A General Framework for Improving Query Processing Performance on Multi-Level Memory Hierarchies

Bingsheng He† Yinan Li‡ Qiong Luo† Dongqing Yang‡
†Hong Kong Univ. of Science and Technology ‡Peking University
{saven,luo}@cse.ust.hk {liyinan,dqyang}@pku.edu.cn

ABSTRACT

We propose a general framework for improving the query processing performance on multi-level memory hierarchies. Our motivation is that (1) the memory hierarchy is an important performance factor for query processing, (2) both the memory hierarchy and database systems are becoming increasingly complex and diverse, and (3) increasing the amount of tuning does not always improve the performance. Therefore, we categorize multiple levels of memory performance tuning and quantify their performance impacts. As a case study, we use this framework to improve the in-memory performance of storage models, B+-trees, nested-loop joins and hash joins. Our empirical evaluation verifies the usefulness of the proposed framework.

1. INTRODUCTION

For the last two decades, processor speeds have been growing at a much faster rate (60% per year) than memory speeds (10% per year) [1]. Due to this widening speed gap, the memory hierarchy has become an important factor for the overall performance of relational query processing [3, 10]. Meanwhile, both relational database systems and hardware platforms are becoming increasingly complex and diverse. It is important and challenging to automatically and consistently achieve a good query processing performance across platforms.

In this paper, we propose a general framework to quantify the relationships between the performance improvement and the automaticity of in-memory query processing techniques. Intuitively, an algorithm that knows much about a specific memory hierarchy can utilize this knowledge to improve its efficiency, but it may require a large amount of tuning due to its dependency on platform-specific parameters, and its performance may also differ on different platforms. Considering these issues, we categorize in our framework the automaticity of an algorithm by the amount of knowledge about the memory hierarchy.

A memory hierarchy has quite a few parameters that affect the query processing performance. The common ones include (1) the number of levels of the hierarchy and (2) the capacity, block size, associativity, and access latency of each level. Other characteristics include prefetching and non-blocking data transfers between two adjacent levels of the memory hierarchy. Some of these characteristics are correlated and others are independent.

So far, various *cache-conscious* techniques [1, 10, 31] have considered one or two of these parameters individually and have demonstrated a high performance with suitable parameter values and fine tuning on a specific memory hierarchy. In contrast, there has emerged initial work on *cache-oblivious* algorithms [5, 7, 11, 12, 18], which assume no knowledge about a specific memory hierarchy and usually have provable upper bounds on the number of block transfers between any two adjacent levels of an arbitrary memory hierarchy.

Considering both the memory hierarchy characteristics and the existing algorithms, we define the tuning levels in our framework corresponding to the memory hierarchy characteristics and study the performance of the algorithms at different tuning levels. Specifically, we start from a cache-oblivious algorithm, which requires no tuning, and gradually add more knowledge about the memory hierarchy and thus more tuning to the algorithm. Finally, we compare the performance of these algorithms at each tuning level and across platforms. The algorithms we studied include in-memory storage models, the B+-tree, the non-indexed nested-loop join and the hash join. Our empirical evaluation verifies the usefulness of the proposed framework.

In brief, this paper makes the following three contributions. First, we propose a general framework for improving the query processing performance on multilevel memory hierarchies. To our best knowledge, this is the first work on quantifying the correlations between the performance improvement and the amount of tuning for the memory hierarchy. Second, we use our framework to study four common data structures and algorithms for in-memory query processing. As a result, we develop a series of algorithms that carry different degrees of tuning for in-memory databases. Third, we empirically evaluate the in-memory performance of the algorithms. Our results demonstrate the effectiveness of our framework.

The remainder of this paper is organized as follows. In Section 2, we briefly review the background and related work. In Section 3, we present our framework. In Section 4, we use our framework to study in-memory storage models, B+-trees, nested-loop joins and hash joins. We experimentally verify our framework in Section 5. Finally, we conclude in Section 6.

2. PRELIMINARY AND RELATED WORK

In this section, we first introduce the background on the memory hierarchy. Next, we review the related work on cache-conscious and cache-oblivious techniques.

2.1 Memory hierarchies

The memory hierarchy in modern computers typically contains

Proceedings of the Third International Workshop on Data Management on New Hardware (DaMoN 2007) June 15, 2007, Beijing, China.

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. *We use the cache and the memory to represent any two adjacent levels in the memory hierarchy.*

We summarize the following *static* characteristics of a memory hierarchy.

P_0. The number of levels in the hierarchy.

P_1. The cache configuration: *<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.

P_2. The transfer latency of the cache, l.

P_3. Transfer characteristic between two adjacent levels: supporting software prefetching or not and the non-blocking capability to support multiple transfers simultaneously. We use the number of concurrent transfers supported, D, to quantify the non-blocking capability.

Compared with static characteristics, the dynamic ones such as the number of concurrent threads are more difficult to capture but are important in multi-task systems, such as databases [22, 23]. In this study, we focus on the static characteristics and leave the study on the dynamic characteristics as future work.

The notations used throughout this paper are summarized in Table 1. For readability, we will simply use C, B, A, l, d, D (i.e., without subscript i) whenever we refer to any level of the memory hierarchy without explicitly specifying a level.

Table 1: Notations used in this paper

Parameter	Description
P_i	Characteristics of the memory hierarchy, $0 \leq i \leq 3$
Γ_i	The knowledge about the memory hierarchy, $\Gamma_0 = \phi$ or $\Gamma_i \subseteq \{P_0, ..., P_i\}, 1 \leq i \leq 3$.
T_i	The levels of tuning corresponding to Γ_i in our framework, $0 \leq i \leq 3$
L	Number of levels in the memory hierarchy considered for tuning
C_i	Cache capacity of the i^{th} level (bytes)
B_i	Cache line size of the i^{th} level (bytes)
A_i	Cache associativity of the i^{th} level
l_i	Access latency of the i^{th} level for random accesses (ns)
d_i	Prefetching distance (number of cache blocks to prefetch ahead)
D_i	Number of concurrent transfers supported by the non-blocking capability
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)

2.2 Cache-centric query processing

Due to the widening speed gap between the processor and the main memory, the CPU caches, especially the L2 cache, have become a bottleneck for in-memory relational query processing [3, 10]. Consequently, many contributions have focused on optimizing the L2 cache performance using cache-centric techniques including cache-conscious [10, 13, 31] and cache-oblivious ones [7, 24].

Cache-conscious techniques have been the leading approach to optimizing the cache performance. Specialized data structures, such as cache-conscious B+-trees [9, 30], R-trees [27] and storage models [2, 20], have been proposed to reduce cache misses. Typical cache-conscious techniques, including blocking [31], data partitioning [28, 31], compression [9], data clustering [31], prefetching [13–16], staging [23] and buffering [33], were proposed for improving the cache behavior of traditional database workloads. Most of these studies optimize a single level on a multi-level memory hierarchy, e.g., the L2 data cache.

With the same focus on reducing cache stalls, cache-oblivious algorithms do not require the knowledge of cache parameters or any tuning on the cache parameters. Representatives of existing cache-oblivious techniques include recursive partitioning [18] and buffering [11, 18] for temporal locality, and recursive clustering [5] for spatial locality. For relational query processing, He et al. [24, 25] proposed cache-oblivious join algorithms, including nested-loop joins with and without indexes, sort-merge joins and hash joins. Both theoretical results and empirical evaluation show that cache-oblivious algorithms can match the performance of their manually optimized, cache-conscious counterparts [7, 12, 24].

In contrast with the existing cache-centric techniques that are tuned based on a certain amount of knowledge about a specific memory hierarchy, we propose a general framework to quantify the correlations between the amount of tuning and the potential performance gain. The framework serves as a guide for optimizing a cache-centric algorithm given a certain amount of knowledge about a specific memory hierarchy.

3. FRAMEWORK

In this section, we present our framework and a cost model for applying the framework to an algorithm on the memory hierarchy. Our framework quantifies the performance impact of the tuning using a certain amount of knowledge about the memory hierarchy. The basic idea of our categorization is that the more knowledge about the memory hierarchy considered, the more tuning can be involved to improve the performance. Given a certain amount of knowledge about the memory hierarchy, we can apply certain kinds of optimizations to improve the query processing performance.

3.1 Categorization

Figure 1 illustrates the spectrum of tuning for the memory hierarchy. Techniques on the left of the spectrum require less tuning than the ones on the right. Based on our categorization on the characteristics of a memory hierarchy, we divide the spectrum into the following four levels of memory performance tuning.

T_0 No knowledge about the cache parameters, i.e., cache-oblivious.

T_1 Having the knowledge of the cache capacity and/or cache block size of the target level.

T_2 Having the knowledge of the latency, in addition to T_1.

T_3 Having the knowledge of software prefetching and/or non-blocking capability of the target level, in addition to T_2.

T_1–T_3 implicitly have the knowledge of P_0. At T_1, we can determine the highest level of cache that can hold the working set of the algorithm. Let this level of cache be x. Thus, T_1–T_3 knows the number of levels of the cache considered for tuning, $L = x$.

In our categorization, T_i *($0 \leq i \leq 3$) requires a set of characteristics of the target level of cache,* Γ_i. Thus, the levels of tuning in our framework are in a total order according to the required amount of knowledge about the memory hierarchy. A T_i with a larger i value requires more information about the memory hierarchy and involves more tuning, i.e., $T_0 < T_1 < T_2 < T_3$. Tuning T_0 is cache-oblivious, since it requires no tuning on the hardware dependent parameters. In contrast, T_1–T_3 are cache-conscious. For

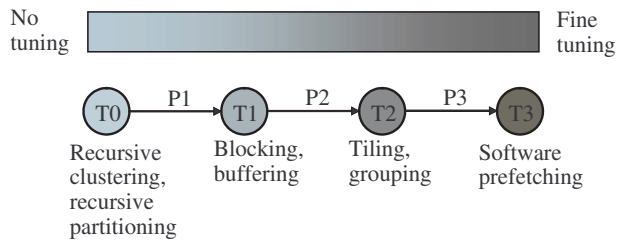


Figure 1: The spectrum of tuning. T_0 on the left of the spectrum is cache-oblivious, and T_1–T_3 on the right of the spectrum are cache-conscious. On the bottom of the figure show some existing techniques belonging to each level of tuning.

instance, if T_3 applies software prefetching to a technique, it requires the tuning based on the cache capacity and the block size (T_1) as well as the latency (T_2). The block size is used to determine the size of the data to be prefetched. The cache capacity and the latency are used to determine the prefetch distance so that the memory latency is fully hidden by prefetching.

We have categorized existing techniques according to our framework, as shown in Table 2. The majority of cache-conscious techniques belong to T_1 and T_2.

Table 2: Categorizing existing cache-centric techniques

Knowledge	Tuning	Representative techniques
Γ_0	T_0	CO B+-trees [5, 6], funnel sort [11], CO nested-loop join [24, 25] and storage models [6]
Γ_1	T_1	Blocking [31], buffering [33], partitioning [10, 31], compression [9, 27], clustering [17]
Γ_2	T_2	Tiling [19], grouping [2, 20, 29, 30]
Γ_3	T_3	Loop unrolling [28], prefetching [13–16]

In this tuning hierarchy, a higher level of tuning can *potentially* achieve a higher performance at the price of a larger amount of tuning. Given a technique at the tuning level, T_i, we consider higher tuning levels, T_j ($j > i$), to optimize this technique with more knowledge about the memory hierarchy. We briefly describe the basic use for each level of tuning: (a) T_0 mainly uses the divide-and-conquer methodology to improve the cache locality. (b) T_1 packs the data into the cache or a cache block. Additionally, we can estimate the number of cache misses on each level of cache. (c) T_2 determines the most significant levels of cache for the total execution time and applies techniques to those levels of cache. (d) T_3 applies software prefetching to hide the cache stalls. As we will demonstrate in Section 4, we apply our framework to four case studies. We optimize each base technique with the knowledge about the memory hierarchy and without much modification to the base technique.

3.2 Determining the target levels of caches

Because a memory hierarchy consists of multiple levels, we need to determine the target level for a cache-conscious algorithm. Note that cache-oblivious techniques do not require determining the target levels due to their automaticity.

Since the complexity of tuning dramatically increases with the number of levels of caches considered, we choose one or two levels of caches that are most significant for the overall performance to be the target levels of caches for tuning. The following are two representative cases:

Case $\Gamma = \{P_0, P_1\}$. Since we do not know the latency information, we can not determine which levels of caches are significant in the overall performance. In practice, we choose the lowest level of caches that can not hold the working set of the algorithm, $(L-1)$, to be the target level, because the lower levels have a larger latency (even though the actual latency is unknown) and are likely to be significant in the overall performance.

Case $\Gamma \supseteq \{P_0, P_1, P_2\}$. With the latency information of the memory hierarchy, we develop our overall cache performance model for a memory hierarchy. This model estimates the overall cache performance, which is defined to be the total cache stalls of an algorithm on all levels of the memory hierarchy. Suppose the cost function $F_i(P_1, ..., P_3, T_0, ..., T_3)$ gives the number of cache misses on the i^{th} level of the memory hierarchy caused by the algorithm. Since caches at different levels of the memory hierarchy are independent with each other, the cost functions for two distinct cache levels may be different due to different levels of tuning applied. Eq. 1 gives the cache stalls on the i^{th} level. Thus, the overall cache performance of the algorithm on the entire memory hierarchy is given in Eq. 2.

$$\tau_i = F_i(P_1, ..., P_3, T_0, ..., T_3) \times l_{i+1} \quad (1)$$

$$\tau = \sum_{i=1}^{L-1} \tau_i \quad (2)$$

To determine the target levels, we rank the levels of caches according to τ_i. The larger τ_i, the more significant the i^{th} level of caches.

4. CASE STUDIES

In this section, we use in-memory storage models, the B+-tree, the nested-loop join without indexes (NLJ) and the hash join as case studies to illustrate the applicability of our framework. We start the tuning process at a certain level of tuning and apply the upper levels of tuning to the algorithm. Additionally, we have developed cost functions for each algorithm at different tuning levels. These cost functions are used in our cost model to determine the significant level of cache.

4.1 Storage models

We consider two kinds of storage models, the array for static data and the linked list (LL) for dynamic data. Especially, we consider optimizing the scan on the storage models.

Array scan. Since the array has a good spatial locality on any level of the memory hierarchy, T_1 and T_2 do not have any performance improvement on the array scan. We consider T_3 to see whether software prefetching helps reduce the number of cache misses for loading the array. Given the prefetching distance, d, the algorithm issues a prefetching instruction on the $(i+d)$th cache block, before processing the ith cache block.

LL scan. At T_1, the algorithm determines the suitable node size for the linked list according to the cache block size of the lowest level of caches so that the spatial locality of each node is maximized. At T_2, the algorithm determines the suitable node size for the linked list according to the cache block size of the target level. At T_3, the algorithm determines the suitable prefetching distance. Similar to the array scan, the algorithm prefetches the $(i + d/z)$th node (z is the node size in number of cache blocks) when it processes the ith node. The algorithm keeps a jump-pointer array, J, to maintain the addresses of the nodes in the linked list. The idea

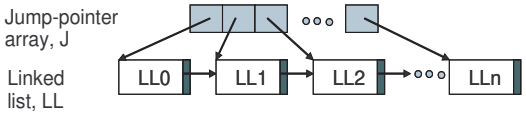


Figure 2: The jump-pointer array for the linked list.

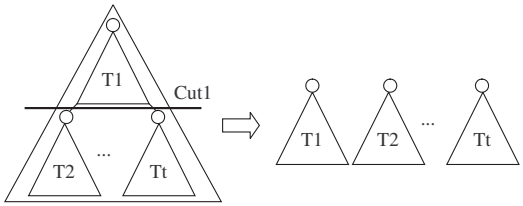


Figure 3: The VEB layout.

of the jump-pointer array is shown in Figure 2. $J[i]$ stores the start address of the ith node in the linked list.

With the jump-pointer array, we apply the prefetching technique to scan the linked list. Given the prefetching distance, d, the algorithm issues prefetch instructions for the node at $J[i+d/z]$ in order to prefetch the $(i + d/z)$th node.

4.2 B+-trees

We start with a cache-oblivious B+-tree at T_0. A COB+-tree [6] consists of two arrays. One stores the data leaves of the B+-tree, and the other one stores the directory. The index nodes in the directory are organized into a binary tree stored in the van Emde Boas (VEB) layout [5] without considering any specific memory parameters.

The VEB layout proceeds as follows. Let h be the number of levels in the tree. We split the tree at the middle level (Cut 1 in Figure 3) and obtain around $N^{1/2}$ subtrees, each of which contains roughly $N^{1/2}$ index nodes (T_1, ..., T_t in Figure 3). The resulting layout of the tree is obtained by recursively storing each subtree in the order of T_1, ..., T_t. The recursion stops when the subtree contains only one node. In 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.

At T_1, each tree node has the exact size of B. It is a small binary tree stored in the VEB layout. Additionally, the entire tree is stored according to the VEB layout. This idea of "tree within a tree" is similar to the fractal tree structure [16]. The difference is that we use the VEB layout to store each node so that the node has a good spatial locality for all levels of caches above the target level. At T_2 and $T3$, the B+-tree is similar to that at T_1 except that the node size is the cache block size at the target level of cache at T_2, and is $(D \times B)$ at T_3.

4.3 Nested-loop joins

We use the cache-oblivious non-indexed nested-loop joins (CO_NLJ) [24] as NLJ at T_0. CO_NLJ first divides each of the inner and outer relations (denoted as S and R, respectively) into two equal-sized sub-relations. Next, it performs joins on the pairs of inner and outer sub-relations. This partitioning and joining process goes on recursively, until it reaches the base case when $|S|$ is no larger than the base case size, C_S (default $C_S = 1$). It then applies the tuple-based non-indexed nested-loop join algorithm to evaluate the base case.

At T_1 or T_2, the algorithm sets the base case to be the cache capacity of the target level depending on the level of cache to be optimized. Thus, the inner relation of the base case can fit into the cache. At T_3, the algorithm sets the prefetching distance to be a small constant so that prefetching does not interfere with the cache locality tuned at T_1. The optimal prefetching distance is $\frac{l}{w}$, given the computation time on each cache block, w. With this prefetching distance, the prefetching can fully hide the cache stall.

4.4 Hash joins

We start with the simple hash join. We use two techniques to improve its cache performance, cache partitioning [10,31] and prefetching [13]. The former one belongs to T_1 or T_2 depending on its target level of cache, whereas the latter one belongs to T_3.

Since these two techniques are independent with each other, we have implemented two variants of cache-optimized hash joins: (1) we first implement the partitioned hash join, and then apply prefetching within the join on each partition pair. (2) we apply the prefetching in both the partitioning and the probing.

4.5 Summary

We derive the cost function of each level of tuning for each case, as shown in Table 3. These cost functions are used to determine the target levels of caches.

5. EVALUATION

We verified the usefulness of the proposed framework by implementing and evaluating the case studies in Section 4.

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 4. The row `DTLB` gives the number of entries in the data TLB (Translation Look-aside Buffer). The Ultra-Sparc does not support hardware prefetching data from the main memory [32], whereas both P4 and AMD do. AMD performs prefetching for ascending sequential accesses only [4] whereas P4 supports prefetching for both ascending and descending accesses [26].

Table 4: Machine characteristics

Name	P4	AMD	Ultra-Sparc
OS	Windows XP	Linux 2.6.15	Solaris 8
Processor	Intel P4 1.8GHz	AMD Opteron 1.8GHz	Ultra-Sparc III 900Mhz
L1 DCache	<32K, 64, 4>	<64K, 64, 2>	<64K, 32, 4>
L2 cache	<2M, 64, 8>	<1M, 128, 16>	<8M, 64, 8>
DTLB	64	1024	64
Memory	1.0 GB	15.0 GB	8.0 GB

We performed calibration on these machines to obtain the cache latency. For instance, the latency of the L1 and L2 caches on P4 is 12 and 42 cycles, respectively.

Workload design. The workloads in our study contain (1) one selection query on table R, and (2) two join queries each on two tables R and S. Each of tables R and S consists of n integer attributes, a_1, a_2, ..., and a_n. Each field was a randomly generated 4-byte integer. We varied n to scale up or down the tuple size of the table. These workloads are similar to those in a the previous study [3].

We consider the following selection query, "`SELECT` $R.a_1$ `FROM` R `WHERE` $R.a_1 = 1$ *and...and* $R.a_n = 1$". We used a full scan on R to evaluate this selection query. All fields of the table are involved in the predicate so that an entire tuple is brought into the cache for the evaluation of the predicate.

The join queries considered in our experiments are "`SELECT` $R.a_1$ `FROM` R, S `WHERE` <predicate>". There are two

Table 3: Cost functions. z is the size of an index entry in the B+-tree node (bytes).

Cases	Array scan	LL scan	NLJs	B+-Trees	Hash join (1)	Hash join (2)
T_0	$\frac{\|\|R\|\|}{B}$	$\|R\| \cdot B$	$\frac{3\|\|R\|\cdot\|\|S\|\|}{C \cdot B}$	$2\log_{\frac{B}{z}} \|R\|$	$\|R\| \cdot (1 + \frac{s}{B}) + \frac{\|\|R\|\|}{B}$	$\|R\| \cdot (1 + \frac{s}{B}) + \frac{\|\|R\|\|}{B}$
T_1	$\frac{\|\|R\|\|}{B}$	$\frac{\|\|R\|\|}{B}$	$\frac{\|\|R\|\cdot\|\|S\|\|}{C \cdot B}$	$\log_{\frac{B}{z}} \|R\|$	$(\|\|R\| + \|\|S\|\|) \cdot \log_{\frac{C}{B}} \|S\|$	$(\|\|R\| + \|\|S\|\|) \cdot \log_{\frac{C}{B}} \|S\|$
T_2	$\frac{\|\|R\|\|}{B}$	$\frac{\|\|R\|\|}{B}$	$\frac{\|\|R\|\cdot\|\|S\|\|}{C \cdot B}$	$\log_{\frac{B}{z}} \|R\|$	$(\|\|R\| + \|\|S\|\|) \cdot \log_{\frac{C}{B}} \|S\|$	$(\|\|R\| + \|\|S\|\|) \cdot \log_{\frac{C}{B}} \|S\|$
T_3	$\frac{\|\|R\|\|}{D \cdot B}$	$\frac{\|\|R\|\|}{D \cdot B}$	$\frac{\|\|R\|\cdot\|\|S\|\|}{D \cdot C \cdot B}$	$\log_{D \cdot \frac{B}{z}} \|R\|$	$(\|\|R\| + \|\|S\|\|) \cdot \log_{\frac{C}{B}} \|S\|$	$\frac{1}{D} \cdot (\|\|R\| + \|\|S\|\|) \cdot \log_{\frac{C}{B}} \|S\|$

predicates, $R.a_1 = S.a_1$ for an equi-join and $R.a_1 < S.a_1$ *and ...and* $R.a_n < S.a_n$ for a non-equijoin. We used the non-indexed NLJ algorithm to evaluate the non-equijoin, and the hash join as well as the B+-tree to evaluate the equi-join.

Metrics. Table 5 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 only, because we did not have privileges to perform profiling on AMD or Ultra-Sparc.

Table 5: Performance metrics

Metrics	Description
TOT_CYC	Total execution time on all three platforms in milliseconds (ms)
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)

5.2 Results

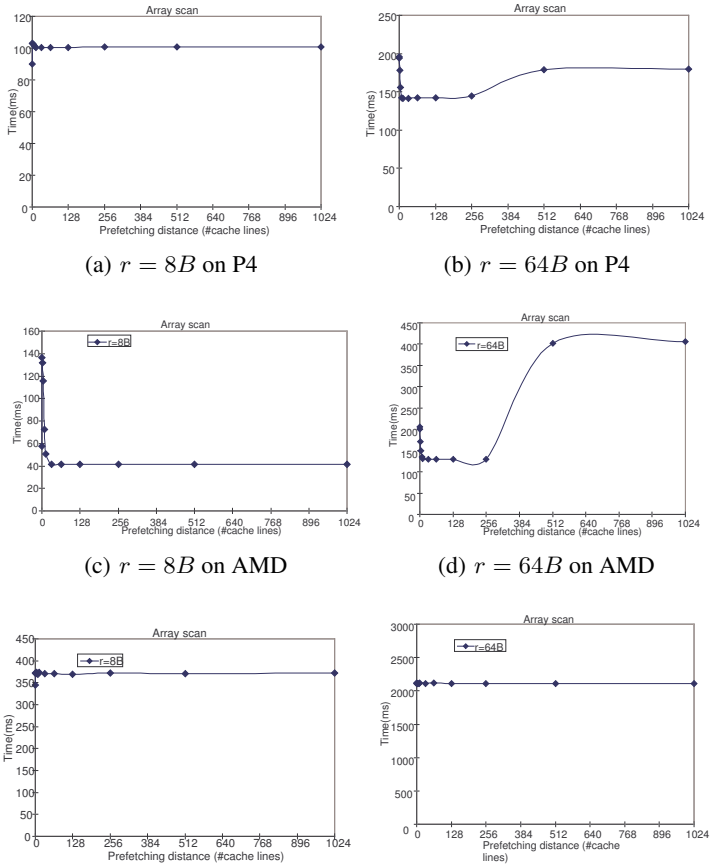
We present the experimental results on the three platforms. In general, the results on AMD are similar to those on P4. Additionally, the prefetching technique achieves a considerable performance improvement on P4 and AMD, whereas the prefetching technique has little performance improvement on Ultra-Sparc.

5.2.1 Storage models

Array. Figure 4 shows the execution time of the array scan with the software prefetching when $|R| = 8M$. We varied the tuple size. For each tuple size, we varied the prefetching distance in number of L2 cache lines. Software prefetching improves the array scan on P4 and AMD, whereas it has little performance impact on Ultra-Sparc.

Since hardware prefetching is enabled on P4 and AMD, software prefetching does not necessarily improve the performance. When the tuple size is small, the memory stalls are fully hidden with sufficient computation in the presence of hardware prefetching. For example, when $r = 8B$, each cache line contains 8 tuples. Software prefetching degrades the performance due to its computation overhead on P4. In contrast, when the tuple size is large, software prefetching further improves the scan performance in addition to hardware prefetching. Figure 5 shows the performance of array scan with the tuple size varied. The performance improvement of software prefetching increases as the tuple size increases. On P4, when the tuple size is larger than 16 bytes, software prefetching starts to improve the performance of the array scan. Note, on AMD, software prefetching improve the performance of the array scan when the tuple size is larger than 8 bytes. One possible reason is that the AMD has a larger memory latency than P4.

Software prefetching requires tuning on the prefetching distance in order to achieve the best performance on P4 and AMD. When the prefetching distance is small, the memory stalls are not completely hidden. When the prefetching distance is larger than the L1 cache capacity, a performance slowdown occurs due to the cache thrashing in the L1 cache. A similar performance slowdown is observed when the prefetching distance is larger than the number of cache lines in the L2 cache. Thus, to develop an efficient prefetching scheme, we need to (1) the latency and the cache block size to determine whether software prefetching can help hide the memory stalls; (2) the cache capacity to avoid prefetch too many cache lines. This validates our framework that T_3 includes the lower levels of tuning, T_1 and T_2.



(a) $r = 8B$ on P4 (b) $r = 64B$ on P4

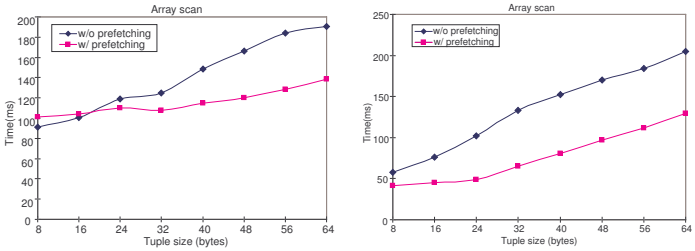
(c) $r = 8B$ on AMD (d) $r = 64B$ on AMD

(e) $r = 8B$ on Ultra-Sparc (f) $r = 64B$ on Ultra-Sparc

Figure 4: Array scan at the tuning level T_3: varying the prefetching distance.

Linked list. We first investigated the performance of the linked list scan without software prefetching. We varied the node size and found that the stable node size is 128B on P4 and AMD, and 64B on Ultra-Sparc. When the node size is the stable node size, the execution time becomes stable.

We next evaluated the prefetching technique on the linked list scan. The execution time of the linked list scan with software



(a) On P4 (b) On AMD

Figure 5: Array scan with and without prefetching: varying the tuple size.

prefetching is shown in Figure 6. Due to the random nature of the linked list scan, hardware prefetching has little performance impact. The software prefetching technique helps reduce the cache stalls on P4 and AMD, whereas it does not help on Ultra-Sparc. The performance improvement on a relation with a large tuple size is larger than that with a small tuple size. This is because, memory stalls are more significant and software prefetching better hides the memory stalls on a relation with a large tuple size. Similar to the array scan, software prefetching on the linked list requires the tuning on the cache block size, the cache capacity and the latency in order to determine the suitable prefetching distance.

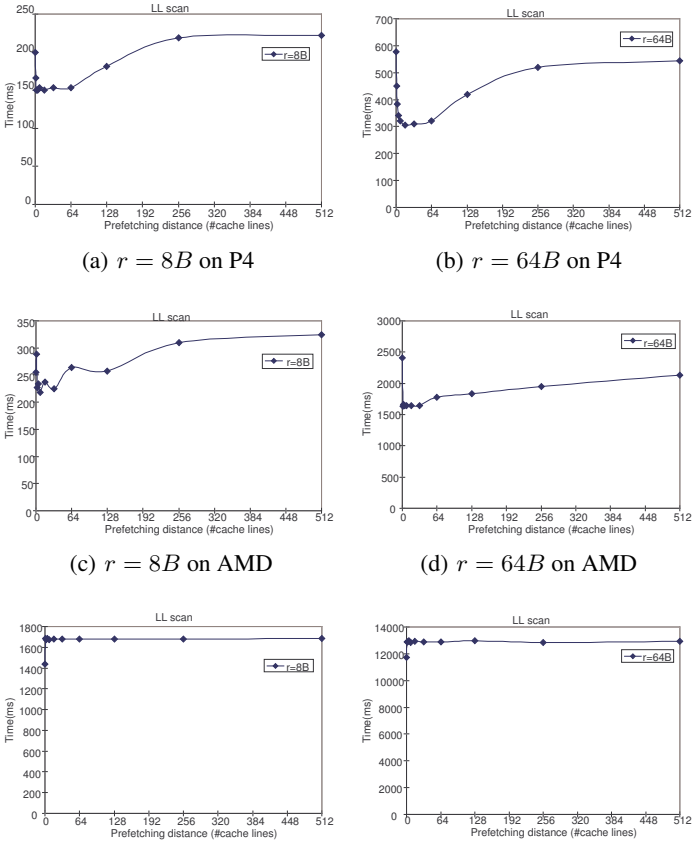
5.2.2 *B+-trees*

We used B+-tree indexing to evaluate the equijoin query. The measurements are shown in Figure 7. The reported results were obtained when $|R| = 200K$, $|S| = 32M$ and $r = s = 8$ bytes. $|R|$ was much smaller than $|S|$, since we focused on the spatial locality of the B+-tree index. This setting was comparable to the previous study [29, 30]. Since the tree index is static, we did not store the pointers in its internal nodes and used implicit addressing like CSS-trees [29]. In this implementation, the performance of our B+-trees at T_1 was similar to that of CSS-trees.

On all platforms, T_3 is the best among all variants; T_0 is 20%–30% slower than T_3. The reason for this phenomenon is that B+-trees at T_0 has a good spatial locality with the VEB layout using implicit addressing. In our experiments, the COB+-tree is a binary tree. Each internal node is 4 bytes. Suppose a L2 cache line is 128 bytes (the cache line size on P4 or AMD). It can hold 32 nodes. Ignoring the cache block alignment, a subtree of five levels can fit into one cache line. This good spatial locality of VEB layout greatly reduces the cache misses on the index probes.

Comparing the performance gain of each level of tuning, we find that T_2 has little performance impact on the B+-trees. Finally, the software prefetching technique, T_3, considerably improves the overall performance (except on Ultra-Sparc). The performance improvement is due to (1) the software prefetching hiding the cache stalls, and (2) the small tree height. The performance improvement is 20%–30%, which is comparable to that shown in previous studies on simulators [15].

We investigated the cache performance of the index probes on P4. Figure 8 shows the time breakdown of the index probes at different levels of tuning. T_1–T_3 have a stable busy time, and T_0 has a larger busy time than other levels of tuning due to the larger amount of computation required by the VEB layout. T_0–T_2 have a similar cache performance. Among all levels of tuning, T_3 has the best cache performance.



(a) $r = 8B$ on P4 (b) $r = 64B$ on P4

(c) $r = 8B$ on AMD (d) $r = 64B$ on AMD

(e) $r = 8B$ on Ultra-Sparc (f) $r = 64B$ on Ultra-Sparc

Figure 6: Linked list scan at the tuning level T_3: varying the prefetching distance.

5.2.3 *NLJs*

Figure 9 shows the time comparison for non-indexed nested-loop joins with different levels of tuning. The reported results were obtained when $||R|| = ||S|| = 32M$ bytes and $r = s = 128$ bytes (both relations have $256K$ tuples).

According to our cost model, T_1 chooses the L2 cache as its target level, whereas T_2 chooses the L1 cache as its target level. T_1 is even slower than T_0, because T_1 chooses the incorrect target level at the absence of the knowledge of the latency. Note that the L1 cache is the most significant level of the cache according to our cost function in Table 3. This is evidence that a higher level of tuning does not guarantee a higher performance. We illustrate this result by comparing the performance of T_1 when the target level of cache is the L2, the L1 or the TLB, as shown in Figure 10.

T_3 applies software prefetching to both of the outer and the inner relations in the join. The prefetching distance was set to be one on the three platforms. The join performance improvement by software prefetching is insignificant, because the blocking technique has achieved a good cache locality.

5.2.4 *Hash joins*

Figure 11 shows the performance of hash joins when $|R| = |S| = 8M$ and $r = s = 8$ bytes. Note, in Figure 11 (a)(d), the prefetching distance being zero means that the result is obtained from the simple hash join (without prefetching). We do not show the results on Ultra-Sparc, because the performance impact of soft-

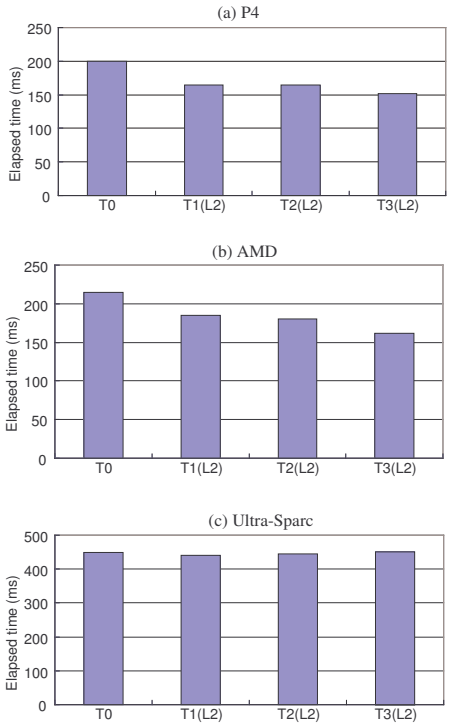


Figure 7: B+-trees

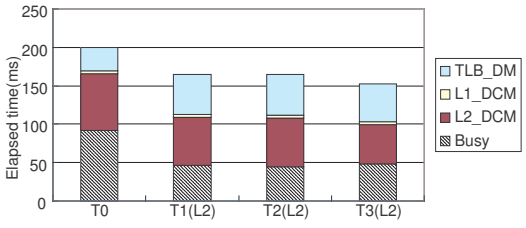


Figure 8: B+-trees: time breakdown on P4

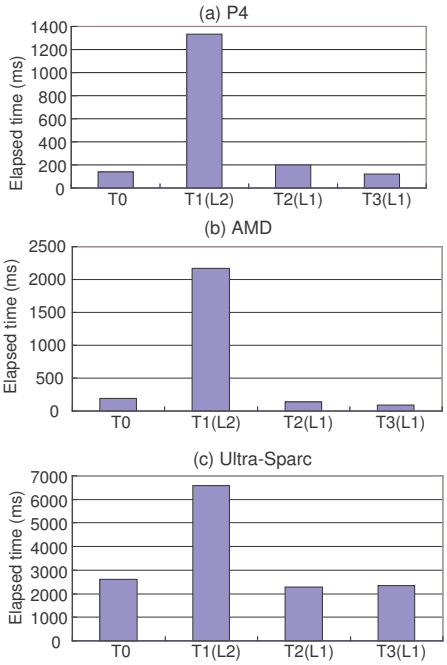


Figure 9: Non-indexed nested-loop joins

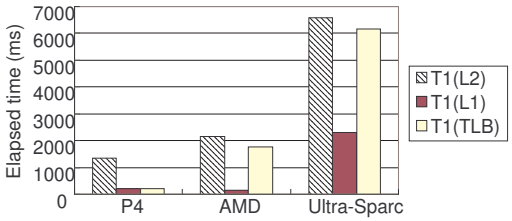


Figure 10: Non-indexed nested-loop joins at T_1

ware prefetching is insignificant.

We summarize the results in two aspects. First, both partitioning and prefetching improve the performance of the hash join. The prefetching hash join achieves the best performance when the prefetching distance is 16 and four on P4 and AMD, respectively. The partitioned hash join achieves the best performance when the partition granularity is 4K and 16K tuples, respectively. The performance trend with a single technique is concave. Thus, the suitable setting of these techniques requires tuning according to the cache capacity.

Second, the performance improvements of applying prefetching only, applying partitioning only and applying both techniques over the simple hash join are 40%, 47% and 30%, respectively, on P4 and are 40%, 85% and 69%, respectively, on AMD. The cumulative performance improvement of the two techniques can be smaller than that of applying a single technique. It indicates that prefetching hurts the performance of the optimized partitioned hash join.

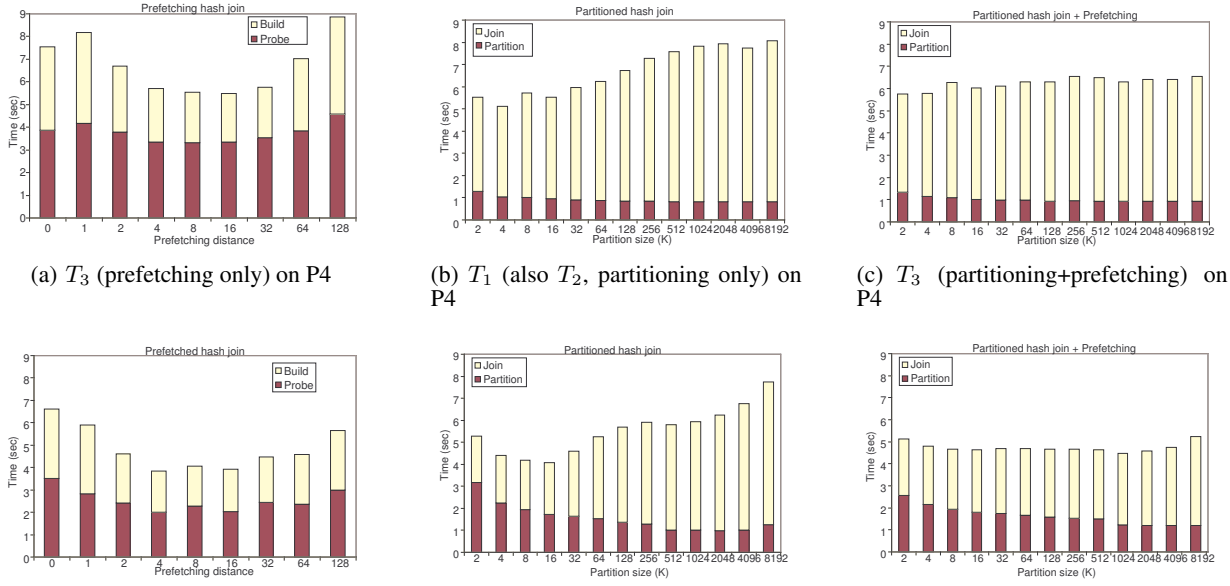
5.2.5 Summary

Through the four case studies, we observed the following three results. First, among the four levels of tuning in our framework, T_3 utilizes the complete knowledge of a specific memory hierarchy and achieves the best performance on all the four case studies except the hash join. Second, T_1 and T_2 in our framework do not necessarily improve the performance over T_0 due to the possible ineffective tuning based on the incomplete knowledge of the memory hierarchy. Third, T_0 achieves a comparable performance to its higher levels of tuning, whose execution time is less than twice that of the fine-tuned algorithm.

6. CONCLUSION

As the memory hierarchy becomes an important factor for the performance of database applications, it is imperative to improve the memory performance of relational query processing. Cache-oblivious techniques optimize all levels of any memory hierarchies without knowledge of cache parameters of a specific memory hierarchy, whereas cache-conscious techniques can potentially achieve a better performance with careful tuning based on the cache characteristics. Considering the strengths and weaknesses of both techniques, we propose a general framework to quantify the performance impact of different degrees of tuning. Through studying on several basic data structures and algorithms in query processing, we show that our framework is useful in this process of tuning.

As future work, we are interested in extending our framework to



(a) T_3 (prefetching only) on P4

(b) T_1 (also T_2, partitioning only) on P4

(c) T_3 (partitioning+prefetching) on P4

(d) T_3 (prefetching only) on AMD

(e) T_1 (also T_2, partitioning only) on AMD

(f) T_3 (partitioning+prefetching) on AMD

Figure 11: Hash joins

the dynamic characteristics of the memory hierarchy on the chip-multiprocessors [21]. We are also interested in tuning the cache-conscious algorithm adapting to the runtime dynamics of the memory hierarchy based on the hardware profile.

7. ACKNOWLEDGEMENT

We thank the anonymous reviewers for their comments on the earlier versions of this paper. This work was supported by grants DAG05/06.EG11, HKUST6263/04E, and 617206, all from the Hong Kong Research Grants Council.

8. REFERENCES

[1] A. Ailamaki. Database architectures for new hardware. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, page 1148, Washington, DC, USA, 2005. IEEE Computer Society.

[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] AMD Corp. *Software Optimization Guide for AMD64 Processors*, 2005.

[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/, 2002.

[9] P. Bohannon, P. Mcllroy, 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Inspector joins. In *VLDB '05: Proceedings of the 31st*

international conference on Very large data bases, pages 817–828. VLDB Endowment, 2005.

[15] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. *SIGMOD Rec.*, 30(2):235–246, 2001.

[16] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B+-trees: optimizing both cache and disk performance. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 157–168, New York, NY, USA, 2002. ACM Press.

[17] 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.

[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] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey. Cache-conscious Frequent Pattern Mining on a Modern Processor. VLDB, 2005.

[20] R. A. Hankins and J. M. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *VLDB*, pages 417–428, 2003.

[21] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, S. Harizopoulos, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR '07: Proceedings of the Third International Conference on Innovative Data Systems Research*, Asilomar, CA, USA, 2007.

[22] S. Harizopoulos and A. Ailamaki. Improving instruction cache performance in OLTP. *ACM Trans. Database Syst.*, 31(3):887–920, 2006.

[23] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD Conference*, pages 383–394, 2005.

[24] 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.

[25] B. He and Q. Luo. Cache-oblivious query processing. In *CIDR '07: Proceedings of the Third International Conference on Innovative Data Systems Research*, Asilomar, CA, USA, 2007.

[26] Intel Corp. *Intel(R) Itanium(R) 2 Processor Reference Manual for Software Development and Optimization*, 2004.

[27] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. *SIGMOD Rec.*, 30(2):139–150, 2001.

[28] S. Manegold, P. Boncz, and M. Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709–730, 2002.

[29] 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.

[30] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 475–486, New York, NY, USA, 2000. ACM Press.

[31] 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.

[32] Sun Corp. *UltraSPARC (R) III Cu Users Manual*, 1997.

[33] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *VLDB '03: Proceedings of the 29th International Conference on Very Large Data Bases*, pages 405–416, 2003.