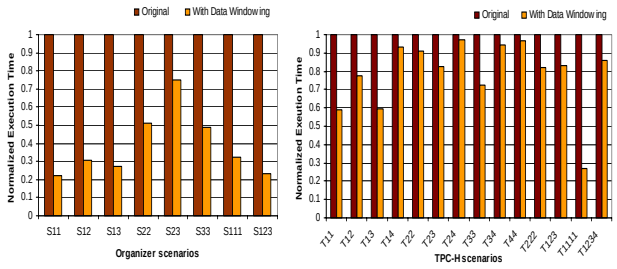

Figure 6. Overall impact of data windowing strategy: Normalized values of average execution times for our scenarios implemented on a memory-resident database.

5.2. System-Wide Impact

We bring the overall aspects of a real system into play by executing our code on a real memory database (hosted on Solaris 2.8, Ultra 10 Sparc system with 1GB memory and the cache configurations of Table 4) and identifying the corresponding response times. Figure 6 shows the actual execution times of our scenarios. The execution times shown here also considers the various overheads involved in data windowing like window creation, and consistency checks. When data windowing is used, there is an average 61% reduction in the execution time for organizer queries and 21% for TPC-H queries.

6. Conclusion

We propose and evaluate a new optimization strategy called *data windowing* that uses the data and not the query execution flow to improve performance. *Data window* is a block of data being accessed by a processor at any given point of time. We move from one window to the next only when all queries finish using the particular window of data. All queries complete execution when all windows of all tables are covered. An average 37% reduction is seen in the overall cache misses when our technique is applied to TPC-H and hand-held device like queries. Thus, our technique targets a data-centric approach and achieves significant improvements in the latency of memory hierarchy. We also studied the impact of our technique on the overall system. Our results showed that data windowing improves the overall execution times as well.

7. References

[1] A. Ailamaki, D.J. DeWitt, M.D. Hill, and D.A. Wood. DBMSs on a Modern Processor: Where Does Time Go?. *The VLDB Journal*, 1999.

[2] A. Ailamaki, D.J. DeWitt, and M.D. Hill. Data Page Layouts for Relational Databases on Deep Memory Hierarchies. *The VLDB Journal*, 11(2), 2002.

[3] Berkeley DB V4.1.25, Sleepycat Software, 2003. Available HTTP: http://www.sleepycat.com/docs/index.html

[4] P. A. Boncz. Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications. *Ph.D. Thesis*, Universiteit van Amsterdam, The Netherlands, May 2002.

[5] R. Choenni, M.L. Kersten, J.F.P. Van den Akker, and A. Saad. On multi-query optimization. *Technical Report CS-R9638*, Centrum voor Wiskunde en Informatica, 1996.

[6] *International Technology Roadmap for Semiconductors - 2002 Update*, ITRS, 2003. Available HTTP: http://public.itrs.net/

[7] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*, 1986.

[8] Main Memory Databases, white paper, Bloor Research Ltd., November 1999.

[9] *Main Memory vs. RAM-Disk Databases*, McObject LLC, 2003. Available HTTP: http://www.mcobject.com/whitepapers.htm

[10] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet. AlphaSort: A RISC Machine Sort. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[11] W. Pugh and D. Won. Constraint-based array dependence analysis. *ACM Transactions on Programming Languages and Systems*, 20(3):635-678, May 1998.

[12] R. Ramakrishnan and J. Gehrke. *Database Management Systems*, third edition, McGraw-Hill publishers, 2002.

[13] K. A. Ross. Conjunctive selection conditions in main memory. In *Proceedings of the 21st ACM Symposium on Principles of Database Systems*, 2002.

[14] Shade V6, Sun Microsystems, 2003. Available HTTP: http://www.sun.com/microelectronics/shade/

[15] A. Shatdal, C. Kant, and J.F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994.

[16] TPC-H Benchmark Revision 2.0.0, Transaction Processing Performance Council, 2003.

[17] P. Trancoso and J. Torrellas. Cache Optimization for Memory-Resident Decision Support Commercial Workloads. In *Proceedings of the International Conference on Computer Design*, 1999.