Cache Conscious Trees: How Do They Perform on Contemporary Commodity Microprocessors?

Kyungwha Kim[1], Junho Shim[2,*], and Ig-hoon Lee[3]

[1,2] Dept of Computer Science, Sookmyung Women's University, Korea
{kamza81, jshim}@sookmyung.ac.kr
[3] Prompt Corp., Seoul, Korea
ihlee@prompt.co.kr

Abstract. Some index structures have been redesigned to minimize the cache misses and improve their CPU cache performances. The Cache Sensitive B+-Tree and recently developed Cache Sensitive T-Tree are the most well-known cache conscious index structures. Their performance evaluations, however, were made in single core CPU machines. Nowadays even the desktop computers are equipped with multi-core CPU processors. In this paper, we present an experimental performance study to show how cache conscious trees perform on different types of CPU processors that are available in the market these days.

1 Introduction

Modern desktop computing environment has been in on-going evolution in terms of its architectural features. Two of the most noticeable features in last few years may be observed in areas of main memory and CPU.

Random access memory becomes more condensed and cheaper. Nowadays it becomes common to equip a new PC even for home uses with 1 giga bytes or more of random access memory[1]. A recent launch of new PC operation system[2] has accelerated the minimal memory requirement for a system. Such a trend that PCs need and therefore are equipped with more amount of memory than ever before is expected to last for a while.

As a hardware system contains larger amount memory, it becomes feasible to store and manage database within main memory. Researchers have paid attention to various aspects of main memory databases. The index structure for main memory is one area in which T-Trees were proposed as a prominent index structure for main memory [9]. In [12,13], Rao et al claimed that B-Trees may outperform T-Trees due to the increasing speed gap between cache access and main memory access. CPU clock speeds have

[*] Corresponding author.

[1] For example, Hewlett-Packard and Dell, two leading companies with respect to the worldwide PC market shares, recommend their customers to have at least 1 GB memory for their middle-line home desktop computers. See http://www.shopping.hp.com/ or http://www.dell.com/.

[2] Windows Vista™, http://www.microsoft.com/windows/products/windowsvista

increased at a much faster rate than memory speeds [1,4,11]. The overall computation time becomes more dependent on cache misses than on disk buffer misses.

In the past we considered the effect of buffer cache misses to develop an efficient disk-based index structure. The same applies to the effect of cache misses. A design of index structure with regard to its cache behavior may lead to the improvement in terms of cache hits. A most well-known cache optimized index structure for main memory database systems has been CSB+-Trees (Cache Sensitive B+-Trees) [13], a variant of B+-Trees. Recently, Lee et al [10] claimed that T-Trees index may be also redesigned to better utilize the cache, and they introduced a new index structure CST-Trees (Cache Sensitive T-Trees). In their experiment, CST-Trees outperform CSB+-Trees on searching performance and also show comparable performance on update operations.

A feature in a contemporary CPU architecture comes along with the industry that has launched multi-core CPU microprocessors in the market. It has been only about one year since the first dual-core PC processor was introduced in the market. Very recently, two leading manufacturers in the industry again announced that their upcoming processors will be redesigned to double the number of cores within a processor[3,4]. Experts expect that we will have eight-or 16-core microprocessors in a near feature [8,7]. The trend concurs in the industry that manufactures processors for workstations and server-levels as well[5,6]. What it has meant to the software research community is to investigate the performance impact that a multi-core processor may offer, and to change the software architecture to exploit a higher performance benefit of the design of new processor. The database community is one of the early birds which found the trend [2,8].

In this paper, we provide an experimental study to show how the traditional index structures and recently developed cache conscious versions actually perform in modern computer environments. We conduct the experiment to check the performances of T-Trees, B+-Tress, CST-Trees, and CSB+-Trees, on contemporary available computer systems equipped with single-core and multi-core CPUs.

In short, the experimental result shows that cache conscious designs for index structures may achieve the performance gain in hardware systems with multi-core CPUs as they do in hardware systems with single-core CPUs. The experiment is worthy not only because we show the empirical study in a real modern hardware system equipped with brand new CPU configuration, but also because the result may be used in future as an comparable source to an analytical model of cache index structure.

The rest of this paper is structured as follows. In Section 2 we present the related work on cache conscious tree index. The cache conscious B+-Trees and the original T-Trees are briefly introduced for explanation purpose. We also provide a structural sketch on cache conscious T-trees. In Section 3 we present a recent trend on CPU technology and illustrate an architectural view of multi-core CPU processor. In Section 4 we present the experimental performance study of four competitors: T-Trees, B+-Tress, CST-Trees, and CSB+-Trees. And finally, conclusions are drawn in Section 5.

[3] Intel Ignites Quad-Core Era, http://www.intel.com/pressroom/archive/releases/20061114comp. htm
[4] AMD Details Native Quad-core Design Features, http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543_544~115794,00.html
[5] IBM PowerPC Microprocessor, http://www.chips.ibm.com/products/powerpc/
[6] Sun Microsystems, Inc.: UltraSPARC Processors, http://www.sun.com/processors/

2 Background

2.1 Related Work on Index Structures

Most widely used tree-based index structures may include B+-Trees, AVL-Trees, and T-Trees [9]. B-Trees are designed for disk-based database systems and need few node accesses to search for a data since trees are broad and not deep, i.e., multiple keys are used to search within a node and a small number of nodes are searched [6]. Most database systems employ B+-Trees, a variant of the B-Tree.

In [12,13], Rao et al showed that B+-Trees have a better cache behavior than T-Trees, and suggested to fit a node size in a cache line, so that a cache load satisfy multiple comparisons. They introduced a cache sensitive search tree [12], which avoids storing pointers by employing the directory in an array. Although the proposed tree shows less cache miss ratio, it has a limitation of allowing only batch updates and rebuilding the entire tree once in a while. They then introduced an index structure called CSB+-Tree (Cache-Sensitive B+-Tree) that support incremental updates and retain the good cache behavior of their previous tree index structure [13]. Similar to their previous tree structure, a CSB+-Tree employs an array to store the child nodes, and one pointer for the first child node. The location of other child nodes can be calculated by an offset to the pointer value.

The AVL-Tree is a most classical index structure that was designed for main memory [6]. It is a binary search tree in which each node consists of one key field, two (left and right) pointers, and one control field to hold the balance of its subtree (Figure 1-(a)). The left or right pointer points the left or right sub-trees of which nodes contain data smaller or larger than its parent node, respectively. The difference in height between the left and right sub-trees should be maintained smaller or equal to one.

The major disadvantage of an AVL-Tree is its poor storage utilization. Each tree node holds only one key item, and therefore rotation operations are frequently performed to balance the tree. T-Trees address this problem [9]. In a T-Tree, a node may contain *n* keys (Figure 1-(b)). Key values of a node are maintained in order. Similar to

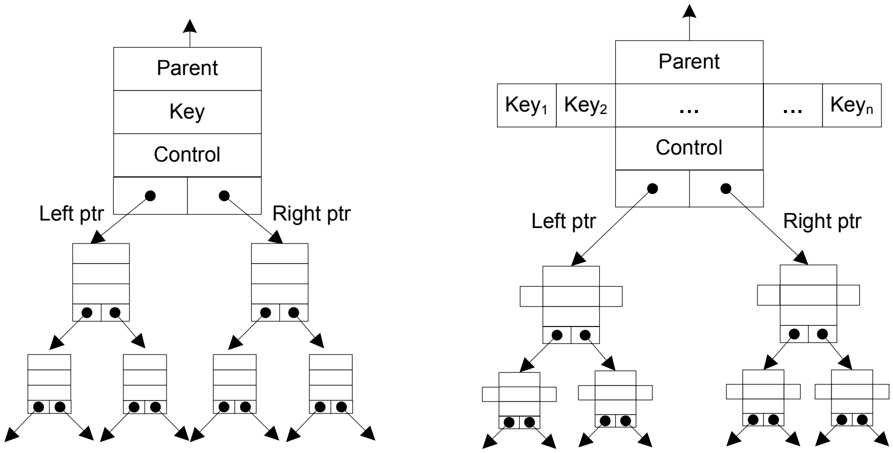


Fig. 1. (a) AVL-Tree (b) T-Tree : The node structure of AVL and T-Trees

an AVL-Tree, any key stored within a left and right sub-tree should be smaller or larger than the least and largest data of a node, respectively. The tree is kept balanced as for the AVL-Tree.

2.2 Cache Sensitive T-Trees

T-Trees are not so cache sensitive either as the following reasons [10]. First, cache misses are rather frequent in that a T-Tree has a deeper height than a B+-Tree, and that it does not align the node size with the cache line size. Secondly, a T-Tree uses only two keys (maximum and minimum keys) for comparison within the copied data in cache while a B+-Tree use $|log_2 n|$ keys that are brought to the cache for comparison.

In [10], Lee et al modified the original T-Tree to improve the cache behavior and introduced a CST-Tree (Cache Sensitive T-Tree), which is a n-way search tree consisting of node groups and data nodes. Figure 2 shows a node structure of CST-Trees.

A CST-Tree consists of data nodes and node groups. A data node contains keys while a node group consists of maximal keys of data nodes. Each node group is a binary search tree represented in an array. It works as a directory structure to locate a data node that contains an actual key. The size of the binary search tree is not big and great portion of it may be cached. More importantly, the cache utilization can be high since every search needs to explore the tree. The child node groups of a node group are stored contiguously as well. A CST-Tree is balanced by itself, and a binary search tree of any node group is also balanced. As recommended in [3,5,12], in a CST-Tree the size of each node group is aligned with cache line size, so that there will be no cache miss when accessing data within a node group.

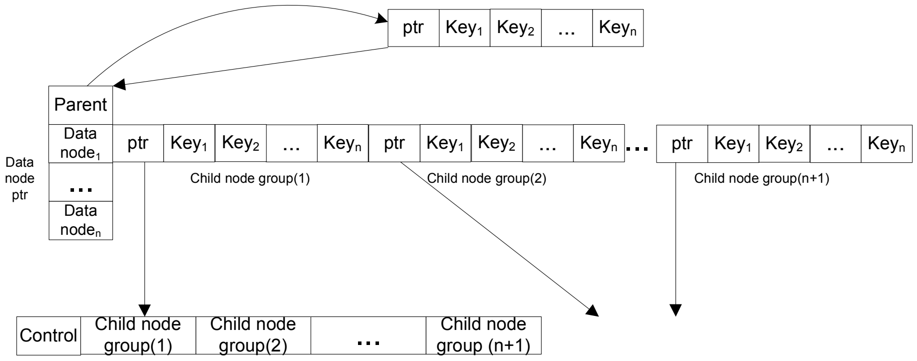


Fig. 2. The node structure of CST-Trees

3 Trends in CPU Processor Technology

Over the past decade, processor speeds have drastically increased according to Moore's law, while DRAM speeds have not. Memory latency tends to decrease by half every six years [2]. This incurs a so-called *memory wall* problem that causes a processor to keep waiting more time for the completion of main memory access. The processor utilization becomes much less as it runs a program with lower memory or cache locality. A noticeable change appears in a processor design. The clock speed growth is no longer high, i.e., it hits a wall two years ago [14], while the number of transistors on a processor continues to climb, i.e., it doubles every 18 months [2]. Another trend is to let a processor enable higher level of parallelism without compensating power constraints. Then major CPU manufactures have shifted their processor designs toward chip multiprocessors (CMPs).

While some early CMPs employed private per-core cache designs, more recent ones employed shared last-level on-chip caches [7]. Sharing a cache may provide the multiple threads with more flexible allocation of the cache space, and is also expected to achieve higher performance when cores share data. Figure 3 illustrates an architectural view of a multi-core processor which shares a cache located outside the cores yet on the processor chip. Note that a processor in the figure is dual-core, i.e., the number of cores in a processor chip is 2, and the last-level on-chip cache is L2. As mentioned in Introduction, the industry recently began to deliver 4-core processors and also processors with L3 shared.

Database research community has already begun to explore higher performance that might be offered by new multi-core processors. Ailamaki et al's tutorial [2] provides a good survey on the modern architecture of commodity processors and related issues on database systems. In their previous work [1], they perform the experiment to analyze the query execution time by several commercial DBMSs. From

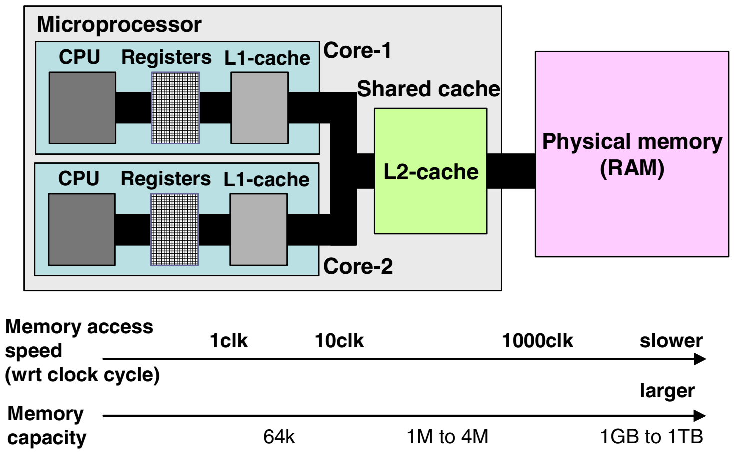


Fig. 3. Architectural view of a multi-core processor (dual-core in this figure) and its memory hierarchy[7]

[7] Actual memory speeds and capacities vary from a processor to another. We referenced Ailamaki et al's report [2], and two recent dual-core microprocessor product lines: Intel® Core™2 Duo Processors and AMD Opteron™ Processors.

the results they suggest that database developers need to pay more attention to optimize data placement for L2 cache, rather than L1, because L2 data stalls are a major component of the query execution time. The hardware systems that they performed the experiment all contain single-core processors, although they are the most up-to-date by then. Their suggestion is still valid by now or becomes more important in a sense that we now have larger speed gaps between processor clock and memory in most hardware systems.

4 Performance Evaluation

4.1 Experimental Environment

We performed an experimental comparison of the B+-Trees, T-Trees, and their cache conscious versions CSB+-Trees and CST-Trees. For the performance comparison, we implemented all the methods. For the implementation of CSB+-Trees and T-Trees, we referred to the sources [9, 13] that are proposed by the original authors. For the implementation of CST-Trees, we referred to the source [10] that we previously built. Originally, the source codes were built and tested on Sparc machines, and therefore we should modify some codes accordingly to the hardware platforms that were equipped with multi-core CPUs.

The hardware platforms that we chose for experiment are listed in Figure 4[8]. Both machine A and B are equipped with one dual-core CPU microprocessors of which architectures are different and manufactured by different corporations. The CPU processor contained in machine-A employs a shared L2 cache while one in machine-B employs separate L2 caches per core. Note that for comparative study we performed our experiment on hardware machines with single-core CPU as well. Both machine C and D are equipped with single-core CPU processors. Machine-C has one processor while machine-D has two processors.

We implemented all the codes in C, and the programs were compiled and built by GNU cc compiler, which are available for every platform that we used in the experiment. For the performance comparison, we implemented all the methods including T-Trees, CST-Trees, B+-Trees, and CSB+-Trees. All the methods are implemented to support search, insertion, and deletion.

In the original CSB+-Tree, node groups are allocated dynamically upon node split. Memory allocation calls can be saved if we pre-allocate the space for a full node group whenever a node group is created. CST-Trees also adopt a scheme to pre-allocate the whole space for a node group. In order to conduct a fair performance comparison, we also implemented a variant of CSB+-Trees in which the whole space of a node group is pre-allocated when keys are inserted. In our insertion experiment, we call it CSB+-(full), while we call the original CSB+-Tree as CSB+-(org). For deletion, we used “lazy” policy as it is practically used [13,10].

[8] We used a free-software to check the details of chipsets employed in machine A, B, and C. The program is available at http://www.cpuid.com/, and the version we used is v1.39.

	Machine-A	Machine-B	Machine-C	Machine-D
No. of CPU processors	1	1	1	2
Multi-Core? (No. of cores per processor)	Yes (2)	Yes (2)	No (1)	No (1)
Cache structure	Shared L2 cache across dual cores	Separate L2 cache per core	Separate cache per processor	Separate cache per processor
CPU clock speed	2.66GHz	2.0GHz	2.40GHz	1.20GHz
L1 cache <cache size, cache line size>	2×<32K bytes, 64bytes> (Data) 2×<32K bytes, 64bytes> (Code)	2×<64K bytes, 64bytes> (Data) 2×<64K bytes, 64bytes> (Code)	<8K bytes, 64bytes> (Data) <12 Koups> (Trace)	<64 Kbytes, 64bytes> (Data) per chip <32 Kbytes, 64bytes> (Code) per chip
L2 cache <cache size, cache line size>	<4096K bytes, 64bytes>	2×<512K bytes, 64bytes>	<512K bytes, 64bytes>	2×<8M bytes, 64 bytes>
RAM	2G bytes DDR2	1G bytes DDR2	1.5G bytes DDR	2G bytes DDR
Operating system	Redhat Enterprise Linux ES v3	Redhat Enterprise Linux ES v3	Redhat Enterprise Linux ES v3	SunOS 5.9

Fig. 4. The CPUs and their cache specifications of four different machines that are used in the experiment[9]

In order to measure the number of CPU cache misses, we used the Valgrind debugging and profiling tool for Linux operating system and the Performance Analysis Tool for Sun operating system.[10] We only considered the L2 level cache misses as in [13,10].

In all experiments we set the keys and each pointer to be 4 bytes integers and 4 bytes. All keys are randomly chosen as integer values of which ranges are from 1 to 10 million. The keys are generated in advance before the actual experiments in order to prevent the key generating time from affecting the measurements. The node sizes of all the methods are chosen to 64 bytes, same to the cache line size of each machine, since choosing the cache line size to be the node size was shown close to optimal [12, 13, 10]. We repeated each test three times and report the average measurements.

[9] Note that we do not include the actual model names of the microprocessors, since the purpose of our experiment is not to reveal the precise benchmark of each microprocessor.

[10] The versions that we used are the Valgrind 3.2.3 and the Sun ONE Studio 8. The Valgrind is freely available under GNU license at http://www.valgrind.org.

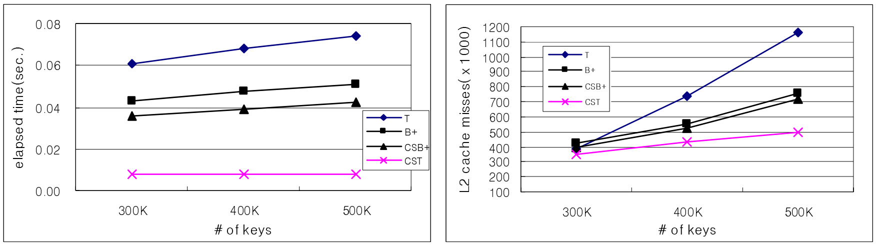
4.2 Results

Searching

In the first experiment, we compared the search performance of each index structure. We generated the different number of keys and insert all the keys into each index, and then measured the time and the number of cache miss that were taken by 200,000 searches. All search key values were randomly chosen among the generated keys. Figure 5 to 8 show the results[11].

In general, CST-Trees show the best both in terms of speed and cache miss rate. CSB+-Trees, B+-Trees, and T-Trees follow the next in order. In a machine-A (1CPU, dual-cores, separate L2 cache), CST-Trees are on average 79.8%, 83.3%, and 88.3% faster[12] than CSB+-Trees, B+-Trees, and T-Trees (Figure 5-(a)). CST-Trees also show the least number of cache misses among the methods, i.e., on average 20.5%, 25.0%, 35.4% less[13] than CSB+-Trees, B+-Trees, and T-Trees, respectively (Figure 5-(b)). CSB+-Trees also outperform the original B+-Trees in terms of both speed and cache misses. In another machine-B that is equipped with a dual-core processor yet separate L2 cache, CST-Trees also show the fastest in speed and the least in number of cache misses, while CSB+-Trees, B+-Trees and T-Trees follow the next in order (Figure 6). In other machine-C and D, each method shows a similar pattern in their performance ranks (Figure 7 and 8).

We may observe two particular interesting results in these experiments. Firstly, as the number of searches becomes larger, the difference between CST-Trees and other methods in their cache miss numbers becomes larger too. Then among the methods, T-Tree shows steeper slope than others in its cache miss graphs, although the number of cache misses are linearly incremented as others. Secondly, the number of cache misses may greatly vary with the machine architectures. For example, in Figure 5-(b), the average cache miss numbers of four trees on machine-A with 500K search keys is about



(a) CPU elapsed time (b) Cache misses

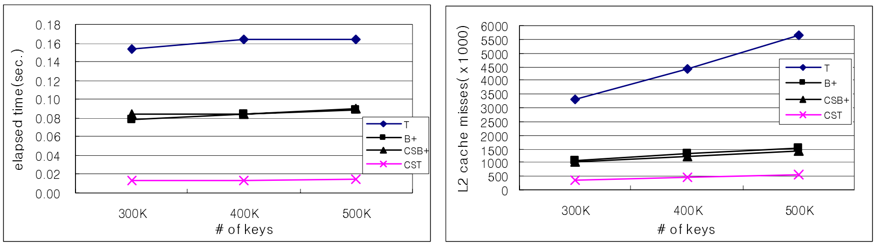
Fig. 5. Search performances in machine-A (1CPU, dual-cores, and shared L2 cache)

[11] As mentioned before, we do not attempt to directly compare the performances of four microprocessors by drawing all graphs in a chart, since it may misguide some readers to directly consider the results as the performance benchmark of each microprocessor. Note that for comparative study we also include the results of our experiment on machine-D of which result data previously appeared in [10] in part.

[12] We use a relative performance ratio, i.e., (A-B)/A. For example, (elapsed_time by CSB+ - elapsed_time by CST) / elapsed_time by CSB+.

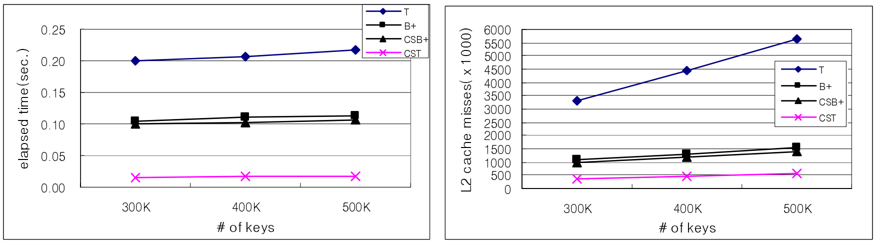
[13] Here again, we use a relative performance ratio, i.e., (A-B)/A.

782K, while it is 2,278K and 2,276K on machine-B and C with same search keys, respectively. Note that the total L2 cache size of machine-A is 4 times bigger than B, and 8 times bigger than C, although their cache line sizes are same to 64bytes. The machine-D that has a much larger L2 cache size significantly decreases the average number of cache misses for all cases. According to the result that both machine-B and C show a similar number of cache misses; just to have a double-cores without sharing the L2 cache may not affect the number of cache misses.



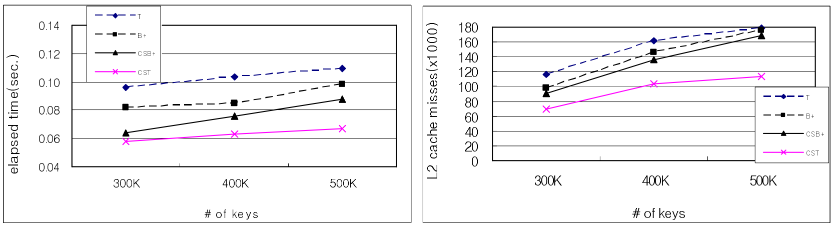
(a) CPU elapsed time (b) Cache misses

Fig. 6. Search performances in machine-B (1CPU, dual-cores, and separate L2 caches)



(a) CPU elapsed time (b) Cache misses

Fig. 7. Search performances in machine-C (1CPU, single-core)



(a) CPU elapsed time (b) Cache misses

Fig. 8. Search performances in machine-D (2CPUs, separate caches)

Insertion and Deletion

In the next experiment, we tested the performance of insertion and deletion. Before testing, we first stabilized the index structure by bulk-loading 1 million keys, same as in

[13,10]. Then we performed up to 20K operations of insertion and deletion and measure the time that were taken for the given number of operations (Figure 9-(a) to 12-(b)).

Full CSB+-Trees show the best in insertion, while B+-Trees, CST+-Trees show comparable performance in their insertions. T-Trees are among the worst in machines except one (machine-D) where original CSB+-Trees also perform poor.

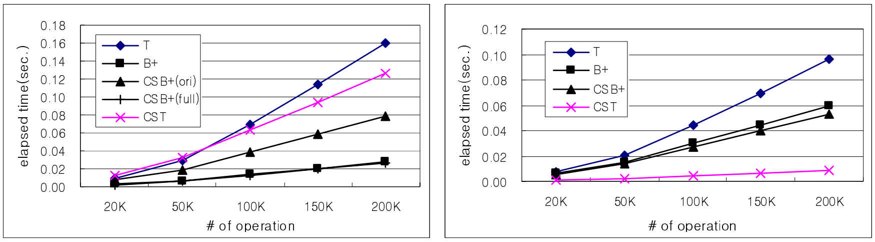


Fig. 9. (a) Insertion (b) Deletion : CPU elapsed times in machine-A

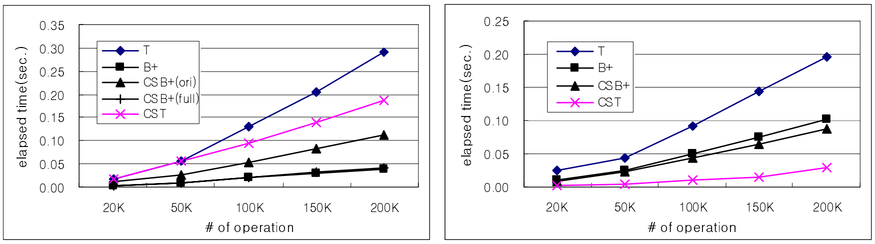


Fig. 10. (a) Insertion (b) Deletion : CPU elapsed times in machine-B

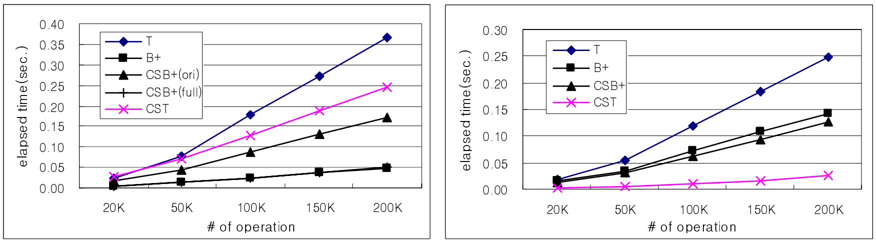


Fig. 11. (a) Insertion (b) Deletion : CPU elapsed times in machine-C

The delete performance also showed a similar pattern to that of search, in that the “lazy” strategy was employed for deletion. Most of the time on a deletion is spent on pinpointing the correct entry in the leaf node. In all experiments (Figure 9-(b) to 12-(b)), CST-Trees show the best both in terms of speed and cache miss rate. CSB+-Trees, B+-Trees, and T-Trees follow the next in order.

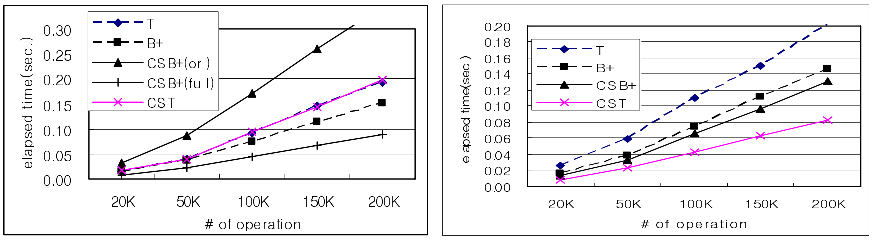


Fig. 12. (a) Insertion (b) Deletion : CPU elapsed times in machine-D

5 Conclusion

In this paper, we present an experimental evaluation of tree-based index structures on multiple conventional processors. CST-Tree is one of the index structures that we especially care for the performance on multi-core CPU processors.

Our experimental results show that cache sensitive trees provide much better performance than their original versions. In searching operations, CST-Trees show much superior performance than CSB+, B+-Trees, and T-Trees. CSB+-Trees also show better performance than B+-Trees. CST-Trees and CSB+-Trees also show good performance on insertion operations and better performance on deletion operations, although the performance benefits over their original versions are less than in searching.

The experiment is worthy because the experimental results show that cache sensitive index structures may benefit of the designs of modern commodity microprocessors. It is, however, limited in that we have not developed an analytical model of our cache sensitive index on a multi-level shared cache architecture, so that we can mathematically compare the empirical results to the theoretically-expected behavior of the model. This should be one of the works we shall deal with in future.

It is one of the hottest research topics in database community to tune a database management system to perform well enough to benefit the commodity microprocessors. Building an index structure more cache-conscious is a way to decrease the cache miss and therefore to benefit more the larger size of shared cache. However, those cache conscious technologies employed in either CST-Trees or CSB+-Trees may not inherently resolve a problem of so called *cold miss*. We are developing a CST-Tree version which employs a prefetching technology to reduce the cold miss rate.

References

1. Ailamaki, A., DeWitt, D.J., Hill, M.D., Wood, D.A.: DBMSs On A Modern Processor: Where Does Time Go? In: Proc. of the 25th International Conference on Very Large Database Systems, pp. 266–277 (1999)
2. Ailamaki, A., Govindaraju, N.K., Harizopoulos, S., Manocha, D.: Query co-processing on commodity processors. In: Proc. of the 32nd International Conference on Very Large Database Systems, Tutorials, pp. 1267–1267 (2006)

3. Bohannon, P., Mcllroy, P., Rastogi, R.: Main-Memory Index Structures with Fixed-Size Partial Keys. In: Proc. of the 2001 ACM SIGMOD Int'l Conf. on Management of Data, pp. 163–174. ACM Press, New York (2001)
4. Boncz, P., Manegold, S., Kersten, M.L.: Database Architecture Optimized for the new Bottleneck: Memory Access. In: Proc. of the 19th International Conference on Very Large Database Systems, pp. 54–65 (1999)
5. Chilimbi, T.M., Davidson, B., Larus, J.R.: Cache-Conscious Structure Definition. In: Proc. of the ACM SIGPLAN 1999 conference on Programming language design and implementation, pp. 13–24. ACM Press, New York (1999)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge (1990)
7. Hsu, L.R., Reinhardt, S.K., Iyer, R., Makineni, S.: Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In: Proc. of the 15th International Conference on Parallel Architectures and Compilation Techniques, pp. 13–22 (2006)
8. Ghoting, A., Buehrer, G., Parthasarathy, S., Kim, D., Nguyen, A., Chen, Y.-K., Dubey, P.: Cache-conscious frequent pattern mining on modern and emerging processors. The VLDB Journal 16(1), 77–96 (2006)
9. Lehman, T.J.: A Study of Index Structures for Main Memory Database Management System. In: Proc. of the 12th International Conference on Very Large Database Systems, pp. 294–303 (1986)
10. Lee, I.-h., Shim, J., Lee, S.-g., Chun, J.: CST-Trees: Cache Sensitive T-Trees. In: DASFAA 2007. Proc. of the 12th International Conference on Database Systems for Advanced Applications, pp. 398–409 (2007)
11. Manegold, S., Boncz, P.A., Kersten, M.L.: Optimizing database architecture for the new bottleneck: memory access. The VLDB Journal 9(3), 231–246 (2000)
12. Rao, J., Ross, K.A.: Cache Conscious Indexing for Decision-Support in Main Memory. In: Proc. of the 19th International Conference on Very Large Database Systems, pp. 78–89 (1999)
13. Rao, J., Ross, K.A.: Making B+ Trees Cache Conscious in Main Memory. In: Proc. of the 2000 ACM SIGMOD International Conference on Management of Data, pp. 475–486. ACM Press, New York (2000)
14. Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, available at http://www.gotw.ca/publications/concurrency-ddj.htm