Evaluating the Performance of CSB+-Trees on Multithreaded Architectures

Layali K. Rashid and Wessam M. Hassanein
Department of Electrical and Computer Engineering
University of Calgary
{lrashid@, hassanein@enel.}ucalgary.ca

ABSTRACT

Modern architectures have made considerable increases in processor speed and performance. However, Database Management Systems (DBMSs) fall far short from achieving their ideal performance. DBMSs are widely used in almost every large organization. Therefore, it is important to achieve fast data retrieval and processing. Recent studies have shown that more than 50% of the execution time in database operations is spent waiting for data. CSB+-trees were introduced to speedup index structure operations, mainly the search and update. In this paper we propose a multithreading technique to utilize the two threads available in an Intel Pentium 4 Hyperthreaded (HT) platform. Our technique gains speedup ranging from 29% to 70% for dual-threaded CSB+-tree on an HT enabled platform compared to a single-thread version running on HT disabled architecture.

1. INTODUCTION

Hiding the gap between the memory hierarchy and CPU speeds has been the aspiration of many prior researches. As DRAM sizes are getting larger, more research is targeting memory-resident data, i.e. datasets reside entirely in main memory. Considerable efforts have been made to hide cache access latency by either reducing the number of cache misses [14] or overlapping latencies with other useful work [16]. DBMS, in particular, data retrieval and update is an attractive candidate for these optimizations since it usually undergoes high memory load and store miss rates. Modern architectures such as Simultaneous Multithreaded architectures (SMT) aids the use of multiple threads executing the same program. Therefore, special understanding of the underlying architecture should pave the way onto generating more cache-friendly programs. Cache Conscious B+-trees (CSB+-trees) improve the traditional B+ tree by storing the child nodes sequentially. Therefore, only the address of the first child has to be kept in the node, while other child nodes will be accessed implicitly by using the first child address. This improves cache line utilization. While CSB+-tree proves to have significant speedup over B+-trees, experiments show that large fraction of its execution time is still spent waiting for data. SMT allows multiple execution streams to share some resources in one physical processor. Although several papers have studied the CSB+-tree behavior, there have been few papers studying the interaction of multiple threads running CSB+-tree on SMT platform. In this paper we evaluate the CSB+-trees widely used search operation on Intel Pentium 4 Hyperthreaded (dual thread SMT) architecture. Our dual-threaded CSB+-tree search achieves speedup ranging from 29% to 70% compared to a single thread with HT-off architecture. Most of the performance we gained is due to constructive patterns observed between threads at the unified secondary cache level. The rest of the paper is arranged as follows: Section 2 gives a background about B+-tree and CSB+-tree index structure. Section 3 surveys the previous work done to improve CSB+-trees. In Section 4 we propose our multithreaded CSB+-tree. Section 5 explains our experimental methodology. In Section 6 we analyze our results. Finally we conclude in Section 7.

2. BACKGROUND

B+-tree [7] is an index data structure. It consists of a root, internal nodes and leaves. It's designed to manage data efficiently and supports entry retrieval, addition and removal. In a B+-tree, which is a variant of the B-tree, each internal node is of the form <key k, pointer ptr>, where the k directs the search operation towards the next proper node, and ptr points to a child node in the tree. The leaf is of the same structure; given that k is the key for the tuple, and ptr is the tuple pointer. Therefore, the actual data resides on leaves (external nodes) only. All leaves are connected together by forward and backward pointers. If a B+-tree is of x order, then each internal node has between x and 2x+1 keys. A node with y keys has y+1 children. To insert into a B+-tree, a search for the proper leaf to which the new item should be inserted occurs. If the leaf has enough space then the new item is added to it and the insert function terminates. Otherwise, another leaf needs to be allocated and the entries redistributed between the two leaves equally. A copy of the middle key and the new leaf pointer is saved in the parent node. If the parent node is full then it is split using the same technique. To delete an item, usually lazy deletion is used, since other operations (e.g. search) are used more frequently. In lazy deletion a search for the specified entry occurs, and it is de-allocated. No further tree adjustment is needed. In contrast to lazy deletion, other deletion algorithms might require keys redistribution to ensure that each node has at least x (where x is the order of the tree) keys. This can be done by borrowing from a sibling node. Toward making B+-tree more cache conscious for in-memory indexing techniques, Rao and Ross [14] introduced Cache-Sensitive B+-tree (CSB+). In contrast to the B+-tree, each internal node in a CSB+-tree has one pointer to the first child in a group of children nodes. Each node in the

group is of size one cache line (e.g. 128 byte). Thus, keys inside the node are stored physically adjacent in one cache line. The head of each group is found explicitly by referencing its pointer in a parent node, other nodes are visited by offsetting this address. This technique reduces the number of child node pointers in internal nodes. As result, search, insert and delete operations will process using a lower number of cache lines and the tree consumes less memory. Leaf nodes in CSB+ trees are similar to B+-trees.

In SMT architectures [15] multiple logical processors share or duplicate the resources of one physical processor. The efficiency of this technology is largely dependent on the application properties; this is essentially identified by how these programs utilize the available resources. For example, if one application is computation-intensive meaning that it uses execution units heavily, while the other thread is spending most of its time waiting for data from disk storage, then we expect SMT to show high throughput compared to a same environment used by one application only. Intel's implementation of SMT is called Hyper-Threading Technology (HTT) [4]; it allows two threads to issue their instructions simultaneously, sharing mainly L1 and L2 caches, the Trace Cache (TC), the Data Translation Lookaside Buffer (DTLB), and execution units. The Instruction Translation Lookaside Buffer (ITLB) and some other buffers are duplicated.

3. RELATED WORK ON IMPROVING CSB+-TREE

This section provides a survey on the related work that has been made to enhance the performance of cache conscious index structures. Rao and Ross [13] presents a Cache Sensitive Search (CSS), they eliminate all child pointers to effectively increase cache line utilization by storing the tree in an array data structure called directory. Therefore, nodes are accessed by performing computation on array offsets rather than dereferencing child pointer as in B+-trees. As a CSS-tree has to rebuild the whole tree on every insert operation, Rao and Ross in [14] propose CSB+-tree, which is an update-friendly cache conscious B+-tree. For both CSS and CSB+-trees, the authors argue that cache line size is the optimal node size. Whereas R. Hankins et al in [9] show that a CSB+-tree with a node size of 512 bytes and more will be optimal for a machine with 32 byte cache line size. Chen, Gibbons and Mowry in [6] proposed pB+-tree. They rely on creating larger node sizes and arrays of pointers to children nodes to assist prefetching data ahead of its usage. All previous papers present algorithms and memory access methodologies to improve index structure, mainly CSB+-tree, assuming that their code will be executed by a single thread. Current hardware platforms often have more than one processor working in one system. These systems can be in a form of Symmetric Multiprcocessors (SMP), or Simultaneous Multithreading (SMT). Authors in [5] present a latch-free index traversal (OLFIT) concurrency control design to facilitate the execution of multiple search and insert operations running concurrently on an SMP platform. Their results for search operation show good scaling while increasing the number of CPUs. However, we expect SMT platform to have different memory behavior since some vital resources such as L1, L2 and execution units are shared between running threads. J. Zhou et al in [16] use a prefeching thread that works simultaneously with the main thread which executes a staged version from CSB+-tree, they were able to decrease main thread cache misses by transferring them to the helper one. In this research we use SMT environment to allow two threads to access the same memory index data structure simultaneously to perform data retrieval, depending on the fact that multiple data reads will not create any type of data-hazards.

4. MULTI-THREADED CSB+-TREE

A CSB+-tree is designed to force serialized execution of any requested queries. When using the SMT environment, running CSB+-tree queries serially neglects the fact that there are two streams of execution that can be initiated simultaneously to carry out multiple operations. If one thread is used in an SMT enabled platform, resources divided between the two threads will be significantly underutilized. On the other hand, some shared resources such as caches and execution units might be contended when serving two threads, possibly resulting in slowdowns for both. This paper presents a dual-threaded CSB+-tree implementation optimized for SMT architectures. The OpenMP Application Program Interface (API) [1] is designed to assist FORTRAN and C/C+ programmers to parallelize applications in a shared memory environment, by providing a set of compiler directives, functions and environment variables. We use the OpenMP library to initiate two threads to execute queries that arrive to the CSB+-tree. To be fair when comparing any potential improvement SMT would give, we run our experiment on the same machine with SMT enabled and disabled. In this way we ensure that for the case of one thread, all the machine resources will be devoted to process instructions which belong to this thread only, while when enabling SMT, we use our multithreaded version of the CSB+-tree. To implement our dual-threaded version of the CSB+-tree we use the following steps: (1) Since the bulkloading is done only once when building the tree and before any queries appear, therefore one thread is enough to perform bulkloading. (2) We implement simultaneous execution of multiple searches. Similar to B+-tree, searches involve reading keys and computing which route to traverse until the target leave is reached, then the tuple pointer is dereferenced. Multiple concurrent reads of the tree do not generate hazards of any kind. For inserts, first we have to locate the appropriate node for the new entry in the tree, which is achieved by a search operation, if this node has space then the new key is added, otherwise the node is split as described earlier. This means that a new node might be allocated and some keys will be moved during an insert, so the tree will appear in a non-stable condition to the other

thread. Unless some synchronization directives are used, we don't carry out multiple inserts at the same time. For deletes, CSB+-tree uses a lazy deletion technique, where the target key is found then removed. Previous research [8] shows that semaphores have a positive effect on performance when used to limit memory access to one thread at a time. In this research we illustrate CSB+-tree simultaneous search operations, we used basic search approach [14]. Basic search implemented using a while loop to perform a binary search [12]. Rao and Ross analyze other search techniques depending on code expansion. We concentrate on the basic approach, since it's widely used in index searches due to its simplicity and small code size.

5. EXPERIMENTAL METHODOLOGY

We conduct our experiments on a 3.4GHz Pentium 4 processor with HyperThreading technology (HT). Our Pentium 4 processor has a 64KB on-chip data cache, a 12K micro-operation instruction trace cache, and a unified 2MB secondary cache with 128 byte cache lines. Our machine has a 1GB 533MHz DDR2 SDRAM memory, an 800MHz front side bus, and a 160GB SATA, 7200 RPM hard disk drive. All our experiments fit in main-memory. We use the Scientific Linux version 4.1 operating system which is based on the Redhat Linux Enterprise version 4.0. We implement CSB+-tree bulkload and search operations in C and use the OpenMP API. We compile our code using the Intel C++ Compiler for Linux version 9.1 [2] with full optimizations. The Intel compiler supports OpenMP C/C++ [1] version 2.5. Our bulkload implementation is similar to [14], where the tree is filled level by level. The keys are generated randomly from 1 to 10 million. The node size is 128 bytes (one cache line). Each internal node has 30 keys, the number of keys used, and one pointer to the first child node in a group that has a maximum of 31 children. A leaf node contains a maximum of 14 pairs of <tuple pointer, key>, the number of items in the current leaf, and backward and forward pointers. All keys, pointers, and tuple identifiers are 4 bytes each. We use the VTune Performance Analyzer 3.0 for Linux [3] to collect our events using the performance counters available in the Pentium 4 processor. We repeat every run five times, remove the outliers and take the average. Timing measurements are done through our CSB+-tree program and use wall-clock time. We switch hyperthreading on and off using the system BIOS.

6. RESULTS

To compare our dual-threaded CSB+-search operation against the original single-threaded version, we perform experiments similar to those done by [14]. First we bulkload the CSB+-tree with a number of items (records) ranging from 100 and up to 10^7, then we run 200,000 searches; 100,000 on each thread. Figure 1 shows that the proposed dual-threaded CSB+-tree has a speedup ranging from 29% for a 100-item tree and up to 70% for a 1000-item tree. For 10^7 entries, we obtained a speedup of approx. 67%.

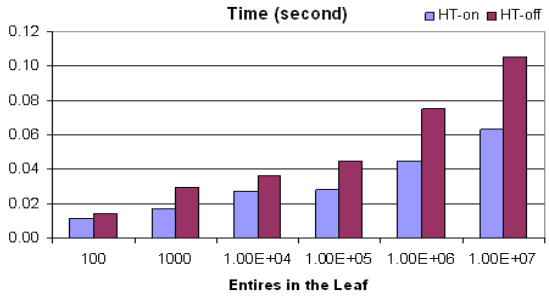

Figure 1: CSB+-tree search timing.

Figure 2 shows mostly a constructive effect for the 2MB unified L2 cache load miss rate. For trees bulkloaded with items less than 10^6, the space used is less than 2MB (refer to [14] for memory space equations), so most loads generated by the search function hit either in the L1 or the L2 cache. While for trees with large number of entries (>10^6), we can see that having two threads working on the same dataset (CSB+-tree in our case) will result in one thread pre-fetching data for the other, this agrees with results in [10] showing constructive behavior at L2 cache level.

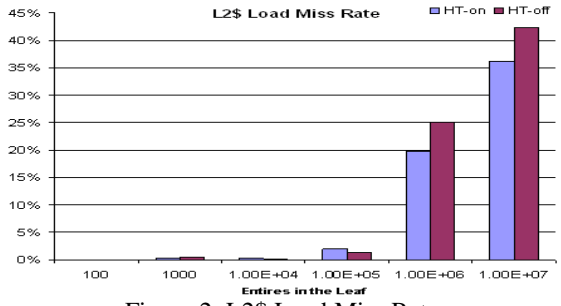

Figure 2: L2$ Load Miss Rate

Figure 3 shows the L1 data cache load miss rate. Using two threads gives a slightly worse results than a single thread version for large datasets. We expect that the 64KB L1 data cache is too small to store data fetched by thread one until the data used by thread two. Trees with items less than 10^3 are using memory space less than 64KB. Therefore, they are exhibiting very small L1 data cache miss rates. The Trace Cache (TC), used as the L1 instruction cache, stores decoded instructions. Figure 4 shows the trace cache miss rates. Although the TC is a shared resource, access to the TC is granted to one thread at a time in a clock-cycle alternating fashion [4]. This fact together with the destructive behavior of the two threads create a significant increase in the TC miss rates for the two-thread version of the search operation. However, the absolute values of the TC miss rates are small. Temporal locality plays a good role in reducing these effects for large trees.

The Pentium 4 Processor has a 64 fully associative Data Translation Lookaside Buffer (DTLB); it's used to translate virtual memory addresses to physical addresses. It is also a shared component in the memory subsystem [4]. Figure 5 shows an increase in the DTLB miss rates using dual

threads while using items more than 10^4. Figure 6 shows the Instruction Translation Lookaside Buffer (ITLB) miss rates. The ITLB is visited on a TC miss. Our results show that the two-thread version is experiencing low ITLB miss rates since each thread has its own ITLB component.

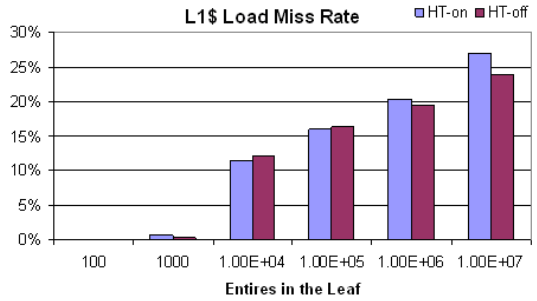

Figure 3: L1$ Load Miss Rate

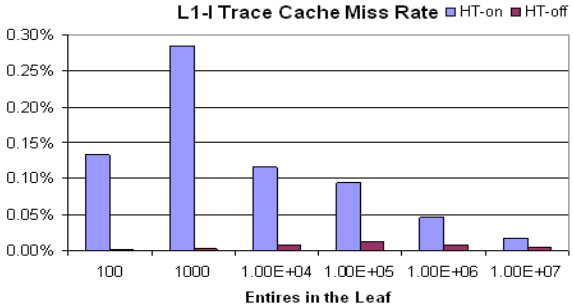

Figure 4: L1-I Trace Cache Miss Rate

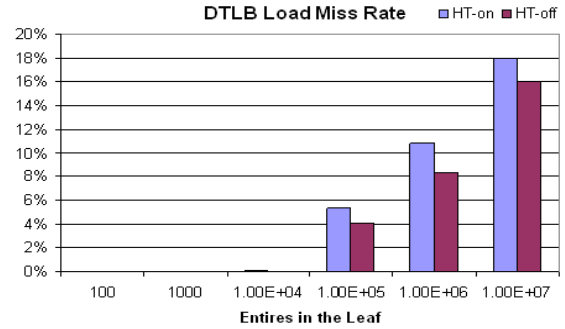

Figure 5: DTLB Load Miss Rate

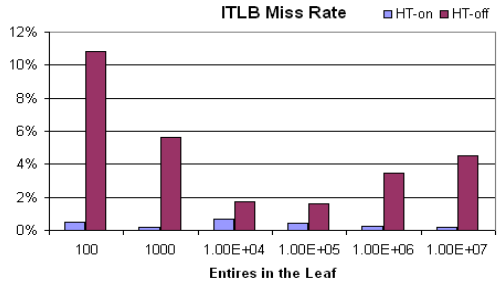

Figure 6: ITLB Miss Rate

In summary, we analyzed the behavior of a dual-threaded search operation on CSB+-tree using HT technology. We concentrate on the memory system activities since CSB+-tree spends it's time mainly loading and storing data [14]. We find that HT slightly decreases the performance of L1 data cache and DTLB. Our results show a larger performance degradation for the TC. However, the thread constructive pattern in the L2 cache (saving in L2 load miss rate is up to 26% for 10^6 items) of the dual thread CSB+-tree creates a speedup from 29% to 70% over the single threaded version.

7. CONCLUSION

In this paper we propose a parallelized version of CSB+-tree for the search operation, where two threads share the same memory index structure and retrieve data in parallel. We compare our dual-threaded CSB+-tree while HT is enabled to a single thread version from CSB+-tree while HT is disabled for the same architecture. Our results show a constructive behavior between the two threads at the L2 cache level and ITLB, and destructive pattern in Trace Cache and DTLB. The L2 and ITLB constructive behavior is able to outweigh other negative effects and result in a speedup from 29% to 70%.

REFERENCES

[1] http://www.openmp.org/drupal/

[2]http://www.intel.com/cd/software/products/asmo-na/eng/compilers/277618.htm

[3] http://www.intel.com/software/products/vtune/.

[4]http://www.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf

[5] S. Cha, S. Hwang, K. Kim, K. Kwon. "Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems". In Proceedings of Very Large Data Base (VLDB), 2001.

[6] S. Chen, P. Gibbons, T. Mowry. "Improving Index Performance through Prefetching". In Proceedings of Special Interest Group on Management of Data (SIGMOD), 2001.

[7] D. Comer. The ubiquitous B-tree. ACM Computing Surveys, 11(2), 1979.

[8] P. Garcia, H. Korth. "Multithreaded Architectures and The Sort Benchmark". First International Workshop on the Data Management on New Hardware (DaMoN) ,2005.

[9] R. Hankins, J. Patel. "Effect of Node Size on the Performance of Cache Conscious B+trees". In Proceedings of Special Interest Group On Management of Data (SIGMOD), 2003.

[10] W. Hassanein, M. Hammad, L. Rashid. "Characterizing the Performance of Data Management Systems on Hyper-Threaded Architectures". Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) ,2006.

[12] D. Knuth. "The Art of Computer Programming". Volume 3: Sorting and Searching, Third Edition. Addison-Wesley, 1997.

[13] J. Rao and K. Ross. "Cache Conscious Indexing for Decision-Support in Main Memory". In Proceedings of the Very Large Data Base (VLDB),1999.

[14] J. Rao and K. Ross. "Making B+-trees Cache Conscious in Main Memory". In Proceedings of Special Interest Group on Management of Data (SIGMOD), 2000.

[15] D. Tullsen, S. Eggers, H. Levy, "Simultaneous Multithreading: Maximizing on-Chip Parallelism", In Proceedings of the 22nd Annual International Symposium on Computer Architecture, (ISCA),1995.

[16] J. Zhou, J. Cieslewicz, K.Ross, M. Shah "Improving Database Performance on Simultaneous Multithreading Processors". In Proceedings of Very Large Data Base (VLDB), 2005.