Optimizing Read Convoys in Main-Memory Query Processing

Kenneth A. Ross*
Columbia University, New York, NY 10025
kar@cs.columbia.edu

ABSTRACT

Concurrent read-only scans of memory-resident fact tables can form convoys, which generally help performance because cache misses are amortized over several members of the convoy. Nevertheless, we identify two performance hazards for such convoys. One hazard is underutilization of the memory bandwidth because all members of the convoy hit the same cache lines at the same time, rather than reading several different lines concurrently. The other hazard is a form of interference that occurs on the Sun Niagara T1 and T2 machines under certain workloads. We propose solutions to these hazards, including a local shuffle method that reduces interference, preserves the beneficial aspects of convoy behavior, and increases the effective bandwidth by allowing different members of a convoy to concurrently access different cache lines. We provide experimental validation of the methods on several modern architectures.

1. INTRODUCTION

The classic example of a convoy is when traffic on roads without passing lanes forms clusters, with many cars "stuck" behind slow cars [2]. Convoys are a well-known phenomenon in databases [2], occurring when many transactions queue on a lock request. Once a critical number of requests are queued, the system can enter a stable state in which all requests pass through the lock repeatedly, in sequence, slowing down the system.

Even in read-only databases, convoys can occur during query processing. For example, Zhou at al. [8] describe a method in which a worker thread is aided by a helper thread whose job is to preload needed data into the cache. The worker thread writes memory locations to a data structure called the work-ahead set. The helper thread reads from this data structure to find out which addresses to preload. Since the helper thread suffers cache misses, it often happens that the helper thread was only just ahead of the worker thread (which did more computation, but experienced cache hits). The threads thus form a convoy with two members. Unfortunately, when the helper thread and worker thread access the same cache lines in the work-ahead set (even if they accessed different data elements), an expensive memory-ordering event is generated, slowing down both threads dramatically. The solution explored in [8] is somewhat counterintuitive: make the helper thread work *backwards* in the work-ahead set structure relative to the worker thread. This behavior avoided the convoy effect that led to the expensive memory-ordering events. See Appendix B for further discussion of backwards processing.

A second kind of convoy occurs when multiple threads are processing records in the same order, and updating a common data structure. For example, Cieslewicz et al. [4] consider many threads cooperating to compute a grouped aggregate query by scanning fragments of an input table and updating a shared hash table with data from the processed records. Threads use atomic operations or latches to protect against destructive updates for the short period in which the data in the hash table is updated. Even if the data values are well distributed, when there is a common reference pattern in the data stream there is a performance hazard caused by convoying behavior [4]. Consider a repeated run distribution in which the group-by column takes the values $1, 2, \ldots, n, 1, 2, \ldots$ in a repeating sequence. Because of contention, a thread will slow down when it tries to access and update an element accessed by another thread. This effect leads to a convoy in which all threads repeatedly contend for the same items in the same sequence, dramatically reducing performance. One solution is to cycle through the input table fragments using a different stride[1] in each thread, while still reading each input cache line only once [4]. Because of the different strides, an occasional contention event does not lead to repeated contention as before.

In both of these examples there is conflicting read/write or write/write behavior on a shared data structure. Convoys can also occur for read scans of data. When there is only reading of data, the conventional wisdom has been that a convoy is at worst harmless, and in some cases even helpful because of improved locality. For example, Qiao et al. [6] describe how convoys naturally form between multiple threads scanning a common table. Leading threads progress slowly due to cache misses, while trailing threads have the opportunity to catch up as they experience cache hits resulting from the loading of the data by the earlier threads. The resulting

*Supported by NSF Grants IIS-0534389 and IIS-0915956, and by a Jim Gray seed grant from Microsoft Corp.

Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010) June 7, 2010, Indianapolis, Indiana.

[1]Each stride must be relatively prime to the number of cache lines in the input.

convoy improves performance by allowing cache misses to be amortized over the member threads.

Nevertheless, there remain performance hazards for read convoys, as we demonstrate in this paper. We identify two distinct kinds of hazard. The first hazard is caused by the fact that in a convoy of many threads, all threads end up synchronized, waiting on the same cache line. Since modern processors allow multiple outstanding memory requests, it seems that the machine is being underutilized. If several cache lines were being requested concurrently, the convoy could be making faster progress through the input data. Prefetching (either in software or in hardware) does not solve this problem, because all threads end up concurrently prefetching the same cache lines too, and memory transfer rates are limited to what one thread can sustain.

The second hazard is more surprising, because it involves a read-only convoy on a shared data stream that performs significantly worse than threads working on independent data streams. In other words, there is *interference* between read-only threads. We demonstrate such a hazard on the Sun Niagara T1 and T2 machines, which are, respectively, 32-thread and 64-thread symmetric multithreading processors. The kinds of workloads that cause this effect are common ones, involving multiple threads scanning a common input table and performing a limited amount of processing on each record in sequence.

After demonstrating the convoy-related performance hazards, we investigate the two prior solutions mentioned above for avoiding convoys by changing the order of record access in some threads. We also demonstrate how prefetching can avoid the second hazard, even though it does not change the order of access. To our knowledge, this is the first use of prefetching as a convoy-avoidance technique.

We then propose a new way of reordering accesses called the *local shuffle* that solves both hazards. The main idea is to preserve the cache sharing behavior of the convoy by retaining a common *global* access pattern across threads. However, each thread will have a different *local* access pattern, where different threads trigger cache loads for different parts of the input data.

We verify that the local shuffle solves the second performance hazard on the Niagara machines, and performs even better than prefetching. We also study the local shuffle on two additional platforms, the Intel Core i7 (Nehalem) processor and the AMD Opteron (Barcelona) processor. The local shuffle improves performance on the Nehalem, but not the Barcelona. The Barcelona performance is unchanged due to its unusual cache architecture: the lower level caches are victim caches and prefetched data is sent directly to the L1 cache, meaning that there is little opportunity for threads to share prefetched data.

The remainder of this paper is structured as follows. Section 2 outlines the context of the paper, and discusses related work. In Section 3 we identify and characterize read interference between threads in convoys on the Sun T2 machine; we also implement and evaluate several methods to overcome this interference. In Section 4 we develop a new scheme for scan-like operations that preserves the global convoying behavior, while changing the local access pattern to improve performance. Additional query types are discussed in Section 5. We conclude in Section 6. Information about the experimental configurations used in the paper can be found in Appendix A.

2. FRAMEWORK AND RELATED WORK

Our memory-resident database consists of a fact table and a collection of small dimension tables. The tables are stored as arrays as they might be stored by an in-memory database system [1, 3]. The fact table is stored columnwise as a collection of arrays, one per column. Fact table entries for a given column may be interpreted as foreign keys into a particular dimension table; a value of i in the fact table column means that the fact table record is referencing the ith record of the dimension table. The dimension tables could be stored row-wise or column-wise, but for the purposes of this paper, where the entire dimension tables will be L1-cache resident, this choice is unimportant. We assume a fact table that is larger than the lowest-level cache of the machine, which on current architectures is typically 2–8MB.

The class of queries we have in mind are those that can be answered with a single scan of the fact table while (a) doing minimal computation and (b) holding just a small amount (less than the L1 cache) of state. The state would include accessed dimension tables, as well as any intermediate structures such as hash tables for grouped aggregation. Because they do minimal computation, such queries are likely to be memory-bound rather than compute-bound. Further, since the state is L1-resident, the memory bottleneck will typically be the lowest-level cache misses on the fact table.

When large fact tables are processed sequentially, it pays to process the data in batches of records [5, 9, 3]. Within a batch of records, data should be processed one column at a time [3]. The payoff for such a choice is a tighter inner loop, better opportunities for compression, and better instruction cache locality. Batch sizes are typically chosen to be about 1024 records. We follow such an approach and choose a batch size of 1024 rows.

Given a scan query Q, we run Q on every thread in the system in parallel. Further, when Q completes on a thread, we immediately restart Q over again. After a sufficient number of instances of Q have been run, convoys will form in which each thread accesses the same fact table elements as other members of the convoy at the same time.

We choose this pattern (the same query repeated over and over) to make the analysis of the convoy phenomena straightforward. When different scan queries with different rates of progress are run on different threads, one would expect to see additional more complex phenomena such as convoy splitting and merging, phenomena that are beyond the scope of the present analysis.

Qiao et al. argue that one can achieve better throughput than convoying scans by grouping queries into batches that are explicitly coordinated to progress together through the fact table [6]. Nevertheless, such an approach may give poor response time, since all queries in a batch progress only as fast as the slowest query in the batch. In contrast, if queries progress independently, slow queries will naturally drop out of a convoy while fast queries make rapid progress.

3. READ INTERFERENCE ON THE T2

Our discovery of read interference on the Sun Niagara machine was serendipitous. We were measuring the performance of scan-like queries that aggregated values found in foreign-key dimension tables. When the dimension cardinality was small, so that dimension tables were L1 cache resident, the throughput curve had several unexpected dis-

continuities when plotted against the number of concurrent threads.

To isolate the problem, we progressively simplified the query, while observing whether the discontinuities persisted in the simplified query. We found that the problem was apparent even for a scalar aggregate of a single fact table column, without dimension table references. The performance discontinuities are shown in Figure 1 for this simple scalar-sum aggregate.

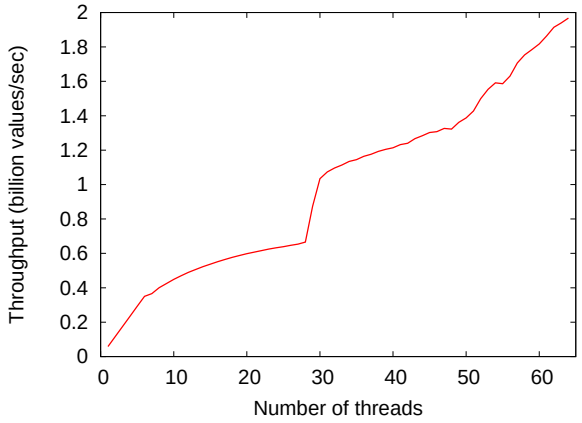


Figure 1: Throughput discontinuities for a single-column scalar aggregate on the Sun Niagara T2.

There are several interesting features of this graph. The first feature is the sudden change in slope from the region with 6 or fewer threads to the region from 7 to 28 threads. Since there are 8 cores on the machine, and threads are distributed among cores by the operating system, this transition does not seem to coincide with a natural core/thread boundary. The second interesting feature is the sudden improvement in performance from 28 to 31 threads. After that, performance improves slowly again until 48 threads. From 49 to 64 threads, performance improves about twice as fast as from 32 to 48 threads. This last feature is surprising because on a simultaneous multithreading (SMT) machine like the T2, the value one gets from the last threads is generally less than from the earlier threads.

We repeated the experiment on several other machines, and did not see any similar effects on an Intel Nehalem processor or an AMD Opteron processor. However, we did see similar behavior (with different transition points) on a Sun Niagara T1 machine having 32 threads. We concluded that the observed phenomena might depend on some specific aspect of the Niagara architecture.

To further illuminate the performance issue, we used Sun's cpc performance counter libraries to measure various performance related architectural events. The L2 miss events, shown in Figure 2, were particularly informative.

As the fact table is scanned, data is brought into the L2 cache in 64-byte L2 cache lines. Thus, in the absence of any shared misses, one would expect to see one L2 cache miss for each sixteen 4-byte data values. As Figure 2 shows, we do encounter one cache miss per cache line for 1 to 6 threads. We can thus infer that there is little sharing or convoy formation in this range.

However, when we hit 7 threads, the L2 miss count per cache line jumps to about 4, and stays above 3 for the rest

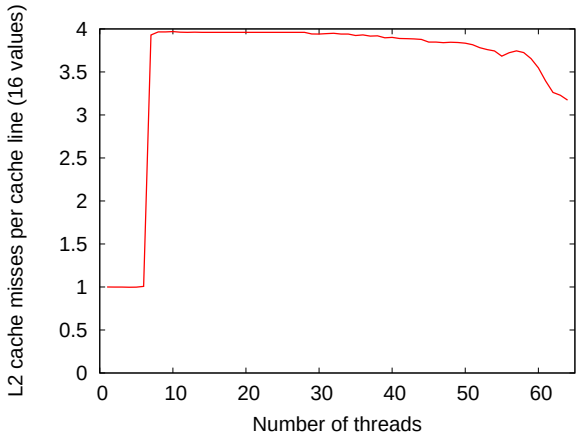


Figure 2: L2 misses per fact table cache line for a single-column scalar aggregate on the Sun Niagara T2.

of the graph. This discontinuity corresponds to the sudden change in throughput from 6 to 7 threads in Figure 1. There does not seem to be a sudden change in L2 miss behavior to explain the jump at 28 threads. A slight drop in L2 misses at the right of the graph seems to be associated with the improved throughput in the range.

What is causing these extra L2 cache misses? A critical clue comes from the fact that the number of misses per cache line jumps from 1 to almost exactly 4. The L1 cache of the T2 has a 16-byte cache line, and so one expects four L1 misses for each L2 cache line.[2] To find out why these L1 misses might result in additional L2 misses, we consulted the Niagara micro-architecture specification [7]. We found in Section 5.3.3 the following description of the load miss queue (LMQ):

> The LMQ checks for common addresses for load misses across threads to prevent duplication of tags in the dcache. The LMQ compares an incoming load from any thread against all valid entries in the LMQ. If there is a match, the incoming load is termed a secondary miss. Secondary misses cause a request to the L2, but they are marked as non-cacheable to prevent cache pollution.

Based on this description, it appears that two threads that access the same address within the timespan of an L2 cache miss can interfere with each other, even if both are reading. In particular, the L2 line is marked non-cacheable. If this event occurs repeatedly, an L2 miss occurs for each of the four L1 accesses. Note that it takes the special coordination evident in a convoy to generate this kind of access pattern in which many threads touch the same cache line within a very narrow timeframe.

Returning to the throughput graph, we can now explain the transition from 6 to 7 threads. The relative speedup at the right of the graph can be explained based on the fact that when cores become close to fully loaded with threads, each thread slows down slightly as they take turns being

[2]In contrast, the Nehalem and Opteron machines have 64-byte L1 cache lines.

processed. This decrease in speed reduces the rate at which conflicting accesses of the sort mentioned above are generated, leading to a slight improvement in the L2 miss rate and throughput. The transition between 28 and 31 threads remains mysterious, and will be explained later, once additional information is at hand.

Eliminating Read Interference

To address the read interference problem, we must somehow reduce the contention between threads for the same fact table items. We first apply the technique of [8] in which half of the threads progress forwards through the array, while half of the threads progress backwards. The results are shown in Figure 3. It appears that the performance problems remain, but that the thresholds are simply deferred until twice the number of threads.

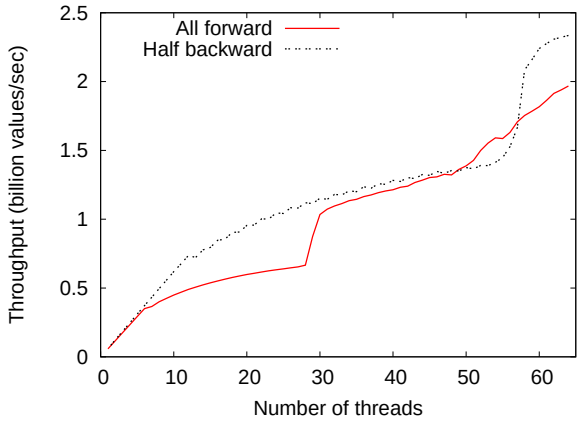


Figure 3: Throughput with half the threads progressing backwards.

Figure 3 also suggests an explanation for the performance jump at 28 threads in Figure 1. The performance of the "all forward" technique matches the performance of the "half backward" technique from 31 threads to 48 threads, suggesting that the "all forward" technique undergoes a change of dynamics at that point, splitting into two independent convoys.

We next apply a solution motivated by [4], where cache lines were accessed with different strides in different threads. Instead of using cache lines as the striding unit, we will use batches of 1024 values as the unit. Each thread gets a different prime stride that is relatively prime to the number of batches in the data set. Threads wrap around when they reach the end of the data, until every batch has been processed. The results are shown in Figure 4. Performance is significantly better than before, by more than a factor of two in some regions, and the performance curve does not show any discontinuities. This performance improves because the convoying behavior has been destroyed, eliminating the interference.

Our final solution for this section is prefetching. The discovery of this solution was also serendipitous. When making a small, apparently inconsequential change[3] the performance

[3]The change was putting slightly different aggregate computations inside a `case` statement. Only the first block of the `case` statement showed improved performance.

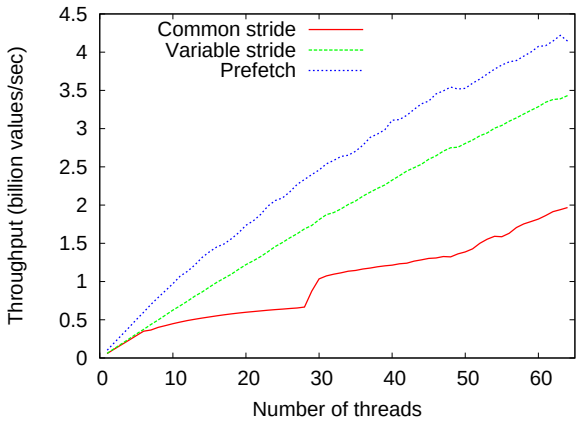


Figure 4: Throughput with varied batch stride, and with prefetching.

seemed to dramatically improve. Investigation of the generated assembly code revealed that the compiler had optimized the inner loop by inserting a prefetch statement to load the fact table data into the L2 cache ahead of its use.[4]

In a convoy, one might suspect that the same conflict pattern would remain even with prefetching, since threads are synchronized. However, as the Niagara micro-architecture specification [7] explains:

> Once the [prefetch] packet is sent to the PCX, lsu_complete can be signaled and the entry in the LMQ retired.

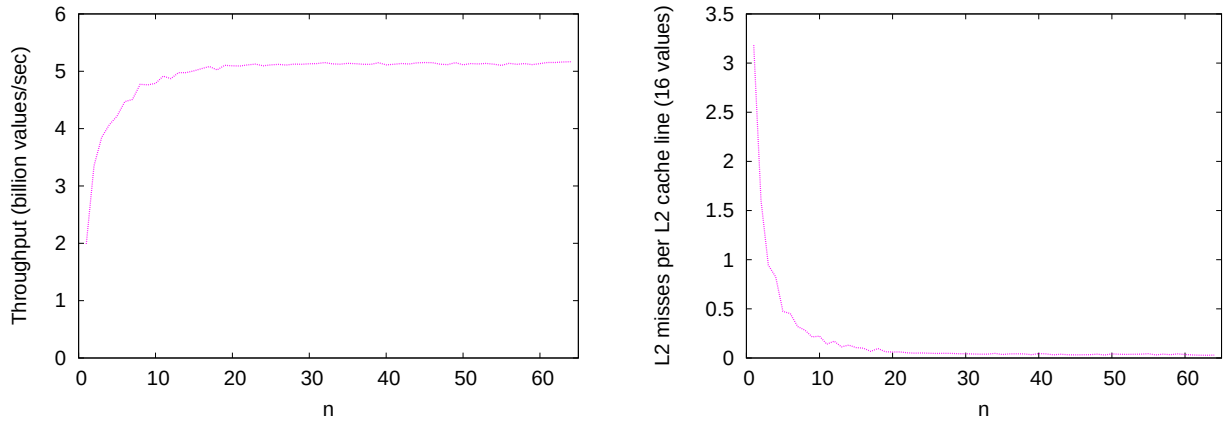
The LMQ entry for the prefetched location is retired once the prefetch instruction is complete, well before the prefetched data is actually retrieved. As a result, read interference is avoided. The performance of the prefetching method (implemented by simply using the `case` statement variant mentioned above) is shown in Figure 4. The prefetch-based method performs even better than the variable stride method, because it eliminates the interference without destroying the convoying behavior. Convoying does have benefits, namely the amortization of cache misses across threads.

4. THE LOCAL SHUFFLE

As outlined in Section 1, even an interference-free convoy may be underutilizing the available bandwidth of the machine. In this section, we aim to improve the performance of convoys by allowing threads to read different data items concurrently. However, unlike the variable-stride method in Section 3, we wish to preserve the convoying behavior so that cache misses are amortized over many threads.

The essence of the idea is to think of the scan at two levels: a coarse granularity and a fine granularity. At the coarse granularity, all threads progress in the same sequence, moving from segment to segment together as a convoy. At the fine granularity, however, threads iterate within the segment using different access sequences. If the segment size

[4]In general, we could not predict when the compiler will generate code with prefetch instructions, and the compiler options to "force" prefetch optimization were only partially effective. Manually inserting prefetch instructions is possible, but requires significant tuning.



(a) Throughput (b) L2 misses

Figure 5: Local shuffle on the T2 with 64 threads, for various n.

is small relative to the lowest-level cache, then all threads in the convoy should be able to benefit from other threads' cache misses within the segment. They remain in the convoy because any thread that gets too far ahead will suffer cache misses, while the trailing threads encounter cache hits.

Like the variable stride method, we again use a batch of 1024 values as the unit of access. By using a reasonably large unit, we allow processors with hardware prefetching to benefit from fixed-stride access. We define *shuffle groups* to be disjoint sets of n contiguous batches of array values. For example, the first shuffle group would contain the first $1024n$ values, the second would contain the next $1024n$ values, and so on.

Threads all move from shuffle group to shuffle group in the same sequence. Within a shuffle-group, threads proceed in a fashion similar to the variable-stride method using different strides, each relatively prime to n. The start batch within the shuffle group is set to $i \mod n$ for thread i.

The value n must satisfy several constraints. It must be small enough that $1024n$ values can fit well within the lowest-level cache. For example, with 4-byte values and a 4MB cache, n can be at most 1024. n should be large enough that threads are spread across different regions within the shuffle group. Nevertheless, once n reaches a certain point, the memory bandwidth will be saturated, and additional spreading will not improve performance.

For the Sun T2, we tried various n for the 64-thread case, and obtained the performance shown in Figure 5(a) for the scalar sum query. It appears that $n \geq 16$ performs well in practice. The cache misses per input cache line are given in Figure 5(b). The low miss rates verify that convoys are forming and that misses are being shared between threads.

We set $n = 16$ and generate performance graphs for the local shuffle method at various numbers of threads. The results are shown in Figure 6, together with the prefetching and variable-stride results. The local shuffle performs well, exceeding the performance of even the prefetch method.

We repeated the local shuffle analysis for the Nehalem machine. With 16 threads on the Nehalem, the best performance was achieved with n in the range 3 to 5. We set $n = 3$ and measured the performance of the local shuffle method on the Nehalem, together with the basic forward scan and the variable-stride method. The results are shown in Figure 7. Note that since the Nehalem machine does hardware prefetching for fixed-stride access to memory, the forward scan on the Nehalem is analogous to the prefetch method on the T2. Further, the local shuffle performance benefits from hardware prefetching on the Nehalem, unlike on the T2.

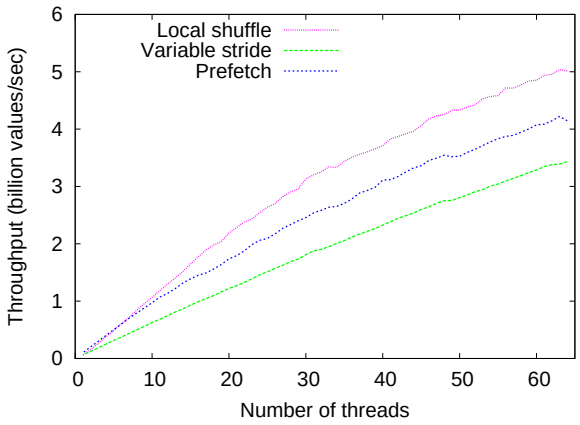
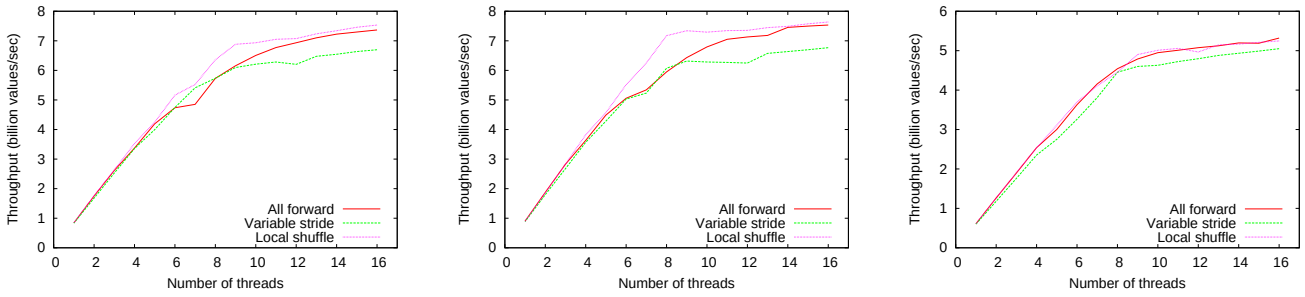


Figure 6: T2 throughput of the local shuffle method for various numbers of threads.

The variable-stride method performance levels off at about 8 billion values per second, which corresponds precisely with the 32GB/sec memory bandwidth rating of the machine. Since there is no convoying in the variable-stride method, the bandwidth wall limits the aggregate throughput. The forward scan performs better. Because of convoying, some of the memory accesses help service multiple threads, allowing an apparent processing rate that is higher than the machine's rated bandwidth. The local shuffle does even better, ensuring that all of the machine's memory bandwidth is effectively used.

In a final set of experiments, we repeated the performance comparison on the Opteron, and found essentially no difference in performance between the local-shuffle and all-forward methods. The Opteron performance is unchanged due to its cache design: the lower level caches are victim caches and prefetched data is sent directly to the L1 cache.



(a) Scalar aggregate of dimension table value

(b) Count aggregate with simple local selection

(c) Grouped count aggregate

Figure 8: Scan query variants on the Nehalem.

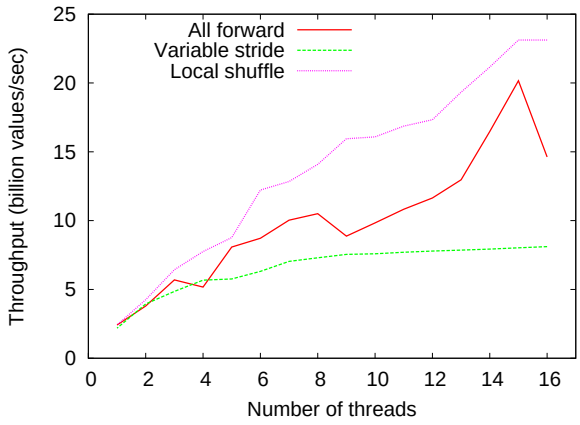


Figure 7: Nehalem scalar aggregate throughput of three methods for various numbers of threads.

There is less opportunity for threads to share prefetched data because the L1 caches are private to each thread.

5. MORE GENERAL QUERIES

So far, our performance results have focused on a single-column scalar aggregate query. This query is a good candidate for convoying behavior because it does only one operation per value, and needs just a single register for state. While a complete study of more general queries is beyond the scope of this paper, we also considered the following query variants:

1. The value being aggregated comes from an L1-resident dimension table referenced via a foreign key.
2. A simple equality filter with selectivity 0.1 is applied to fact table rows before a count aggregate.
3. Grouped count aggregation is performed, with group-by values used directly as offsets into a array of counts, so that no hashing is required. The group-by cardinality was 10.

Results for these queries on the Nehalem are shown in Figure 8. There was still an improvement for the local shuffle method relative to the other methods, but the gap was significantly smaller, and all methods performed slightly below the bandwidth rating of the machine. For all methods, there was little improvement after 8 threads. Since there is more work being performed for each record, the memory transfer cost represents a smaller fraction of the overall work done by the query.

6. CONCLUSIONS

We have demonstrated that even for read-only workloads, there are performance hazards associated with convoys. The two hazards identified here are *interference* on the Niagara architecture, and an underutilization of the memory bandwidth. We have demonstrated solutions to these problems, proposing the local shuffle method to preserve the global convoying behavior while spreading out the local access pattern.

As the number of cores per chip increases in future, issues such as convoys will become even more significant. Dealing with associated performance hazards will be critical for the efficient parallel utilization of the available resources.

7. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD '06*, pages 671–682, 2006.

[2] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The convoy phenomenon. *SIGOPS Oper. Syst. Rev.*, 13(2):20–25, 1979.

[3] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[4] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye. Automatic contention detection and amelioration for data-intensive operations. In *SIGMOD*, 2010.

[5] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, pages 567–574, 2001.

[6] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core CPUs. *Proc. VLDB Endow.*, 1(1):610–621, 2008.

[7] Sun Microsystems. Opensparc T2 core microarchitecture specification, 2007.

[8] J. Zhou et al. Improving database performance on simultaneous multithreading processors. In *VLDB*, 2005.

	Sun T2	Dual Intel Nehalem Xeon X5550	Dual AMD Opteron 2350
Chips	1	2	2
Cores	8	8	8
Threads	64	16	8
Clock Frequency (GHz)	1.2	2.66	2
RAM (GB)	32	24	16
RAM type	667 MHz DDR2 ECC	1333 MHz DDR3 ECC	667 MHz DDR2 ECC
Cache line size (bytes)	16 (L1), 64 (L2)	64	64
Lowest-level cache size	4MB (L2)	8MBx2 (L3)	2MBx2 (L3)
Associativity	16	16	32
Latency (cycles)	20	39–49	38–43
Middle-level cache size		256KBx8 (L2)	512KBx8 (L2)
Associativity		8	8
Latency (cycles)		10–11	12–15
L1 data cache size	8KBx8	32KBx8	64KBx8
Associativity	4	8	2
Latency (cycles)	2	4	3

Table 1: Characteristics of test platforms. Cache latency ranges indicate that different measurements have been reported by different sources.

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

APPENDIX

A. EXPERIMENTAL CONFIGURATION

We test the performance of our algorithms using three modern machines: the Sun Niagara T2, the Intel Xeon (Nehalem), and the AMD Opteron (Barcelona). The characteristics of these machines are summarized in Table 1. The same code base was used on all platforms, compiled with `g++`[5] under maximum optimization and using pthreads for parallelism. All experimental measurements are computed as the average over ten runs at each data point.

The underlying fact table is stored as an array of 4-byte integer values. The size of the arrays used in the experiments was 8 million values, four times larger than the largest of the lowest-level caches of our target machines. Larger inputs did not significantly change the performance graphs, and would have been more time-consuming to run for a given number of passes through the input.

On the Nehalem, with 16 threads, we repeated each query about 250 times in each thread. A budget of 16*250 passes was allocated, and each time a thread started over, it decremented this budget. It was therefore possible for some threads to do a few more passes than others. On the T2, with 64 threads, we repeated each query about 25 times in each thread. We noticed that it sometimes took several passes of the data before convoys reliably formed. When we used only 5 passes per thread, there was some variability in the performance results presumably due to occasional instances where convoys formed late.

B. BACKWARDS PROCESSING

In [8], the authors use a helper thread to preload data into the cache for the main worker thread. The work-ahead-set contains information about all of the pending computations that the main thread has in flight. One can think of a pending computation as a single stage of a multistage operator, applied to a single row. The main thread is responsible for adding and removing items to/from the work-ahead set in a first-in first-out fashion. The helper thread traverses the work-ahead set with the aim of making the data needed by the pending computations cache-resident.

When the helper thread proceeded ahead of the main thread, there were cases where the main thread caught up to the helper thread. This is a convoying effect: the helper thread suffers cache misses while the main thread encounters cache hits, so that the main thread progresses faster through the data. As a result, the helper thread and main thread read/wrote the same cache line, triggering the expensive memory-ordering pipeline flush.

By making the helper thread proceed backwards, the convoy effect was avoided. As long as the work-ahead-set's active data was smaller than the cache size, it still helped to preload data that was going to be accessed relatively far into the future. The main thread encountered some cache misses for the segment of the input that the helper thread had not yet reached. Nevertheless, the helper thread was able to take a substantial fraction of the cache miss burden away from the main thread.

[5]Version 4.3.3 on the Nehalem, version 4.2.4 on the Opteron, and version 4.0.4 on the T2.