Cache-Oblivious Nested-Loop Joins

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

ABSTRACT

We propose to adapt the newly emerged cache-oblivious model to relational query processing. Our goal is to automatically achieve an overall performance comparable to that of fine-tuned algorithms on a multi-level memory hierarchy. This automaticity is because cache-oblivious algorithms assume no knowledge about any specific parameter values, such as the capacity and block size of each level of the hierarchy. As a first step, we propose recursive partitioning to implement cache-oblivious nested-loop joins (NLJs) without indexes, and recursive clustering and buffering to implement cache-oblivious NLJs with indexes. Our theoretical results and empirical evaluation on three different architectures show that our cache-oblivious NLJs match the performance of their manually optimized, cache-conscious counterparts.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing, Relational Databases*

General Terms

Algorithms, Measurement, Performance

Keywords

Cache-Oblivious, Nested-Loop Join, Recursive Partitioning, Recursive Clustering, Buffering

1. INTRODUCTION

The cache-oblivious model [17] was first proposed by M. Frigo *et al.* in 1999. Since then, a few cache-oblivious algorithms and data structures [3, 4, 5, 6, 10, 11, 12, 17] have been studied. The most interesting feature of this line of work is that, even though the model has no knowledge about the capacity and block size of each level of a multi-level memory hierarchy, a cache-oblivious algorithm has a provable upper bound on the number of block transfers for any two adjacent levels of the hierarchy. Furthermore, this memory efficiency asymptotically matches those more knowledgeable external memory algorithms in many cases. Encouraged by their clean theoretical properties, we explore how cache-oblivious algorithms can be applied to in-memory relational query processing to achieve an efficient overall performance automatically.

Automatic performance tuning aims at good and stable performance with little or no manual administration effort regardless of changes in workloads and environments [16, 21]. As the memory hierarchy becomes an important factor for database performance [1, 9], it is essential to investigate self-optimizing techniques in query processing on the memory hierarchy.

Cache-conscious [8, 9, 13, 15, 24, 27], or cache-aware techniques have been the leading approach to the database performance optimization for the memory hierarchy. 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, as both the database systems and the memory systems become more complex and diverse, this cache-conscious tuning becomes a daunting task [5, 23]. In our experiments, we find that the performance of a cache-conscious algorithm varies greatly with its parameter values and the hardware platform. Moreover, the suitable parameter values for the algorithm may be none of the parameter values of the memory hierarchy and therefore requires careful tuning even with the knowledge of the cache parameters.

Having the goal of automatic performance tuning in mind, we investigate if cache-oblivious techniques can be applied to in-memory query processing to achieve a performance comparable to those of their cache-conscious counterparts. Since existing work has focused on basic computation tasks, e.g., sorting and matrix operations [17], and basic data structures, e.g., cache-oblivious B-trees [4, 5, 11], our first task is to find out how to apply cache-oblivious techniques to basic query processing operations.

Joins are the major query operator in relational databases, and nested loops are a simple but widely applicable form of join implementation. Therefore, as a first step, we examine cache-oblivious NLJs, either with or without indexes.

First, we propose recursive partitioning to implement the cache-oblivious non-indexed NLJ algorithm. This recursive partitioning follows the divide-and-conquer methodology [17] of cache-oblivious techniques. Specifically, a problem is divided into a number of subproblems recursively and this recursion will not end until the smallest unit of computation is reached. The intuition is that, at some level of the recursion, a sub-problem will fit into some level of the memory hierarchy and will further fit into one block as the recursion continues. In our cache-oblivious non-indexed NLJ, each relation is partitioned into two halves and the join is decomposed into four smaller joins. This process goes on until a join is fully contained in some level of the memory hierarchy.

We next apply recursive clustering and buffering to implement the cache-oblivious indexed NLJ algorithm. Recursive clustering aims at improving the spatial locality of a tree index. It recursively places related data together so that a cluster fits into one block at some level of the recursion. Buffering improves the temporal locality of a tree index. Our idea of buffering is similar to the previous work [27]: Buffers are created for non-root nodes in the tree index and batch up the accesses to each index node. The difference is that we propose a novel mechanism to recursively determine the size of each buffer in our cache-oblivious buffering.

Compared with cache-conscious techniques for NLJs, e.g., blocking [24] and buffering [27], our cache-oblivious algorithms do not need to calibrate or measure the cache parameters of a specific machine or adjust algorithm settings accordingly. They are automatically optimized for all levels of the memory hierarchy, and this automatic optimization holds on different architectures without any manual tuning.

In brief, this paper makes the following three contributions. First, we propose cache-oblivious algorithms for nested-loop joins. To our best knowledge, this is the first work on cache-oblivious algorithms for relational query processing. Second, we prove that our cache-oblivious algorithms asymptotically match the cache performance of their cache-conscious counterparts. Third, we empirically evaluate the in-memory performance of our cache-oblivious algorithms on three different architectures. Our results show that our cache-oblivious algorithms provide a good and robust performance on all three architectures, and our algorithms can be faster than the carefully tuned cache-conscious algorithms.

The remainder of this paper is organized as follows. In Section 2, we briefly review the background on the memory hierarchy, and compare cache-conscious and cache-oblivious techniques. In Sections 3 and 4, we present our cache-oblivious algorithms for the non-indexed and the indexed NLJs, respectively. We experimentally evaluate our algorithms in Section 5. Finally, we conclude in Section 6.

2. BACKGROUND AND RELATED WORK

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

2.1 Preliminaries

The memory hierarchy in modern computers typically contains multiple levels of memory from bottom up: disks, the main memory, the L2 cache, the L1 cache and registers. Each level has a larger capacity and a slower access speed than its higher levels. A higher level serves as a cache for its immediate lower level and all data in the higher level can be found in its immediate lower level (known as cache-inclusion [18]). 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 ($1 < n < \frac{C}{B}$). In addition to the cache configuration, the average access time and the replacement policy, e.g., LRU, MRU and Random [18], are also important parameters of the cache.

The cache-oblivious model [17] assumes a simplified two-level hierarchy, the cache and the memory. Under this model [17], algorithms are designed without the knowledge of the cache parameters. For analysis, the model has the following assumptions on the cache. First, the cache is tall, $C \geq B^2$ (known as the tall cache assumption) and is fully associative. Second, the cache uses an *optimal* replacement policy: If the cache is full, the ideal cache line will be replaced based on the future access.

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. For example, a search on a cache-oblivious B-tree [4] has an optimal cache complexity $O(\log_B N)$, where N is the number of nodes in the B-tree. 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 [17].

We study cache-oblivious algorithms for the memory hierarchy, in particular, the levels above the main memory. This is because caches at these levels, e.g., L1 and L2 caches, are managed by the hardware. As a result, the accurate state information of these caches are difficult to obtain due to the system runtime dynamics and the hardware complexity. Moreover, even with the knowledge of the cache parameters, the performance of cache-conscious algorithms needs to be tuned carefully. In contrast, we investigate whether and how cache-oblivious algorithms can automatically improve the in-memory performance of query processing. Since LRU is a common cache replacement policy [18], we assume LRU in our cache complexity analysis.

The notations used throughout this paper are summarized in Table 1.

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 NLJ
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	Base case size in number of tuples of S
N	Number of nodes in the tree index on S
nz	the index node size (bytes)
bz	the size of the query item, <search key, RID> (bytes)

2.2 Cache-conscious and cache-oblivious algorithms

Cache-conscious techniques have been the leading approach to the performance optimization of relational query processing on a memory hierarchy. For the last few decades, a number of cache-conscious techniques, e.g., the NSM (N-ary storage model) [22] and the B+-tree [15], have been applied to improve the performance of disk-based applications. Recently, the CPU caches, especially the L2 cache, have become a new bottleneck for in-memory relational query processing [1, 9]. Consequently, many contributions have focused on optimizing the L2 cache performance using cache-conscious techniques [8, 9, 13, 15, 24, 27].

We categorize cache-conscious algorithms into *capacity aware* and *block size aware*. Capacity aware techniques mainly utilize the knowledge of the capacity to improve the temporal locality of the cache, whereas block size aware techniques mainly improve the spatial locality of each block. Representatives of capacity aware techniques include blocking [24], buffering [27] and partitioning [9, 24]. Representatives of block size aware techniques include compression [8] and clustering [14]. An algorithm can be both capacity aware and block size aware, e.g., prefetching based algorithms [13].

Similar to our categorization on cache-conscious algorithms, we categorize cache-oblivious algorithms into *capacity oblivious* and *block size oblivious*. To date, representatives of capacity oblivious algorithms include matrix multiplication and transposition [17], funnel sort and distribution sort [12, 17]. Representatives of block size oblivious algorithms are cache-oblivious B-trees [4, 5, 11] and R-trees [3]. The cache-oblivious non-indexed NLJ we propose is capacity oblivious, and our indexed NLJ is both capacity oblivious and block size oblivious.

Compared with the previous work on cache-oblivious algorithms, our paper focuses on relational query processing, specifically nested-loop joins. As such, the techniques we propose inherit both the divide-and-conquer methodology of cache-oblivious techniques and the data-centric considerations of cache-conscious query processing [24]. Moreover, we pay attention to both the cache complexity of our algorithms and their performance on real systems.

3. CACHE-OBLIVIOUS NON-INDEXED NLJS

The blocked nested-loop join is a cache-conscious algorithm for nested-loop joins without indexes [24] (denoted as CC_NLJ in this paper). The basic idea of blocking is that the inner relation is divided into multiple cache-sized blocks, each of which resides in the cache throughout a loop. In contrast, we propose *recursive partitioning* to design a cache-oblivious algorithm for the non-indexed NLJ. Both relations are recursively partitioned so that the join on the sub-relations can fit into the cache at some level of the recursion.

3.1 Algorithms

We start with a cache-oblivious NLJ algorithm on two relations of an equal size. This algorithm is denoted as RP_NLJ, as shown in Algorithm 1. Next, we use RP_NLJ as a component and implement a cache-oblivious NLJ algorithm for two arbitrary relations. This algorithm is CO_NLJ, as shown in Algorithm 2.

We apply recursive partitioning to both relations in RP_NLJ. The algorithm first divides the inner and outer relations into two equal-sized sub-relations (If the cardinality of the relation is odd, the first sub-relation has one more tuple than the second). Next, it performs joins on the pairs of inner and outer sub-relations. Recursive calls are ordered to maximize the cache reuse among them. The recursion reaches the base case when $|S|$ is no larger than C_S. We apply the tuple-based simple nested-loop join algorithm (denoted as TB_NLJ) to evaluate the base case.

We have two considerations on $||R||$ and $||S||$ for the efficiency of recursive partitioning. First, $||R|| \geq ||S||$, i.e., the inner relation should be no larger than the outer relation of the join. Second, $||R||$ should not be much larger than $||S||$. Since both sub-relations R_1 and R_2 are scanned twice in the four recursive calls of RP_NLJ, this consideration bounds the cost of scanning the outer relations in these recursive calls. With these two considerations, we set the precondition for RP_NLJ to be $||R|| = ||S||$ for simplicity.

In order that both relations to be partitioned have no less than two tuples (Lines 4–5 in Algorithm 1), C_S should be larger than or equal to $|S|/|R|$. Given the precondition $||R|| = ||S||$, we have $|S|/|R| = r/s$. Since C_S should be larger than or equal one, the default value of C_S is the larger value of one and r/s.

Figure 1 compares the order of tuples generated by CC_NLJ and RP_NLJ. The dots represent tuples generated by the join, some of which may be eliminated by the join predicate. CC_NLJ explicitly performs blocking on S. In contrast, RP_NLJ recursively decomposes the join into four smaller joins. In Figure 1 (b), these smaller joins are represented using boxes of different sizes. If the tuples within a box fit into the cache, cache thrashing does not occur in the evaluation of the join represented by the box, and the temporal locality is increased.

Algorithm 1 Cache-oblivious non-indexed NLJ algorithm with constraints

```
Algorithm: RP_NLJ($R$,$S$)
Precondition: $||R|| = ||S||$.
1: if $|S| \leq C_S$ then
2:     TB_NLJ($R$,$S$);
3: else
4:     divide $R$ into two halves, $R_1$ and $R_2$;
5:     divide $S$ into two halves, $S_1$ and $S_2$;
6:     RP_NLJ ($R_1$,$S_1$);
7:     RP_NLJ ($R_2$,$S_1$);
8:     RP_NLJ ($R_2$,$S_2$);
9:     RP_NLJ ($R_1$,$S_2$);
```

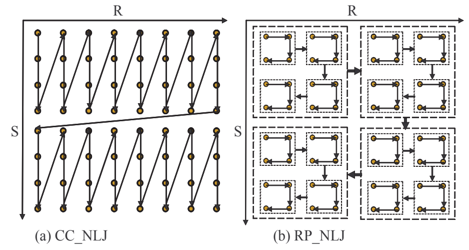

Figure 1: The order of tuples generated in CC_NLJ and RP_NLJ

We now use RP_NLJ as a component in a cache-oblivious algorithm for two arbitrary relations, as shown in Algorithm 2. This algorithm first checks whether the cardinality of S is smaller than or equals C_S. If so, it uses TB_NLJ to evaluate the join. Otherwise, it divides R into p partitions $R_i (1 \leq i \leq p, p = \lceil \frac{||R||}{||S||} \rceil$) so that the size of the first $(p-1)$ partitions equals $||S||$ and that of the last one is smaller than or equals $||S||$. Next, it applies RP_NLJ to evaluate the joins of the first $(p-1)$ partitions and S. Finally, it evaluates the join on the last partition R_p and S. If $||R_p|| = ||S||$, it directly uses RP_NLJ to evaluate the join. Otherwise, it takes S and R_p as the outer and inner relations, respectively, and evaluates the join using CO_NLJ recursively.

Algorithm 2 Cache-oblivious non-indexed NLJ algorithm

```
Algorithm: CO_NLJ($R$,$S$)
1: if $|S| \leq C_S$ then
2:     TB_NLJ($R$, $S$);
3: else
4:     divide $R$ into $p$ partitions, $R_1$, $R_2$, ..., and $R_p$ ($p = \lceil \frac{||R||}{||S||} \rceil$) so that
       $||R_i|| = ||S||$ $(1 \leq i < p)$ and $||R_p|| \leq ||S||$;
5:     for each partition $R_i$ $(1 \leq i < p)$ do
6:         RP_NLJ($R_i$,$S$);
7:     if $||R_p|| = ||S||$ then
8:         RP_NLJ($R_p$,$S$);
9:     else
10:        CO_NLJ($S$, $R_p$);
```

3.2 Cache complexity

Having presented our algorithms, we give their cache complexity results, as shown in Propositions 1 and 2.

PROPOSITION 1. *The cache complexity of RP_NLJ is* $O(\frac{||R|| \cdot ||S||}{CB})$.

Proof. We denote $Q(||R||, ||S||)$ as the cache complexity of RP_NLJ on R and S. When $||S|| < 2C$, cache reuse on the inner relation occurs among every two adjacent recursive calls on the sub-partition of S, and $Q(||R||, ||S||)$ is $\frac{2||R||}{B} + \frac{||S||}{B}$. Based on the recursion of the algorithm, we have:

$$Q(||R||, ||S||) = \begin{cases} 4Q(\frac{||R||}{2}, \frac{||S||}{2}) & , ||S|| \geq 2C \\ \frac{2||R||}{B} + \frac{||S||}{B} & , ||S|| < 2C \end{cases} \quad (1)$$

Suppose R' and S' are the outer and inner relations at the first occurrence of $||S'|| < 2C$. Hence, $C \leq ||S'|| < 2C$. The number of recursion levels is $\log_2 \frac{||S||}{||S'||}$, and the number of $Q(||R'||, ||S'||)$ for $Q(||R||, ||S||)$ is $4^{\log_2 \frac{||S||}{||S'||}} = (\frac{||S||}{||S'||})^2$. Therefore, we have:

$$\begin{aligned} Q(|R|, |S|) &= (\frac{||S||}{||S'||})^2 Q(||R'||, ||S'||) \\ &= (\frac{||S||}{||S'||})^2 (\frac{2||R'||}{B} + \frac{||S'||}{B}) \\ &= \frac{3||S||^2}{||S'||B} \quad \text{(because } ||R'|| = ||S'||) \\ &\leq \frac{3||S||^2}{CB} \end{aligned}$$

Therefore, $Q(|R|, |S|)$ is $O(\frac{||S||^2}{CB}) = O(\frac{||R|| \cdot ||S||}{CB})$.□

PROPOSITION 2. *The cache complexity of CO_NLJ is* $O(\frac{||R|| \cdot ||S||}{CB})$.

Proof. The cache complexity of CO_NLJ is no larger than $\lceil \frac{||R||}{||S||} \rceil$ multiplied by that of RP_NLJ. Thus, according to Proposition 1, the cache complexity of CO_NLJ is $O(\lceil \frac{||R||}{||S||} \rceil \frac{||S||^2}{CB}) = O(\frac{||R|| \cdot ||S||}{CB})$. □

Since the cache complexity of CC_NLJ with a block size C is $O(\frac{||R|| \cdot ||S||}{CB})$ [24], both RP_NLJ and CO_NLJ match the cache complexity of CC_NLJ. Moreover, the cache complexity of CO_NLJ holds on all levels of the memory hierarchy, whereas CC_NLJ optimizes the performance for only one level in the memory hierarchy [24].

3.3 Base case

The size of the base case, C_S, is important for the efficiency of RP_NLJ. On one hand, without the knowledge of the cache capacity, C_S must be small in order to avoid cache thrashing in the base case. On the other hand, a small C_S value results in a large number of recursive calls, which can yield a significant overhead. Thus, we develop a cost-based way to estimate a suitable base case size.

The basic idea of our estimation is to compare the total size of data transferred between the cache and the memory in two cases: (1) the join is evaluated as a base case, and (2) the join is divided into four smaller joins and each of these smaller joins is evaluated as a base case. We denote these two costs as T and T' (in bytes), correspondingly. Through comparing T and T', we determine whether it is cost-effective to evaluate the join as a base case.

Recall that the base case is evaluated using TB_NLJ. Consider the cost of TB_NLJ(R, S). If $||S|| < C$, the relation S can fit into the cache and the size of transferred data is $(||R|| + ||S||)$. Otherwise, it is $(||R|| + |R| \cdot ||S||)$. In addition, fc is the size of the data (in bytes) brought into the cache for a recursive call.

Given the precondition of RP_NLJ ($||R|| = ||S||$), we estimate T and T' in the following four scenarios:

• 1: $||S|| \geq 2C$. Relation S can not fit into the cache, and each partition from S can not either. We have $T = (||R|| + |R| \cdot ||S||)$ and $T' = (2||R|| + |R| \cdot ||S|| + 4 \cdot fc)$.

• 2: $C \leq ||S|| < 2C$. Relation S can not fit into the cache, but each partition from S can. We have $T = (||R|| + |R| \cdot ||S||)$ and $T' = (2||R|| + ||S|| + 4 \cdot fc)$.

• 3: $C/2 < ||S|| < C$. Relation S can fit into the cache, and each partition from S can fit into the cache. We have $T = (||R|| + ||S||)$ and $T' = (2||R|| + ||S|| + 4 \cdot fc)$.

• 4: $||S|| \leq C/2$. R and S together can fit into the cache. We have $T = (||R|| + ||S||)$ and $T' = (||R|| + ||S|| + 4 \cdot fc)$.

In a cache-oblivious setting, we compare the average cost of these four scenarios for T and T', denoted as $E(T)$ and $E(T')$, respectively. We determine the boundary base case size such that $E(T) > E(T')$. Note, we have an under-estimation of the C_S value, because the computation overhead of recursive calls is not included in our estimation of $E(T')$. Therefore, we simply set the C_S value in RP_NLJ to be twice of the boundary value.

We take our implementation as an example to illustrate the estimation. The recursive call of RP_NLJ is in the form of `RP_NLJ(R, S, rStart, rEnd, sStart, sEnd)`. Parameters `rStart` and `rEnd` (or `sStart` and `sEnd`) represent the RID range on the relation `R` (or `S`). Each parameter is coded as four bytes in our implementation. When a function call is issued, the data that are pushed into the recursion stack include a function pointer, the address of the returned result and one copy of all parameters in the recursive call. The size of these data is 32 bytes in total. When the function call is processed, the data having been pushed into the recursion stack are popped. Again, 32 bytes of data are brought into the cache. Thus, $fc = 64$.

Suppose $r = s = 128$ bytes. Given the precondition of RP_NLJ ($||R|| = ||S||$), we have $|R| = |S|$. When $|S|$ is larger than 8, we have $E(T) > E(T')$. The C_S value is set to be 16 in this example. With this C_S value, we eliminate more than 97% of the recursive calls in the RP_NLJ with $C_S = 1$.

4. CACHE-OBLIVIOUS INDEXED NLJS

After presenting cache-oblivious NLJs in the absence of indexes, we discuss cache-oblivious NLJs in the presence of B-tree indexes. Without the knowledge of the block size, we use 2-3 B-tree indexes (each non-leaf node has two or three child nodes). Since the spatial and the temporal locality of such a tree index is important for the performance of indexed NLJs, we apply recursive clustering and buffering techniques to improve the spatial and the temporal locality, respectively. Additionally, we prove that with these techniques, the cache complexity of our cache-oblivious indexed NLJ algorithm matches that of the cache-conscious indexed NLJ algorithm. In our cache complexity analysis, we assume that each index node has exactly two child nodes. Since the tree index with this assumption is taller, this is the worst case analysis for the cache complexity of the indexed NLJ on the 2-3 tree index.

4.1 Clustering

Clustering is to store related data together and results in a layout with a better spatial locality. With the knowledge of the block size, an effective way of clustering is to pack the related data into exactly one block. In contrast, lacking the block size information, recursive clustering recursively puts together the related data items. At some level of the recursion, a cluster of data fits into one block.

An example of recursive clustering is a cache-oblivious B-tree [4], which is a weight-balanced binary tree with the van Emde Boas (VEB) layout [26]. The VEB layout proceeds as follows. Let h be the number of levels (assuming h is a power of two for simplicity). We split the tree at the middle level (Cut 1 in Figure 2 (a)) and we have around $N^{1/2}$ subtrees, each of which roughly contains $N^{1/2}$ index nodes (B_1, ..., B_t in Figure 2 (b)). The resulting layout of the tree is obtained by recursively storing each subtree in the order of B_1, ..., B_t. The recursion stops when the subtree contains only one node. Figure 2 (b) shows the VEB layout of a tree with $h = 4$ and

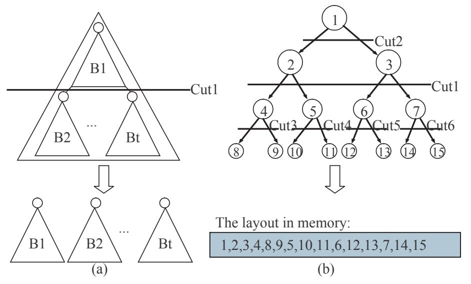

Figure 2: The van Emde Boas layout (left) and the layout of a tree with a level of four (right)

$N = 15$. With the VEB layout, an index node and its child nodes are stored close to each other. Thus, the spatial locality of the tree index is improved.

We apply the VEB layout to the 2-3 tree index. Proposition 3 states that the cache complexity of one probe on the 2-3 tree index with the VEB layout matches that of a probe on one cache-conscious B-tree with the index node size B.

PROPOSITION 3. *The cache complexity of one probe on the 2-3 tree index with the VEB layout is $O(\log_B N)$.*

Proof. The proof of this proposition is the same as the one that proves the cache complexity of the cache-oblivious B-tree [4]. □

4.2 Buffering

The indexed NLJ in its basic form evaluates one tuple at a time. Its major problem is that cache thrashing severely degrades the join performance when the tree index is not cache resident. To improve the temporal locality for the index access in the NLJ, a buffering technique has been proposed [27]. In this technique, an index tree is organized into multiple subtrees (called *virtual node*s [27]) and the root of each subtree, except the root of the original tree, is associated with a buffer. Each buffer is a memory area that temporarily stores the query items that have been passed down from the upper levels. Each query item is in the form of <search key, RID>. With these buffers, the accesses to a virtual node are in a batch. Thus, the temporal locality of the index is improved.

For the completeness of the presentation, we briefly describe how the buffers work in the context of an NLJ [27]. These buffers can be either fixed-size or variable-size. If fixed-size buffers are used, the size of each buffer is set to be the size of a virtual node. These buffers are created before the join begins and stay until the join finishes. As the join proceeds, the query items from the outer relation are distributed to the buffers in the tree index of the inner relation until a buffer becomes full. When a buffer is full, we flush the buffer by distributing the buffered query items to its child buffers recursively. When a query item reaches a leaf node, the leaf node is scanned for matching tuples. At the end of the join, we flush all buffers of the tree index in the depth-first order.

Variable-size buffers are dynamically created during the join process. Each buffer is a linked list of segments, each of which can hold a number of query items. At the beginning of the join, the query items from the outer relation are distributed to the buffers, until scanning the outer relation is done. Then, buffers are flushed one by one in the depth-first order.

Through buffering, the cost of loading a virtual node into the cache is amortized among the buffered query items. Hence, we study the amortized cache complexity of one probe on the 2-3 tree index with buffers.

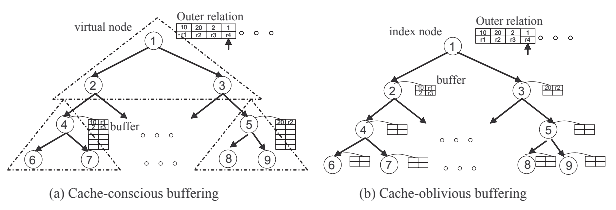

Figure 3: Buffering schemes

4.2.1 CC_BUF

Cache-conscious buffering [27] (denoted as CC_BUF) determines the size of each virtual node according to the cache capacity of the target level in a specific memory hierarchy (e.g., the L2 cache) so that each virtual node can fit into the cache. This buffering scheme is illustrated in Figure 3 (a). Note that the size of two query items is equal to the size of one index node in Figure 3.

PROPOSITION 4. *The amortized cache complexity of one probe on the 2-3 tree index with CC_BUF is $O(\frac{1}{B} \cdot \log_C N)$.*

Proof. The number of levels in a virtual node in CC_BUF is $\log_2 \frac{C}{nz}$. Hence, the number of levels of the buffers in the tree index is $\log_2 N / \log_2 \frac{C}{nz} = \log_{\frac{C}{nz}} N$.

If fixed-size buffers are used, the amortized cost of each probe on a virtual node is $O(1/B)$, because the cost of loading the virtual node into the cache is amortized among $\frac{C}{bz}$ probes. Since the number of levels of the buffers in the tree index is $\log_{\frac{C}{nz}} N$, the amortized cost of each probe is $O(\frac{1}{B} \cdot \log_{\frac{C}{nz}} N) = O(\frac{1}{B} \cdot \log_C N)$.

If variable-size buffers are used, the amortized cost of each probe on a virtual node is $O(1/B)$, because each node is loaded exactly once. Each query item is copied $\log_{\frac{C}{nz}} N$ times among the buffers. Thus, the amortized cache complexity of each probe is $O(\frac{1}{B} \cdot \log_C N)$. □

4.2.2 BASIC_COBUF

The basic algorithm of cache-oblivious buffering (denoted as BASIC_COBUF) performs buffering at each level, because we do not assume knowledge of the cache capacity. Each virtual node contains only one index node. The number of levels of the buffers in the tree index is $\log_2 N$. This buffering scheme is illustrated in Figure 3 (b).

BASIC_COBUF is the same as CC_BUF [27], except that they have different virtual node sizes. However, in BASIC_COBUF with fixed-size or variable-size buffers, the amortized cost of one probe on a virtual node is $O(1/B)$. Since the number of levels of the buffers in the tree index is $\log_2 N$, the amortized cache complexity of one probe in BASIC_COBUF is $O(\frac{1}{B} \cdot \log_2 N)$, which does not match the cache complexity of CC_BUF.

4.2.3 VEB_COBUF

Due to the nature of cache-oblivious algorithm, buffering must be performed on each level of the tree index in BASIC_COBUF. In this case, we can not do much about BASIC_COBUF with variable-size buffers. In contrast, we develop a novel technique to define the size of each buffer for BASIC_COBUF with fixed-size buffers. With this technique, the amortized cache complexity of one probe on the tree index with cache-oblivious buffering matches that with cache-conscious buffering.

We define the size of each fixed-size buffer following the VEB recursion [26]. Thus, we denote this algorithm as VEB_COBUF.

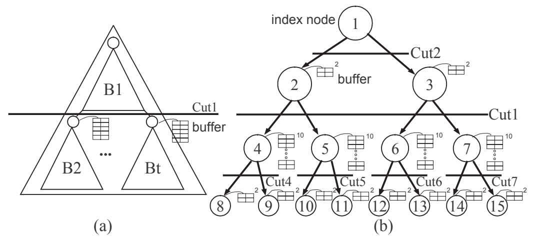

Figure 4: The VEB buffer size definition

The basic idea of our definition is that at each level of the recursion, the size of the buffer for the root node of a subtree is set to be the total size of the index nodes and the buffers for the non-root nodes in the subtree. If the subtree and all its buffers (excluding the buffer for its root node) fit into the cache, the subtree and all these buffers can be reused before they are evicted from the cache.

Following the VEB recursion, we cut the tree at the middle level and obtain a *top tree* as the subtree on top of the cut and a number of *bottom trees* below the cut. If the tree contains N index nodes, we set the buffer size for the root node of each bottom tree to be $N^{\frac{1}{2}} \log_2 N^{\frac{1}{2}}$ in number of index nodes (we will discuss this size in more detail). The sizes of buffers in the top tree and the bottom trees are recursively defined following the VEB recursion. This process ends when the tree contains only one index node. If the size of the buffer for this node has not been determined, it is set to be the index node size.

Denote $S(N)$ to be the total size (in number of index nodes) of buffers on the index consisting of N nodes in VEB_COBUF. Note, the size of the buffer for the root node is not included in $S(N)$. Since each bottom tree has around $N^{1/2}$ nodes, the buffer size for the root node of each bottom tree equals $(N^{1/2} + S(N^{1/2}))$. According to the VEB recursion, we have the formula, $S(N) = N^{1/2}(N^{1/2}+S(N^{1/2}))+(N^{1/2}+1)\cdot S(N^{1/2})$. We solve this formula, eliminate lower-order terms and obtain $S(N) = N(\log_2 N - 1)$. Thus, we set the buffer size for the root node of the bottom tree to be $N^{\frac{1}{2}} \log_2 N^{\frac{1}{2}}$ in number of index nodes.

An example of the VEB buffer size definition on an index tree of four levels is shown in Figure 4 (b). The first cut of the VEB recursion on this tree is between levels 2 and 3. This results in one top tree and four bottom trees. Since the size of two query items equals the size of one index node, the size of the buffer for the root of each bottom tree is $2 \times 3 \log_2 3 = 10$. A further cut on each subtree completes setting the buffer sizes in the tree.

Proposition 5 states that the cache complexity of one probe on the 2-3 tree index with VEB_COBUF matches that with CC_BUF.

PROPOSITION 5. *The amortized cache complexity of one probe on the 2-3 tree index with VEB_COBUF is* $O(\frac{1}{B} \cdot \log_C N)$.

Proof. We consider the VEB recursion of the index tree until a subtree (either the top tree or a bottom tree) and its buffers (excluding the one for its root node) can fit into the cache. These subtrees are denoted as base trees [10]. Thus, a subtree at the previous recursion level and its buffers can not fit into the cache. Suppose each base tree contains k index nodes. Thus, a subtree at the previous recursion level contains approximately k^2 index nodes. Under our buffer size definition, the total size of index nodes and buffers (excluding the one for the root node) is $(k \log_2 k \cdot nz)$ and $(k^2 \log_2 k^2 \cdot nz)$ for these two trees, correspondingly. Therefore, $k \log_2 k \cdot nz < C$ and $k^2 \log_2 k^2 \cdot nz \geq C$. Taking "$\log_2$" on both sides of these two equations, we have $\frac{1}{4} \log_2 \frac{C}{nz} \leq \log_2 k < \log_2 \frac{C}{nz}$.

The amortized cost of each probe on a base tree is $O(1/B)$, because the cost of loading the base tree is amortized among $\frac{k \log_2 k \cdot nz}{bz}$ probes. The number of base trees that each probe should go through is $O(\log_2 N / \log_2 k) = O(\log_2 N / \log_2 C) = O(\log_C N)$. Therefore, the amortized cost of each probe on the index tree is $O(\frac{1}{B} \cdot \log_C N)$. □

5. EVALUATION

We have implemented both cache-conscious and cache-oblivious NLJ algorithms and studied their in-memory performance on three different platforms.

5.1 Experimental setup

Our empirical study was conducted on three machines of different architectures, namely P4, AMD and Ultra-Sparc. Some features of these machines are listed in Table 2. Both the L1 and L2 caches on all three platforms are non-blocking, and the L2 cache is unified. The Ultra-Sparc does not support hardware prefetching data from the main memory [25], whereas both P4 and AMD do. AMD performs prefetching for ascending sequential accesses only [2] whereas P4 supports prefetching for both ascending and descending accesses [19].

Table 2: 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

All algorithms were implemented in C++ and were compiled using g++ 3.2.2-5 with optimization flags (*O3* and *finline-functions*). As we studied the in-memory performance, the data in all experiments were always memory-resident and the memory usage never exceeded 80%.

To date, cache-oblivious B-trees with implicit pointers are complex and not as efficient as the ones with explicit pointers [11, 20]. For a fair comparison between cache-conscious and cache-oblivious algorithms, we chose the B-trees with explicit pointers for our implementation. The representation of tree nodes was the same as the traditional B-tree [15], except that each node contained a four-byte field $height$ and a one-byte field $type$ (a leaf node or a non-leaf node). These two fields were used to facilitate the VEB layout.

Workload design. The workloads in our study contain two join queries on two tables R and S. These workloads are similar to those of the previous study [1]. Tables R and S have a similar schema, defined as follows:

create table R (
a_1 integer not null,
...
a_n integer not null
);

Each field was 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 the table S.

We used the following join queries in our experiments:

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

The equi-join query has the predicate, $R.a_1 = S.a_1$, 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 algorithm to evaluate the non-equijoin and the indexed NLJ algorithm to evaluate the equi-join. We stored join results into an output buffer area, and the storing cost was not significant in our measurements.

Metrics. Table 3 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 [7], to count cache misses on P4 only, because we did not have privileges to perform profiling on AMD or Ultra-Sparc.

Table 3: Performance metrics

Metrics	Description
TOT_CYC	Total execution time on all three platforms in seconds (sec)
L1_DCM	Number of L1 data cache misses on P4 in billions (10^9)
L2_DCM	Number of L2 data cache misses on P4 in millions (10^6)
TLB_DM	Number of TLB misses on P4 in millions (10^6)

For a cache-conscious algorithm in our study, we varied its parameter values to examine the performance variance. Specifically, given a cache parameter value (either C or B), y, of the target level in the memory hierarchy, we varied the parameter value x for the cache-conscious algorithm: $x < y$, $x = y$ and $x > y$. This parameter value tuning of $x \leq y$ simulates the scenario that a programmer has the accurate knowledge of the target level and manually tunes x according to y. $x > y$ may happen if the programmer does not know the cache parameter value or the cache-conscious algorithm is ported from one platform to another without any tuning.

For a given set of performance measurements of a cache-conscious algorithm, we computed the minimum (min), the maximum (max), the average ($mean$) and the variance ($stdev$) values for the set. The min and max values specify the performance range of the cache-conscious algorithm. The $mean$ and $stdev$ values quantify the impact of tuning the cache-conscious algorithm.

5.2 Non-indexed NLJs

We first verified the effectiveness of our estimation on the base case size. Next, we compared the performance of CO_NLJ and CC_NLJ with different block sizes. The reported results were obtained when $||R|| = ||S|| = 32M$ bytes and $r = s = 128$ bytes (both relations have $256K$ tuples) unless otherwise specified. Finally, we varied the tuple size and the relation size to further compare CO_NLJ and CC_NLJ.

Figure 5 shows the time breakdown of CO_NLJ with C_S varied on P4. The busy time is obtained by subtracting the three types of cache stalls from the total elapsed time. We mark the time breakdowns when the size of the inner relation of the base case equals the L1 and L2 data cache capacities, the number of entries in the TLB, and our model estimation ($C_S = 16$). The large gap between the performance of CO_NLJ with ($C_S = 1$) and with ($C_S = 16$) is due to the large difference in their number of recursive calls. This justifies the effectiveness of our estimation. Additionally, the large gap between the performance of CO_NLJ with ($C_S = 16$) and with ($C_S \geq 4096$) indicates that recursive partitioning greatly reduces the number of cache misses of the join. One may conjecture from Figure 5 that the L1 cache capacity is a better guess than our estimation for the base case size on P4; unfortunately, this particular phenomenon does not hold for different platforms, different algorithms, or different data sizes.

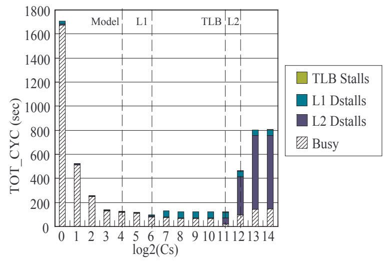

Figure 5: Time breakdowns of CO_NLJ with C_S varied on P4

Figure 6 shows the elapsed time of CO_NLJ and CC_NLJ with different block sizes on the three platforms. On each platform, we mark the measurements of CC_NLJ when its block size equals the L1 and L2 data cache capacities, and the number of entries in the TLB.

We analyze these results on two aspects. First, we study the performance variance of the cache-conscious algorithm. The performance variance of CC_NLJ with different block sizes is large. In particular, the performance variance on P4 and AMD is larger than that on Ultra-Sparc. The performance variance $stdev$ is more than 90% of the mean performance on P4 and AMD, and the ratio is around 40% on Ultra-Sparc. The max value is around four times of the min value on Ultra-Sparc, and can be more than ten times larger than the min value on P4 and AMD. This large performance variance quantifies the potential performance loss with ineffective tuning or without any tuning. Additionally, the best block size for CC_NLJ differs on the three platforms. The best block size is between the L1 and the L2 data cache capacities on P4, is smaller than the L1 data cache capacity on AMD, and is equal to the L1 data cache capacity on Ultra-Sparc. This shows the difficulty in determining the best block size for CC_NLJ on different platforms even with the knowledge of the cache parameters.

Second, we compare the overall performance of CC_NLJ and CO_NLJ on the three platforms. CO_NLJ is faster than CC_NLJ with the L1 or L2 data cache capacity on all three platforms. Regardless of the differences in the three platforms, CO_NLJ provides a much more robust performance, which is close to or even better than the best performance of the carefully tuned CC_NLJ.

We further examine the cache performance of CO_NLJ and CC_NLJ. Figure 7 shows the cache performance of CO_NLJ and CC_NLJ with different block sizes on P4. On one hand, when the block size is small, the number of cache misses caused by CC_NLJ is large due to the large number of scans performed on the outer relation. On the other hand, when the block size increases, we observes the following cache thrashing in order: (a) when the block size is larger than the L1 data cache capacity ($8K$ bytes), thrashing occurs on the L1 data cache; (b) when the number of memory pages in one block of S is larger than the number of entries in the TLB, thrashing occurs on the TLB (the page size is $4K$ bytes, and the block size is $256K$ bytes); (c) when the block size is larger than the L2 cache capacity ($512K$ bytes), thrashing occurs on the L2 cache. This is evidence of a disadvantage of cache-conscious algorithms, i.e., they often optimize the performance for a specific level in the memory hierarchy, but do not optimize for all levels of the memory hierarchy.

Compared with CC_NLJs, CO_NLJ has a consistently good performance on both the L1 and L2 caches and TLB. This performance advantage shows the power of automatic optimization for all levels of the memory hierarchy achieved by cache-oblivious techniques.

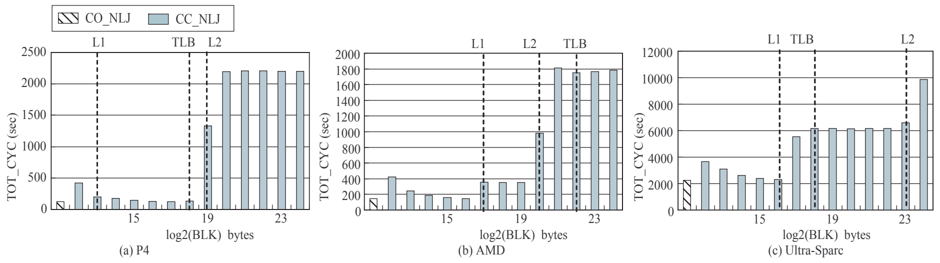

Figure 6: TOT_CYC measurements of CO_NLJ and CC_NLJ

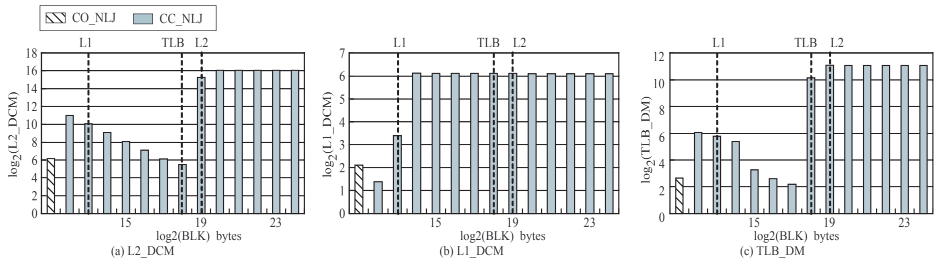

Figure 7: The cache performance of CO_NLJ and CC_NLJ on P4

Figure 8 shows the time breakdown of CO_NLJ and the best CC_NLJ on P4. The total cache stalls of the best CC_NLJ are significant. In contrast, the cache stalls of CO_NLJ are less significant due to its automatic optimization for the entire memory hierarchy.

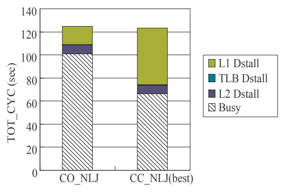

Figure 8: Time breakdowns of CO_NLJ and CC_NLJ(best) on P4

Finally, we study the TOT_CYC ratio of CO_NLJ over the best CC_NLJ on the three platforms. If the ratio is smaller than one, CO_NLJ outperforms the best CC_NLJ. Figure 9 (a) shows the TOT_CYC ratio when both r and s are varied, and $|R| = |S| = 256K$. The performance ratio slightly decreases, as the tuple size increases. Figure 9 (b) shows the TOT_CYC ratio when $r = s = 128$ bytes, $|S| = 256K$ and $|R|$ is varied. Thus, $||R||$ is ranged from $64M$ to $256M$ bytes. The performance ratio is stable when $|R|$ is varied.

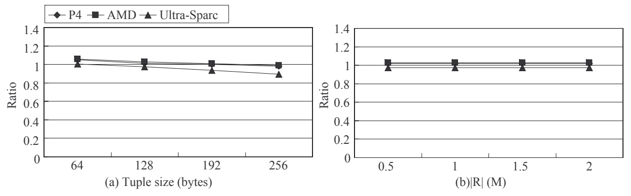

Figure 9: The TOT_CYC ratio of CO_NLJ and CC_NLJ(best)

5.3 Indexed NLJs

We first studied the performance of clustering and buffering separately. The reported results were obtained when $|R| = |S| = 5M$ and $r = s = 8$ bytes unless otherwise specified. This setting was comparable to the previous study [27]. Next, combining clustering and buffering, we compared cache-conscious and cache-oblivious indexed NLJs.

Figure 10 shows the TOT_CYC measurements of the 2-3 tree index with and without clustering, denoted as COB and CCB, respectively. We vary the fanout of CCB to be $F = 2^i + 1$ $(1 \leq i \leq 11)$. Since the node size is $(8F + 1)$ bytes on a 32-bit platform, it is ranged from 25 bytes to around $16K$ bytes. In Figure 10, we mark the TOT_CYC measurements of CCB with the node size equal to the L1 and L2 data cache line sizes and the memory page size on each platform.

We have the following two observations. First, the performance variance of CCB on P4 and AMD is smaller than that on Ultra-Sparc. On P4 and AMD, the performance variance $stdev$ is smaller than 80% of the mean performance, and the max value is four to six times of the min value. On Ultra-Sparc, the performance variance $stdev$ is larger than the mean performance, and the max value is over ten times larger than the min value.

Second, on all platforms, COB outperforms CCB with the fanout of three. This indicates recursive clustering improves the spatial locality of the tree index. When the fanout of CCB increases, CCB outperforms COB. Especially on Ultra-Sparc, CCB can be ten times faster than COB. This is not surprising because the small cache line size reduces the performance impact of clustering. Compared with previous studies that observed that COB can outperform CCB with different fanouts in disk-based applications [5], our study on in-memory performance shows that COB outperforms CCB only when CCB has a relatively small fanout. This may also explain why the performance improvement by clustering on Ultra-Sparc is smaller than that on other two platforms, since its data cache line size (either L1 or L2) is only one half of the other two platforms.

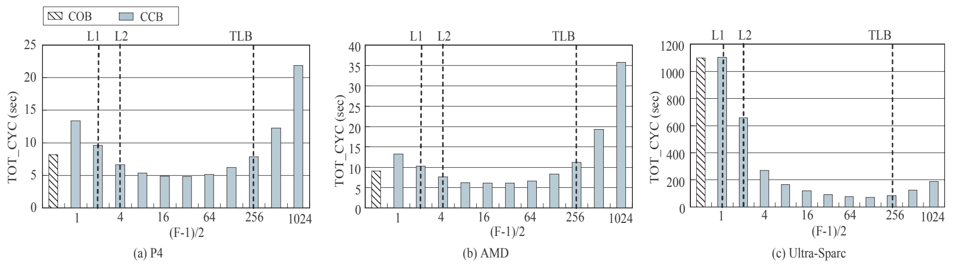

Figure 10: TOT_CYC measurements of COB and CCB

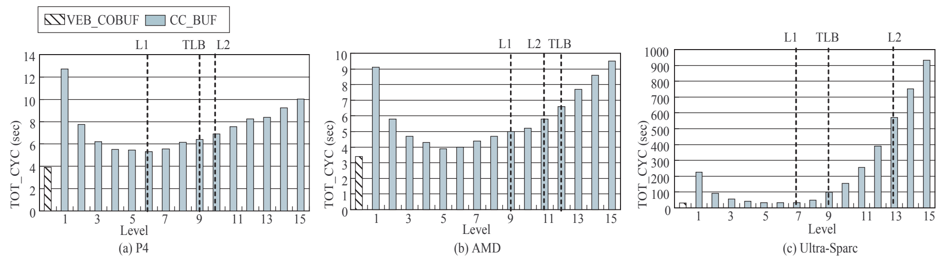

Figure 11: TOT_CYC measurements of the algorithms with fixed-size buffers

Figure 11 shows the TOT_CYC measurements of the algorithms with fixed-size buffers, including VEB_COBUF and CC_BUF with the number of levels in a virtual node varied. Note, the index is a 2-3 tree index without clustering. BASIC_COBUF is equivalent to CC_BUF with each virtual node containing only one level. On each platform, we mark the TOT_CYC value when the virtual node size is determined based on the L1 and the L2 data cache capacities and the number of entries in the TLB.

We summarize our findings on Figure 11 on three aspects. First, the buffering technique significantly improves the overall performance of the join on the three platforms. Comparing the measurements in Figures 10 and 11, we find that even the naive algorithm, BASIC_COBUF, can be more than four times faster than that without buffering on Ultra-Sparc.

Second, the performance variance of CC_BUF on P4 and AMD is smaller than that on Ultra-Sparc. On P4 and AMD, the performance variance $stdev$ is smaller than 35% of the mean performance, and the max value is two to three times of the min value. On Ultra-Sparc, the performance variance $stdev$ is larger than the mean performance, and the max value is around thirty times larger than the min value. The best virtual node sizes are different on the three platforms. It is equal to the L1 cache capacity on P4, but it is smaller than the L1 cache capacity on AMD and Ultra-Sparc.

Third, on all three platforms, VEB_COBUF outperforms CC_BUF with fixed-size buffers. We further examine the time breakdown of VEB_COBUF and the best CC_BUF on P4 (Figure 12). Similar to the results of non-indexed NLJs, the total cache stalls of the best CC_BUF are significant, and those of VEB_COBUF are less significant.

Figure 13 compares the algorithms with fixed-size and variable-size buffers. We show the performance of CC_BUF with the best virtual node size in the figure. Not surprisingly, variable-size buffering is faster than fixed-size buffering in both CC_BUF and BASIC_COBUF. As an algorithm with fixed-size buffers, VEB_COBUF has a good performance, which is better than that of BASIC_COBUF with variable-size buffers and close to or better than that of CC_BUF with variable-size buffers.

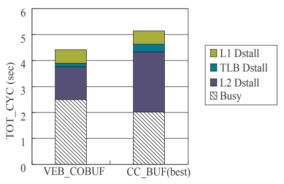

Figure 12: Time breakdowns of VEB_COBUF and CC_BUF(best) on P4

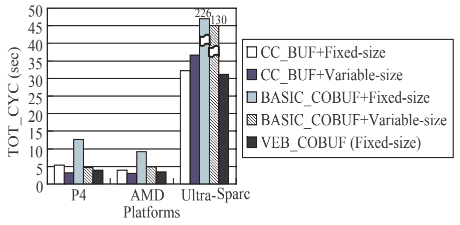

Figure 13: Performance comparison of algorithms with fixed-size and variable-size buffers

To put it together, we study the performance of combining clustering and buffering with the relation size varied. We compute the TOT_CYC ratio of cache-oblivious indexed NLJs with VEB_COBUF over the best of cache-conscious indexed NLJs with fixed-size buffers. Figure 14 (a) shows the results when $|S|$ is varied ($|R| = 5M$), and Figure 14 (b) when $|R|$ is varied ($|S| = 5M$). On all platforms, the ratios are smaller than one. We compute the TOT_CYC ratios of cache-oblivious indexed NLJs with VEB_COBUF but without clustering over the best of cache-conscious indexed NLJs with fixed-size buffers, and find that these ratios are close to those of combining clustering and buffering. This suggests that the temporal locality is more important than the spatial locality in the indexed NLJ.

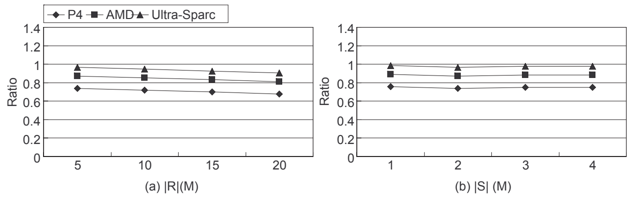

Figure 14: The TOT_CYC ratio of cache-oblivious and cache-conscious indexed NLJs

5.4 Summary

In summary, the performance of CO_NLJ is close to the best performance of CC_NLJ on P4 and AMD, and is better than the best performance of CC_NLJ on Ultra-Sparc. VEB_COBUF is faster than CC_BUF with fixed-size buffers on all these three platforms. Moreover, we find that the best parameter value for a cache-conscious algorithm may be none of the cache parameters, and the performance variance of a cache-conscious algorithm is large. Regardless of platform differences, our cache-oblivious algorithms provide a stable performance, which is similar to or even better than the peak performance of carefully tuned cache-conscious algorithms.

6. 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. As a first step, we have applied recursive partitioning to implement cache-oblivious non-indexed NLJs. Also, we have applied recursive clustering and buffering to implement cache-oblivious indexed NLJs. We prove that the cache complexity of our cache-oblivious NLJs matches that of their cache-conscious counterparts, and that this cache complexity holds on all levels of the memory hierarchy. Based on our evaluation on real systems, we find that the performance of cache-conscious NLJs varies greatly with their parameter values and the hardware platform, and that the best parameter value for the cache-conscious algorithm may be none of the cache parameters. In contrast, our cache-oblivious NLJs provide a much more robust performance, which is similar to or even better than their cache-conscious counterparts with suitable parameter values.

Acknowledgment

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

7. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? VLDB, 1999.
[2] AMD Corp. *Software Optimization Guide for AMD64 Processors*.
[3] L. Arge, M. de Berg, and H. Haverkort. Cache-Oblivious R-Trees. In *Proceedings of ACM Symposium on Computational Geometry (SoCG)*, 2005.
[4] M. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, 2000.
[5] M. Bender, M. Farach-Colton, and B. Kuszmaul. Cache-Oblivious String B-trees. PODS, 2006.
[6] M. Bender and H. Hu. An Adaptive Packed-Memory Array. PODS, 2006.
[7] R. Berrendorf, H. Ziegler, and B. Mohr. PCL: Performance Counter Library. http://www.fz-juelich.de/zam/PCL/.
[8] P. Bohannon, P. McIlroy, and R. Rastogi. Main-memory Index Structures With Fixed-Size Partial Keys. SIGMOD, 2001.
[9] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB Journal*, pages 54–65, 1999.
[10] G. S. Brodal and R. Fagerberg. Cache Oblivious Distribution Sweeping. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP)*, 2002.
[11] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache Oblivious Search Trees via Binary Trees of Small Height. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete Algorithms (SODA)*, 2002.
[12] G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a Cache-Oblivious Sorting Algorithm. Technical Report ALCOMFT-TR-03-101, November 2003.
[13] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving Hash Join Performance Through Prefetching. ICDE, 2004.
[14] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. *SIGPLAN*, 1999.
[15] D. Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.
[16] S. Elnaffar, W. Powley, D. Benoit, and P. Martin. Today's DBMSs: How Autonomic Are They? DEXA, 2003.
[17] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, 1999.
[18] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach. 2nd Edition*.
[19] Intel Corp. *Intel(R) Itanium(R) 2 Processor Reference Manual for Software Development and Optimization*.
[20] R. E. Ladner, R. Fortna, and B.-H. Nguyen. A Comparison of Cache Aware and Cache Oblivious Static Search Trees Using Program Instrumentation. In *Experimental Algorithmics, LNCS 2547*, 2002.
[21] G. M. Lohman and S. S. Lightstone. SMART: Making DB2 (more) Autonomic. VLDB, 2002.
[22] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 3 edition, 2003.
[23] M. Samuel, A. U. Pedersen, and P. Bonnet. Making CSB+-Trees Processor Conscious. the First International Workshop on Data Management on New Hardware (DaMoN), 2005.
[24] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. VLDB, 1994.
[25] Sun Corp. *UltraSPARC (R) III Cu Users Manual*.
[26] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and Implementation of an Efficient Priority Queue. *Math. Systems Theory*, 10:99–127, 1977.
[27] J. Zhou and K. A. Ross. Buffering Access to Memory-Resident Index Structure. VLDB, 2003.