

Improving Database Performance on Simultaneous Multithreading Processors [Extended Abstract]

John Cieslewicz*
johnc@cs.columbia.edu
Columbia University

Jingren Zhou
jrzhou@microsoft.com
Microsoft Research

Kenneth A. Ross†
kar@cs.columbia.edu
Columbia University

Mihir Shah
msshah@gmail.com
Columbia University

1 Introduction

Simultaneous multithreading (SMT) allows multiple threads to supply instructions to the instruction pipeline of a superscalar processor. Because threads share processor resources, an SMT system is inherently different from a multiprocessor system and, therefore, utilizing multiple threads on an SMT processor creates new challenges for database implementers.

We investigate three thread-based techniques to exploit SMT architectures on memory-resident data. First, we consider running independent operations in separate threads, a technique applied to conventional multiprocessor systems. Second, we describe a novel implementation strategy in which individual operators are implemented in a multi-threaded fashion. Finally, we introduce a new data-structure called a work-ahead set that allows us to use one of the threads to aggressively preload data into the cache for use by the other thread. This work originally appeared in VLDB 2005 [2].

We evaluate each method with respect to its performance, implementation complexity, and other measures. We also provide guidance regarding when and how to best utilize the various threading techniques. Our experimental results show that by taking advantage of SMT technology we achieve a 30% to 70% improvement in throughput over single threaded implementations on in-memory database operations.

2 The Work-ahead Set

A work-ahead set (Figure 1) is created for communication between the main thread and the helper thread. The work-ahead set is a collection of pairs (p, s) , where p is a memory address and s is state information representing the state of the main thread's computation for the data at that memory location. The size of the work-ahead set is the number of pointers it contains. The main thread has one method, `post`, to manipulate the work-ahead set. The helper thread has a single method, `read`, for accessing the work-ahead set.

Because the work-ahead set needs to be particularly efficient and simple, we implement it as a fixed-length circular array. The main thread posts a memory address to the work-ahead set when it anticipates accessing data at that memory location. This address, together with state information for the main thread, is written to the next slot

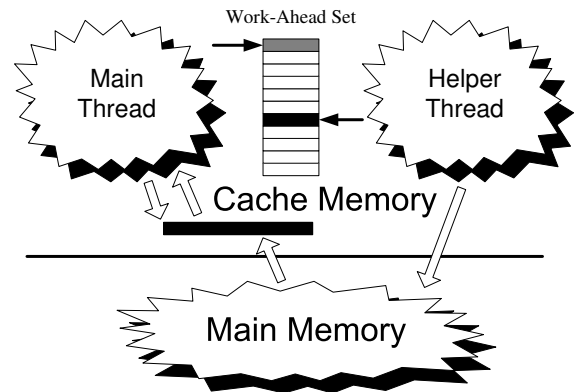


Figure 1. The Work-ahead Set

of the array. If that slot already contained data, that data is returned to the main thread by the `post` operation. The returned data is then be used by the main thread to continue processing. Thus, the main thread cycles through the work-ahead set in a round-robin fashion. At the end of the computation, the main thread may need to `post` dummy values to the work-ahead set in order to retrieve the remaining entries.

During the period between a data item's posting to the work-ahead set and its retrieval by a subsequent `post` operation, we desire (but do not require) that the helper thread loads the data at the given memory location into the cache. The helper thread has its own offset into the array and accesses the array in sequential fashion without explicitly coordinating with the main thread.

The helper thread executes a read on the work-ahead set to get the next address to be loaded. In practice, we implement the load in C using something like `temp += *((int*)p);` rather than using assembly language. The value of `temp` is unimportant; the statement forces the cache line at address `p` to be loaded. Note that the helper thread does not need to know anything about the meaning (or data type) of the memory reference. After loading the data, the helper offset is moved to the next entry of the array.

The design of the work-ahead set and helper thread is generic and can be used in combination with any database operation. This is a key advantage. If each operation needed its own helper thread that had some special knowledge about its memory access behavior, then the implementation would be much more complex. The helper thread operates without tuning to a particular workload.

* Funding provided by a U.S. Department of Homeland Security Fellowship administered by Oak Ridge Institute for Science and Education.

† This research was supported by NSF Grants IIS-0120939 and IIS-0121239.

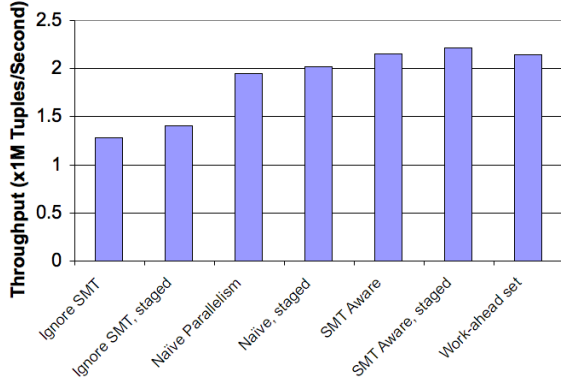


Figure 2. Throughput of a CSB⁺-tree Index Join. Note that SMT aware methods perform much better than naïve methods.

3 Experimental Results

In our experiments we explored the performance of three strategies for using SMT: naïve parallelism where each thread was treated as if it were a separate core, SMT aware parallelism where the input was partitioned and cooperatively processed by the threads, and a work-ahead set implementation that uses a thread to explicitly preload data into the cache. We found that by exploiting SMT technology intelligently, we could improve the performance of core database operations by up to 70% over a single threaded implementation.

Our experiments were conducted on a Pentium 4 with Hyperthreading, Intel’s proprietary name for SMT technology. The Pentium 4 has two hardware thread contexts, so all of our experiments used two threads. Figures 2 and 3 show some of our experimental results. These figures show the throughput and L2 cache misses, respectively, for a tree index traversal using a CSB⁺ Tree [1], which is an important database operation used to find tuples that match a specified key value.

Figure 3 demonstrates that intelligent use of SMT technology results in higher performance. This figure highlights the importance of high latency cache misses on performance. In naïve parallelism, where there is no cooperation among the threads (see the “two

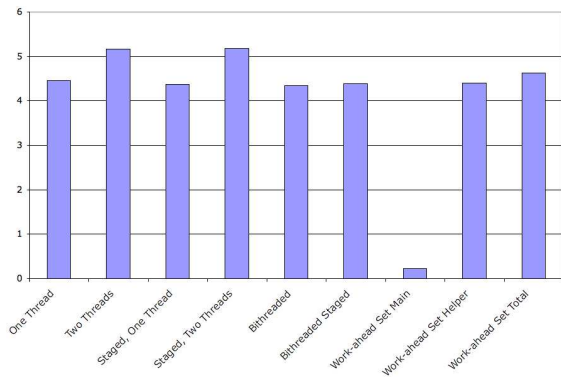


Figure 3. L-2 Cache Misses per probe tuple processed during a CSB⁺-tree Index Join. In the case of the work-ahead set implementation, the helper thread incurs most of cache misses and the main thread rarely misses in the cache.

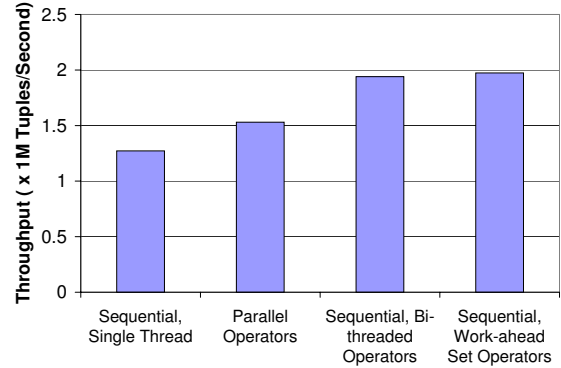


Figure 4. Throughput of a heterogeneous workload. The SMT aware techniques, bithreading and work-ahead set, perform much better than naïve parallelism.

threads” data point), the cache misses are higher because of contention in the cache. The “bithreaded” implementation has fewer cache misses because the threads cooperatively process the input. Finally, the work-ahead set implementation is shown to effectively hide cache misses from the main thread by having the helper thread perform explicit preloading. These results show that an intelligent use of SMT technology can effectively overlap computation with high latency cache misses in core database operations.

In the case of heterogeneous workloads, the importance of using SMT technology intelligently is even more important, as shown in Figure 4. In these experiments both an index join and a hash join were performed. The leftmost datapoint is a baseline and shows the performance of both joins executed sequentially by one thread. The second data point represents the throughput of the two joins executed in parallel, each using one of the Pentium 4’s thread contexts. The rightmost two data points represent bithreaded and work-ahead set implementations, respectively. In these two experiments the index join and hash join are multithreaded and these multithreaded implementations are run sequentially. In all experiments, the amount of work performed is the same. The throughput is much higher for the SMT aware implementations because they suffer less contention for resources, such as the cache, than the naïve parallel implementation and they effectively use thread level parallelism to overlap computation with high latency memory accesses.

4 References

- [1] J. Rao and K. A. Ross. Making B+ trees cache conscious in main memory. In *Proceedings of SIGMOD Conference*, 2000.
- [2] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. Improving database performance on simultaneous multithreading processors. In *Proceeding of VLDB Conference*, 2005.