Cache-Oblivious Query Processing

Bingsheng He, Qiong Luo
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
{saven,luo}@cse.ust.hk

ABSTRACT

We propose a radical approach to relational query processing that aims at automatically and consistently achieving a good performance on any memory hierarchy. We believe this automaticity and stableness of performance is at times more desirable than some peak performance achieved through careful tuning, especially because both database systems and hardware platforms are becoming increasingly complex and diverse. Our approach is based on the cache-oblivious model, in which data structures and algorithms are aware of the existence of a multi-level memory hierarchy but do not assume any knowledge about the parameter values of the hierarchy, such as the number of levels in the hierarchy, the capacity and the block size of each level. Since traditional database systems are intrinsically aware of these parameters, e.g., the memory page size, our cache-oblivious approach requires us to rethink the query processing architecture and implementation. In this position paper, we present the architectural design of our cache-oblivious query processor, EaseDB, discuss the technical challenges and report our initial results.

1. INTRODUCTION

Both relational database systems and hardware platforms are becoming increasingly complex and diverse. As a result, it is a challenging task to automatically and consistently achieve a good query processing performance across platforms. This problem is even more severe for in-memory query processing, because the upper levels of a memory hierarchy, such as the CPU caches, are hard, if feasible at all, to manage by software.

Since CPU caches are an important factor for database performance [3, 11], cache-conscious techniques [10, 11, 14, 22, 35, 41] have been proposed to improve the in-memory query processing performance. In this approach, the capacity and block size of a target level in a specific memory hierarchy, e.g., the L2 cache, are taken as explicit parameters for data layout and query processing. As a result, cache-conscious techniques can achieve a high performance with suitable parameter values and fine tuning. Nevertheless, it is difficult to determine these suitable parameter values at all times and across platforms [7, 20, 21, 33], since they are affected by the characteristics of the memory hierarchy and the system runtime dynamics.

Considering the difficulties faced by the cache-conscious approach, we propose another alternative, namely cache-oblivious query processing, to achieve the goal of improving in-memory performance automatically. Our approach is based on the cache-oblivious model [18] proposed by M. Frigo *et al.* in 1999. A cache-oblivious algorithm is aware of the existence of a multi-level memory hierarchy, but assumes no knowledge about the parameters of the hierarchy. By eliminating the dependency on the memory parameters, cache-oblivious data structures and algorithms [5, 7, 12, 13, 18] usually have provable upper bounds on the number of block transfers between any two adjacent levels of an arbitrary memory hierarchy. Furthermore, this memory efficiency asymptotically matches those more knowledgeable external memory algorithms in many cases.

Despite the nice theoretical properties of the cache-oblivious approach, it is uncertain whether a relational query processor can be implemented in a cache-oblivious manner. One reason is that traditional database systems are designed to be aware of memory parameters. Moreover, most of the cache-oblivious work is on basic data structures and computational algorithms, for example, cache-oblivious B-trees [5, 7], sorting and matrix operations [18]. As such, even if a cache-oblivious query processor is implemented, it is unclear how well it will perform on real machines, especially in comparison with a fine-tuned, cache-conscious counterpart.

In this paper, we present our initial design of EaseDB, the first cache-oblivious query processor for memory-resident relational databases. We also discuss the technical challenges in cache-oblivious query processing and report our experience with designing, implementing, and evaluating cache-oblivious query processing techniques.

2. TECHNICAL CHALLENGES

In this section, we identify five technical challenges in cache-oblivious query processing and discuss them in order.

The first technical challenge lies in the design of cache-oblivious query processing algorithms that asymptotically match the memory efficiency of their cache-conscious counterparts. Existing cache-oblivious work has focused on the memory efficiency, but none of it is about relational query processing. Relational query processing algorithms, such as partitioned hash joins [11, 35], can be both computation and data-intensive. Moreover, these algorithms are carefully designed to take into account the memory parameters and achieve a high memory efficiency.

In contrast, assuming no knowledge about the memory parameters, cache-oblivious techniques are usually designed to divide and conquer [18]. Specifically, a problem is divided into a number of sub-problems recursively and the recursion will not end until the

3^{rd} *Biennial Conference on Innovative Data Systems Research (CIDR)* January 7-10, 2007, Asilomar, California, USA.

smallest unit of computation is reached. The intuition is that, at some point of the recursion, the sub-problem will fit into some level of the memory hierarchy and will further fit into one block as the recursion continues. However, due to the absence of memory-specific parameters, the base case of the recursion is usually set as small constants, e.g., the binary fanout of a cache-oblivious B+-tree. Consequently, the divide-and-conquer process may be unnecessarily long. If this process involves a significant amount of data access, as in query processing, the algorithms need to be carefully designed to improve the memory efficiency.

The second challenge is about the reduction of the recursion overhead in cache-oblivious algorithms without the knowledge of cache parameters. As the base case size in the recursion is usually small, the recursion process goes unnecessarily deep. Consequently, the recursion overhead, especially the computational overhead, becomes significant. For instance, the cache-oblivious non-indexed nested loop join had three times less CPU busy time when the base case increased from one to two tuples in our experiments. Moreover, the dominance of computational cost in the overall time remained for all base cases smaller than 64 tuples. Thus, it is necessary to determine a suitable base case size to avoid the unnecessary recursions.

The third challenge is also a major one in traditional query processing - cost estimation for query optimization. The difference, again, is the absence of memory parameters. Traditional query optimizers estimate costs on CPU instructions [37] and disk accesses [31], and a recent cache-conscious cost model estimates CPU cache stalls with given cache parameters of all levels of caches in the memory hierarchy [11]. However, it is infeasible for a cache-oblivious query processor to estimate the absolute cost without the cache parameter values. Therefore, we need to consider relative costs and to compute expected values for an arbitrary memory hierarchy. Additionally, this cost estimation will be useful for the determination of the base case size.

The fourth challenge is on a cache-oblivious storage model. Existing storage models, e.g., the slotted-page storage model [31], are inherently conscious of the disk, the memory, or the CPU cache parameters. In order to make a query processor completely cache-oblivious, the storage model must be redesigned. A few cache-oblivious in-memory data structures, e.g., the Packed Memory Array (PMA) [5], have been proposed so far, but none of them have been studied in consideration of query processing workloads. Additionally, even though we focus on read-only queries at the current stage, it is desirable to also leave room for update efficiency in our storage model.

The final challenge is on the performance evaluation of our cache-oblivious query processor in comparison with its cache-conscious, fine-tuned counterparts. Most of the cache-oblivious work focused on establishing theoretical bounds of its memory efficiency. In contrast, we emphasize on empirically evaluating the cache performance as well as the overall performance of our query processor on various hardware platforms in addition to studying its theoretical bounds. Furthermore, to facilitate a fair comparison, we need to implement representative cache-conscious algorithms and to tune their parameter values for the best performance. In our experiments, we have found that the performance of cache-conscious algorithms varies greatly with the parameter values and the best parameter values for these algorithms are often none of the parameter values of the memory hierarchy.

3. BACKGROUND AND RELATED WORK

In this section, we first review the background on the memory hierarchy and then cache-centric techniques including both cache-conscious and cache-oblivious techniques.

3.1 Memory hierarchy

The memory hierarchy in modern computers typically contains multiple levels of memory from top down [23]:

- Processor registers. They provide the fastest data access, usually in one CPU cycle. The total size is hundreds of bytes.
- Level 1 (L1) cache. The access latency is a few cycles and the size is usually tens of kilobytes (KB).
- Level 2 (L2) cache. It is an order of magnitude slower than the L1 cache. Its size ranges from 256 KB to a few megabytes (MB).
- Level 3 (L3) cache. It is present in the Intel Itanium [25]. It has a higher latency than the L2 cache. Its size is often several megabytes.
- Main memory. The access latency is typically hundreds of cycles and the size can be several gigabytes (GB).
- Disks. The access latency is hundreds of thousands of cycles. The capacity of a single disk can be up to several hundred gigabytes.

Each level in the memory hierarchy has a larger capacity and a slower access speed than its higher levels. *In this paper, we use the cache and the memory to represent any two adjacent levels in the memory hierarchy whenever appropriate.*

We define a *cache configuration* as a three-element tuple $<C, B, A>$, where C is the cache capacity in bytes, B the cache line size in bytes and A the degree of set-associativity. The number of cache lines in the cache is $\frac{C}{B}$. A=1 is a direct-mapped cache, $A=\frac{C}{B}$ a fully associative cache and $A=n$ an n-associative cache ($1 < n < \frac{C}{B}$).

For the design and analysis of algorithms on the memory hierarchy, a number of cache models have been proposed, such as the external memory model (also known as the cache-conscious model) [1] and the cache-oblivious model [18]. Both models assume a two-level hierarchy, the *cache* and the *memory*. Especially, the cache-oblivious model assumes an ideal cache, which is fully associative and uses an optimal replacement policy [18]. The *cache complexity* of an algorithm is defined to be the asymptotical number of block transfers between the cache and the memory incurred by the algorithm. Frigo *et al.* showed that, if an algorithm has an optimal cache complexity in the cache-oblivious model, this optimality holds on all levels of a memory hierarchy [18].

3.2 Cache-conscious techniques

Cache-conscious techniques [10, 11, 14, 22, 35, 41] have been the leading approach to optimizing the cache performance of in-memory relational query processing. Representatives of cache-conscious techniques include blocking [35], buffering [22, 41] and partitioning [11, 35] for temporal locality, and compression [10] and clustering [15] for spatial locality.

Cache-conscious algorithms explicitly take cache parameters as input. A common way of obtaining these cache parameters is to use calibration tools [27]. One problem of calibration is that some calibration results may be inaccurate or missing, especially as the memory system becomes more complex and diverse. For example, the calibrator [27] does not give the characteristics of TLB on the P4 or Ultra-Sparc machines used in our experiments. Furthermore, the best parameter values for a cache-conscious algorithm may be none of the cache parameters, as our experiments showed. In contrast, cache-oblivious techniques are an automatic approach to performance optimization for the entire memory hierarchy without the need for any cache parameters.

3.3 Cache-oblivious techniques

Two main methodologies of a cache-oblivious algorithm are divide-and-conquer and amortization [16].

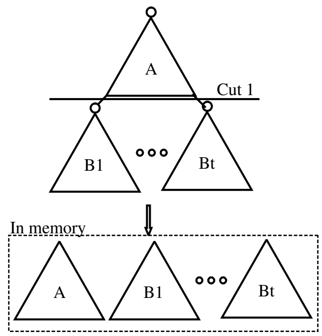

Figure 1: The cache-oblivious B+-tree

The divide-and-conquer methodology is widely used in general cache-oblivious algorithms, because it usually results in a good cache performance. In this methodology, a problem is recursively divided into a number of subproblems. At some point of the recursion, the subproblem can fit into the cache, even though the cache capacity is unknown.

Following the divide-and-conquer methodology, we proposed to use two cache-oblivious techinques, *recursive clustering* [21] and *recursive partitioning* [20], for the spatial and temporal locality of data access, respectively. As their names suggest, recursive partitioning recursively partitions data so that at some level of the recursion a subpartition is fully contained in some level of the memory hierarchy, and recursive clustering recursively places related data together so that a cluster fits into one block at some level of the recursion. An example of recursive partitioning is the quick sort, one of the most efficient sorting algorithms in the main memory.

An example of recursive clustering is the cache-oblivious B+-tree (COB+-tree) [5, 6]. A COB+-tree is obtained by storing a complete binary tree according to the van Emde Boas (VEB) layout [38], as illustrated in Figure 1. Suppose the tree consists of N nodes and has a height of $h = \log_2(N+1)$. The VEB layout proceeds as follows. It first cuts the tree at the middle level, i.e., the edges below the nodes of height $h/2$. This cut divides the tree into a *top* subtree, A, and approximately $\sqrt{N}$ *bottom* subtrees below the cut, B_1, B_2, ..., and B_t. Each subtree contains around $\sqrt{N}$ nodes. Next, it recursively stores these subtrees in the order of A, B_1, B_2, ..., and B_t. The memory efficiency of one search on a complete binary subtree stored in the VEB layout matches that of a cache-conscious B+-tree (CCB+-tree) [5, 6].

The amortization methodology is used to reduce the average cost per operation for a set of operations, even though the cost of a single operation may be high. We use amortization to improve the reuse of the data that reside in the cache. Specifically, we have designed a novel cache-oblivious buffer hierarchy to amortize data accesses to the buffers [20, 21], as shown in Figure 2. These buffers are organized into a hierarchy. When a buffer is full, we flush the buffer by distributing the buffered items to its child buffers recursively.

The sizes of buffers in the hierarchy are recursively defined. This definition follows the VEB recursion [38]. At each cut of the recursion, we define the sizes of the buffers at the middle level. If the tree contains N buffers, we set the size of a buffer at the middle level to be $N^{\frac{1}{2}} \log_2 N^{\frac{1}{2}}$, which equals the total size of the buffers in the bottom subtree. This process ends when the tree contains only one buffer. If the size of the buffer for this node has not been determined, it is set to a small value. For example, if the buffer is used in a tree index, it is set to the index node size [21]. At some level of the recursion, buffers can fit into the cache and are reused before they are evicted from the cache. Thus, the average cost of each operation is reduced.

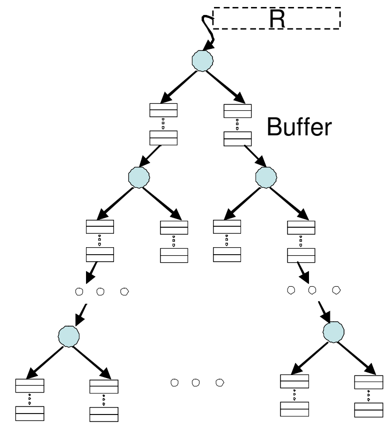

Figure 2: A buffer hierarchy in EaseDB. The buffers form a binary tree structure. Their sizes are recursively defined.

Table 1: Notations used in this paper

Parameter	Description
C	Cache capacity (bytes)
B	Cache line size (bytes)
R, S	Outer and inner relations of the join
r,s	Tuple sizes of R and S (bytes)
$\|R\|$,$\|S\|$	Cardinalities of R and S
$\|\|R\|\|$,$\|\|S\|\|$	Sizes of R and S (bytes)
C_S	The base case size in number of tuples

Representatives of existing cache-oblivious techniques include recursive partitioning [18] and buffering [12, 18] for temporal locality, and recursive clustering [5] for spatial locality. Especially for query processing, the following cache-oblivious data structures and algorithms match the cache complexity of their cache-conscious counterparts. The notations used throughout this paper are summarized in Table 1.

- B+-trees [7]: The cache complexity of a search on the B+-tree for relation R is $O(\log_B |R|)$.
- Sorting [13]: The cache complexity of sorting relation R is $O(\frac{|R|}{B} \log_C |R|)$.
- Nested-loop joins (NLJ) [21]: The cache complexity of joining relations R and S without and with the B+-tree index is $O(\frac{|R| \cdot |S|}{C \cdot B})$ and $O(\frac{|R|}{B} \log_C |S|)$, respectively. For indexed NLJs, a COB+-tree is built on S in advance.
- Hash joins [20]: The cache complexity of joining relations R and S is $O(\frac{|R|}{B} \log_C |S|)$, where R and S are the probe and build relations, respectively.

Our previous work [20, 21] has designed cache-oblivious algorithms for NLJs and hash joins. In this paper, we discuss architectural design and implementation issues in developing a full-fledged cache-oblivious query processor instead of focusing on individual algorithms.

Table 2: Main memory relational query processors

	FastDB	*TimesTen*	*MonetDB*	*EaseDB*
Category	cache-conscious	cache-conscious	cache-conscious	cache-oblivious
Query optimizer	rule-based	instruction cost-based	cache cost-based	cache cost-based
Supported indexes	T-tree, hash index	B+-Tree, T-tree, hash index	B+-Tree, T-tree, hash index	B+-Tree, hash index
Access to database file	virtual memory	main memory	virtual memory	main memory
Storage model	row-based	row-based	column-based	row-based

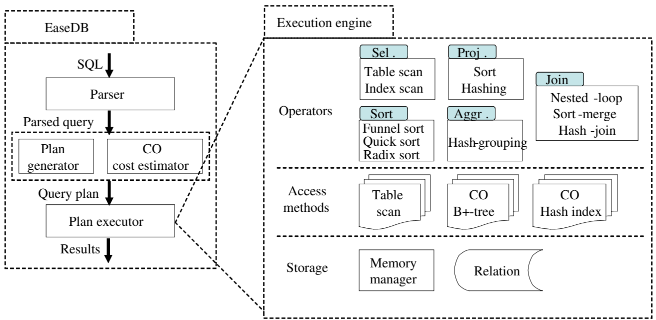

Figure 3: The system architecture of EaseDB

4. EASEDB

In this section, we describe the architectural design and implementation status of EaseDB, our cache-oblivious query processor. The goal of EaseDB is to automatically and consistently achieve a good performance on various machines at all times.

4.1 System Overview

Figure 3 shows the system architecture of EaseDB. There are three major components, namely the SQL parser, the query optimizer, and the plan executor. The query optimizer in turn consists of a query plan generator and a cache-oblivious cost estimator. At the time of writing, we have finished the implementation of most of the plan executor and the cost estimator. We will reuse the parser component of PostgreSQL [30] for our SQL parser and will use a Selinger-style optimizer [34] for plan generation. Additionally, a two-phase optimization strategy [24] will be used to limit the plan search space in the optimizer.

EaseDB currently runs in a single process (thread). We will apply a multi-threading mechanism at the operator level, similar to that in staged databases [19]. We expect this multi-threading mechanism will boost the performance advantage of cache-oblivious algorithms over cache-conscious ones, because cache-oblivious algorithms are more robust on cache coherence among concurrent threads in a processor [20]. Moreover, the self-optimizing nature of cache-oblivious algorithms eliminates the tuning work required in a multi-threaded environment.

Table 2 summarizes the main features of EaseDB in comparison with three existing main memory relational query processors, including FastDB [17], TimesTen [37] and MonetDB [29]. These three query processors are cache-conscious, whereas EaseDB is cache-oblivious. EaseDB uses cache cost as the cost metric in the query optimizer, because the memory hierarchy has become an important factor in the performance of relational query processing.

In the following, we focus our discussion on the plan executor and the cost estimator.

4.2 Execution engine

We divide the execution engine into three layers, including the storage, access methods and query operators.

4.2.1 Storage

Currently EaseDB uses arrays to represent relations in a row-based manner. Recent work [2, 29, 36] has shown that column-based and row-based storage models have their own pros and cons. Our decision towards a row-based storage model is mainly due to its simplicity, because relational rows map naturally to arrays in the main memory.

We consider both static data (no updates) and dynamic data in our storage model. A generic array for static data is sufficient; however, its performance suffers from updates. In comparison, the linked list is a common storage model for dynamic data in a cache-conscious setting. The node size of the linked list is determined according to the cache block size. For example, the slotted page model is essentially a linked list with a node size equal to the page size. However, without the knowledge of the cache block size, the linked list may have a very bad scan performance. Consider a node of size s. This node spans $\lceil s/B \rceil + 1$ cache lines in the worst case, even though s can be very small. Therefore, an efficient cache-oblivious storage model that balances the scan and update costs is required.

We use the packed memory array (PMA) [5] as a storage model for dynamic data. To make our presentation self-contained, we briefly describe how PMA works [5]. PMA is essentially an array with some unoccupied (or free) slots. The density of an array is defined as the ratio of the number of elements over the total number of slots in the array. A generic array has a density of 100%.

The basic idea of PMA is to define windows of different sizes as contiguous areas of different sizes on the array, and to further define the density thresholds for these windows. The window is defined after its size: a k-window means the size of the window is k, whose slots start from index $(j-1)*k$ ($1 \le j \le N/k$, N is the number of slots in the array). For simplicity, $k = 2^i$ ($i = log_2 log_2 N, log_2 log_2 N + 1, ..., log_2 N$). That is, the smallest window is of a size $log_2 N$ and the largest one of N. The density thresholds are defined with the following two rules: (1) the upper-bound density threshold *decreases* linearly as the window size increases; (2) the lower-bound density threshold *increases* linearly as the window size increases. Given the lower-bound and upper-bound density thresholds of the $log_2 N$-window and the N-window, all density thresholds for windows of different sizes can be determined.

Inserting a tuple at index j of PMA is handled as follows. We examine the 2^i-windows containing $j, i = log_2 log_2 N, log_2 log_2 N$+1, ..., $log_2 N$ until the first window is found whose density is smaller than its upper-bound density threshold. After inserting the tuple into this window, we redistribute the elements evenly within the window. Deletions are handled similarly. By design, PMA has the following memory efficiency bounds: (1) scanning any n consecutive elements causes $O(n/B + 1)$ cache misses; (2) inserting or

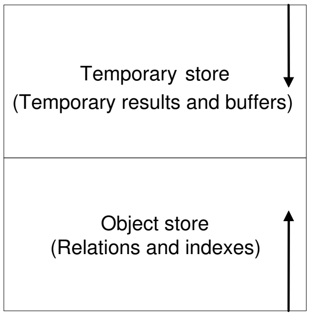

Figure 4: The memory pool in EaseDB

deleting an element causes $O(\frac{(\log_2 N)^2}{B} + 1)$ cache misses on average.

We have two considerations for using PMA in EaseDB. First, we will adjust the density thresholds of the $log_2 N$- and N-windows. This adjustment will be based on the frequency of insertion and deletion. Second, we will use strict two-phase locking [31] for concurrency control on PMA. A PMA window is locked when the redistribution of elements is propagated to this window.

After introducing the storage model in EaseDB, we discuss our design on memory management. EaseDB has a memory manager to handle memory allocation and deallocation. Similar to other main memory databases [17, 29, 37], EaseDB does not have a global buffer manager. When EaseDB starts up, it pre-allocates a large memory pool. The memory space required for query processing is allocated from the pool on demand.

We further divide the memory pool into two parts, the *temporary store* and the *object store*, as shown in Figure 4. The temporary store allocates the memory from top down, whereas the object store from bottom up. If these two stores overlap, the memory pool is running out of free memory. When this happens, either some memory will be released from the pool or a new memory pool will be allocated.

The temporary store is for storing temporary results and auxiliary data structures such as buffers for query processing. The nature of these data is dynamic. Therefore, it uses the sequential fit scheme [9] in memory allocation for high memory utilization. In the sequential fit scheme, the process of handling a memory request is to traverse all memory blocks until it reaches a free block that is large enough to satisfy the memory request. To release a memory block, it locates the block, releases it and may merge it with adjacent blocks of free memory. The original sequential fit scheme uses a linked list to maintain the state information of memory blocks [9]. In EaseDB, we use PMA and a cache-oblivious B+-tree built on top of the PMA to speed up the search process for a fit free block. Note, the space allocated in the temporary store is private to a single thread so that the overhead of concurrency control on these space is avoided.

The object store is used to store the base relations and the indexes in EaseDB, which are more stable than the data in the temporary store. This object store is shared among threads. Since the relations and the indexes are all stored using PMA, the start addresses of their corresponding PMAs are located via a hash index on the relation name or index name.

4.2.2 Access methods and query operators

In the current status, EaseDB supports three common access methods, including the table scan, the B+-tree and the hash index.

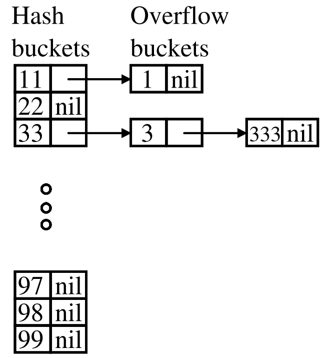

Figure 5: The cache-oblivious hash index

- *Table scan*. The relation is sequentially scanned or a binary search is performed on the relation according to the sorted key in the relation (if applicable).
- *B+-trees*. The COB+-tree has the same memory efficiency bounds as the CCB+-tree, and its performance on disk-based applications is close to or even better than the fine-tuned CCB+-tree [7]. For in-memory query processing, the COB+-tree has a very similar performance to the CCB+-tree.
- *Hash indexes*. The hash index for R consists of $|R|$ buckets so that each bucket is expected to contain one tuple. An example of such a hash index is shown in Figure 5. The memory efficiency bound of a probe on the hash index is $O(1)$, which matches the memory efficiency bound of a probe on the cache-conscious hash index [31]. However, the cache-oblivious hash index has a space overhead due to the large number of hash buckets.

We next briefly discuss cache-oblivious algorithms for common query processing operators.

- *Selection*. In the absence of indexes, a sequential scan or binary search on the relation is used. In the presence of indexes, if the selectivity is high, a data scan on the relation is performed. Otherwise, the B+-tree index or the hash index can be used.
- *Projection*. If duplicate elimination is required, we use either sorting or hashing to eliminate the duplicates for the projection.
- *Sorting*. Two cache-oblivious sorting algorithms, namely the funnel sort [12, 18] and the distribution sort [18], have been proposed. The funnel sort is essentially the merge sort implementation with cache-oblivious buffers. Experimental results [13] show that the funnel sort can outperform the fine-tuned quick-sort in the main memory. Therefore, we chose the funnel sort as one of our sorting methods.

  Additionally, we have developed a cache-oblivious radix sort, because a cache-conscious radix sort has been developed [11]. Our cache-oblivious radix sort is the binary radix sort with our buffer hierarchy. The sorting process is similar to that of the cache-oblivious hash join [20] except for two differences. First, at the ith phase of the radix sort ($i \geq 0$), we use the $(\log_2 Max - i)$th bit from the right, where Max is the maximum key value of the tuples in the relation. Second, we use the insertion sort to sort the base case.

In this paper, we used the radix sort and leave the comparison study on the two sorting algorithms as future work.

- *Grouping and aggregation.* We use the build phase of the hash join [20] to perform grouping and aggregation.

- *Joins.* We have implemented cache-oblivious nested-loop joins with or without indexes [21], sort-merge joins, and hash joins [20]. We briefly introduce the cache-oblivious techniques used in our join algorithms.

  The non-indexed NLJ (denoted as CO_NLJ) performs recursive partitioning on both relations. With recursive partitioning, both relations are recursively divided into two equal-sized sub-relations, and the join is decomposed into four smaller joins on the sub-relation pairs. This recursive process goes on until the base case is reached.

  The indexed NLJ uses cache-oblivious buffers to improve its temporal locality. Buffering is performed on each level of the tree index. Each non-leaf node is associated with a buffer, which temporarily stores query items for the node.

  The sort-merge join sorts both relations using a cache-oblivious sorting algorithm, and merges the sorted relations on a per-tuple basis.

  The hash join has been implemented using recursive binary partitioning and cache-oblivious buffers.

  Each of these join algorithms has the same memory efficiency bound as its cache-conscious counterpart.

4.3 Cost estimator

When multiple query plans are available, the optimizer chooses the one of the minimum cost. In the optimizer of EaseDB, the cost estimator estimates the cache cost of a query plan in terms of the data volume transferred between the cache and the memory. Compared with the cache-conscious cost model [11], our estimation assumes no knowledge of the cache parameters of each level or the number of levels in a specific memory hierarchy.

Our cost estimator estimates the expected volume of data transferred (in bytes) between the cache and the memory in the cache-oblivious model. This two-level memory hierarchy has the following characteristics for the simplicity of the cost estimation. First, both the cache block size and the cache capacity are a power of two. Second, B is no larger than $\sqrt{C}$ according to the tall cache assumption ($B \leq \sqrt{C}$) [18]. Note that these assumptions are based on some fundamental properties of the memory hierarchy rather than specific parameter values. Consequently, there is no tuning or calibration involved.

To compute the expected volume of data transferred between the cache and the memory caused by the algorithm, we need a cost function of the algorithm. This cost function estimates the number of cache misses caused by the algorithm for a given cache capacity and cache block size. We denote this cost function to be $F(C, B)$.

Suppose a query plan has a working set size ws bytes, which is the total size of the data (e.g., relations and indexes) involved in the query plan. We consider all possible combinations of the cache capacity and the cache block size to compute the expected data volume transferred between the cache and the memory on an arbitrary memory hierarchy. If $C \geq ws$, this working set can fit into the cache. Once data in the working set are brought into the cache, they stay in the cache for further processing. That is, further processing does not increase the data volume transferred. We estimate the data volume of the cases that the cache capacity is larger than the working set of the query plan to be zero. Thus, given a certain C_x value, we consider all possible B values and compute the expected volume of data transferred to be $(\frac{1}{\log_2 \sqrt{C_x}+1}(1 \cdot F(C_x,1) + 2 \cdot F(C_x,2) + 4 \cdot F(C_x,4) + ... + \sqrt{C_x} \cdot F(C_x, \sqrt{C_x}))) = \frac{1}{\log_2 \sqrt{C_x}+1} \sum_{i=0,B_x=2^i}^{i=\log_2 \sqrt{C_x}} (B_x \cdot F(C_x, B_x))$.

Therefore, we estimate the expected volume of data transferred to be $\mathcal{Q}(F)$.

$$\mathcal{Q}(F) = \frac{1}{\log_2 ws + 1} \times \sum_{k=0,C_x=2^k}^{k=\log_2 ws} \frac{1}{\log_2 \sqrt{C_x}+1} \sum_{i=0,B_x=2^i}^{i=\log_2 \sqrt{C_x}} (B_x \cdot F(C_x, B_x)).$$

We use the cost estimator to determine the query plan in the optimizer. Given an input query, the plan generator generates multiple possible query plans. Then, we use the cost estimator to estimate the expected volume of data transferred for each query plan. Thus, the optimizer determines which of those plans will be the most efficient, and recommends it to the execution engine.

We also use the cost estimator to determine the suitable base case size for a cache-oblivious algorithm. The base case size is important for the efficiency of divide-and-conquer algorithms. A small base case size results in a large number of recursive calls, which can yield a significant overhead. A large base case size may cause cache thrashing. Since the cost estimator gives the cache cost of an algorithm, we use it to compare the costs with and without the divide-and-conquer operation for a given problem size. We obtain the suitable base case size to be the maximum size of which the problem is small enough to stop the divide-and-conquer process.

Take CO_NLJ as an example. We use the cost estimator to compute the minimum sizes of the relations that are worthwhile to be partitioned in the NLJ. Specifically, we compare the cache costs for the NLJ without and with partitioning: (1) the join is evaluated as a base case, and (2) we divide the join into four smaller joins and evaluate each of these smaller joins as a base case. Since CO_NLJ uses the tuple-based simple NLJ to evaluate the base case, the cost functions of the NLJ without and with partitioning are given as F and F', respectively. Note, we define fc to be the size of the data (in bytes) brought into the cache for a recursive call.

$$F(C_x, B_x) = \begin{cases} \frac{1}{B_x}(||R|| + |R| \cdot ||S||) & , ||S|| \geq C_x \\ \frac{1}{B_x}(||R|| + ||S||) & , otherwise \end{cases}$$

$$F'(C_x, B_x) = \begin{cases} \frac{1}{B_x}(2||R|| + |R| \cdot ||S|| + 4fc) & ,||S|| \geq 2C_x \\ \frac{1}{B_x}(||R|| + ||S|| + 4fc) & ,||S|| \leq \frac{C_x}{2} \\ \frac{1}{B_x}(2||R|| + ||S|| + 4fc) & ,otherwise \end{cases}$$

With the cost estimator, we obtain $\phi = \mathcal{Q}(F)$ and $\phi' = \mathcal{Q}(F')$. Given the condition of recursive partitioning in CO_NLJ, $||R|| = ||S||$, we obtain the minimum base case size when $\phi > \phi'$.

Finally, we note that a cache-oblivious cost model for the mesh layout was proposed by Yoon *et al.* [39, 40]. Yoon's model estimates the expected number of cache misses caused by the accesses to a mesh. In contrast, our cost model estimates the expected volume of data transferred between the cache and the memory for general query processing algorithms.

5. PRELIMINARY RESULTS

We present our preliminary performance results on a few core components of EaseDB, specifically the PMA storage model, the access methods and the join algorithms.

5.1 Experimental setup

Our experiments were conducted on three machines of different architectures, namely P4, AMD and Ultra-Sparc. The main features of these machines are listed in Table 3. The L2 caches on all three platforms are unified. The Ultra-Sparc does not support hardware prefetching data from the main memory, whereas both P4 and AMD do. AMD performs prefetching for ascending sequential accesses only whereas P4 supports prefetching for both ascending and descending accesses. In modern CPUs, a translation lookaside buffer (TLB) is used as a cache for physical page addresses, holding the translation for the most recently used pages. We treat a TLB as a special CPU cache, using the memory page size as its cache line size, and calculating its capacity as the number of entries multiplied by the page size.

Table 3: Machine characteristics

Name	P4	AMD	Ultra-Sparc
OS	Linux 2.4.18	Linux 2.6.15	Solaris 8
Processor	Intel P4 2.8GHz	AMD Opteron 1.8GHz	Ultra-Sparc III 900Mhz
L1 DCache	<8K, 64, 4>	<128K,64,4>	<64K, 32, 4>
L2 cache	<512K,128,8>	<1M, 128, 8>	<8M, 64, 8>
DTLB	64	1024	64
Memory	2.0 GB	15.0 GB	8.0 GB

Intel P4 supports the SMT (Simultaneous Multi-Threading) feature to allow two concurrent threads running in one processor [28]. Since the L2 cache is shared by the two running threads in SMT, we can evaluate the performance impact of cache coherence on our algorithms. All experiments were conducted in a single-threaded environment except the one evaluating the performance impact of SMT.

All algorithms were implemented in C++ and were compiled using g++ 3.2.2-5 with optimization flags (*O3* and *finline-functions*). In our experiments, the memory pool was set to be $1.5G$ bytes on all platforms. The data in all experiments were always memory-resident and the memory usage never exceeded 90%.

The workload contains two selection queries and two join queries on relations R and S. Both tables contain n fields, $a_1, ..., a_n$, where a_i is a randomly generated 4-byte integer. We varied n to scale up or down the tuple size. The tree index was built on the field a_1 of R and S. The hash index was built on the field a_1 of R.

The selection queries in our experiments are in the following form:

Select $R.a_1$
From R
Where <predicate>;

One of the two selection queries is with a non-equality predicate ($x - \delta < R.a_1 < x + \delta$) and the other an equality predicate ($R.a_1 = x$). Note that x is a randomly generated 4-byte integer. We used the B+-tree index to evaluate the non-equality predicate and the hash index to evaluate the equality predicate. Since we focused on the search performance of the tree index, we set δ to a small constant such as ten in the non-equality predicate.

The join queries in our experiments are:

Select $R.a_1$
From R, S
Where <predicate>;

One of the two join queries is an equi-join and the other non-equijoin. The equi-join takes $R.a_1 = S.a_1$ as the predicate and the non-equijoin $R.a_1 < S.a_1$ *and...and* $R.a_n < S.a_n$. All fields of each table are involved in the non-equijoin predicate so that an entire tuple is brought into the cache for the evaluation of the predicate. We used the non-indexed NLJ to evaluate the non-equijoin and used the indexed NLJ or the sort-merge join or the hash join to evaluate the equi-join.

Table 4 lists the main performance metrics used in our experiments. We used the C/C++ function $clock()$ to obtain the total execution time on all three platforms. In addition, we used a hardware profiling tool, PCL [8], to count cache misses on P4.

Table 4: Performance metrics

Metrics	Description
TOT_CYC	Total execution time in seconds (sec)
L1_DCM	Number of L1 data cache misses in billions (10^9)
L2_DCM	Number of L2 data cache misses in millions (10^6)
TLB_DM	Number of TLB misses in millions (10^6)

For the cache-conscious algorithms in our study, we varied their parameter values to examine the performance variance. Given a cache parameter, y, of a target level in the memory hierarchy (either C or B) of an experiment platform, we varied the parameter value x in a cache-conscious algorithm within a range so that the three cases $x < y$, $x = y$ and $x > y$ were all observed. Given a multi-level memory hierarchy, we considered all cache levels and varied the cache parameter value in a cache-conscious algorithm for every level of the cache.

5.2 PMA

First, we compared the performance of PMA and the linked list for our consideration on the choice of the storage model. We began with an empty relation and performed random insertions into the relation. The relation was stored using PMA or the linked list.

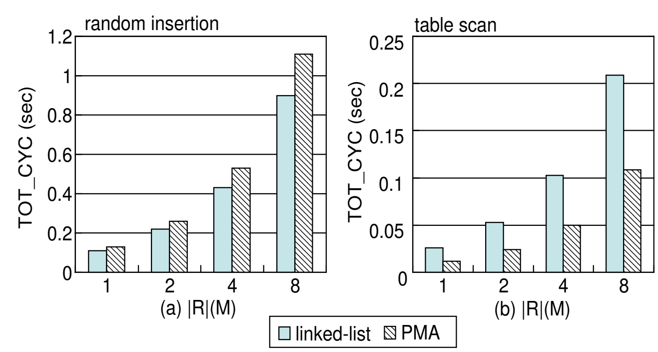

Figure 6: Performance of PMA and the link list on P4. On the left is the insertion performance. On the right is the scan performance.

Figures 6 (a,b) show the performance of PMA and the linked list for insertion and scan, respectively, when the upper-bound density thresholds for the $\log_2 |R|$- and $|R|$-windows are set to be 100% and 90%. These high density thresholds may cause a large redistribution overhead so that we can observe the behavior of these two choices under stress tests on updates. We obtained similar results when the density thresholds were varied. We fixed r to be 8 bytes and varied $|R|$ in millions of tuples (M) to examine the effect of the data size. The figures show that insertion with PMA is slightly slower than that with the linked list, whereas the scan on

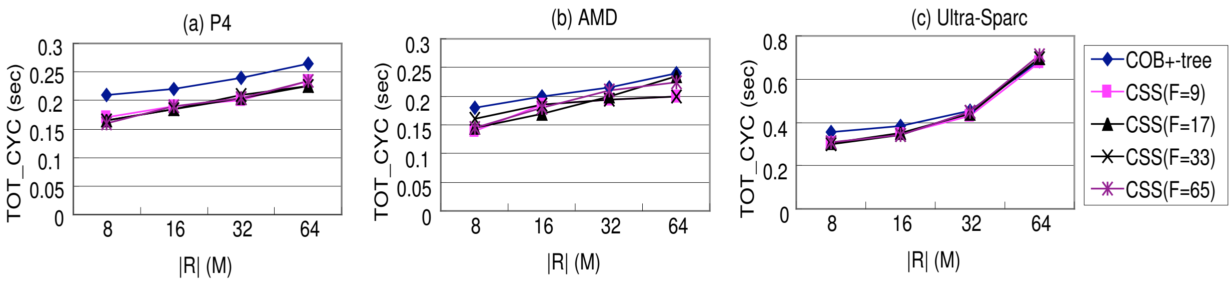

Figure 7: Performance comparison of COB+-trees and CSS-trees

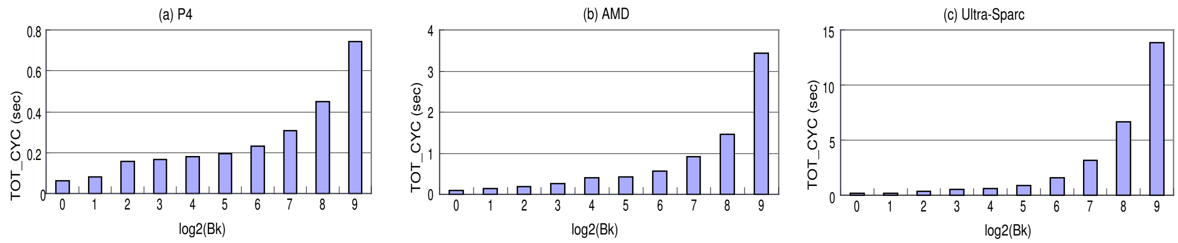

Figure 8: Performance of hash indexes. The hash index with $Bk = 1$ is our cache-oblivious hash index in EaseDB.

PMA is over twice faster than that on the linked list. This result clearly indicates that PMA is the choice for the storage model in our cache-oblivious setting.

5.3 Access methods

We evaluated our access methods including the B+-tree and the hash index. We executed a number of selection queries on each access method.

5.3.1 B+-trees

We compared the search performance of COB+-trees and cache sensitive search trees(CSS-trees) [32]. On each platform, we varied the fanout of the CSS-tree from 9 to 65. The node size was varied from 32 to 256 bytes. Figure 7 shows the performance comparison with the number of tuples in the relation varied. The number of selection queries executed was $200K$.

On all platforms, COB+-trees have a similar performance to CSS-trees. The performance gap between them becomes smaller as the relation size increases. When $|R| = 64M$, the COB+-tree is around 20% slower than the best CSS-tree on P4 and AMD, and it is less than 1% slower than the best CSS-tree on Ultra-Sparc.

5.3.2 Hash indexes

We evaluated the hash index with the average number of tuples in each bucket, Bk, varied. The number of selection queries executed was $200K$. Figure 8 shows the performance of hash indexes when Bk is varied from one to 512. Since each tuple takes 8 bytes (four bytes for the value and four bytes for the record identifier), the bucket size is around $(8 \times Bk)$ bytes. That is, the bucket size is varied from 8 to $4K$ bytes. Note, the cache-oblivious hash index is the one with $Bk = 1$.

On all platforms, the performance degrades dramatically as the Bk value increases. Due to the hardware prefetching capability, P4 has a smaller performance degradation than Ultra-Sparc. The cache-oblivious hash index is faster than the cache-conscious one, because its bucket size is smaller and the computation time on each bucket is likely to be smaller. However, it has a larger space overhead due to its larger number of hash buckets.

5.4 Join algorithms

We verified the effectiveness of our cost estimator and evaluated the performance of our cache-oblivious join algorithms in comparison with the best performance of their cache-conscious counterparts. The reported results for non-indexed NLJs and indexed NLJs were obtained when $|R| = |S| = 256K$ and $r = s = 128$ bytes, $|R| = |S| = 5M$ and $r = s = 8$ bytes, respectively. The results for sort-merge joins and hash joins were obtained when $|R| = |S| = 32M$ and $r = s = 8$ bytes. These settings were chosen to be comparable to the previous studies on cache-conscious join algorithms [11, 35, 41]. More detailed results on each individual algorithm can be found in our related papers [20, 21].

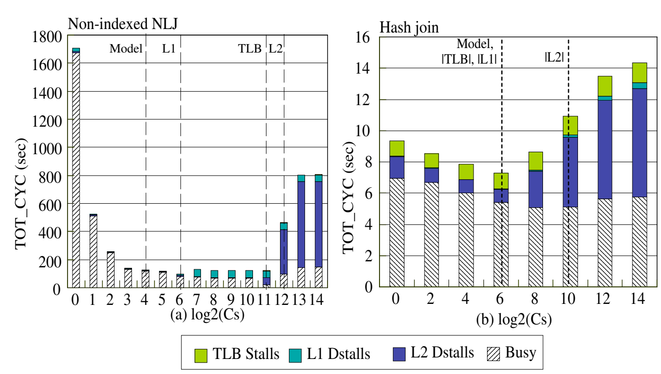

Figure 9: Performance of join algorithms with the base case size varied on P4. On the left is the non-indexed nested-loop join. On the right is the hash join.

5.4.1 *Model verification*

Figure 9 shows the time breakdown of two representative cache-oblivious join algorithms, non-indexed NLJs and hash joins, with the base case size varied on P4. The results for the sort-merge join algorithm are not shown in this figure, since they were similar to those of the hash join. The busy time is obtained by subtracting the three types of cache stalls from the total elapsed time. The base case for the non-indexed NLJ is a join on two equal-sized partitions. The size of each partition is compared with the capacities of the L1 and L2 caches, and the TLB. The base case for the hash join is a join with a small hash table. Since each hash bucket is likely to be one cache line, the size of the hash table is compared with the number of cache lines in the L1 and L2 data caches, and the number of entries in the TLB.

With the estimated base case size, the busy time and cache stalls of both algorithms are greatly reduced. These results verify the effectiveness of our cost estimator. One may conjecture from Figure 9(a) that the L1 cache capacity is a better guess than our cost estimator for the base case size on P4; unfortunately, this particular phenomenon does not hold for different platforms, different algorithms, or different data sizes.

5.4.2 *Single-threaded evaluation*

Figure 10 shows the performance measurements of join algorithms on the three platforms.

Figures 10 (a–c) show the performance for non-indexed NLJs. The cache-conscious algorithm is the blocked NLJ [35], whose parameter is the block size of the inner relation. We varied the block size of the blocked NLJ from $4KB$ to $16MB$.

Figures 10 (d–f) show the performance for indexed NLJs with cache-oblivious and cache-conscious buffering schemes. In cache-conscious buffering [41], an index tree is organized into multiple subtrees (called *virtual node*s [41]) and the root of each subtree is associated with a buffer. The parameter of this cache-conscious buffering is the number of tree levels in a virtual node. Given an index tree of l levels, we varied the number of levels in a virtual node from one to $(l-1)$.

Figures 10 (g–i) show the performance for sort-merge joins. The sorting algorithm is the radix sort. The cache-conscious radix sort works in two phases. First, given the partition granularity, gr bytes, it divides the relation into multiple partitions using the radix cluster algorithm [11]. Each partition is around gr bytes. Second, it sorts each partition using the quick sort. These figures show the best performance obtained in our tuning on the radix cluster algorithm for the given partition granularity.

Figures 10 (j–l) show the performance for hash joins. The cache-conscious hash join is the radix join [11]. In the radix join, we need to tune the partition fanout and the partition granularity. Given a certain partition granularity, gr bytes, we varied the partition fanout and measured the execution time. These figures show the best performance obtained in our tuning for the given partition granularity.

We analyze the results of Figure 10 on three aspects. First, we study the performance variance of each cache-conscious algorithm. On a given platform, the performance variance of a cache-conscious join algorithm with different parameter values is large. Furthermore, the performance variance of a cache-conscious algorithm differs across platforms. For example, the performance variance of cache-conscious indexed NLJs on Ultra-Sparc is larger than the other two platforms, whereas the performance variance of cache-conscious non-indexed NLJs, sort-merge joins and hash joins on Ultra-Sparc is smaller than the other two platforms.

Second, we study the best parameter values for the cache-conscious algorithms. On a given platform, the best parameter value for a cache-conscious join algorithm may be none of the cache parameters, e.g., the sizes of the L1 and L2 data caches, or the number of entries in the TLB. Moreover, for a given cache-conscious algorithm, the best parameter values differ across platforms. These results show it is difficult to determine the best parameter values on different platforms even with the knowledge of the cache parameters.

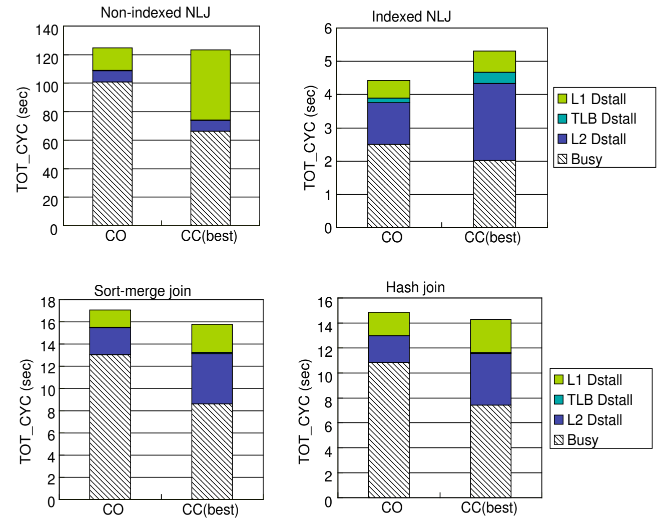

Figure 11: Time breakdown of cache-oblivious algorithms and the best cache-conscious algorithms on P4

Third, we compare the overall performance of cache-conscious and cache-oblivious algorithms on three platforms. Regardless of the architectural differences among the three platforms, our join algorithms provide a robust and good performance. In specific, the performance of our join algorithms is close to, if not better than, the best performance of cache-conscious join algorithms on P4, and is mostly better than the best performance of cache-conscious join algorithms both on AMD and Ultra-Sparc.

We further examine the time breakdown of our cache-oblivious algorithms and the best cache-conscious algorithms in Figure 11. The total cache stalls of the cache-conscious join algorithms are significant, because these algorithms typically optimize only for the cache of a chosen level and cache thrashing may occur at the levels other than the chosen one. In contrast, the cache stalls of our cache-oblivious algorithms are less significant due to their automatic optimization for the entire memory hierarchy. The busy time is significant among all cache-oblivious join algorithms.

5.4.3 *Multi-threaded evaluation*

Finally, we investigated the performance impact of cache interference on cache-conscious and cache-oblivious algorithms. We observed that the execution time of each thread running in the multi-threaded environment was not stable, whereas the system throughput was stable. The variance in the execution time of each thread is because of the resource contention and sharing in SMT at runtime. Therefore, we used the system throughput as the performance metric in the multi-threaded environment. Figure 12 shows the throughput improvement of SMT on P4. Both of the concurrent threads ran the same algorithm, either the cache-oblivious algorithm or the best cache-conscious algorithm.

SMT improves the throughput of both cache-conscious and cache-oblivious algorithms. The improvement to the cache-oblivious al-

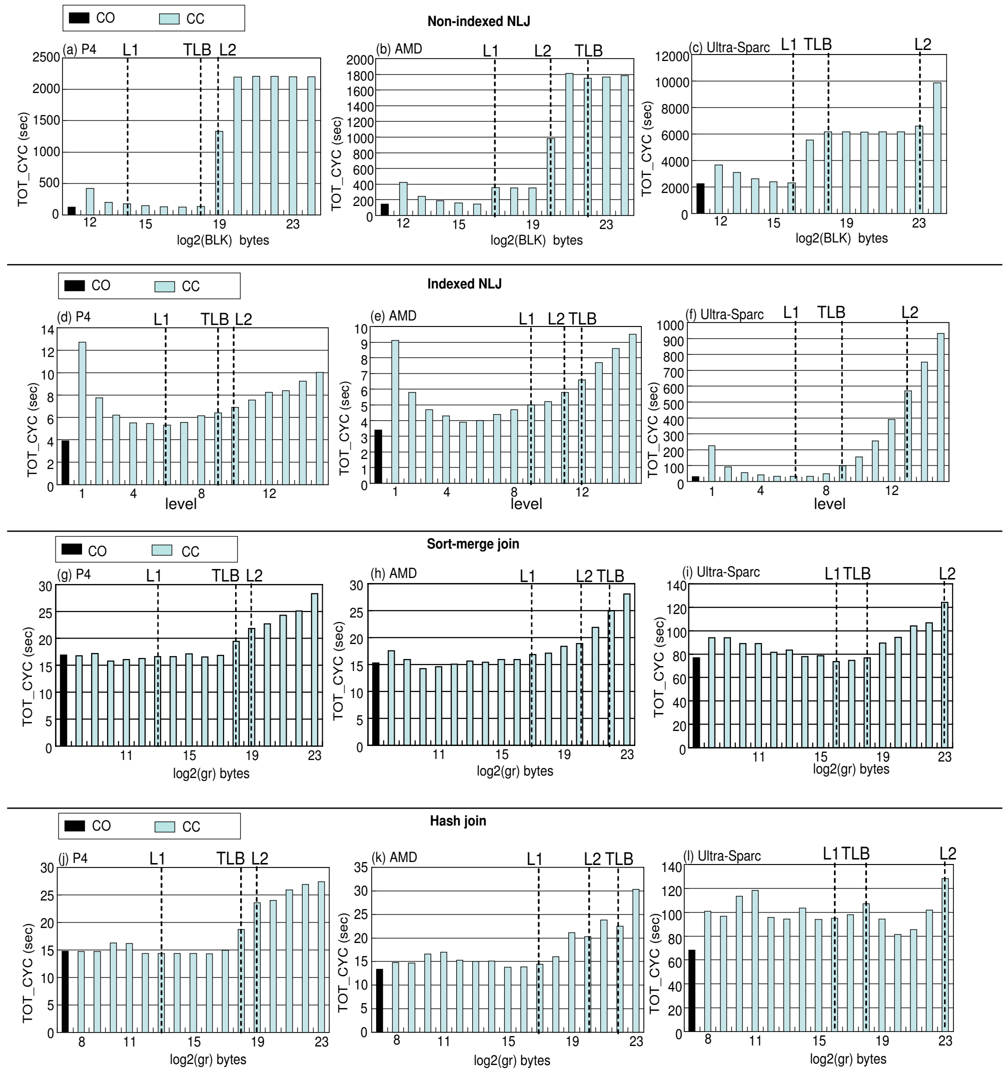

Figure 10: Performance study for join algorithms. From top down: the non-indexed NLJ, the indexed NLJ, the sort-merge join and the hash join.

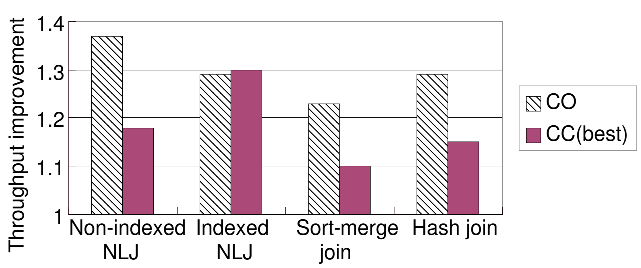

Figure 12: Throughput improvement of SMT on P4

gorithms is larger than that to the cache-conscious algorithms. This is because the cache-oblivious algorithm is more robust than the cache-conscious algorithm on the cache coherence. We conjecture that the advantage of such robustness of the cache-oblivious algorithm will be greater in the future when the processor supports more concurrent threads [28].

6. DISCUSSION

Through implementing and evaluating EaseDB, we have got hands-on experience on developing an efficient cache-oblivious query processor. We have also deepened our understanding on the efficiency of cache-centric techniques. In the following, we discuss the main advantages and limitations of cache-oblivious techniques and outline a few future directions.

The main advantage of cache-oblivious techniques is their self-optimization. As we observed in the experiments, EaseDB had a consistently good performance on any machine without any modification. This automaticity eliminates the need for the cache parameters, which are not always available and are often difficult to obtain automatically. Moreover, this automaticity is achieved without tuning for a specific memory hierarchy. For example, one may develop an algorithm to tune the block size of the inner relation in the blocked NLJ on a target machine. The tuning is based on the performance differences of different block sizes. If a larger block size improves the performance, we continue to increase the block size. However, such tuning is affected by both the static and the dynamic characteristics of the memory hierarchy, which are difficult, if possible, to determine. In contrast, cache-oblivious techniques rely on the divide-and-conquer methodology to achieve the automaticity.

Nevertheless, the efficiency of cache-oblivious techniques needs careful design and optimization. Without knowledge of any cache parameters, cache-oblivious techniques usually employ sophisticated data structures and mechanisms, e.g., the VEB layout and our recursive buffering mechanism, in order to achieve the same cache complexity as their cache-conscious counterpart. Moreover, they require some automatic and machine-independent optimization to improve their efficiency. For instance, a suitable base case size improves the efficiency, but it must be estimated in a cache-oblivious way.

As the first cache-oblivious query processor, EaseDB opens a number of areas for cache-oblivious databases. Take the storage model as an example. Traditional query processors use NSM [31], DSM [11, 29, 36] or PAX (Partition Attributes Across) [2] as storage models, either for main memory databases or for disk-based databases. In contrast, we consider PMA to be a suitable storage model for cache-oblivious query processing. Currently, we use PMA to store the relation in a row-based manner. However, it is an interesting direction to investigate how to support DSM efficiently using PMA. For instance, we consider using some compression schemes to reduce the data volume transferred between the cache and the memory in a cache-oblivious setting.

Finally, new architecture features, especially the emerging processor techniques, create great opportunities for cache-oblivious query processing. In addition to the SMT technique that has been investigated in this study, we are interested in several other architectural features, in particular, the transactional memory [26], multi-core processors and GPUs (Graphics Processing Units) [4].

7. CONCLUSION

As the memory hierarchy becomes an important factor for the performance of database applications, we propose to apply cache-oblivious techniques to automatically improve the memory performance of relational query processing. In this paper, we present our initial efforts on building a cache-oblivious query processor, EaseDB, and report our preliminary results on cache-oblivious storage models, access methods and joins in comparison with their cache-conscious counterparts. Our results show that our cache-oblivious algorithms provide a good performance on various platforms, which is similar to or even better than their fine-tuned cache-conscious counterparts.

Our current work on EaseDB is focused on (1) evaluating and improving the performance of the cache-oblivious access methods and query processing algorithms; (2) developing the cost-based optimizer and verifying its effectiveness; and (3) using our prototype engine to support some data-intensive applications, for example, scientific computing. We expect that the development and continuous evaluation of our prototype system will bring further algorithmic and system research issues.

Acknowledgement

This work was supported by grants HKUST6263/04E and 617206 from the Research Grants Council of the Hong Kong Special Administrative Region, China.

We thank Chang Xu and Jun Zhang for discussions and for reading early drafts of this paper.

8. REFERENCES

[1] A. Aggarwal and S. V. Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[4] A. Ailamaki, N. K. Govindaraju, S. Harizopoulos, and D. Manocha. Query Co-Processing on Commodity Processors. VLDB, 2006.

[5] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 399, Washington, DC, USA, 2000. IEEE Computer Society.

[6] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *J. Algorithms*, 53(2):115–136, 2004.

[7] M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. Cache-oblivious string B-trees. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–242, New York, NY, USA, 2006. ACM Press.

[8] R. Berrendorf, H. Ziegler, and B. Mohr. PCL: Performance Counter Library. http://www.fz-juelich.de/zam/PCL/.

[9] B. Blunden. *Memory Management: Algorithms and Implementation in C/C++*. Wordware Publishing, Inc., 2002.

[10] P. Bohannon, P. McIlroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 163–174, New York, NY, USA, 2001. ACM Press.

[11] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 54–65, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[12] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *ICALP '02: Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 426–438, London, UK, 2002. Springer-Verlag.

[13] G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorith. In *ALENEX/ANALC*, pages 4–17, 2004.

[14] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 116, Washington, DC, USA, 2004. IEEE Computer Society.

[15] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 1999. ACM Press.

[16] E. D. Demaine. Cache-Oblivious Algorithms and Data Structures. Lecture Notes from the EEF Summer School on Massive Data Sets, BRICS, 2002.

[17] FastDB. *http://www.ispras.ru/ knizhnik/fastdb.html*.

[18] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

[19] S. Harizopoulos and A. Ailamaki. A Case for Staged Database Systems. In *CIDR '03: Proceedings of the 1st Biennial Conference on Innovative Data Systems Research*, 2003.

[20] B. He and Q. Luo. Cache-Oblivious Hash Joins. Technical report, HKUST-CS06-04, 2006.

[21] B. He and Q. Luo. Cache-Oblivious Nested-Loop Joins. In *CIKM '06: Proceedings of the ACM Fifteenth Conference on Information and Knowledge Management*, 2006.

[22] B. He, Q. Luo, and B. Choi. Cache-conscious automata for xml filtering. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1629–1644, 2006.

[23] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, 2002.

[24] W. Hong and M. Stonebraker. Exploiting inter-operation parallelism in xprs. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 19–28, New York, NY, USA, 1992. ACM Press.

[25] Intel Corp. *Intel(R) Itanium(R) 2 Processor Reference Manual for Software Development and Optimization*.

[26] B. C. Kuszmaul and J. Sukha. Concurrent Cache-Oblivious B-Trees Using Transactional Memory. Workshop on Transactional Memory Workloads, 2006.

[27] S. Manegold. The Calibrator (v0.9e), a Cache-Memory and TLB Calibration Tool. http://www.cwi.nl/∼manegold/Calibrator/.

[28] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1), 2002.

[29] MonetDB. *http://monetdb.cwi.nl/*.

[30] PostgreSQL. *http://www.postgresql.org/*.

[31] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 3 edition, 2003.

[32] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 78–89, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[33] M. Samuel, A. U. Pedersen, and P. Bonnet. Making CSB+-trees processor conscious. In *DAMON '05: Proceedings of the 1st international workshop on Data management on new hardware*, page 1, New York, NY, USA, 2005. ACM Press.

[34] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, New York, NY, USA, 1979. ACM Press.

[35] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[36] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[37] TimesTen. *http://www.oracle.com/timesten/index.html*.

[38] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and Implementation of an Efficient Priority Queue. *Math. Systems Theory*, 10:99–127, 1977.

[39] S.-E. Yoon and P. Lindstrom. Mesh layouts for block-based caches. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1213–1220, 2006.

[40] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious Mesh Layouts. *ACM Trans. Graph.*, 24(3):886–893, 2005.

[41] J. Zhou and K. A. Ross. Buffering Access to Memory-Resident Index Structure. VLDB, 2003.