Scalable Aggregation on Multicore Processors

Yang Ye, Kenneth A. Ross*, Norases Vesdapunt
Department of Computer Science, Columbia University, New York NY
(yeyang,kar)@cs.columbia.edu, nv2157@columbia.edu

ABSTRACT

In data-intensive and multi-threaded programming, the performance bottleneck has shifted from I/O bandwidth to main memory bandwidth. The availability, size, and other properties of on-chip cache strongly influence performance. A key question is whether to allow different threads to work independently, or whether to coordinate the shared workload among the threads. The independent approach avoids synchronization overhead, but requires resources proportional to the number of threads and thus is not scalable. On the other hand, the shared method suffers from coordination overhead and potential contention.

In this paper, we aim to provide a solution to performing in-memory parallel aggregation on the Intel Nehalem architecture. We consider several previously proposed techniques that were evaluated on other architectures, including a hybrid independent/shared method and a method that clones data items automatically when contention is detected. We also propose two algorithms: partition-and-aggregate and PLAT. The PLAT and hybrid methods perform best overall, utilizing the computational power of multiple threads without needing memory proportional to the number of threads, and avoiding much of the coordination overhead and contention apparent in the shared table method.

1. INTRODUCTION

The number of transistors in microprocessors continues to increase exponentially. Power considerations mean that chip designers cannot increase clock frequencies. Instead, chip designers have shifted the design paradigm to multiple cores in a single processor chip. Application developers thus face the challenge of efficiently utilizing the parallel resources provided by these multi-core processors.

We consider data intensive computations such as aggregation, a central operation in database systems. The essential question is whether such computations should be organized on multi-core processors as disjoint independent computations, or as coordinated shared computations. Independent computations avoid coordination overhead and contention, but require resources proportional to the number of threads. Compared with shared computations, independent computations on n threads have effective use of only $1/n$th of the cache.

Shared computations allow multiple threads to use a common data structure, meaning that larger data sets can be handled with the same amount of RAM or cache. This enhanced scalability comes with the burden of making sure that threads do not interfere with each other, protecting data using locks or atomic instructions. This coordination has overheads, and can lead to contention hot-spots that serialize execution.

1.1 Prior Work

Adaptive parallel aggregation has been investigated in the context of shared-nothing parallelism [13]. On chip multiprocessors, previous papers have proposed a variety of methods for aggregating a large memory-resident data set [7, 4]. These papers focused on the Sun Niagara T1 and T2 architectures because they offered a particularly high degree of parallelism on a single chip, 32 threads for the T1 and 64 threads for the T2. The high degree of parallelism made issues such as contention particularly important. We examine four previous hash-based algorithms. These proposals are:

Independent. Perform an independent hash-based aggregation on disjoint subsets of the input, and combine data from the tables at the end. Each active thread has its own full-sized hash table.

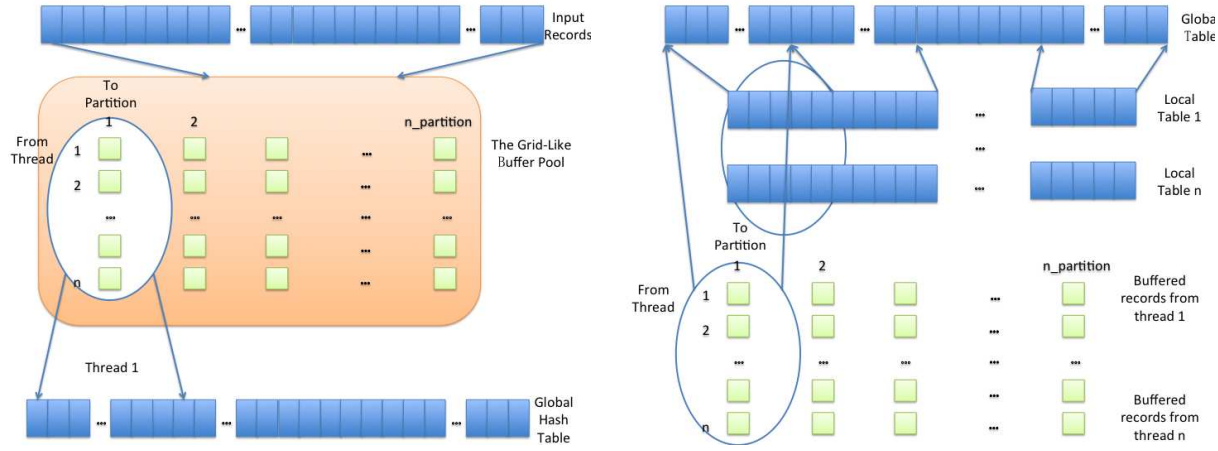
Shared. Use a single common hash table for all threads, and protect access to data elements by using atomic instructions to update hash cell values.

Hybrid. In addition to a global hash table, each thread has a small private local table that is consulted first. If there is a match in the local table, then the update happens there without atomic instructions. A miss in the local table leads to an eviction of some partial-aggregate element from the local table into the global table, and the creation of a new aggregate for the record in the local table.

Contention Detection. The programmer specifies a set of basic operations to handle the initialization and combination of aggregate values. The system automatically detects contention on frequently accessed data

*This work was supported by the National Science Foundation under awards IIS-0915956 and IIS-1049898.

Proceedings of the Seventh International Workshop on Data Management on New Hardware (DaMoN 2011), June 13, 2011, Athens, Greece.



(a) Partition-then-Aggregation Method (b) PLAT

Figure 1: Illustration of Partition-and-Aggregate method and PLAT

items, and clones them so that fewer threads access each copy, reducing contention. Infrequently accessed items are not cloned. A "local" variant of this scheme allows threads to make their own local clones when facing contention, while a "global" variant shares clones across a balanced number of threads.

The contention detection method counts the number of compare-and-swap attempts needed to update an item. Appendix A gives the code for an atomic add instruction that also returns this number of attempts. If this number is greater than a threshold (set to 1 by default), cloning is triggered. Note that the contention detection method allows computations more general than aggregation, and that this generality comes with a performance overhead [4].

1.2 Contributions

In an effort to broaden prior results, we aim in this paper to study aggregation on a more conventional architecture, the Intel Nehalem processor. Compared with the Niagara processors, the Nehalem processor has a higher effective clock speed per thread, and fewer threads (8 per chip, 16 in a dual-chip system).

We ported the code for the methods discussed above from the Niagara to the Nehalem architecture and quantified their performance. Some of the performance results are expected. The independent method performs well, but cannot scale to the largest data sets. The shared method performs poorly for small group-by cardinalities. As on the Niagara machines, contention for a small number of hash cells severely limits performance.

One unexpected result is that the contention detection mechanism failed to ameliorate the contention for small group-by cardinalities. Because there are fewer threads than on the Niagara, and because each thread operates much faster, observable contention (via a failed compare-and-swap) becomes relatively rare. As a result, clones are not generated at a sufficiently rapid rate. We also measure several indirect indicators of contention, including cache coherence overhead and MOMC events.

In light of these observations, we consider two additional aggregation methods that minimize thread sharing and partition work without creating a global hash table for each thread. These methods are illustrated in Figure 1.

Partition-and-Aggregate. First partition the input into fragments based on the group-by value, and aggregate each fragment on a single thread. Different fragments can be processed in parallel.[1] To avoid contention when writing the partition output, the partition buffers are arranged in a grid-like manner.

PLAT. Extending the Partition-and-Aggregate method, each thread is given a private local table in which to perform aggregation. Once this table fills up, input records that miss in the local table are partitioned as before. The local table is particularly useful when there is a small number of frequent group-by values. PLAT is an acronym for "Partition with a Local Aggregation Table."

The best methods overall are the hybrid method and PLAT. Both methods generally match the performance of the independent tables method for small group-by cardinalities, while being able to scale to larger data sets. For some distributions with group-by cardinalities somewhat beyond the L1 cache size, the hybrid method outperforms PLAT, because its local table adapts over time using dynamic eviction. PLAT does not evict data from the local table. On the other hand, PLAT outperforms hybrid when data cannot fit in the L3 cache because the partitioning enhances the locality in the second phase.

Our results show that on the Nehalem architecture, one can achieve the performance of independent computation without duplicating large hash tables across threads. As more threads become available in future generations of machines, the importance of such scalability increases. The use of a cache resident local table is a key element of achieving scalability while retaining high performance.

Our results also show that, despite efforts to hide architectural details, architecture matters. The contention detection method that worked well on the Niagara T2 machine does not work as well on the Nehalem processor because directly

[1]A version of the Partition-and-Aggregate method was previously evaluated on the Niagara architecture [12], but it did not outperform direct aggregation using a large hash table.

Platform	Sun T2	Intel Nehalem Xeon E5620
Operating System	Solaris 10	Ubuntu Linux 2.6.32.25-server
Processors	1	2
Cores/processor (Threads/core)	8 (8)	4 (2)
RAM	32GB	48 GB
L1 Data Cache	8KB per core	32 KB per core
L1 Inst. Cache	16KB per core	32 KB per core
L2 Cache	4MB, 12-way Shared by 8 cores	256 KB per core
L3 Cache	N.A.	12MB, 16-way Shared by 4 cores

Table 1: Experimental Platforms

measurable contention (via a failed compare-and-swap) is rare. We show that a lock-based implementation of the contention detection method does work well on the Nehalem architecture because the lock attempt coincides with the time-consuming cache load, making contention more easily observable.

The remainder of the paper is organized as follow. Section 2 presents an overview of the architecture and its influence on aggregation algorithms. Section 3 experimentally evaluates the algorithms. Section 4 concludes this paper.

2. ARCHITECTURAL OVERVIEW

With large main memories, main memory access has taken the role of I/O bandwidth as the new bottleneck in many data-intensive applications, including database systems [1, 2]. Caches are designed to speed up memory access by retaining frequently used data in smaller but faster memory units. There are multiple cache levels, each successively larger but slower. The higher level caches are typically private to a core, and the lowest level caches are typically shared between cores on a chip. The specific cache configurations of the Sun Niagara T2 and the Intel Nehalem processors are given in Table 1.

On a multi-chip machine, the caches of the various chips are kept consistent via a cache coherence protocol. Within a chip, the private caches also communicate to make sure that they store consistent results for items accessed by multiple threads.

On the Intel Nehalem machine, if one thread is updating a cache line and another thread wants to access the same cache line, there is a potential memory order hazard. To avoid potential out-of-order execution, the machine has to flush the pipeline and restart it. This event is called a "Memory Order Machine Clear" (MOMC) event. When an MOMC event is triggered, it can induce more MOMC events because of the delay caused by the first event. MOMC events can be quite expensive, and can effectively serialize execution. Previous research also discovered the influence of MOMC events on Intel's Pentium 4 SMT processor [3].

The L3 cache on the Nehalem, and the L2 cache on the Niagara T2 are inclusive, meaning that data in the higher level caches also reside in these caches. Thus the lowest-level cache is the natural place to resolve accesses within a chip. Accessing cache memory in another processor in multi-processor system is more complicated. The Nehalem uses the QuickPath Interconnect (QPI) to control the request to and from the memory of another processor. The Nehalem processor implements the MESIF (Modified, Exclusive, Shared, Invalid and Forwarding) protocol [5] to manage cache coherency of caches on the same chip and on other chips via the QPI. Because of the different path lengths to different kinds of memory, the Nehalem exhibits non-uniform memory access (NUMA) time. Table 2 summarizes the memory latencies of the Nehalem 5500 processor; this data is taken from [10]. (Our experimental machine uses 5620-series processors, but the latencies are likely to be similar.) Note that accessing L3-resident data becomes more expensive if the data has been modified by threads on a sibling processor.

Since the unit of transfer between levels of the memory hierarchy is the cache line, it is possible for false sharing to impact performance. False sharing occurs when two threads access disjoint items that happen to reside in the same cache line. Because of the performance pitfalls of modifying shared cache lines, the Intel optimization manuals recommend that sharing of data between threads be minimized [11]. Nevertheless, our goal is to take advantage of shared data to better utilize memory, and so some degree of sharing may be necessary.

3. EXPERIMENTAL EVALUATION

3.1 Experiment Setup

We consider an aggregation workload (count, sum, and sum squared) similar to that in [4, 7]. The query is

```
Q1: Select G, count(*), sum(V), sum(V*V)
    From R
    Group By G
```

where `R` is a two-column table consisting of a 64-bit group-by integer value `G` and a 64-bit integer value `V` for aggregation. The input size is $2^{28} \approx 270$ million records which fits in 4GB of RAM. We measure the throughput of aggregation as our performance metric. We consider a variety of input key distributions and vary the number of distinct group-by keys in each distribution. We use the synthetic data generation code from [7] (based on the work of Gray et al. [8]). The distributions are: (1) uniform, (2) sorted, (3) heavy hitter, (4) repeated runs, (5) Zipf, (6) self-similar, and (7) moving cluster.

3.2 Independent Table Method

The performance of the independent table method is particularly good for small group-by cardinalities: over 1,000 million tuples per second as shown in Figure 2 for 8 concurrent threads. The downward steps in performance occur as the group-by cardinality causes the active part of the hash table to exceed successive cache sizes.

The independent table method does not scale well because for 8 threads it needs 8 times the memory to accommodate hash tables for all threads. For instance, if records with the same key appear in the inputs of all threads, the threads all need to store this key in their individual tables and merge

Access Type	Exclusive			Modified			Shared			RAM
	L1	L2	L3	L1	L2	L3	L1	L2	L3	
Local	4	10	38	4	10	38	4	10	38	195
On Chip	65			83	75	38	38			
Across Chip	186			300			170			300

Table 2: Read Cache Latency for Different Access Types on the Nehalem 5500 Processor

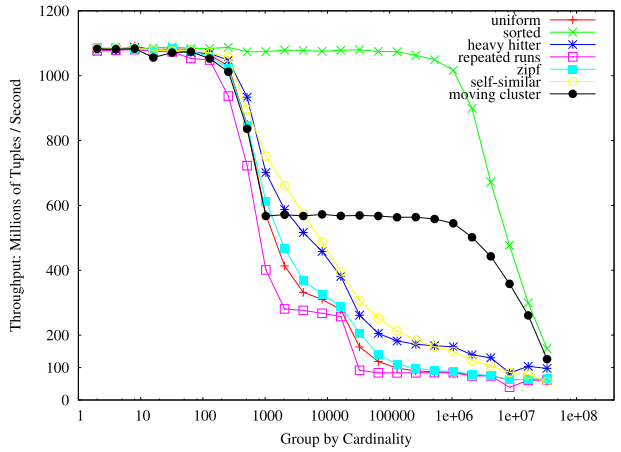


Figure 2: Independent Tables on all Distributions, 8 Threads

the results at the end. In our experiment, when the group-by cardinality is 2^{26}, the independent table method fails because the memory required is 64GB, more than the 48GB available on the machine.

3.3 Contention Detection and Shared Table

Figure 3(a) shows the performance of the contention detection aggregation method using uniform input on the T2 platform with all 64 threads. The No-Detection curve is a simple shared table with no cloning, which performs badly (due to contention) for small group by cardinalities. On the T2, both the local and global cloning methods are effective at eliminating the poor performance at small group-by cardinalities.

Figure 3(b) shows the performance of the contention detection aggregation on the Nehalem platform with 16 threads enabled. (The results were similar for 4 or 8 threads, even when all four threads were mapped to a single chip.) Contrary to our initial expectations, the contention detection method does not achieve similar benefits on the Nehalem platform as on the T2 platform. We kept track of the number of clones created. On the Nehalem processor, the cloning methods create relatively few clones, often just 2 or 3 even for very small group-by cardinalities with high levels of contention.

To understand why cloning is rarely triggered, we created a micro-benchmark in which a single shared variable is updated using an atomic compare-and-swap operation by all threads in parallel on both the T2 and Nehalem architectures. On the T2, the average number of failed compare-and-swaps was about 53, which seems reasonable given that there are 64 threads. On the Nehalem, however, this number was 0.1. This sharp difference is due to the smaller number of much faster threads on the Nehalem platform. (We measured the single threaded performance of the Nehalem to be seven times the single-threaded T2 speed for a simple scalar aggregation.)

Figure 12 in Appendix B shows the performance counters for the number of MOMC events per record, and the number of snoops hitting another cache in modified states (HITM events) per record as measured using the Vtune performance analysis tool [9]. The global method has higher event numbers than the local method.

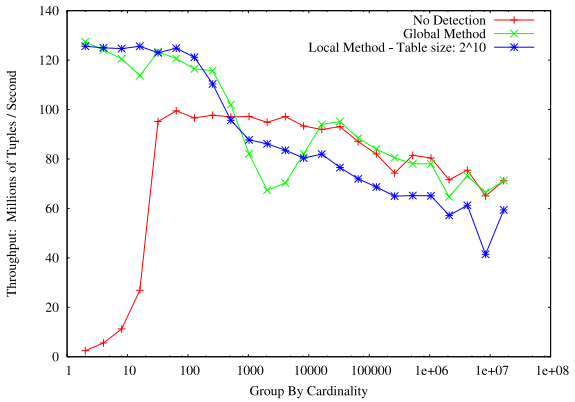
To understand whether clone triggering issues are the only performance issues, we re-ran the contention detection methods for small group-by cardinalities with the contention threshold set to zero, so that clones are always created. The performance of these methods is also shown in Figure 3(b). While the local method performance mirrored that of the T2, the global method performs even worse than without always cloning. After analyzing the code, we realized that the global method allocates clones in contiguous memory. Since the aggregate state is 24 bytes, smaller than a cache line, false sharing was occurring between clones. When we padded the aggregate state to be exactly 64 cache-line aligned bytes, the global performance improved, as shown in Figure 3(b).

3.3.1 *Locking vs. Atomic Operations*

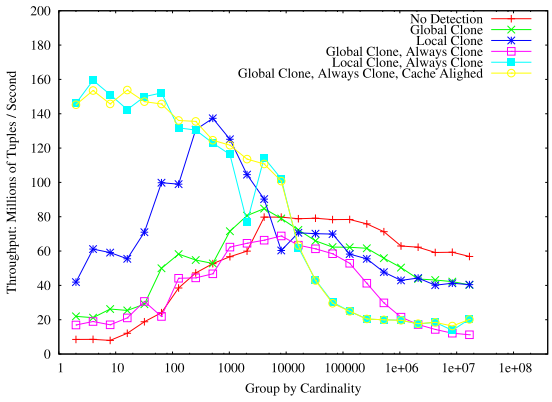
Cieslewicz et al. [4] suggested (but did not evaluate) an alternative contention detection framework using locks rather than atomic operations. A single mutex can protect all aggregate operations for a given group. A failed attempt to obtain a lock would indicate contention, analogously to a failed compare-and-swap. We implemented this alternative version of contention detection by using the `try_lock` system call that returns control to the calling context if a lock attempt is unsuccessful. Each failed `try_lock` adds to the contention count.

On the Niagara architecture, locks are more expensive than atomic operations; it takes six or more atomic operations before the lock-based implementation performs better [7]. This observation remains true in our contention detection method: On the T2, the atomic operation based method performs better (data not shown).

Figure 4 shows that for the Nehalem machine, the lock-based methods were able to detect contention and perform well even in cases where the original implementation failed to detect contention. The critical difference between the two methods is that the `try_lock` method induces a time-consuming coherency-related cache-miss to load the cache line containing the mutex and aggregate values. This delay makes lock contention much more likely to be observed. In contrast, the atomic operations decouple the initial cache miss of the aggregate values from the compare-and-swap, leading to the absence of observable contention as discussed above.



(a) T2, 64 threads



(b) Nehalem, 16 threads

Figure 3: Performance of Contention Detection on the T2 and Nehalem Processors, Uniform Distribution

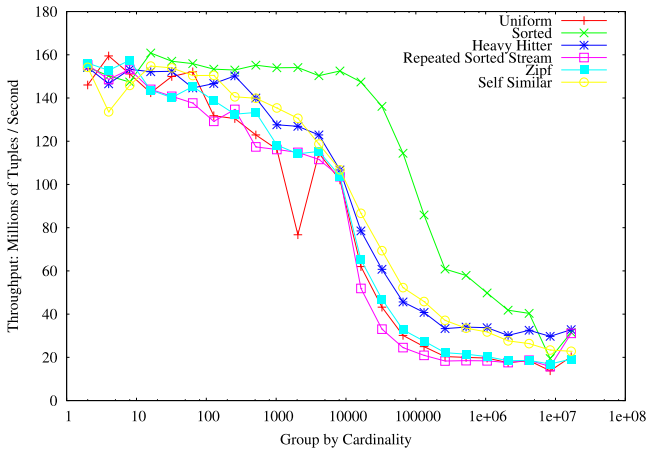


Figure 4: Contention Detection Method using `try_lock`

3.4 Partition-and-Aggregate

We first vary the fanout from 2^3 to 2^9-way partitioning on all distributions to empirically determine the optimal fanout. As shown in Figure 6, the performance difference is noticeable when the group-by cardinality is 2^{20} or larger: the throughput difference is up to 50 million records per second. Fewer partitions suffer from large partition size that needs more hash buckets for each thread and thus yields worse performance due to cache misses.

In our implementation, the size of the pointer is 8 bytes and the size of the buffer head each pointer points to is 64 bytes. Therefore, when the fanout is 256, the size of all buffer headers together is 16 KB (fanout times the header size). The pointers also need space and thus to fit into the L1 cache (32KB), 256-way partitioning is the largest fanout. Furthermore, the Nehalem architecture in our experimental setup has 2 levels of translation look-aside buffers for each core [6]. The L1 DTLB has 64 entries and the L2 TLB (Data and Instruction together) has 512 entries and thus 256-way partitioning does not thrash the TLB. But 512-way (2^9) partitioning suffers from both L1 cache misses and TLB misses. Beyond 512-way partitioning, larger partitioning fanout will lead to further performance degradation.

Therefore, we use 256-way partitioning, which is the optimal fanout for our experimental setup: the partitioning buffers fit into the L1 cache and the group-by cardinality in each partition is reduced to enhance data locality. The performance of the Partition-and-Aggregate method is stable at 120-140 million tuples per second for all distributions, as is shown in Figure 11 in Appendix B.

Figure 8 shows that the partition-and-aggregate method outperforms the independent table method when the group-by cardinality is larger than 2^{15}, which corresponds to the L3 cache limit in the independent table method.

3.5 The PLAT Method

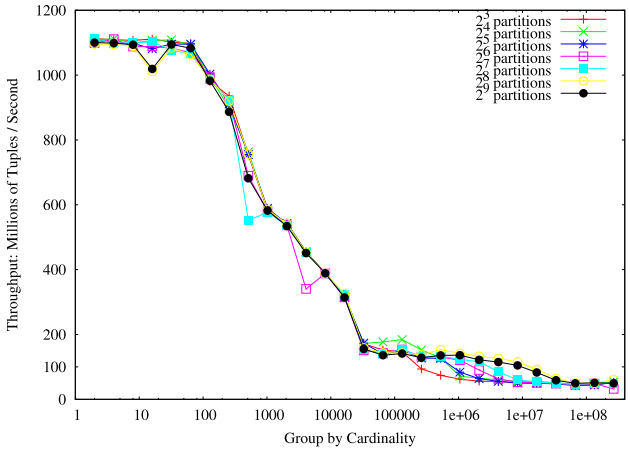
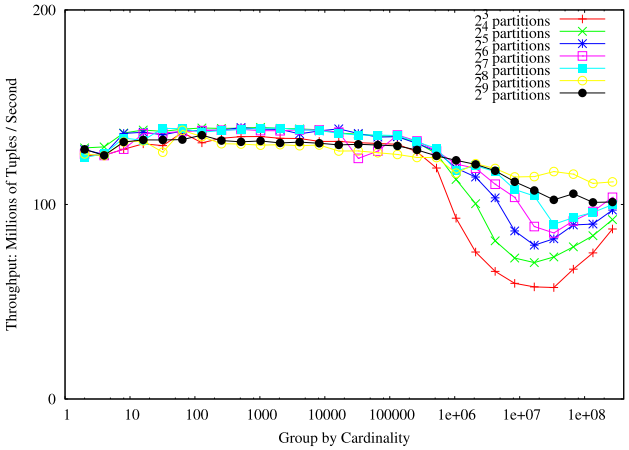
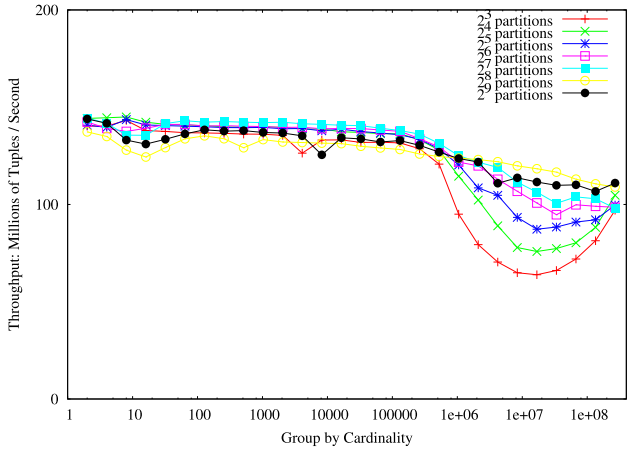


Figure 5: Varying the Fanout on PLAT, 8 Threads, Uniform Distribution

We run PLAT on different number of partitions (fanout) from 2^3 to 2^9. As shown in Figure 5, the results agree with the results from the partition-and-aggregate algorithm that the best fanout is 256. The performance difference is noticeable when the group-by cardinality is around 2^{20} or larger: the throughput difference is up to about 80 million records per second. When the group-by cardinality is very large (larger than 2^{24}) there is no discernible difference in performance with different fanouts. 512-way partitioning



(a) Uniform



(b) Zipf

Figure 6: Varying the Fanout on Partition-and-Aggregate Method, 8 Threads

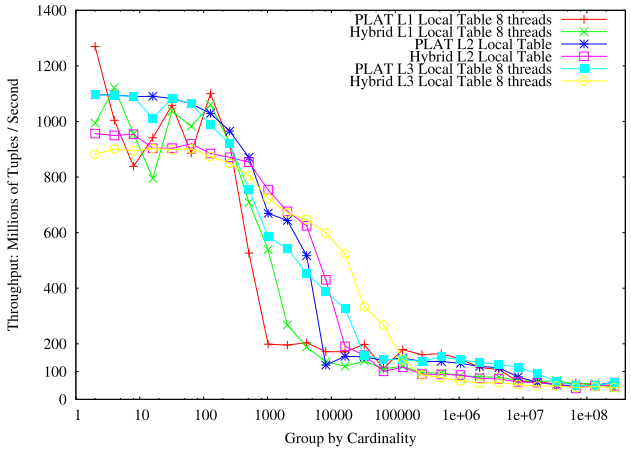


Figure 7: Impact of Local Table Size on PLAT and Hybrid Methods, Uniform Distribution, 8 Threads

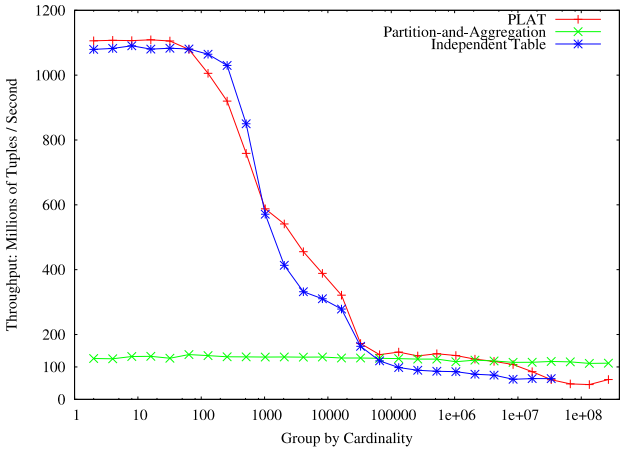


Figure 8: Comparison of Independent Table, Partition-and-Aggregate, and PLAT methods on Uniform Distribution, 8 Threads

performs slightly worse than 256-way partitioning.

The size of the local table in PLAT was set at 2^{15} to ensure L3-cache residence. Figure 7 shows that a smaller (L1-resident or L2-resident) local table performs slightly better for small group-by cardinality, but worse for cardinalities between 8000 and 30000.

Figure 10 shows the performance of the PLAT aggregation method. Figure 8 shows that, like partition-and-aggregate, PLAT outperforms the independent method on large group-by cardinalities by roughly a factor of 2, and scales to larger group-by cardinalities due to lower memory consumption.

One potential optimization we considered was turning off the local table if the hit rate to the local table is sufficiently low. However, even when the hit rate is very low the local table processing overhead was sufficiently small that no performance gain was apparent.

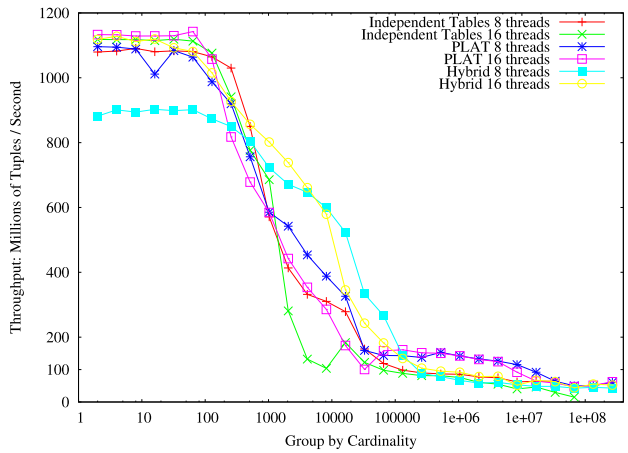
When the data is skewed, PLAT is able to aggregate the most frequent items in the local table in the first phase. Without such pruning, the partition-and-aggregate method will send all records with a frequent common key to one thread; this thread will become the performance bottleneck. As shown in Figure 11 and Figure 10, the partition-and-aggregate method performs worst on the heavy-hitter distribution whereas PLAT performs relatively well.

3.6 Hybrid Method

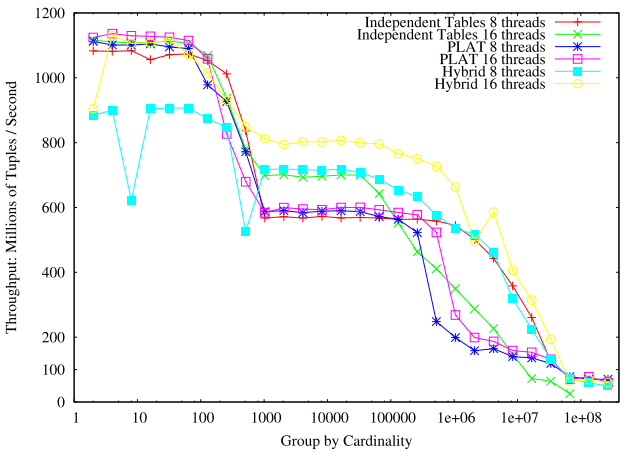
The Hybrid method [7] reduces sharing by making use of a small local table for each thread. The table also adapts to the input because it spills the oldest entry to the global table when the bucket is full. Although it does not eliminate sharing completely, it effectively reduces sharing so that it has minimal impact on performance.

Figure 7 shows the impact of local table size on the performance of the hybrid method using the uniform distribution. An L1-resident local table can accommodate 2^9 entries, and an L3-resident local table can accommodate 2^{15} entries. The trade-off is similar to that of PLAT, and we choose to size the local table based on the L3-cache size.

We show the performance comparison of the hybrid method, independent method and PLAT in Figure 9 for uniform and moving-cluster distributions. (Comparisons for other distri-



(a) Uniform



(b) Moving Cluster

Figure 9: Comparing Hybrid with PLAT and Independent methods

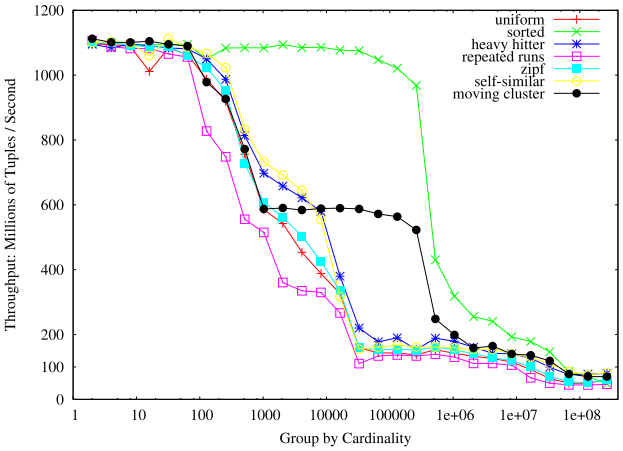


Figure 10: PLAT Method on all Distributions, 8 Threads

butions are shown in Figure 13 in Appendix B.)

The hybrid method does well at intermediate cardinalities relative to PLAT for two reasons. First, PLAT needs to store both a local table and partitioning data in cache, whereas the Hybrid method can use all of the cache just for the local table. Second, as is apparent for distributions with changing locality such as the moving cluster distribution, the local table in the hybrid method is able to adapt to the changing reference pattern. PLAT is much more sensitive to the initial data values, because it does not evict items from the local table.

On the other hand, PLAT does better at high cardinalities for distributions without locality. The investment of cache in the first phase for partitioning pays off at the second phase when aggregation has better locality.

3.7 Tuple Width

One subtle difference between the hybrid and PLAT caching schemes is that PLAT passes down an input record when there's a miss, whereas the hybrid method passes down a partially aggregated result. In the experiments above, an input record is 16 bytes wide, while a partially aggregated result is 32 bytes wide. In situations with poor locality, the hybrid method will be copying twice as many bytes. In this particular example, both kinds of record fit in one cache line, and so the difference between the two kinds of copying is unlikely to be significant. However, one can imagine more extreme examples in which (a) many aggregates are computed from a few columns, or (b) a few aggregates are computed based on many columns. Case (a) would favor PLAT, whereas case (b) would favor the hybrid method.

4. CONCLUSIONS

We have investigated the performance of various aggregation methods on the Nehalem processor. Unlike the Niagara processors previously studied, the Nehalem performance is sensitive to data sharing. This difference is fundamentally due to different memory models. The Niagara processor does not enforce memory access order: it is the responsibility of the programmer to insert suitable `fence` instructions if a particular order is required. For computations like aggregation the order of operations is not important. In contrast, the Nehalem processor includes hardware to detect and resolve potential out-of-order execution. For general purpose computing such out-of-order events are rare, as long as threads do not modify each others' data. For aggregation, such considerations make sharing of data more expensive than alternatives that do not share data.

We also showed that detecting contention can be challenging on the Nehalem processor. Explicit contention detection via failed compare-and-swap operations is not sufficient. An alternative implementation based on the `try_lock` primitive was effective, illustrating that different architectures call for different implementation primitives for optimal performance.

It may be possible to further improve the performance of the methods presented here. For example, using a staged computation, prefetching can overlap the latency of multiple cache misses [14].

As the number of cores continues to increase in future processors, techniques such as the ones studied here will become even more important for efficient machine utilization.

5. REFERENCES

[1] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory Access. In *VLDB*, 1999.

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

[3] J. Zhou et al. Improving database performance on simultaneous multithreading processors. In *VLDB*, 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] D. Kanter. The Common System Interface: Intel's Future Interconnect. http://www.realworldtech.com/page.cfm?ArticleID=RWT082807020032&p=5

[6] D. Kanter. Inside Nehalem: Intel's Future Processor and System. http://www.realworldtech.com/page.cfm?ArticleID=RWT040208182719&p=7

[7] J. Cieslewicz and Kenneth A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, 2007.

[8] J. Gray et al. Quickly generating billion-record synthetic databases. In *SIGMOD*, 1994.

[9] D.d Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. http:/software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[10] D. Molka, D. Hackenberg, R. Schone, and M.S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[11] Intel 64 and IA-32 Architecture Optimization Reference Manual. www.intel.com/Assets/ja_JP/PDF/manual/248966.pdf.

[12] J. Cieslewicz, and K. A. Ross. Data Partitioning on Chip Multiprocessor In *DaMoN*, 2008.

[13] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *SIGMOD*, 1995.

[14] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving Hash join performance through prefetching. In *Proc. Int. Conf. Data Eng.,* 2004

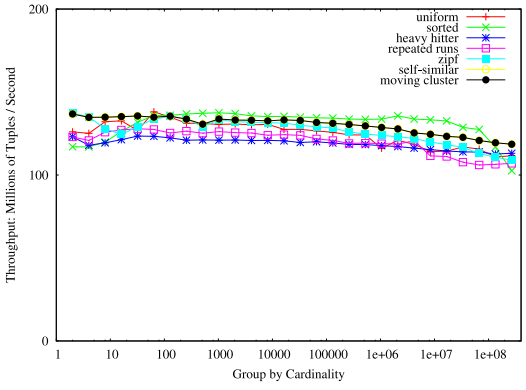


Figure 11: Partition-and-Aggregate Method on all Distributions, 8 Threads, 256-way partitioning

APPENDIX

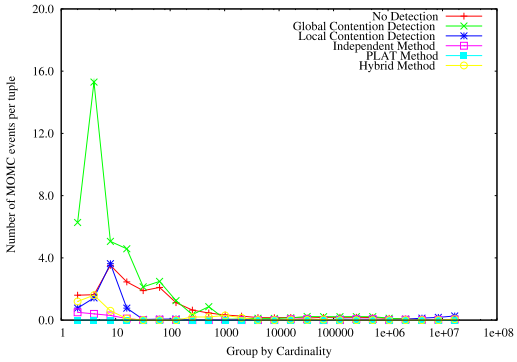
A. CONTENTION DETECTION DETAILS

The code for an atomic 64-bit add using the compare-and-swap primitive is given below as a `C` intrinsic using x86 assembly language. The `cmpxchg` opcode is the compare-and-swap instruction, and the `lock` prefix requires the instruction to be executed atomically.

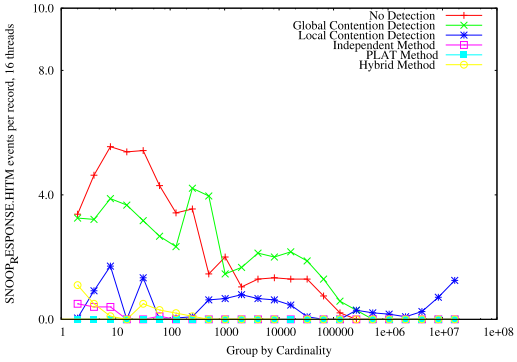
```
inline void atomic_add_64(volatile uint64_t *des,
                          int64_t src)
{
 __asm__ __volatile__("spi1:\tmovq %0, %%rax\n\t"
            "movq %%rax, %%rdx\n\t"
            "addq %1, %%rdx\n\t"
            "lock\n\t"
            "cmpxchg %%rdx, %0\n\t"
            "jnz spi1\n\t"
            :"=m"(*des)
            :"r"(src),"m"(*des)
            :"memory","%rax", "%rdx"
            );
}
```

B. ADDITIONAL EXPERIMENTS

Figure 11 shows the performance of the partition-and-aggregate method with 256-way partitioning. Figure 12 shows MOMC and HITM performance counter measurements for all methods. Figure 13 shows the performance of the various methods on a variety of distributions.

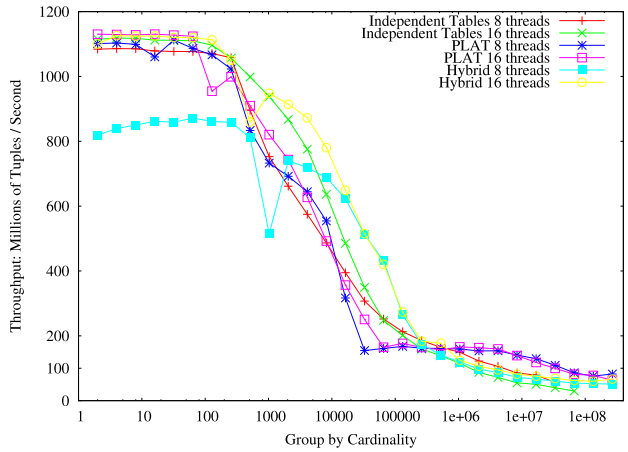


(a) MOMC events

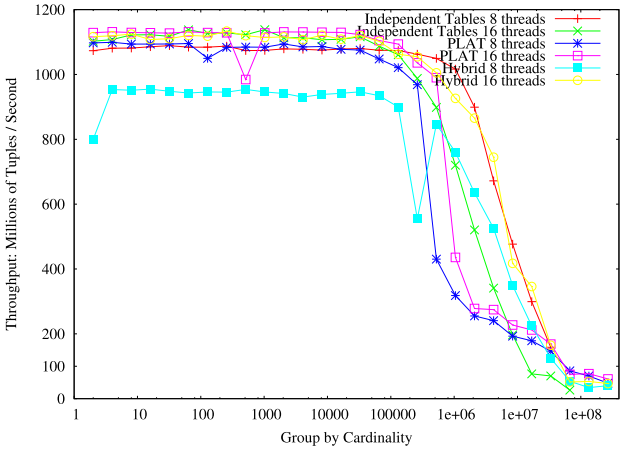


(b) Cache HITM events

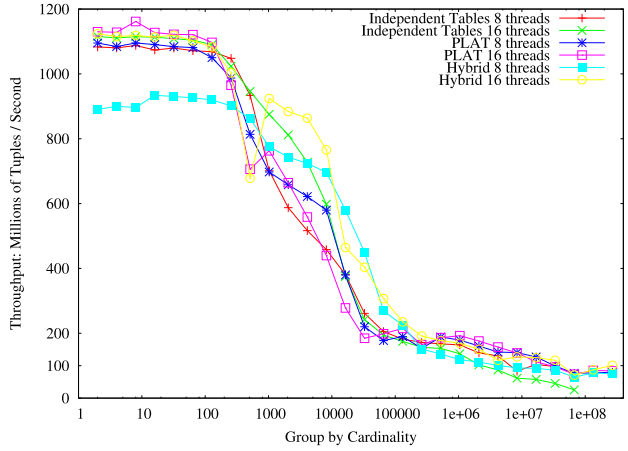
Figure 12: Performance counter events for all methods, 16 threads



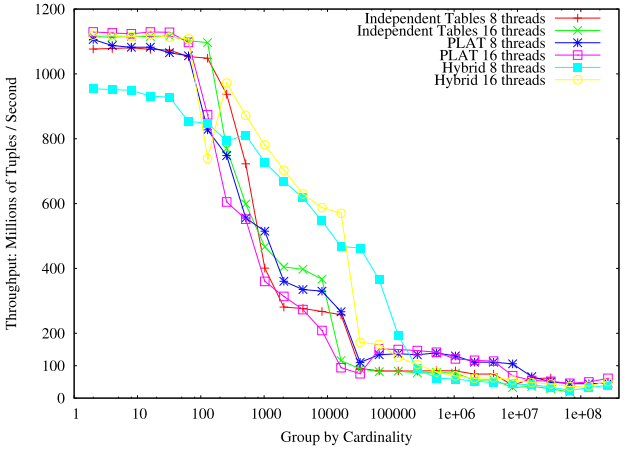
(a) Self-similar



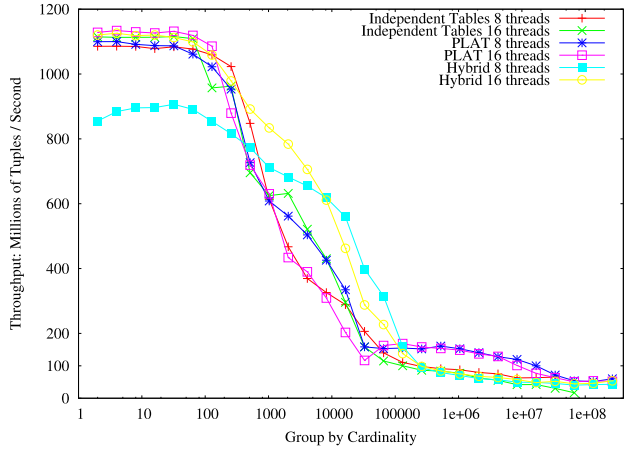
(b) Sorted



(c) Heavy Hitter



(d) Repeated Runs



(e) Zipf

Figure 13: Independent Tables, PLAT, and Hybrid, on various data distributions.