Tech Report CUCS-017-05

Improving Database Performance on Simultaneous Multithreading Processors

Jingren Zhou
Microsoft Research
jrzhou@microsoft.com

John Cieslewicz
Columbia University
johnc@cs.columbia.edu

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

Mihir Shah
Columbia University
ms2064@cs.columbia.edu

Abstract

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.

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.

1 Introduction

Simultaneous multithreading (SMT) improves CPU performance by supporting thread-level parallelism on a single superscalar processor [24]. An SMT processor pretends to be multiple *logical* processors. From the perspective of applications running on an SMT system, there appear to be multiple processors. Multiple threads may issue instructions on each cycle and simultaneously share processor resources. The resulting higher instruction throughput and program speedup are beneficial for a variety of workloads, including web servers and multimedia applications.

SMT technology has been incorporated into CPU designs by most CPU vendors. Intel's version of SMT, called "Hyper-Threading Technology," has been commercially available since 2002 [18]. SMT is supported in Intel's Xeon and Pentium 4 processors, in IBM's latest POWER5 processors [12], and in Sun Microsystems' forthcoming processors [22].

An SMT system is different from a shared-memory multiprocessor system because the different threads share many of the execution resources, including the memory bus and caches. Thus the performance of an SMT system is intrinsically lower than that of a system with two physical CPUs. (Threads that compete with each other for a shared resource may even show a *decrease* in performance relative to a single thread [14].) On the other hand, an SMT system performs better than a single-threaded processor because it can do some work in parallel, and because it can be doing useful work even when one of the threads is blocked waiting for a cache miss to be resolved.

Cache misses are an increasingly important issue because advances in the speed of commodity CPUs far outpace advances in memory latency. Main memory access is a performance bottleneck for many computer applications, including database systems [1, 5]. In response to the growing memory latency problem, the database community has explored a variety of techniques to perform database operations in a cache-conscious way [20, 21, 8, 15, 4, 9, 28, 7, 29].

In this paper, we investigate three thread-based techniques to exploit SMT architectures for databases operating on memory-resident data. In all three cases, we force the operating system to schedule two threads on different logical processors; this scheduling is straightforward in modern operating systems.

The simplest option is to think of logical processors as real physical processors and treat an SMT system as a multiprocessor system. Independent operations are

performed by separate threads and are scheduled to different logical processors in parallel. This approach requires minimal code changes for database systems that already run on multiprocessors. However, such an approach ignores the sharing of resources between the logical processors. Complex operations running simultaneously may interfere, leading to a less-than-ideal utilization of the processor.

The second approach we consider is to implement each database operator in a multi-threaded fashion. The workload is partitioned and processed by different threads that cooperatively *share* input and output data in the cache. While this option requires re-implementing database operations, partitioning the workload is relatively easy to implement in a way that minimizes synchronization between the two threads. The main challenge of this approach is deciding how to partition the workload, and how to merge the results from different threads.

We also propose a new, general thread-based *preloading* technique for SMT processors. We utilize one thread, the *helper thread* to perform aggressive data preloading. The main computation is executed in the other thread, which we call the *main thread*. The helper thread works ahead of the main thread, triggering memory accesses for data elements that will soon be needed in the main thread. For reasons discussed in Section 2.3, the helper thread performs an explicit load instruction, and not a prefetch instruction.

If the preloading happens in an optimal manner, the main thread will almost always find the requested data in the cache and experience a higher cache hit rate. The data preloading performed by the helper thread can be overlapped with the CPU computation done by the main thread. The helper thread suffers most of the memory latency, while the main thread is free to work on the real computation.

Several aspects of simultaneous multithreading make the design of this two-thread solution challenging.

1. *Communication between the threads is expensive.* In particular, if two threads access the same cache line, and at least one of these accesses is a write, then the hardware needs to do additional work to ensure the consistent and correct execution of the two threads. On an Intel machine, such conflicting accesses trigger a "Memory Order Machine Clear" (MOMC) event that can be relatively time-consuming. As a result, we wish to design the threads in such a way that (a) direct synchronization is avoided, and (b) when the two threads do communicate, they do not access common cache lines at the same time.

2. *The speed of the helper thread is hard to predict.* The helper thread may be slower or faster than the main thread, and the relative speed of the two threads may depend on run-time factors that are difficult to measure. As a result, we must implement the main thread in such a way that it does not depend on the helper thread for correctness. The main thread should not block for the helper thread, and it should make progress even if the helper thread has not prefetched a memory reference. Further, we should implement the helper thread in a way that is robust with respect to speed, so that it does not need to adjust its behavior depending on the relative thread speed.

3. *Boundary conditions must be checked.* If the helper thread gets too far ahead, it may pollute the cache by loading so much data that other preloaded data is evicted from the cache before it is utilized by the main thread. Conversely, if the main thread catches up to the helper thread and we do not change the behavior of the helper thread, then the helper thread provides no benefit as it is no longer ahead of the main thread. Further, the helper thread may even slow down the main thread for the reasons mentioned in item 1 above, since the two threads may be operating on common data elements.

4. *The helper thread must be simple.* Because both threads issue instructions to the same pipeline, a complex helper thread may compete with the main thread for computation units, and slow down the execution of the main thread.

We define a data structure called the *work-ahead set* that is shared between the main thread and the helper thread. The main thread is responsible for inserting soon-to-be-needed memory references (together with a small amount of state information) into the work-ahead set. The helper thread accesses the work-ahead set and loads the referenced data into the cache. Since the helper thread exclusively performs read-only operations, the main thread is guaranteed to achieve the correct result, regardless of the status of the helper thread. This design separates the work of data preloading from the work of CPU computation.

To benefit from the work-ahead set, the main thread does need to be implemented to access the data in stages. For example, in a hash join one would separate the probe of the hash table and the construction of the resulting output tuple(s) into different stages [7]. The main thread submits the address m of the probed hash bucket to the work-ahead set, but does not immediately try to access the probed bucket. Instead, it proceeds with other work, such as computing hash functions, submitting additional probes, and/or accessing "older" hash buckets. At a later time, the main thread removes the entry for m from the work-ahead set, and continues work on the contents of the hash bucket. The order of work is determined by the order in which

memory accesses are submitted to and retrieved from the work-ahead set.

A staged implementation may be more involved than a direct implementation. We argue that the additional code complexity is moderate, and worth the investment when there are measurable benefits. Restructuring of data accesses to reduce the impact of latency bottlenecks is a commonly used technique; double-buffering is one example. Database query operators tend to do similar, relatively simple operations on large numbers of records. As a result, they can be pipelined relatively easily with a small amount of state per pipeline stage. If every record goes through the same number of stages then the original record order can also be preserved, although order preservation is not required for many database operations.

We apply our proposed techniques to several data-intensive database operations, such as may be used for decision support queries, and experimentally compare the performance. We study the performance of CSB$^+$ tree index traversal and hash join operations for each of our proposed methods. In addition, we provide a detailed analysis of factors affecting the performance of the work-ahead set.

The rest of this paper is organized as follows. In Section 2, we provide an overview of SMT technology as well as survey related work in both the compiler and database research communities. Section 3 presents a new technique to exploit thread-level parallelism by partitioning work. Section 4 presents the *work-ahead set*, our general technique for aggressive data preloading on SMT processors. The database operations used to evaluate our approach are described in Section 5. We then present our experimental results in Section 6 and conclude in Section 7.

2 Related Work

2.1 Simultaneous Multithreading Technology

Lo, et al. simulate an SMT processor with multiple hardware contexts using traces from database workloads [16] and show that SMT can be useful for databases. They do not, however, consider ways of reimplementing database operations.

Our implementation and experimental study uses Intel's Hyper-Threading technology on a Pentium 4 machine. Details of Intel's Hyper-Threading Technology can be found in [18, 3].

An important issue for programming on a Hyper-Threading system is cache coherence. Similar to a conventional parallel system, which must ensure data consistency across multiple caches, a Hyper-Threading system must ensure data consistency across the pipeline. For example, when one thread modifies a shared variable, the cache line becomes "dirty" and must be written out to RAM before the other logical processor accesses it. The hardware needs to make sure the correct memory order is maintained. Due to potential out-of-order execution, modified shared variables can lead to accessing memory in an incorrect order. If that happens, the entire pipeline of the machine has to be cleared. Excessive pipeline flushing can be a significant overhead and may outweigh any potential gain from parallelism. In a Pentium 4 processor, these pipeline flushing events are called "Memory Order Machine Clear" (MOMC) events.

It is important to share a minimum amount of information between logical processors and reduce any *false sharing*, in which different threads modify different variables that reside on the same cache line.

2.2 Speculative Precomputation

Recently, several thread-based prefetching paradigms have been proposed. "Speculative Precomputation" targets a small set of static delinquent loads that incur the most cache miss penalties [10, 27, 26, 14]. A post-pass binary adaptation tool is used to analyze an existing application binary and identify the dependent instruction slice leading to each delinquent load. Helper threads are created to speculatively precompute the load address and perform the data load. Finally, an augmented new application binary is formed by attaching the helper threads to the original binary. Hardware-based speculative precomputation has also been explored [19, 23].

What distinguishes our work-ahead set technique from speculative precomputation is that helper threads in speculative precomputation execute a *subset* of the original program. We, in contrast, do not require helper threads to have the same access pattern as the original program. In speculative precomputation, the subset of the program executed by helper threads skips the expensive computation and only includes delinquent loads. However, if the original program does not have any expensive computation, both the main thread and the helper thread execute the same program (instructions). Also, if the original program has a high degree of data dependency and future memory references depend on current computation, naive speculative precomputation would not work.

2.3 Software Prefetching

Modern processors also provide prefetch and cacheability instructions. Prefetch instructions allow a program to request that data be brought into the cache without blocking. One would typically execute prefetch instructions prior to the actual use of the data. Several software prefetching techniques have been proposed for database systems, including prefetching B$^+$-Trees [8], fractal prefetching B$^+$-Trees [9], and prefetching hash join [7]. These techniques require manual computation of prefetching scheduling distance and manual insertion of explicit prefetch instructions into the code.

The process is cumbersome and hard to tune. Furthermore, on commercially available processors such as the Pentium 4, prefetch instructions are not guaranteed to actually perform the data prefetch. Under various conditions, such as when prefetching would incur a TLB miss, prefetch instructions are dropped [13]. Prefetch instructions are not completely free in terms of bus cycles, machine cycles, and resources. Excessive usage of prefetch instructions can also worsen application performance [13].

Our work-ahead set technique employs preloading rather than prefetching. The data is guaranteed to be loaded into the cache, even if a TLB miss occurs.

2.4 Staged Execution

Our work-ahead set technique requires implementing an operator in a staged fashion. Harizopoulos et al. [11] show that staged operators can improve pipeline behavior for OLTP workloads. Their work does not focus on SMT processors.

We restructure the hash join to use staged memory accesses in a manner similar to the techniques described in [7]. However, [7] evaluates the performance of hash join on a simulated architecture in which prefetch instructions are never dropped and large numbers of simultaneous outstanding memory requests are allowed. In contrast, our implementation achieves significant performance improvements on a real commodity CPU that does not conform to the simulation assumptions of [7].

2.5 Cache Conscious Index Structures

Recent studies have shown that cache-conscious indexes such as CSS-Trees [20] and CSB$^+$-Trees [21] outperform conventional main memory indexes such as B$^+$-Tress. By eliminating all (or most) of the child pointers from an index node, one can increase the fanout of the tree. Multidimensional cache-conscious indexes such as CR-Trees [15] use compression techniques to pack more entries into a node. These techniques effectively reduce the tree height and improve the cache behavior of the index. Bohannon et al. reduce cache misses by storing partial key information in the index [4]. Zhou and Ross propose buffering memory accesses to index structures to avoid cache thrashing between accesses for bulk access patterns [28].

Unlike most of these methods, the techniques proposed in this paper do not attempt to reduce the total number of cache misses. Instead, they try to distribute the misses among the threads so that the cache miss latency can be overlapped with other work.

Chen, et al. show improvements in simulated index performance by using prefetch instructions in the traversal code [8, 9]. As for the discussion of hash join in Section 2.3, this technique could be implemented by preloading data in the helper thread. One does not need to make assumptions about whether software prefetch instructions can be dropped.

3 Multithreading By Partitioning

Running one operation at a time on an SMT processor might be beneficial in terms of data and instruction cache performance. In order to take advantage of two logical threads, we implement each operator in a two-threaded[1] fashion. One thread processes the even tuples on the (probe) input stream, and the other processes the odd tuples. Because access to the input is read-only, there is no performance degradation when they read the same cache lines. Similarly, read-only access to a shared hash table or index does not cause contention. An addition benefit of this kind of approach is that both threads use the same instructions, meaning that there is less contention for the instruction cache.

The difficulty with a bi-threaded implementation is handling the output. If the two threads were to write to a single output stream, they would frequently experience write-write contention on common cache lines. This contention is expensive and negates any potential benefits of multithreading. Instead, we implement the two threads with separate output buffers, so that they can write to disjoint locations in memory.

The use of two output buffers changes the data format, and places extra burden on subsequent operators to merge the data. Further, this kind of implementation no longer preserves the order of input records, and cannot be used if the order must be maintained. Thus, it is not totally fair to compare the performance of such an operation with one that uses a standard data format. Nevertheless, we will measure the performance with the understanding that there may be additional costs not captured by those measures.

4 The Work-ahead Set

A work-ahead set 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,

[1]We also tried implementing operators using more than two threads, but those implementations did not perform as well.

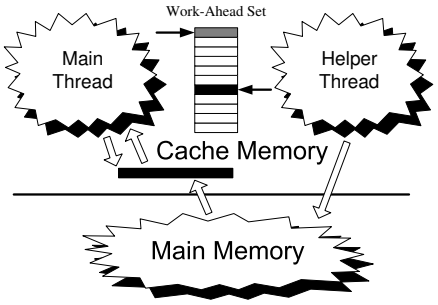


Figure 1: Work-Ahead Set

is written to the next slot of the array. If that slot already contained data, that data is returned to the main thread by the *post* operation. The returned data can 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 may proceed in the same direction as the main thread ("forward"), or in the opposite direction from the main thread ("backward"); we will compare these two alternatives later.

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 operation of the work-ahead set is illustrated in Figure 1.

If the helper thread runs faster than the main thread, the helper thread may load the same data multiple times, potentially wasting resources. Thus we implement a spin-loop wait mechanism in the helper thread. The helper thread keeps a local copy of the last entry processed in each slot of the work-ahead set. If the entry read is different from the local copy, the helper thread executes a load on the memory address as before. We put the helper thread into a spin-loop checking the current entry of the work-ahead set, until a different entry, written by the main thread, is found. We will experimentally evaluate the costs and benefits of spin-loop waiting.

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 any special need for tuning to a particular workload.

There are four main configuration parameters that will affect the performance of the system. They are whether to put multiple pointers into each entry of the work-ahead set, the size of the work-ahead set, whether the helper thread moves forward or backward, and whether the helper thread performs a spin-loop wait when it sees the same memory address a second time in a given slot. We discuss these choices below, and evaluate them experimentally in Section 6.

4.1 Entry Structure

Our discussion so far has assumed that an entry in the work-ahead set is a pointer plus some state. However, there are cases where one might need multiple memory references to execute an operation. The most obvious case is when a data type is longer than a cache line. For a second example, consider a `C` instruction like `*p = *q + *r;`. One statement needs three memory references for execution, and may, in the worst case, generate three cache misses.

It is possible to divide such a statement into three stages, in which `p`, `q`, and `r` are preloaded by the helper thread one by one. However, by the time the operation is executed in the main thread, a relatively long time has passed since the first elements were preloaded, and they may have since been evicted from the cache. We observed such an effect experimentally.

As an alternative, one could alter the structure of an entry of the work-ahead set to allow three pointers plus state. The helper thread would then preload three memory references at a time. The work-ahead set would have a factor of three fewer entries than the single-pointer version in order to cover the same fraction of the cache in a single loop of the array. When the main thread retrieves that entry from the work-ahead set, it can expect that all three memory locations have been touched by the helper thread.

When the number of references per stage is constant throughout the operation, this approach is adequate. However, if one stage needs n_1 references, and another needs $n_2 \neq n_1$, the work-ahead set needs some way to balance them. (It is also possible that two entries in a single stage require different numbers of references.) In this case, entries with different numbers of references are present at the same time in the work-ahead set.

Suppose that an operator has a minimum number m of references per stage, and a maximum number M. The number of entries in the work-ahead set is determined by dividing the fraction of the cache we intend to address by m. That way, even if all entries refer to only m addresses, we can use the available cache mem-

ory. Each entry in the work-ahead set has M slots for memory references. If $k < M$ of the slots are actually used, then the remaining slots are padded with a single valid address to a dummy cache line that is never touched by the main thread. The helper thread always preloads all M slots. Although this might seem to be extra work, the helper thread does not have to perform /conditional tests and suffer branch misprediction penalties. Further, since the dummy cache line will be accessed often, it will almost always yield an L1 cache hit, and generate little overhead in the helper thread.

To maintain constant usage of cache memory, we keep track of the total number N of valid pointers in the work-ahead set. N is incrementally maintained by the post method each time entries are added to or removed from the work-ahead set. Suppose our memory threshold is t cache lines.[2] If $N \geq t$, then we should not admit new work from the main thread, even if there is an available slot in the work-ahead set. In this case, we fill an available slot with (M copies of) the dummy cache line address, and null state information.

4.2 The Size of the Work-Ahead Set

If the work-ahead set is too small, then the performance may suffer for two reasons. First, there may not be enough time for the helper thread to load the requested cache line between when the main thread posts the memory reference and when it retrieves that reference. Second, when the work-ahead set is small, the probability that the main thread and the helper thread are accessing the same cache line within the work-ahead set itself becomes high. As discussed previously, this kind of access will lead to a relatively high number of expensive MOMC events.

If the work-ahead set is too large, then the loads performed by the helper thread may evict still-useful data from the cache. This effect will reduce the benefits of preloading, and could lead to *more* cache misses than a single-threaded implementation. For this reason, the expected amount of preloaded data for the entire work-ahead set should be smaller than the L2 cache size. Since the main thread may have older cache-resident data that is still being used, and in order to avoid conflict misses, the threshold should be lower, such as one quarter of the cache size.

An additional advantage of the work-ahead set is that it allows global coordination of the use of the data cache. By sizing the work-ahead set for each active operator appropriately, one can ensure that the total active memory is smaller than the cache capacity. In this way, we can avoid cache interference both between operators in a single plan, and between operators in concurrently executing queries. Without such control, it is possible for concurrently executing operators to expel each others' cache lines, leading to thrashing behavior between the operators' scheduled time-slices.

[2]We consider t to be the appropriate measure of the "size" of the work-ahead set, since it represents the number of valid pointers in the work-ahead set.

4.3 Forwards Versus Backwards

Forward processing by the helper thread seems intuitive: the helper thread "paves the way" for the main thread. However, there are some pathological behaviors that can arise with forward processing.

First, suppose that the helper thread is faster than the main thread. The helper thread eventually succeeds at preloading the entire work-ahead set, and catches up to the main thread. At this point, the helper thread and the main thread are accessing the same slot in the work-ahead set, and will suffer MOMC events as a result. If the helper thread goes into a spin-loop, it will do so for a short time, until the main thread advances one entry. The helper thread will wake up, make progress of one entry, and then go back to spin-looping. For all of this time, the helper thread and main thread are almost certainly accessing the same cache line, and will therefore encounter a kind of MOMC thrashing.

If the helper thread does not spin-loop, then the situation may not be quite as bad, since the helper thread may eventually overtake the main thread and be accessing different cache lines in the work-ahead set. Nevertheless, the overtaking period itself may dominate, because the MOMC events generated will slow it down dramatically.

If the helper thread is slower than the main thread, then the main thread will eventually catch up to the helper thread. The threads will then interfere in a similar way as before. What is more, the main thread may be faster than the helper thread because it gets cache hits rather than cache misses. Once the main thread catches up to the helper thread, it will begin to experience cache misses and slow down. As a result, the main thread and helper thread may find themselves operating pathologically in lock-step.

Backward processing may seem less intuitive than forward processing, because data needed in the far future is preloaded before data needed in the near future. Nevertheless, the pathological behaviors described above are avoided.

We can mathematically model the difference between forward and backward processing. Suppose that the speed of the main thread (measured in cache lines per second) is m without MOMC interference, but m' with MOMC interference. Let the corresponding numbers be h and h' for the helper thread. Suppose that the size of the work-ahead set is d cache lines. A straightforward analysis shows that the net speed of the main thread when the helper thread is running

forwards is

$$s_f = \left(\frac{m'}{h'-m'} + \frac{m(d-1)}{h-m}\right) / \left(\frac{1}{h'-m'} + \frac{(d-1)}{h-m}\right)$$

while the net speed running backwards is

$$s_b = \left(\frac{m'}{h'+m'} + \frac{m(d-1)}{h+m}\right) / \left(\frac{1}{h'+m'} + \frac{(d-1)}{h+m}\right).$$

If h' is close to m', which is likely when the MOMC event cost is dominant, then the $\frac{m'}{h'-m'}$ term in s_f is large, and drives s_f close to m'. In the backwards case, however, there is no such "catastrophic cancellation" since the corresponding term in s_b adds h' and m'.

Note that the helper thread does not know h' or m', unless it decides to devote extra cycles to measuring the relative rates of progress. These rates may change over time. As a result, the safest choice is to preload backwards, because we do not even need to check h' and m', and since the performance of the main thread for large d is typically close to m.

4.4 Spin-Loop Waiting

When the helper thread is in a spin-loop wait, it uses fewer resources that it would if it continued preloading data that was already memory-resident. Thus we might expect less interference with the main thread in terms of competition for CPU resources. On the other hand, as we have discussed in Section 4.3 above, combining a spin-loop wait with forward preloading could cause more MOMC thrashing than would be experienced without the spin-loop wait.

One could augment the spin-loop wait with a call to a `sleep` function so that the helper thread checks the slot in the work-ahead set less often, thus using fewer resources. However, the amount of time to sleep would be a parameter that would need to be tuned, and the benefits of even a well-tuned choice would be relatively small. Further, a `sleep` function suspends a thread until the operating system chooses to reschedule it, which could be excessively disruptive. Our implementation of efficient spin-loop waiting uses a different technique that is described in Section 6.

When there are multiple pointers per work-ahead set entry, the spin-loop checks only the first pointer. As discussed in Section 4.1, slots may be filled with a dummy cache line address. We prevent the helper thread from spin-looping on this dummy address.

5 Workload

We evaluate the three thread-based approaches by applying them to three database query operations: join using a join index, index traversal, and hash join. Due to space limitations, the join-index experiments are not described here; their results were similar to those for the other operations.

Transforming a database operation into stages is relatively straightforward. When one memory location is only available after retrieving an earlier memory location, the two memory references naturally belong to separate stages. For the reasons described in Section 4.1, it is a good idea to preload all available data needed by a stage, rather than dividing the work up into additional stages. For database operations, the simplicity of the computation per record means that the number of memory references needed per stage is often one. When it is more than one, the number of references is almost always bounded by a small constant corresponding to the number of inputs/outputs of an operator.

5.1 Index Traversal

Searching over tree indexes requires repetitively searching internal nodes from the root to the leaf nodes. Within each internal node, one must perform some computation to determine which child node to traverse. The next memory reference, i.e., the child node address, cannot be determined until that computation is finished.

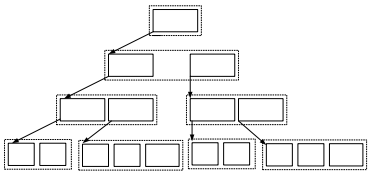


Figure 2: An In-Memory CSB$^+$-Tree

Figure 2 shows an in-memory CSB$^+$-Tree [21]. Each index node has a size of one cacheline. The work-ahead set in this case contains entries of a search key (the state) and the memory reference to the next child node to be traversed. During index traversal, the main thread performs the computation within an index node, posts the child reference into the work-ahead set, and resumes a previously registered request from the set. Each stage corresponds to processing one index node. If the current request is finished, the main thread begins to process a new probe into the index.

We consider two kinds of probe streams for the index. A *clustered* probe stream is a contiguous sequence of (key,RID) pairs. An *unclustered* probe stream is a sequence of RIDs, where the key must be obtained by dereferencing the RID, and the RIDs are in random order. We expect these two kinds of probe streams to have different memory access patterns.

5.2 Hash Join

Figure 3 shows an in-memory hash table. It contains an array of hash buckets. Each bucket is composed of a pointer to a list of hash cells. A hash cell represents a build tuple hashed to the bucket. It contains the tuple pointer and a fixed-length hash code computed

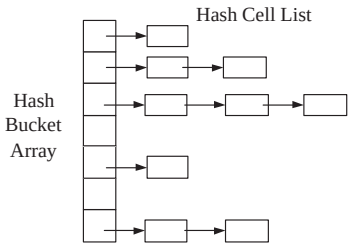


Figure 3: An In-Memory Hash Table

from the join key, which serves as a filter for the actual key comparison.

The probe stream for the hash join is a sequence of RIDs, and the hash key is obtained by dereferencing the RID. A *clustered* probe stream is one where the RIDs appear in physical memory order. An *unclustered* probe stream has RIDs in random order.

Similar to [7], we break a probe operation into 4 phases. Each phase performs some computation and issues memory references for the next phase. Phase 1 computes the hash bucket number for every tuple and issues the memory reference for the target bucket. Phase 2 visits the hash bucket and issues the memory reference for a hash cell. Phase 3 visits the hash cell, and compares the hashed key. Depending on the comparison of hashed keys, it may also issue the memory reference for the matching build tuple. Finally, phase 4 visits the matching build tuple, compares the keys, and produces the output tuple. Stages 3 and 4 repeat until the list of hash cells is exhausted.

Each slot in the work-ahead set contains a hashed key of a probe tuple, a phase number, and memory references. In phase 3, we also need a pointer to the current position in the list of hash cells. During execution, the main thread finishes one phase for one probe tuple at a time. Then, the main thread posts to the work-ahead set and resumes on the entity the work-ahead set returned. If a probe is finished, the main thread continues with the next probe tuple.

6 Experimental Evaluation

We implemented all algorithms on a Pentium 4 machine enabled with Hyper-Threading technology. Table 1 lists the specifications of our experimental system. We use Calibrator [17] to measure the L1, L2 and TLB miss latency. We are unable to measure the latency for MOMC events directly. When an MOMC event happens, the "dirty" cache line must be written out to memory and updated for each processor sharing the bus [13]. We observe that the latency is close to the L2 miss latency, roughly 200 to 300 cycles.

Modern machines have hardware performance counters that measure statistics without any loss in performance. We used Intel's VTune Performance Tool to measure the hardware counters during our experiments. We measured many potential sources of latency, including L1 and L2 cache misses, branch mispredictions, TLB misses, MOMC events, etc. For the operations studied in this paper, only the L2 cache misses and the MOMC events were significant contributors to the overall time. For that reason, we omit the other measurements from the experimental results.

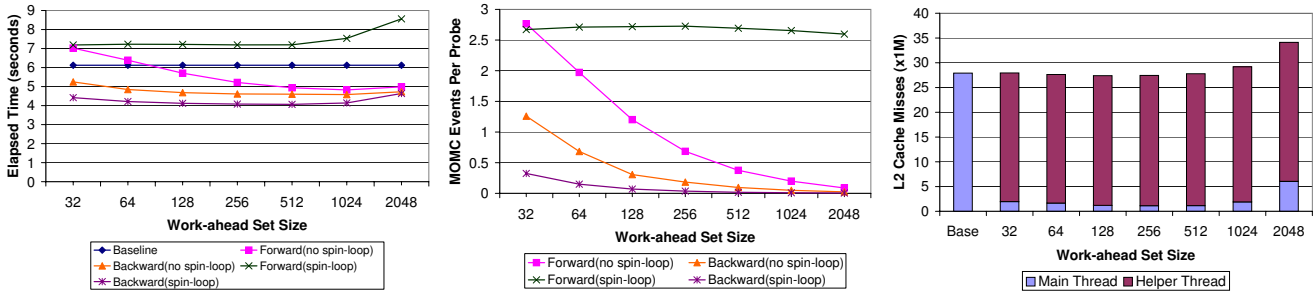
CPU	Pentium 4 3.4 GHz with Hyper-Threading
OS	Windows XP Pro (SP 2)
Main-memory size	2 GB DDR
L1, L2 data cache size L1, L2 data cacheline size TLB entries	8 KB, 512 KB 64 bytes, 128 bytes 64
L1 data miss latency L2 miss latency TLB miss latency	18 cycles 276 cycles 53 cycles
C Compiler	Visual C++ .NET 2003 (maximum optimization)

Table 1: System Specifications

In the case of the work-ahead set and bi-threaded operators, thread synchronization is implemented using shared variables. To avoid race conditions, we could use synchronization primitives, such as locks, mutexes, or semaphores. However, the overhead of implementing mutual exclusion of shared variables typically outweighs the benefit of using a helper thread to hide cache misses [14], as we also observed. Interestingly, in both techniques we do not have to ensure mutual exclusion. The work-ahead set helper thread only reads shared variables, so there is no write-write hazard, and if the helper thread goes wrong, i.e., loads the wrong data, the correctness of the main thread is not affected. The bi-threaded implementation independently processes partitioned input, sharing only read-only data structures so, again, there is no conflict among the threads.

Spin-Loop waiting, as described for the work-ahead set, can be especially wasteful because the logical processors share execution resources. In particular, a tight loop that checks one variable until it changes can fill the execution units with spurious work. Pentium 4 processors provide a low latency instruction, `PAUSE`, designed to combat this kind of behavior [13, 2]. When a thread executes a `PAUSE`, it releases hardware resources to the other thread for a period of time that corresponds roughly to how long it would take for a variable to be changed by an external thread. Our implementation of spin-loop waiting uses the `PAUSE` instruction.

Because the work-ahead set requires a staged operator implementation, for completeness we also provide performance information for the staged operators without a helper thread. The base-line algorithms used for calculating performance improvement are non-staged, single-threaded implementations. We also compare the different threading techniques' relative performances.



(a) Execution Time (b) Interference (c) Cache Miss Distribution

Figure 4: CSB$^+$ Tree Workload for the Work-ahead Set

6.1 A Closer Look at the Work-ahead Set

In this section, we analyze a work-ahead set enabled index traversal using a CSB$^+$ tree, in order to understand the helper thread parameters, and to measure the cache behavior. (The results for hash joins are similar, and can be found in Appendix A.2.)

We evaluate the CSB$^+$-Tree in the context of an index nested loop join, where the index is probed once for each outer relation record. The result of the join is a set of pairs of outer RID and inner RID of matching tuples, where an RID is just an in-memory pointer.

We implement an in-memory CSB$^+$-tree using a 32-bit floating-point key, and 32-bit pointers. The CSB$^+$-tree has a node size of one L2 cache line (128 bytes in a Pentium 4); an index node is able to contain up to 30 keys, while a leaf node is able to contain up to 14 key-RID pairs. The tree indexes 10 million keys. The index is bulk-loaded and nodes are up to 75% full. The index is probed using 10 million (key,RID) pairs from a (hypothetical) outer table. The probe stream is clustered, and the probe keys are randomly generated.

We consider five different scenarios: a single-threaded implementation, simple forward preloading with no spin-loop, simple backward preloading with no spin-loop, forward preloading with spin-loop and backward preloading with spin-loop. For each preloading scenario, we also vary the work-ahead set size.

Figure 4(a) shows the execution time with different work-ahead set sizes. Backward preloading scenarios run faster than the original program, up to 33% faster in the spin-looping case. Backward preloading is faster than forward preloading. Spin-loop waiting improves the performance of backward preloading, but not forward preloading.

The average number of MOMC events per probe is shown in Figure 4(b). The main thread has more computation to do than the helper thread, so the helper thread runs faster, leading to more potential interference in the work-ahead set. As predicted, forward preloading experiences more MOMC events than backward preloading. Using spin-loop waiting for forward preloading increases the number of MOMC events significantly.

In this experiment, backward preloading has the least interference and therefore is faster than forward preloading. The difference in interference between the two scenarios explains the big performance difference. The interference effect decreases with larger work-ahead set sizes because the average distance between the two threads is larger and there is a smaller likelihood that the two threads are operating on the same cache line. In the rest of this paper, we use backward preloading with spin-wait when using a work-ahead set.

Figure 4(c) shows the distribution of cache miss penalties for the backward preloading scenario with spin-looping. The other methods had similar distributions of cache misses. The helper thread absorbs most of the cache misses, up to 97%. The cache miss reduction in the main thread is the reason for the 33% performance improvement. We have generally been successful at transferring cache misses from the main thread to the helper thread without increasing the total number of cache missses. The absence of an increase in cache misses is a consequence of our design, in which the work-ahead set structure itself occupies just a small fraction of the L2 cache.

When a work-ahead set is larger than 1024, the total number of cache misses increases as more conflict cache misses occur. Such increase is sharper for forward preloading than for backward preloading. This is because in forward preloading, the helper thread always follows the main thread resulting in a greater average distance between the main thread and a recently preloaded memory reference than with backward preloading. Preloading with a large work-ahead set is more likely to incur conflict cache misses with other preloaded memory references.

Figure 5 compares work-ahead set performance of clustered probes with that of unclustered probes. We only show the performance of backward preloading with spin-wait; other scenarios have similar performance curves and are omitted. Random probes are generally more expensive than sequential probes and provide a more realistic workload. Using a work-ahead set results in a 41% speedup for unclustered probes. This improvement is even larger than for sequential probes, primarily because there are more cache misses

in the unclustered case.

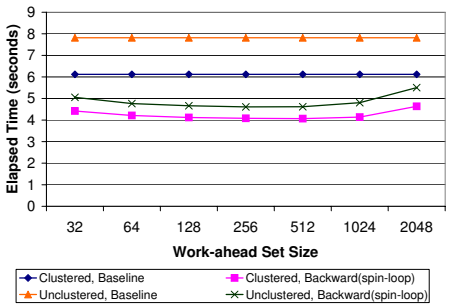


Figure 5: Clustered and Unclustered Probes

As we described in Section 4.2, the performance of preloading is poor when the work-ahead set size is too small, which we see experimentally to be 64 references. The high number of expensive MOMC events outweighs any cache improvement. All preloading methods start to deteriorate due to the increasing effect of cache pollution once the size is larger than 1024 references, which is about one quarter of the L2 cache size. As long as the size of a work-ahead set is in the range between 64 and 1024, the backward preloading performance (with spin-looping) is good, and relatively insensitive to the work-ahead set size. Thus, our preloading solution is robust. One would size the workahead set at the low end of this range, say 128 entries, so that more cache capacity is available for other purposes. In what follows, experiments using the work-ahead set will employ a size of 128.

6.2 Threading Performance

Now we compare all three threading techniques when applied to a CSB$^+$ tree traversal and an in-memory hash join, with unclustered probes in both cases.

The hash join operator joins two tables, each containing 2 million 64-byte records. The hash join produces a table of 128-byte output records. The join keys are 32-bit integers, which are randomly assigned values in the range from 1 to 2 million. An in-memory hash table is pre-built over one table. The probe stream is unclustered. We use join keys from the probe records to probe the in-memory hash table and generate join results. (When we varied the record size, the relative performance of the various methods was similar to what is shown below.)

In these experiments *One Thread* and *Two Threads* refer to non-staged implementations running either alone or in parallel. *Two Threads* means that the same operator is being run in each thread, but each operator has its own input and output data. We also provide similar labels for the staged implementations. *Bithreaded* refers to operators, both staged and unstaged, that partition their input and process it in an interleaved fashion as described in Section 3. Finally, *Work-ahead Set* is a staged operator with a helper thread performing explicit preloading. In some graphs, data for the work-ahead set's main and helper thread are separated to demonstrate their different roles in performance improvement.

We find that all three threading techniques result in substantial increases in the throughput of both an in-memory hash join and a CSB$^+$ tree traversal. As Figure 6(b) shows, in-memory hash join throughput, when compared with a single threaded implementation, increased by up to 48%, 50%, and 55% for independent parallel hash joins, a bi-threaded hash join, and a work-ahead set enabled hash join, respectively. CSB$^+$ performance, Figure 6(a), is comparable, with improvements in throughput of up to 69%, for a bi-threaded CSB$^+$ tree traversal, and 68%, for the work-ahead set enabled implementation.

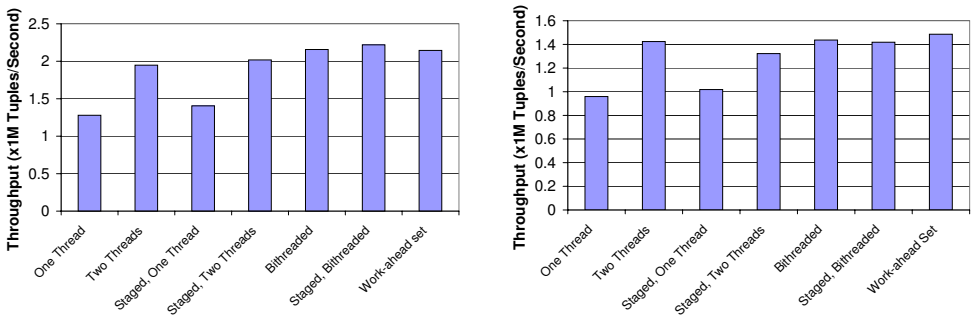
The key benefit of these threading techniques is exploiting thread level parallelism to efficiently overlap blocking cache misses with useful computation. Whereas in a single threaded implementation, a stall caused by a cache miss results in wasted time, having multiple threads available to perform useful computation on the processor's functional units improves overall throughput. Figures 7(a) and (b) show the number of cache misses suffered by each thread, which we find to be the most important source of performance difference among the threading techniques.

Ideally, one would expect independent, parallel hash join or CSB$^+$ tree traversal operators to exhibit the same number of cache misses per thread as the respective single threaded version.[3] The parallel operators, however, suffer elevated cache-misses per thread because they are not designed to be cooperative: there is cross-thread cache pollution. Even with more cache misses, performance improves because the two threads keep the processor more active by overlapping cache misses with computation.

A bi-threaded implementation processes the same amount of data as a single threaded version and therefore suffers almost exactly the same number of cache misses. Though the two threads share input, this does not reduce overall cache misses, since one of the threads must suffer the cache miss required to bring input data into the cache. The important difference is that the bi-threaded operator distributes these cache-misses over the two threads and benefits by overlapping a cache miss in one thread with computation in the other. Additionally, our bi-threaded implementation processes the input in an interleaved manner, an inherently cooperative design choice that allows a thread to benefit if the other thread's cache miss loads needed input data into the cache.

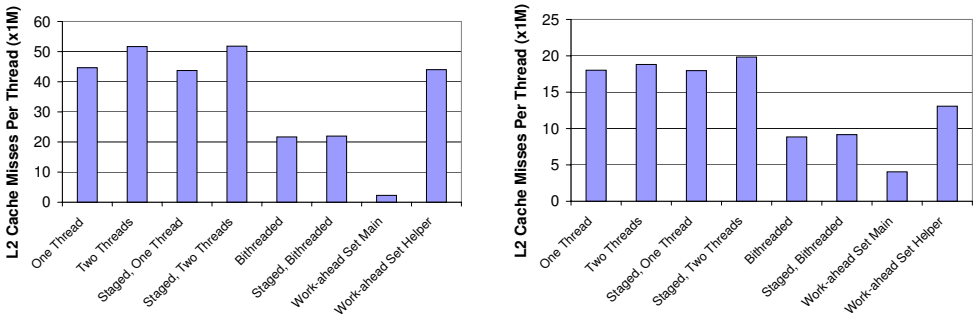
We find that the work-ahead set outperforms naive independent, parallel operators, in terms of through-

[3]Note that in the case of independent parallel operators, twice as much work is processed as in the single threaded implementation so a *per thread* comparison is appropriate.



(a) CSB$^+$ Tree Traversal (b) In-memory Hash Join

Figure 6: Throughput Using Different Threading Techniques



(a) CSB$^+$ Tree Traversal (b) In-memory Hash Join

Figure 7: L2 Cache Misses Using Different Threading Techniques

put, by up to 10% for a CSB$^+$ tree index traversal, and up to 4.3% for an in-memory hash join. The work-ahead set achieves its performance improvement by emphasizing the overlapping of cache misses with useful computation. In Figure 7(a), the work-ahead set helper thread suffers almost as many cache misses as either thread of the parallel operator implementations, but the key difference is that these misses leave the main thread free to work almost exclusively on useful computation.

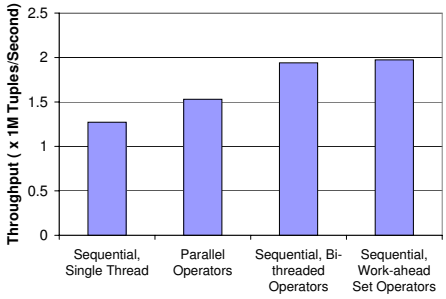


Figure 8: Heterogeneous operator throughput

6.3 Different Operators in Parallel

As described earlier, a naive use of an SMT processor is to treat it as a multiprocessor system, running independent parallel operations. So far, our two-threaded experiments have always been executing the same operator on each thread. To examine what happens when different threads run different operators, we execute a hash join and csb+ tree index traversal in parallel. We ran one operator in each thread, stopping the experiment when the first operator finished. As a result, each logical processor always had an instruction stream to process and was never unused. We note the number of records processed by each of the two threads.

We then compare the measured throughput with sequential single-threaded, bithreaded, and work-ahead set implementations of the hash join and CSB$^+$ tree index traversal operators, processing the same amount of data for each operator.

Figure 8 shows the results of these experiments. All threading techniques improve throughput performance relative to a single-threaded sequential implementation. Throughput for bithreading and work-ahead set implementations exceed naive parallelism by 26% and 29%, respectively. The gain was primarily due to reduced conflicts in the L2 Cache. More complex parallel operators whose instruction footprint exceeds the instruction cache may also suffer further performance loss due to contention for the instruction cache. Bithreaded and work-ahead set enabled operators are less susceptible to this problem. Bithreaded operators share most instructions, and work-ahead set helper threads are designed to be small and to not conflict with the operator's primary code.

6.4 Columnwise Record Layout

The previous experiments focussed on row based record storage. This configuration features limited

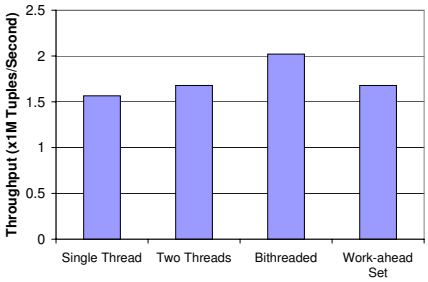


Figure 9: Columnwise Throughput

spatial and temporal locality, particularly in the probe table. This kind of configuration stands to gain the most from the work-ahead set concept because a cache miss rarely benefits more than one data access. In this section we present the opposite configuration: an in-memory hash join implementation with columnwise record storage and a clustered probe stream that includes all records in the table.

Our columnwise implementation stores each attribute of a table in separate in-memory arrays where the k^{th} record's attributes are found at index k of each array. Additionally, the probes are clustered, so subsequent probes benefit from a previous probe's cache miss. This hash join operator joins two tables with integer attributes and its output includes the join key attribute as well as two integers from each table. To assemble the output record, the work ahead set is expanded to hold up to five data pointers per entry as described in Section 4.1.

The throughput of the columnwise hashjoin is not directly comparable to the previous row based experiments, but we provide a similar comparison among the threading techniques described in this paper. As Figure 9 shows, thread level parallelism and the work-ahead set yield similar, modest throughput improvements for the columnwise configuration. In this kind of scenario, the single-threaded application benefits from two related phenomena. Because the data has good spatial locality, there will be fewer cache misses incurred per record. Further, because access to the columns is sequential, hardware prefetching will be able to hide the latency of some cache misses.

Bithreading performs particularly well because it exploits the good record locality and sequential access, which likely triggers hardware prefetching. Unlike naive parallelism, it does not compete for data cache space with other threads.

7 Conclusion

The relative strengths and weaknesses of the three proposed strategies for SMT processors are summarized in Table 2. Bi-threading has good performance in both row-wise and column-wise layouts, but it changes the data format and does not preserve record order. The work-ahead set performs better than naive parallelism, uses the standard data format, and allows for better global coordination of cache usage. Naive parallelism is the easiest to implement, but gives only modest performance improvements.

Because the work-ahead set is advantageous to data dependent work loads, such as those explored in this paper, hardware implementers should consider providing a work-ahead set type feature in hardware. Work along these lines has been proposed by Chappel et. al. in [6] with respect to creating subordinate *microthreads* in hardware to perform operations such as prefetching and cache management, without interfering with main computation threads.

We performed additional experiments that are included in this appendix. In order to pin down the overhead of using a work-ahead set, we implemented a version of backward preloading for the join index case in which the helper thread accessed the join index explicitly. This avoids the overhead of copying the memory references. Further, since the main thread does not write to the join index, there may be fewer MOMC events. On the other hand, the helper thread needs to periodically check whether it has passed the main thread, which introduces additional MOMC events. Surprisingly, this "direct" implementation performed slightly worse that our work-ahead set implementation. We attribute the difference to the difficulty of tuning the various parameters of the direct implementation.

There is a chance that multiple references in the work-ahead set may fall into the same cache line, meaning that the helper thread may unnecessarily load a cacheline more than once. One could design a helper thread that is able to check for duplicate cache line references and perform loading just once. However, this extra computation in the helper thread may compete with the main thread for shared computation units and slow down the execution of the main thread. Further, the cost of loading the cache line the second and subsequent times is low, since it will correspond to a cache hit. We verified experimentally that such extra "intelligence" in the helper thread worsened performance.

A Supplementary Experiments

We also implemented a single-threaded staged version of the join index example that, instead of posting to the work-ahead set, executes appropriate software prefetch instructions available in the Pentium 4 instruction set. This version of the code, see Figure 10, performed no better than code without such prefetch instructions, leading us to conclude that most of the prefetch instructions were dropped.

A.1 Using a Join Index

A join index [25] is a sequence of pairs of tuple pointers representing the pairs of tuples that join according to a known join condition. A join index may be stored in

	Parallel	Bi-threaded	Work-Ahead Set
Implementation Effort	Small	Moderate	Moderate
Change in Data Format	No	Yes	No
Change in Data Order	No	Yes	No (if all data goes through the same number of stages)
Performance Improvement (row-wise)	Moderate	High	High
Performance Improvement (column-wise)	Moderate	High	Moderate
Control of cache resources	No	No	Yes

Table 2: Strengths and Weaknesses of the Proposed Techniques

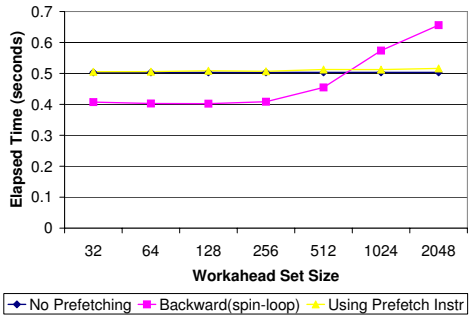


Figure 10: Join Index Using Prefetch Instructions

order to speed up join queries or it may be generated as it is needed.

Composing output tuples using a join index usually has bad cache behavior because the tuple pointers in the index point to "random" physical memory addresses in the two input relations. By using a helper thread to preload the input (and output) cache lines posted by the main thread, the main thread will more often find data already in the cache. As a result, the main thread will make faster progress.

We consider two variants of the join index. In the "inner" variant, the join index has one stage with three memory references: one for each input table and one for the output result. In the "outer" variant, we allow null values for the second pointer in the join index, as if the join index represents a left outer join. This variant is interesting because some entries in the work-ahead set will need two memory references, while some will need three.

A.1.1 Join Index Experiments

We join two tables using a join index. Each table is populated with 2 million 64-byte records. For the "inner" join experiment, the join index is precomputed and contains 1 million pairs of randomly generated record identifiers from the two tables. We perform a join of these 1 million pairs using the join index and copy every pair of matching records to a result record. Each result record is 128 bytes and fits in a 128-byte L2 cache line[4].

[4]Similar to Section 6.2, we conducted experiments with varied record size, but found the results to be similar to what is shown below.

For the "outer" join experiment, we simulate the case of a left outer join. We have the same table configuration except that some fraction of right pointers in the join index are null pointers, indicating the record from the left table joins no records from the right table. In that case, we only copy the record from the left table to the result record.

We consider five different scenarios: a single-threaded implementation, simple forward preloading with no spin-loop, simple backward preloading with no spin-loop, forward preloading with spin-loop and backward preloading with spin-loop. For each preloading scenario, we also vary the work-ahead set size.

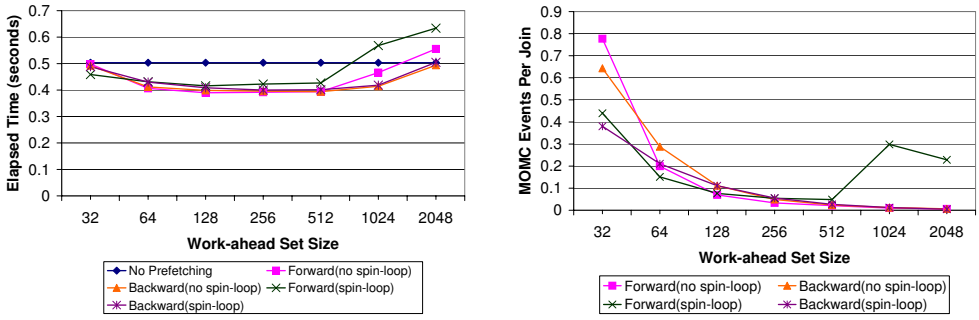
Figure 11(a) shows the execution time for the inner join with different work-ahead set sizes. Both forward and backward preloading run faster than the original program when the size is smaller than 1024 cache lines. Up to 22% of the original execution time is saved. Backward preloading is slightly faster than forward preloading. When the work-ahead set size is too large, the performance of preloading scenarios degrades due to cache pollution effects.

Figure 11(b) shows the effect of interference between the two threads. In both graphs we see little difference between spin-looping and not spin-looping for backward preloading. For this experiment, the main thread does relatively little computation, and catches up to the helper thread. As a result, the helper thread hardly ever spin-loops. The overall impact of interference on performance is relatively small.

Figure 12 shows the distribution of cache miss penalties for the backward preloading scenario with spin-looping. The other methods had similar distributions of cache misses. The helper thread absorbs most of the cache misses, up to 75%. The cache miss reduction in the main thread is the reason for the 22% performance improvement.

For the left outer join experiment, we precompute the join index in which 20% of join pairs contain a null pointer. As described in Section 4.1, each entry of the work-ahead set contains up to 3 references. The main thread can post 2 or 3 references to an entry, depending on whether the right join pointer is null or not. The initial number of the entries is chosen as the work-ahead set size divided by 2. The main thread dynamically decides if there is space to post new references.

Figure 13 shows the execution time. For each sce-



(a) Execution Time

(b) Interference

Figure 11: Preloading a Join Index

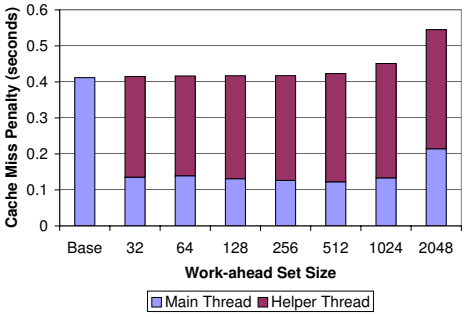


Figure 12: Distribution of Cache Miss Penalties for Backward Preloading a Join Index (Spin-wait)

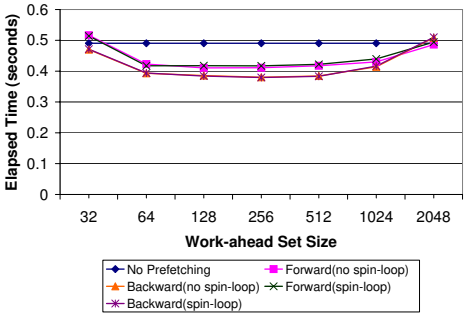


Figure 13: Preloading an Outer-Join Index

nario, we vary the work-ahead set sizes. The performance improvement is similar to that of the inner join experiment. Up to 22% of the CPU time is saved by preloading. We also try different percentages of null pointers and have similar improvements. The analysis of cache miss distribution and interference is also similar to the inner join experiment.

A.2 Hash Join Experiments

The hash join operator joins two tables, each containing 2 million 64-byte records. The hash join produces a table of 128-byte output records. The join keys are 32-bit integers, which are randomly assigned values in the range from 1 to 2 million. An in-memory hash table is pre-built over one table. We use join keys from the other table to probe the in-memory hash table and generate join results.

Figure 14(a) shows the execution times with different work-ahead set sizes. Both preloading methods run faster than the original program and up to 34% of execution time of the original program can be saved by backward preloading. Backward preloading is also faster than forward preloading.

Figure 14(b) compares the effect of interference between the two preloading scenarios. Similar to the case of preloading a CSB$^+$ tree, backward preloading has the least cache interference and is therefore faster than forward preloading. Forward preloading with a spin-loop performs very poorly because it guarantees the pathological case described for forward preloading in Section 4.3.

Figure 15 shows the distribution of cache miss penalty for backwards preloading with spin-looping. (Again, the graphs for the other variants were similar.) As in Section 6.1, the helper thread picks up a significant number of the cache misses. The main thread suffers as few as 25% of the total cache misses, which again explains the performance benefit.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of VLDB Conference*, 1999.

[2] A. Binstock and R. Gerber. *Programming with Multi-threaded Software for Intel IA-32 Processors*. Intel Press, 2004.

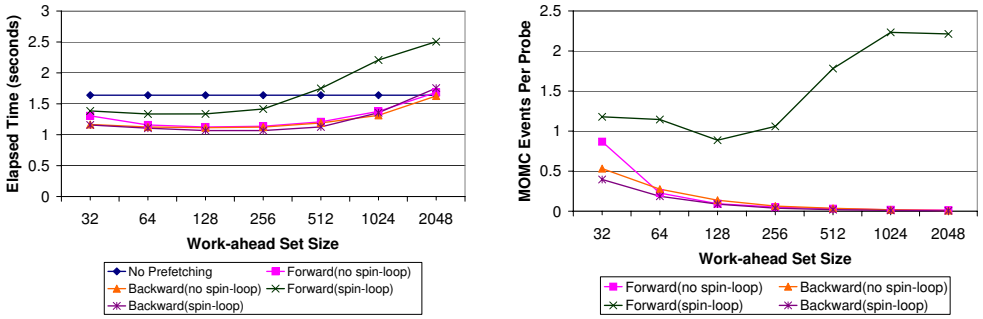
[3] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman. The microarchitecture of the intel pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(1):7–19, 2004.

[4] P. Bohannon, P. McIlroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *Proceedings of SIGMOD Conference*, 2001.

[5] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of VLDB Conference*, 1999.

[6] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the 26th Annual ISCA Conference*, 1999.

[7] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *Proceedings of ICDE Conference*, 2004.



(a) Execution Time (b) Interference

Figure 14: Preloading Hash Join

[8] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proceedings of SIGMOD Conference*, 2001.

[9] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B+-trees: Optimizing both cache and disk performance. In *Proceedings of SIGMOD Conference*, 2002.

[10] J. D. Collins, H. Wang, D. M. Tullsen, C. J. Hughes, Y. fong Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of ISCA Conference*, 2001.

[11] S. Harizopoulos and A. Ailamaki. STEPS towards cache-resident transaction processing. In *Proceedings of VLDB Conference*, 2004.

[12] IBM. IBM POWER Architecture. Available via http://www.ibm.com/technology/power/.

[13] Intel Corporation. IA-32 intel architecture optimization reference manual. Available via http://developer.intel.com.

[14] D. Kim, S. S. wei Liao, P. H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen. Physical experimentation with prefetching helper threads on intels hyper-threaded processors. In *Proceedings of International Symposium on Code Generation and Optimization*, 2004.

[15] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *Proceedings of SIGMOD Conference*, 2001.

[16] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of ISCA Conference*, 1998.

[17] S. Manegold. The calibrator (v0.9e), a cache-memory and tlb calibration tool. http://homepages.cwi.nl/ manegold/Calibrator/.

[18] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, 2002.

[19] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*, page 129. IEEE Computer Society, 2003.

[20] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of VLDB Conference*, 1999.

[21] J. Rao and K. A. Ross. Making B+ trees cache conscious in main memory. In *Proceedings of SIGMOD Conference*, 2000.

[22] Sun Microsystems Inc. Ultrasparc processors. Available via http://www.sun.com/processors/index.html.

[23] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268. ACM Press, 2000.

[24] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of ISCA Conference*, 1995.

[25] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, 1987.

[26] H. Wang, P. H. Wang, R. D. Weldon, S. M. Ettinger, H. Saito, M. Girkar, S. S. wei Liao, and J. P. Shen. Speculative precompuation: Exploring the use of multithreading for latency. *Intel Technology Journal*, 6(1):22–35, 2002.

[27] P. H. Wang, H. Wang, J. D. Collins, E. Grochowski, R. M. Kling, and J. P. Shen. Memory latency-tolerance approaches for itanium processors: Out-of-order execution vs.speculative precomputation. In *Proceedings of International Symposium on High-Performance Computer Architecture*, 2002.

[28] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *Proceedings of VLDB Conference*, 2003.

[29] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *Proceedings of SIGMOD Conference*, 2004.

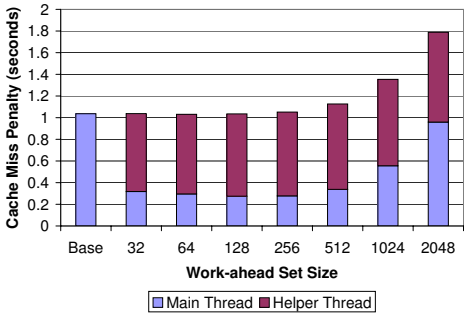


Figure 15: Distribution of Cache Miss Penalties