Adaptive Aggregation on Chip Multiprocessors

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

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

ABSTRACT

The recent introduction of commodity chip multiprocessors requires that the design of core database operations be carefully examined to take full advantage of on-chip parallelism. In this paper we examine aggregation in a multi-core environment, the Sun UltraSPARC T1, a chip multiprocessor with eight cores and a shared L2 cache. Aggregation is an important aspect of query processing that is seemingly easy to understand and implement. Our research, however, demonstrates that a chip multiprocessor adds new dimensions to understanding hash-based aggregation performance—concurrent sharing of aggregation data structures and contentious accesses to frequently used values. We also identify a trade off between private data structures assigned to each thread versus shared data structures for aggregation. Depending on input characteristics, different aggregation strategies are optimal and choosing the wrong strategy can result in a performance penalty of over an order of magnitude. We provide a thorough explanation of the factors affecting aggregation performance on chip multiprocessors and identify three key input characteristics that dictate performance: (1) average run length of identical group-by values, (2) locality of references to the aggregation hash table, and (3) frequency of repeated accesses to the same hash table location. We then introduce an adaptive aggregation operator that performs lightweight sampling of the input to choose the correct aggregation strategy with high accuracy. Our experiments verify that our adaptive algorithm chooses the highest performing aggregation strategy on a number of common input distributions.

1. INTRODUCTION

The number of transistors in microprocessors continues to increase exponentially. Until recently, microarchitects have used larger transistor budgets to achieve higher clock rates, increase cache sizes, and exploit instruction level parallelism (ILP). Higher clock rates have become problematic because of power consumption and heat dissipation issues as well as diminishing returns associated with achieving more ILP. This has caused a paradigm shift in microarchitecture design away from faster uniprocessors toward chip multiprocessors (CMP). For at least the next several years, growth in processor performance will depend on increased thread level parallelism (TLP) [13]. Chip multiprocessors differ from symmetric multiprocessor (SMP) machines in that the multiple cores on a chip often share cache resources and off-chip bandwidth, whereas SMP machines only share physical memory. On-chip cache coherency between CMP cores is much faster than cache coherency between SMP processors.

The available commodity chip multiprocessors can be divided into two groups: a "fat camp" and a "lean camp"[12]. Fat camp multiprocessors have a few wide issue, out-of-order cores with relatively high clock rates. These cores resemble uniprocessors and often share a large L2 cache. Lean camp processors have simpler pipelines and execution units, typically run at somewhat slower clock rates, and are more energy-efficient. The focus of these processors is on providing high overall chip throughput, a feature that is important to OLAP database operations, rather than low latency for single-threaded applications. Because the lean camp processors are simpler, more cores fit on a processor die, thus allowing for more on-chip thread contexts. In this paper we investigate aggregation on a lean camp chip multiprocessor, the Sun UltraSPARC T1. The T1 currently offers the most on-chip thread contexts of any commodity processor, allowing us to explore on-chip TLP in database operations by using many threads on real hardware. Although the T1 has some limitations, such as floating point capability, the parallelism issues experienced with this processor are generalizable to future "lean camp" processor designs. Section 5 contains a discussion of the T1.

Aggregation is a commonly used operator in database systems, particularly for queries typical of On-Line Analytical Processing (OLAP). Aggregation is also useful in other contexts. In stream processing or network monitoring applications, running aggregates of stream data may be maintained so that up-to-date summaries of the data can be generated.

When aggregates are applied to large input streams, the aggregate operator can be a bottleneck, so it is important to make aggregate processing as efficient as possible. At first, aggregation seems simple to implement. Because the T1 shares the L2 cache among its cores, a shared hash table data structure is an obvious solution—a shared hash table

*Supported by a U.S. Department of Homeland Security Fellowship
†Supported by NSF Grant IIS-0534389

VLDB '07, September 23-28, 2007, Vienna, Austria.

can be built and indexed by a hash function of the group-by columns. Each hash cell contains one value per aggregate function being computed. Each input record is hashed to a hash cell, and the corresponding aggregate values are updated. If collisions occur at a bucket, hash cells are added to an overflow list for that bucket.

The parallelism inherent in multicore systems adds one layer of complexity. On the T1, there are 32 threads accessing the hash table concurrently. In order to guarantee consistency, some form of mutual exclusion needs to be enforced to prevent destructive concurrent updates to the same hash cell or to data structure elements such as pointers in the hash bucket.

We study two kinds of concurrency control for parallel access to the hash table. The first employs locking primitives provided by the operating system. The second kind of concurrency control takes advantage of the presence of operations that are guaranteed by the system to be atomic. For example, there is an atomic 64-bit add instruction that can be used to compute sums and counts. Since addition is commutative, the order in which the records are processed does not matter. We show how other aggregates, such as minimum and maximum, can be implemented using atomic primitives, even in the absence of direct atomic min or max instructions. Our performance study suggests that the use of atomic operations is more efficient on the T1 than locking.

A related performance problem for concurrent access is *contention.* When many threads attempt to modify the same hash cell, a contention bottleneck can occur. (Even with the atomic implementation above, there is a time-window during which a competing thread must stall.) As we shall see experimentally, such contention bottlenecks can decrease throughput by an order of magnitude.

An alternative implementation to the shared hash table would be to give each thread its own separate hash table. At the end of the input stream, the data from the various tables would be combined. The advantage is that concurrency control on each table is unnecessary, and contention is not a problem. The disadvantage is that it requires 32 times as much memory as the shared approach. As a result, memory barriers such as the L2 cache size and the RAM size are reached with much smaller group-by cardinalities than with the shared table. Nevertheless, for small group-by cardinalities, we demonstrate that the performance of the independent hash tables is superior to the shared table.

Ideally, we would like to get the best of both worlds: the performance of separate tables for small group-by cardinalities, and the performance of a shared table for large cardinalities. However, group-by cardinalities do not tell the whole story. The reference pattern of group-by values in the input stream plays a major role in the performance of the different algorithms.

The first and most obvious characteristic of this reference stream is locality. If there is temporal locality (a group-by value is repeated within a short time window) or spatial locality (a small number of group-by values) then there is potential for cache reuse. Since cache misses are a large cost factor for modern processors, locality is typically desirable. However, temporal locality can also be undesirable, such as when it causes contention in the shared hash table. This contention can happen even when there is only moderate overall locality, for example, when there are a few "heavy hitters" in the distribution.

In order to distinguish between these situations, we propose to sample small pieces of the input stream during the aggregate computation. After the sampling phase is complete, regular aggregation is performed using a suitable variant of hash-based aggregation. The sampling phase tries to identify three measures of the input stream:

1. The average *run length* of group-by values.
2. The expected L2 cache *miss rate.*
3. The *frequency* of the k most frequently-seen group-by values, for a suitably chosen k.

The way each of these statistics is used to select a suitable variant algorithm will be discussed in Section 4.

Our first contribution is a new hybrid aggregation algorithm that uses a common shared hash table, as well as small local hash tables that together fit in the L2 cache. For many kinds of input streams this hybrid algorithm obtains the benefits of the independent tables approach, while using much less memory.

A second contribution is an adaptive framework in which the data obtained during the sampling phase is used to choose an appropriate algorithm for that data. We demonstrate experimentally that the adaptive algorithm typically tracks the best performance of the shared, hybrid, and independent approaches as the group-by cardinality is varied. In some cases, the adaptive performance is superior to these basic approaches due to the optimized processing of runs of common group-by values. Our experimental study covers a variety of data distributions that occur in practice.

Another contribution is a thorough analysis of how time is spent when performing hash-based aggregation on the Sun T1. For large group-by cardinalities that do not fit in the L2 cache, pure computation consumes roughly 40% of the cycles. An additional 40% of the cycles are spent waiting for cache misses. For L2-resident aggregation, between 60% and 77% of the cycles are spent on instructions, and L1 misses are the most significant source of latency. These results confirm the results of [12] who identify L2 hit time as a significant performance issue on multicore processors.

The alternative to an adaptive aggregate operator is to have a collection of aggregate operators that are available to the query optimizer, and to allow the optimizer to choose a suitable algorithm based on each algorithm's cost function. The disadvantages of such an approach include: (a) the optimizer has to rely on compile-time rather than run-time statistics about the input distribution; (b) performance depends critically on parameters such as run-length and value frequency that are hard to model analytically for outputs of other database operations; (c) the optimizer's search space is larger because it has to consider multiple aggregation implementations rather than one; and (d) the optimizer does not necessarily have access to run-time configuration information, such as the number of threads available and the cache size on the target platform.

2. RELATED WORK

An adaptive query operator is one whose behavior depends on statistics and/or performance measurements obtained during query execution [6]. Adaptive operators have been proposed in several forms. [14, 18] gather cardinality estimates during query execution and reoptimize the query if

those statistics exceed certain optimizer-generated bounds. [2] routes individual records to operators based on running performance estimates of each operator. Sampling to improve join performance is presented in [3]. Adaptive parallel aggregation has been invesitgated in the context of shared-nothing parallelism [22]; our work presents novel adaptive aggregation techniques for chip multiprocessors.

Modern commercial database systems typically employ some form of shared-nothing or shared-memory parallelism [7]. A chip multiprocessor (CMP) is a shared-memory processor, but differs from symmetric multiprocessor (SMP) systems. On an SMP, accesses to common data elements in RAM must be mediated using a cache coherency protocol. Such protocols incur significant latencies (hundreds of cycles) and use resources such as bus bandwidth. In contrast, coherency on a CMP is maintained on-chip, using fast, dedicated hardware. As a result, parallel algorithms that might be impractical on an SMP may be efficient on a CMP.[1]

There has been much recent work on architecture-conscious database systems, e.g., [1, 17, 21]. Most of this work has focused on processors with limited or no TLP. Architecture-conscious database parallelism has been studied using non-traditional computing platforms such as graphics processors [10], network processors [8], and supercomputers [4]. The intra-operator parallelism presented in this paper may be supported using parallel input and output buffers [5]. [4] shows that high latency memory operations can be effectively mitigated by TLP on large shared memory systems with many concurrent threads per processor.

We use conventional hashing rather than variants of cuckoo hashing that have been shown to be more efficient on some processors [25, 20]. This is appropriate for the T1, because [25, 20] are trying to achieve ILP, while the T1 is focused on TLP. Further, [20] is optimizing for a machine with long branch misprediction latencies and hardware support for SIMD instructions. The T1 does not have noticeable branch latencies and does not provide efficient SIMD instructions.

3. AGGREGATION ALGORITHMS

Any aggregate operator has three phases. The first phase is the *startup* phase, in which all data structures are initialized. For example, in a hash table, the "valid bits" would be set to zero for every hash bucket. The second phase is the *computation* phase in which the input data is consumed and used to update the data structures. The third phase is *finalization* in which data structure elements are combined to generate a single final result. For example, if each thread has its own private hash table, then those tables must be merged during finalization. Finalization is also where algebraic aggregates are computed, such as deriving an average from a sum and a count.

We focus on the computation phase in our experimental study, since it is usually the most time-consuming component of the aggregation process. We divide the computation phase into *time-slices*. A time-slice corresponds to uninterrupted aggregation for a certain amount of time (or number of records). Between time-slices, other work may be scheduled for other operators, with the system rescheduling the aggregation operator for more work once more input records are available. Longer time-slices allow the system to benefit from temporal locality in both the data and instruction caches. A time slice also needs to be long enough that overheads such as context-switching and thread start-up are small, but short enough that the batch of input records to be processed is likely to fit within a reasonable RAM budget.

We devote all of the chip's resources to aggregation during a time-slice. It might be possible to devote some threads to aggregation and other threads to other operators, but such an approach is fraught with performance hazards. Since different operators will be accessing different data structures, there could be cache interference between them. Further, if one operator is writing to a structure that another operator reads, some form of synchronization between threads is required. Different operators running on the same core would share the same 16KB L1 instruction cache, and could end up thrashing. While these obstacles might be surmountable in some cases, we leave their study to future work.

In all experiments, we use multiplicative hashing [15] as our hash function. Multiplicative hashing is *universal*, meaning that it is provably robust to adversarial distributions. 64-bit integer multiplication is available on the T1 using an operation with an 11-cycle latency [24]. Multiplicative hashing was extremely simple to implement, and much more efficient on the T1 than other hash algorithms we tried.

Sort-Based Aggregation

Sort-based aggregation has higher complexity ($O(n \log n)$) than hash-based aggregation, whose expected complexity is linear in the number of records n. Further, materializing a complete sorted input may require I/O for large input streams: sorting is a blocking operator. In contrast, hash-based aggregation can pipeline from another database operator, one batch of records at a time. The only time that sort-based aggregation is likely to be competitive is when the input stream is already in sorted order. In that case, our adaptive algorithm will identify that the stream has runs of common group-by values and will have performance close to that of sort-based aggregation (see Section 4.1). As a result, we limit our investigation to hash-based aggregation.

Independent Hash Tables

The simplest parallel approach to aggregation is to give each thread its own independent hash table. An advantage of this approach is that threads do not concurrently access the same hash cell. As a result, the aggregation operator can avoid expensive synchronization primitives. The obvious disadvantage of this approach is memory consumption. As mentioned in Section 1, memory barriers such as the L2 cache size and the RAM size are reached with much smaller group-by cardinalities. Further, startup and finalization costs may also be significant.

Shared Table with Locking

If all threads access a common hash table, the memory requirements are less drastic than for the independent approach. However, accesses to hash cells must be protected to prevent a race condition between threads. One way to protect access is by using a lock, in the form of an OS-supported mutex. Locking is a general solution that can be applied to arbitrary aggregate functions. To simplify processing, we provide locks for each bucket, rather than for each cell.

[1] An SMP system composed of CMP processors would still have expensive off-chip cache coherency between processors. The Sun T1 does not support off-chip coherency and therefore can only be used in single CMP configurations.

Locks are not required during the search phase in which the input record's key is compared against keys in the bucket. Once a match is found, the bucket is locked to guarantee the atomicity of the update to the aggregate value. Locking is also needed when inserting a new element into a hash bucket.

Shared Table with Atomic Aggregation

Modern micro-architectures provide instructions with atomicity guarantees. These instructions are often used to implement synchronization primitives such as semaphores and mutexes. The Sun T1 compiler provides atomic intrinsics for integer addition, increment, bitwise OR and AND, as well as compare-and-swap [23, 19]. Most of these intrinsics are actually compound operations defined in terms of the hardware-supported compare-and-swap operation. While more expensive than ordinary instructions, these instructions are much cheaper than lock operations.

The standard SQL aggregates—Count, Sum, and Average—can be computed using atomic adds or increments. Because addition is commutative, the order in which the updates happen is not important. The following code fragment shows the use of the atomic compare-and-swap instruction for computing the minimum (maximum is analogous).

```
uint64_t current = bucket->min;
uint64_t old = current - 1;
while(new_value < current && current != old){
  old = current;
  current = atomic_cas_64(&(bucket->min),old,new_value);
}
```

The `atomic_cas_64` operation compares the contents of the destination (first argument) with a value (second argument) for equality. If the two values are equal, a new value (third argument) is swapped with the destination. The old contents of the destination are returned as a result of the operation regardless of whether the swap took place.

While many commonly used aggregates are definable using atomic primitives, some aggregate functions (such as user-defined functions) may not be. In such a case, locking is the only option for protecting access. The cost of atomic operations is proportional to the number of aggregates, while the cost of locking is fixed. As a result, we expect that as the number of computed aggregates increases, locking will eventually outperform atomic operations.

Hybrid Aggregation

It is possible to get most of the benefits of the independent tables approach without the overwhelming memory requirements. Each thread has its own small local table, sized so that the total space occupied by all 32 local tables is less than the L2 cache size. Input records first go to the local table, and are accumulated there if the group-by value is present. Since the local table is private, no expensive protection mechanisms are necessary and the update happens very efficiently.

When the group-by value is not present, the current input record is added to the local table. If the bucket it maps to is full, the oldest entry is first removed and "spilled" into a global shared table. The shared table can use either locking or atomic operations for protection against updates from other concurrent threads. The structures used by the hybrid algorithm on the T1 are described in Figure 1.

One would expect the hybrid algorithm to perform well for data sets having good L2 cache locality, even if the group-by cardinality is high. On the other hand, the hybrid algorithm likely performs worse than a shared table approach in the absence of locality, since there is extra work on top of the update to the shared table. In Section 6, we will verify these expectations empirically.

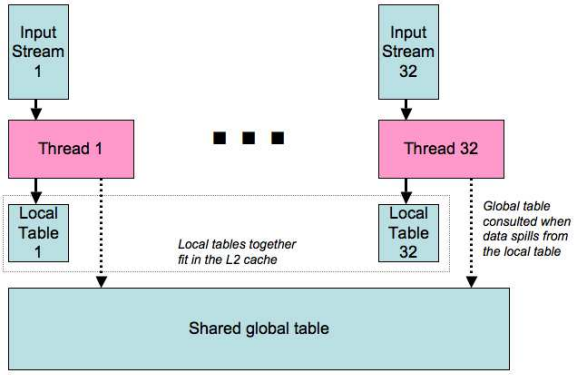


Figure 1: Data structures for hybrid aggregation.

4. MODELING PERFORMANCE

In order to tailor the aggregation to the input distribution, we measure certain statistics derived from the input stream. These statistics allow us to model the potential impact of run-optimization (Section 4.1), cache behavior (Section 4.2) and contention (Section 4.3). The statistics will be gathered by each thread during short sampling windows within the input stream. During sampling, aggregation is performed using the hybrid algorithm and statistics are gathered.

A restriction we place on this statistical analysis is that it be performed *locally*. In other words, the decision one thread makes about locality and contention should be made without coordination with other threads. This choice has the following important advantages:

- Coordination costs, which can be relatively high, are avoided. For example, spawning 32 threads takes about 5 milliseconds on our experimental platform.
- A coordinated scheme would require shared data structures that could themselves have contention points.
- Threads can proceed as fast as their input distributions allow. Threads that see locality will run much faster than threads that do not see locality. A thread that finishes one piece of work can then pick up another piece of work from the input queue, and continue processing without waiting for other threads to reach a coordination point. (Otherwise, one slow thread could cause many threads to stall.) Such a scheme achieves high thread utilization and good load balancing without explicit scheduling.

Because contention is a property of a collection of threads, not just a single thread, our local contention test will only approximate true contention. We will identify a sufficient condition for the *absence* of significant contention based on the following abstraction: "If all threads had probe streams like mine, would there be significant contention?" If the answer is "yes," then that thread is processed using the hybrid approach to eliminate contention on frequently accessed elements. If the answer is "no" for all threads, then it is

relatively easy to see that contention is avoided. Even if all threads had the same frequently-accessed elements, the global frequency of access to those locations in the hash table would be below the contention threshold.

4.1 Runs

Runs of consecutive tuples with the same group-by key often occur trivially in input sorted on the group-by key. Runs may also be created during query processing due to a join in which there are multiple matches for an outer relation record containing the group-by key.

When runs are present in the input, the aggregate(s) of the tuples in the run can be computed directly. This optimization avoids the latency of cache hits in the hash table and a hash computation for each tuple. When one run ends and another begins, the old run's accumulated aggregate is pushed into the hash table. Using this run-based aggregate is not universally optimal because the run-checking is pure overhead if the input contains few or no runs.

During the sampling phase, we compute the average run length. We determine a run-length threshold using a calibration experiment. On our experimental platform, run-based aggregation is best up to a group-by cardinality of 8 for uniformly distributed input. At this point we expect a run length of about $1 + 1/8 + (1/8)^2 + \ldots = 8/7$, which we use as the threshold for choosing run-based aggregation.

4.2 Modeling Cache Behavior

An algorithm running on a processor can be characterized at a cache level of the memory hierarchy by its *hit rate*. The hit rate is the proportion of accesses to the cache that are satisfied by the cache. The *miss rate* is the proportion of accesses that must be satisfied by levels below the cache in the memory hierarchy. Since misses are typically much more costly than hits, even small miss rates can have a noticeable effect on algorithm performance.

Our aim is to model the cache performance of the hybrid algorithm described in Section 3. Because we are using all 32 hardware thread contexts available on the T1, we assume that the local tables are sized so that each occupies $\frac{1}{32}$ of the total L2 cache. During the sampling phase, a successful access to an element in the local table will be counted as a cache hit, while an unsuccessful access will be counted as a cache miss. We measure hits and misses for a number of records that match the local table capacity.

In order to make sure that samples are representative of cache behavior, we "warm up" the hash table before sampling by processing the number of records whose distinct aggregates would fit in the cache. That way, we avoid counting compulsory misses. The number of warm-up records is chosen to be about 30% more than the capacity of the hash table to ensure reasonable coverage of the table by input records.[2]

Figure 2 shows the miss rates for various input distributions (see Section 5.2) using our sampling technique. In this example, the local table capacity is about 1,500 elements. The transition point from mostly hits to mostly misses for most distributions occurs between roughly 1,000 and 2,000 group-by values, where aggregation using private thread tables no longer shares the L2 effectively.

The sorted input does not begin to transition until much later because it has extremely good locality. As the group-by cardinality approaches the input size, there are fewer consecutive repeated values in the sorted input, leading to more misses within a sampling window.

[2]For most experiments the warm up was 2,000 records.

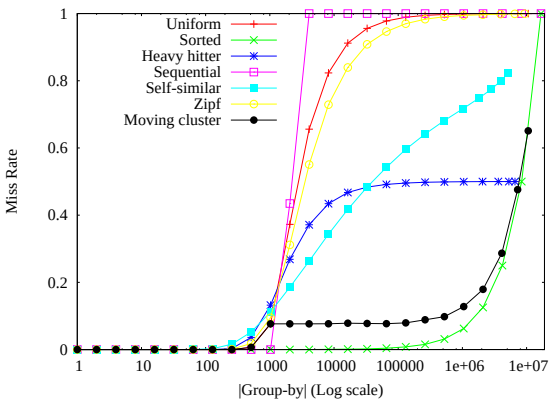


Figure 2: Miss rate sampled on test distributions.

To determine the locality threshold, note that we always pay the local processing cost L, and also pay the global processing cost G with a probability of m where m is the miss rate. The net cost beats the global processing cost when m is smaller than $1 - \frac{L}{G}$. We empirically determine that for L2-resident local tables, $\frac{L}{G} \approx 0.5$ (see Figure 4 for example). Thus we use 0.5 as our locality threshold.

4.3 Modeling Contention

Contention occurs when multiple threads try to access the same memory location concurrently. For contention to be a performance issue, there must be elements that occur frequently in the input streams of many threads. In many cases, frequently occurring elements will be associated with locality of reference. Since the outcome we choose is the same for distributions with contention and distributions with locality (see Section 4.4), this overlap of the categories is not problematic.

One can also have contention without locality. For example, suppose there are a few heavy hitters in an otherwise non-local distribution. The heavy hitters may be frequent enough to cause contention, while not quite frequent enough to meet the locality threshold for the distribution as a whole. As we shall see empirically, the performance penalty for contention is very high, reducing performance by an order of magnitude. Thus we include an explicit test for contention during the sampling phase, separate from the locality test.

We measure contention by counting the number of accesses to each hash bucket for a sample of consecutive input records. A bucket with many accesses is a potential source of contention. We measure accesses to a bucket rather than to a specific element because counting accesses at this coarser granularity incurs less overhead. Because our sampling window is small and our hashing function is universal, the probability is low that two different elements will map to the same bucket and together suggest contention when either element in isolation would not signify possible contention.

To quantify the contention, consider an input distribution in which a proportion p of the elements contained a single common group-by value V, while the remaining proportion $(1 - p)$ were uniformly distributed over a large number of elements distinct from V. In this setting, one would expect contention only on the common group-by value V. We are

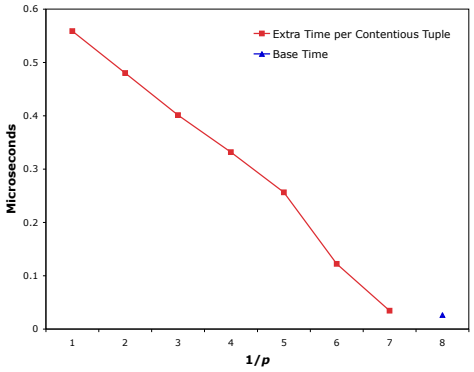


Figure 3: Contention overhead as a function of $1/p$.

interested in how the contention overhead varies with p.

We model this scenario as follows: Let g be the time taken for an input other than V, and w be the time taken by a single thread to process the contentious input, V, including all synchronization latencies. We wish to solve for w in terms of other variables. Let s be the service time for records with V, i.e., the time observed by the system (all threads) to process a contentious record while obtaining and holding exclusive access to needed resources. Unlike w, s does not include the time spent waiting for those resources. Because contentious records are serialized, s is independent of the number of threads. The average time per record (which we can measure) will be

$$A = (1-p)g + pw = g + p(w-g). \quad (1)$$

Equating the service rate for contentious records with the rate at which records are processed gives an equilibrium in which

$$\frac{1}{s} = \frac{Tp}{(1-p)g + pw} \quad (2)$$

where T is the number of threads. Solving for w gives

$$w = Ts + g - \frac{g}{p} \quad (3)$$

Thus, when the service time is too long to keep up with the workload, w should be a linearly decreasing function of $1/p$. To further illustrate this model, consider the case when all tuples are contentious, i.e., $p = 1$. It follows then that $w = Ts$, as shown by Equation 3. This represents a serialization of the threads, with w, the time for one thread to process a tuple, now equal to the time for all other thread to process a tuple plus its time to process a tuple, including the synchronization latencies experienced by all threads.

Figure 3 shows measured values of $(A-g)/p$ as a function of $1/p$ for the atomic-update implementation with three computed aggregates. g was measured at a point where p was sufficiently small that contention was absent. According to the formulas above, $(A-g)/p = w-g$ is a measure of the per-tuple contention overhead. Figure 3 shows that the cost decreases linearly, as predicted by the model, and that contention disappears when the frequency reaches about 1/8.

The contention overhead for the locking implementation also has an initial straight-line behavior like Figure 3, but with a sharp drop when transitioning from slight contention at $p = 1/17$ to almost no contention at $p = 1/20$. We hypothesize that the lock operations are much more time-consuming when locks are not available because operating system calls must be made to manage thread scheduling.

When we varied the number of aggregate functions, the contention performance penalty was very similar to Figure 3. At first glance, it may seem surprising that performing many atomic updates does not add to the overall contention overhead, since there are more potentially contentious updates happening. However, the first of these updates is where the real contention takes place, causing delays for some of the threads. For the second and subsequent updates, the flow of accesses has been limited by the first update to a rate that the hardware can handle without contention.

A straight line cost function f that does not depend on the number of aggregate functions is very easy to store in the database catalog for use in estimating contention for the atomic approach. We estimate overall contention by combining the overhead estimates for all values whose frequencies exceed the x-intercept of the cost function f. In Figure 3, for example, that means that all elements with a frequency greater than 1/8 will contribute to the cost. If there were two such elements, with frequencies 1/5 and 1/7, then the overhead would be $f(1/5)/5 + f(1/7)/7$. The frequencies are estimated using the bucket counters mentioned above. We label the stream as having contention if the contention overhead exceeds a certain fraction of the normal processing cost. For our experiments, we set this threshold to 100%, so that the contention label means that contention more than doubles the processing cost.[3]

For the lock-based approach, we simply set a threshold of $p = 1/20$, and label the stream as contentious if any element has a frequency greater than 1/20.

4.3.1 Duplicate Elimination, Min and Max

For duplicate elimination, we do not need to do any protected updates to aggregate values. As a result, there is potential contention only for the initial insertion of an element into the hash table. Since we expect many accesses per element, this kind of contention will be negligible. Further, since protected updates are relatively expensive, their absence means a potential performance gain relative to conventional aggregation.

A similar observation can be made about Min and Max aggregates. If the values being aggregated are randomly ordered, then we expect to be reasonably close to a Min or Max after a relatively small number of tuples. For example, after probing 99 values, the running minimum is expected to be the first percentile boundary value. The probability that the 100th input value will need to update the running minimum is therefore only one percent. Thus, almost all of the time we can simply read the current minimum (without a protected access) and reject the new value because it does not beat the current value. In rare cases, the current value will beat the old minimum, in which case we do go back and perform a protected access. An adversarial distribution of values (such as a monotonically decreasing sequence) would invalidate this analysis, but such issues are reasonably easy to handle using some initial randomization.

As a result, our contention model should return "no contention" for duplicate elimination, or for aggregates con-

[3]This factor of two corresponds roughly with the measured performance advantage of the atomic method relative to the partitioned method for inputs without locality (see Figure 4 for example).

sisting only of Min and Max aggregates. Other aggregates should be handled as previously described.

4.4 Adaptive Aggregation

Based on the sampled statistics, our decision process is described below. Since atomic aggregation outperforms the lock-based algorithm for most scenarios we have studied, we will choose the atomic algorithm for accessing the global table. (See Section 6.5 for a discussion of when locking beats the atomic algorithm.) For aggregate functions that cannot be expressed atomically, one would need to use the lock-based implementation.

1. For duplicate elimination, or aggregates involving only Min and Max, use atomic aggregation; otherwise
2. If there is locality or contention, then use the hybrid algorithm; otherwise
3. Use atomic aggregation.
4. Whatever method is chosen, if the average run-length exceeds the threshold, apply the run-length optimization.

One could sample once per time-slice, or multiple times per time slice. If the distribution changes often within a time-slice, more frequent sampling will give a better choice of algorithm, at the cost of additional sampling work. We examine the cost/benefit tradeoffs of the sampling frequency in Section 6.4. One could dynamically adjust the sampling frequency based on the similarity between successive samples, but such issues are beyond the scope of this paper.

Several of the thresholds discussed in the previous sections are dependent on the machine on which the aggregation will be run. These thresholds can be estimated in advance using a small number of calibration experiments, such as those used to estimate f in Section 4.3. For standard aggregate functions, the thresholds are relatively insensitive to query parameters such as the number of aggregates being computed. Complex user-defined aggregate functions might change the performance profile sufficiently to change the preferred thresholds. However, in that case the computation cost is likely to dominate the overall processing time.

5. EXPERIMENTAL SETUP

All experiments were conducted on a 1GHz Sun Fire T1000 server with an UltraSPARC T1 processor (Table 1). Unless otherwise stated, we use all 32 available hardware threads.

We measure events, such as cache misses, using performance counters provided by the T1 hardware. Unlike some hardware, branch mispredictions are not problematic for the T1 due to the interleaved nature of the threads: a branch can be resolved within three cycles, when a thread is typically rescheduled. We measured TLB misses but found them to be insignificant because the operating system automatically used large pages for the heap. Instruction cache misses were insignificant due to the inner loop's small code size. The counts that did make significant contributions to latency were the L1 data cache misses and the L2 cache misses.

[4]The miss latency varies with the workload, and with the load on the various processors [13]. When operating out of the L2 cache, our workload stresses the memory unit using all processors, so we use the 155 cycle estimate in our calculations. When the hash table(s) fit in the L2 cache, we use the 90 cycle estimate obtained with Calibrator [16]: reading the input (which is responsible for most L2 misses) does not stress the memory unit.

Operating System	Solaris 10 11/06
Cores (Threads/core)	8 (4)
RAM	8GB
Shared L2 Cache	3MB, 12-way associative Hit latency: 21 cycles Miss latency: 90–155 cycles[4]
L1 Data Cache	8KB per core Shared by 4 threads
L1 Instruction Cache	16KB per core Shared by 4 threads
On-chip bandwidth	132GB/s
Off-chip bandwidth	25GB/s over 4 DDR2

Table 1: Specifications of the Sun UltraSPARC T1.

5.1 Implementation Details

We measure the performance for a single time-slice consisting of $2^{24} \approx 16$ million records. This gives a total time of between 50 and 500 milliseconds per time-slice. The graphed measurements are averages over four repetitions. The experiments are performed for the *first* time-slice of an aggregation, which may have more insertions into the hash table than later time-slices. The cost of insertions will only become noticeable at group-by cardinalities in the millions, as the number of records per group becomes small.

Records are 16 bytes, consisting of a 64-bit integer group-by value, and a 64-bit integer value for aggregation. The input fits in 256MB of RAM.[5] All comparison and arithmetic operations use 64-bit integer instructions. The UltraSPARC T1 design exhibits a number of significant performance tradeoffs. Most notable is the single floating point unit that is shared by all of the cores. This means that floating point intensive code does not perform well on the T1. For this reason we focus on aggregations involving integers. Future generations of the T1 will have one floating point unit per core [9], and we expect our techniques to be applicable to floating point aggregates on such a system.

We tried to use the prefetch instructions to improve performance by hiding some of the L2 cache miss latency. We found that prefetching did not improve performance, so our measured implementation does not use prefetching. There are several reasons why a prefetch may be dropped, and these may have influenced our observations [23, 24]. Note also that gains from prefetching are relatively small on the T1. Each thread has just a 25% share of a 1GHz processor, compared with a conventional single-threaded computation having 100% of a 3GHz processor. As a result, L2 cache latency is an order of magnitude less significant when measured in cycles. Note also that, unlike some "fat camp" processors, the T1 does not perform hardware prefetching for regular-stride access patterns.

Since integer division is very expensive on the T1, we use hash tables whose size is a power of 2, so that bucket calculations can use masks and shifts rather than modulus operations. Given an estimate of the group-by cardinality (something typically provided by the query optimizer), we

[5]For duplicate elimination experiments there is no value for aggregation and records are 8 bytes.

size the hash table so that its occupancy is expected to be at most 50%. Chaining is used to resolve hash collisions. Hash buckets have size matching the size of an L2 cache line (64 bytes), except when the number of aggregates being computed is too large to fit in a cache line. In that case, a small integer number of cache lines is used per bucket. Hash buckets are 64-byte aligned.

During initial experimentation using independent hash tables, we observed poor performance for certain group-by cardinalities. This was due to implicit contention for cache resources. The use of the same hash function in each thread, together with common alignment relative to the L1 cache, caused conflict misses between threads in the L1 cache. We resolved this issue by ensuring that the hash table structures begin at offsets that stripe the hash tables in the L1 cache.

The input is divided into equal sized contiguous chunks that form the unit of work for a thread. When a thread finishes one chunk, it starts work on another chunk if there are any remaining. (Chunk size is discussed in Section 6.4.) We observed in preliminary experiments that this strategy performs better than interleaving thread accesses to the input stream because it allows for more TLP. With interleaved input access, multiple threads may stall on the same cache miss, reducing parallelism, whereas with independent contiguous input chunks each thread's input misses will not be correlated with other threads' misses.

5.2 Input Distributions

We experiment with aggregation performance on multiple input distributions, where the distribution refers to the characteristics of the group-by key in the input relation. For each input distribution, we also vary the number of distinct group-by keys in the distribution. The input is synthetic and the distributions were generated in a manner similar to Gray et al. [11]. The distributions we use are: (1) uniform, (2) sorted, (3) heavy hitter, (4) sequential, (5) Zipf, (6) self-similar, and (7) moving cluster.

In the heavy hitter input, one value accounts for 50% of the group-by keys, while the other values are chosen uniformly from the other group-by keys. The sequential distribution consists of input records in segments, each consisting of a numerically increasing sequence of group-by values. For example, with 10000 group-by values, the sequence of group-by values would be $1, 2, \ldots, 10000, 1, 2, \ldots, 10000, 1, 2, \ldots$. The self-similar distribution uses an 80-20 proportion, and the Zipf distribution uses an exponent of 0.5. In the moving-cluster distribution with $c \geq W$, record number i is chosen uniformly from the range $\lfloor (c-W)i/r \rfloor$ to $\lfloor (c-W)i/r+W \rfloor$, where c is the target group-by cardinality, r is the number of records, and W is a window size. For $c < W$ moving-cluster reverts to a uniform distribution. We use $W = 1024$.

During input generation we specified a target group-by cardinality. However, because of the probabilistic nature of many of the distributions, this target was not met, especially when the requested group-by cardinality approached the size of the input. For each graph we plot the actual group-by cardinality in the input data rather than the target used to generate the input.

Except for Section 6.4, the input is divided into 32 equal chunks. Each thread samples once at the start of its chunk, and then processes the remainder of the chunk. In Section 6.4, we vary the chunk size. The sample size is chosen in the manner described in Section 4.2.

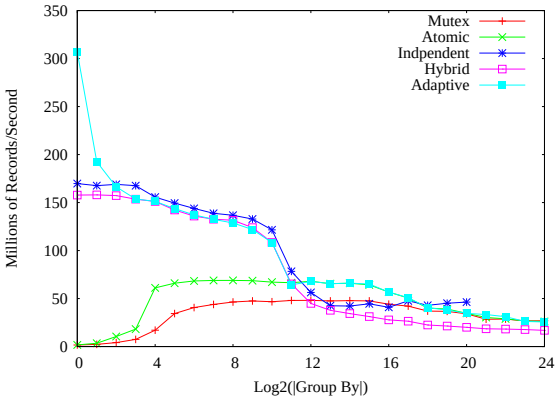


Figure 4: Throughput for `Q1` (uniform data).

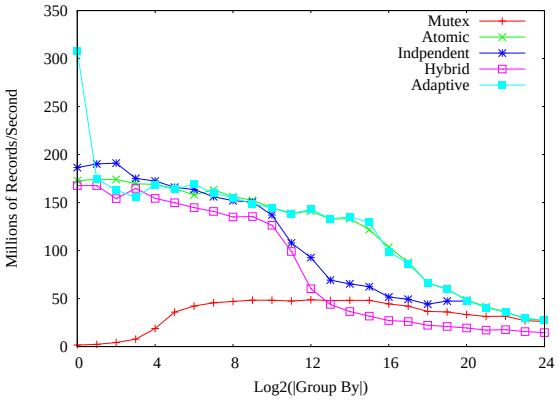


Figure 5: Throughput for `Q2` (uniform data).

6. EXPERIMENTS

Our experiments compute answers to the following queries:

```
Q1: Select G, count(*), sum(V), sum(V*V)
    From R Group By G
Q2: Select G, max(V), min(V), min(V)
    From R Group By G
Q3: Select distinct G From R
```

The repeated `min` in query `Q2` will allow us to more easily compare the performance of queries `Q2` and `Q1` since they have the same number of aggregates and output tuple size.

6.1 Results on Uniform Data

In this section, we show the performance of the various algorithms for queries `Q1`, `Q2`, and `Q3` on uniform data.

Figure 4 presents the throughput for query `Q1` as the group-by cardinality varies. Figure 4 exhibits a performance change at the same point that our sampling model (Figure 2) shows the uniform input distribution to transition from mostly cache-hits to mostly cache-misses. The atomic and mutex implementations exhibit very low throughput for small group-by cardinalities due to contention. Though the independent tables approach performs well when every table fits in the L2 cache, performance suffers significantly when the table sizes exceed the L2 cache size (around a group-by cardinality of 1500). Moreover, the independent tables approach fails at a cardinality of 2 million because it cannot allocate enough physical RAM for all 32 separate tables. Our adaptive method tracks the maximum performing strat-

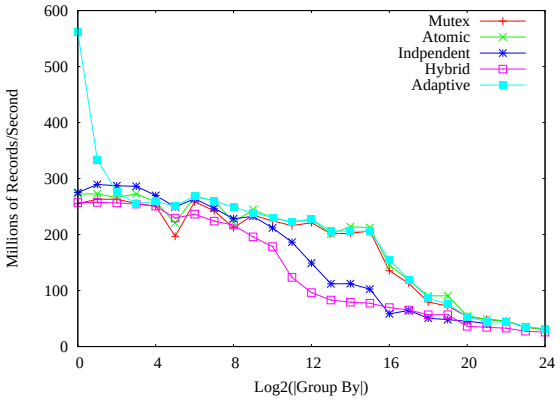


Figure 6: Throughput for Q3 (uniform data).

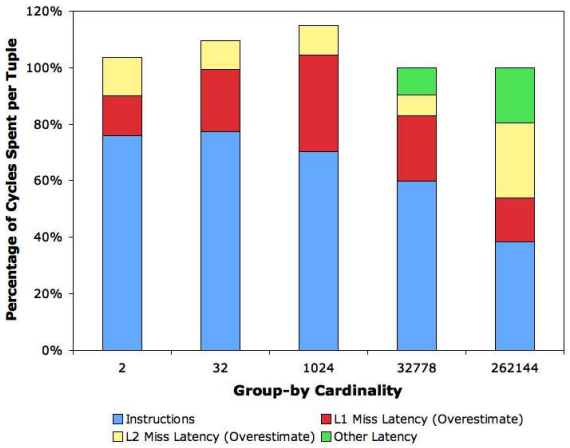


Figure 7: How cycles are spent.

egy for all group-by cardinalities, and exceeds them for small group-by cardinalities due to the run optimization.

Figure 5 shows the throughput for query Q2 as the group-by cardinality varies. As expected, because the Min and Max aggregates rarely require protected access, the atomic method performs best. Our adaptive algorithm chooses this algorithm, with run optimization for small group-by cardinalities. In order to show how important avoiding protected access is, Figure 5 shows the performance of the lock-based algorithm without this reduced-protection optimization.

Figure 6 shows the throughput for query Q3 as the group-by cardinality varies. The picture is similar to that for query Q2, except that the lock-based method and atomic method are identical: there are no aggregate values to update.

Time Breakdown

We try to pinpoint exactly where the T1 is spending its time during the aggregate computation of query Q1 on uniform data (Figure 4). Each cache miss is assigned its full latency, which is an overestimate because some of the latency may be overlapped with other work/latency. As a result, we sometimes obtain a total latency greater than 100% of the cycle budget. We consider our adaptive strategy at five different group-by cardinalities that represent different configurations:

Card.	Algorithm	Comments
2	Hybrid + runs	Average run length = 2
32	Hybrid	L1 resident
1024	Hybrid	Always hit local tables in L2
32768	Atomic	Global table L2 resident
262144	Atomic	Well out of cache

Figure 7 shows the time breakdown at these group-by cardinalities.

When the aggregation data structures are L1 or L2 cache resident, processor utilization exceeds 60%, and hits 77% for L1-resident tables. A majority of cycles perform useful computation. For all of the cache resident aggregations (the four left columns), the L2 miss latency is due almost entirely to compulsory misses on the input records. Four input records fit in one 64-byte L2 cache line, so we expect one L2 cache miss per four input records, and this is, in fact, what we measure for the L1 resident aggregation. When the size of the aggregation hash table exceeds the L2 cache, L2 cache misses become a significant source of latency and processor utilization falls to around 40%, which is still relatively good.

Instructions and cache misses do not account for all of the latency. The "other latency" category includes non cache-related stalls such as long-latency instructions (multiply at 11 cycles and branches at 3 cycles) and waits during atomic add instructions. These latencies may be less effectively overlapped with computation from other threads when processor utilization falls.

6.2 Adaptability

We now investigate how well our adaptive algorithm performs under a variety of input distributions for query Q1. Figure 8 shows the performance results. As for the uniform case, our adaptive method almost always tracks the best algorithm for each group-by range. For sorted data, the run optimization makes a dramatic contribution to high performance. In some ranges, the performance of the independent tables approach outperforms the adaptive method. However, in these ranges, the independent tables make excessive memory demands, 32 times the amount needed by the adaptive approach. These demands may not be reasonable given a fixed RAM budget for the query and/or system. The distribution in Figure 8(d) validates the contention model. Though non local after a group-by cardinality of 16000, the distribution has significant contention and the adaptive algorithm correctly chooses hybrid aggregation. At very high group-by cardinalities, the contention is reduced and the adaptive algorithm correctly chooses shared aggregation.

These results show that the adaptive method is doing a good job of determining when to transition between algorithms for a variety of input distributions. As a result, the performance of the operator is robust with respect to the underlying input distribution.

6.3 Thread Scaling

Figure 9 shows how performance scales with the number of available threads. When interpreting these results, it is important to remember that T1 processor has eight cores, each of which has four threads. Therefore, after eight threads are used, additional threads will compete for L1 and execution resources. That said, scaling is rather good, particularly for larger group-by cardinalities.

Figure 9(a) demonstrates an important aspect of contention: adding more threads naively can actually reduce performance! Based on the term Ts in Equation 3 of our

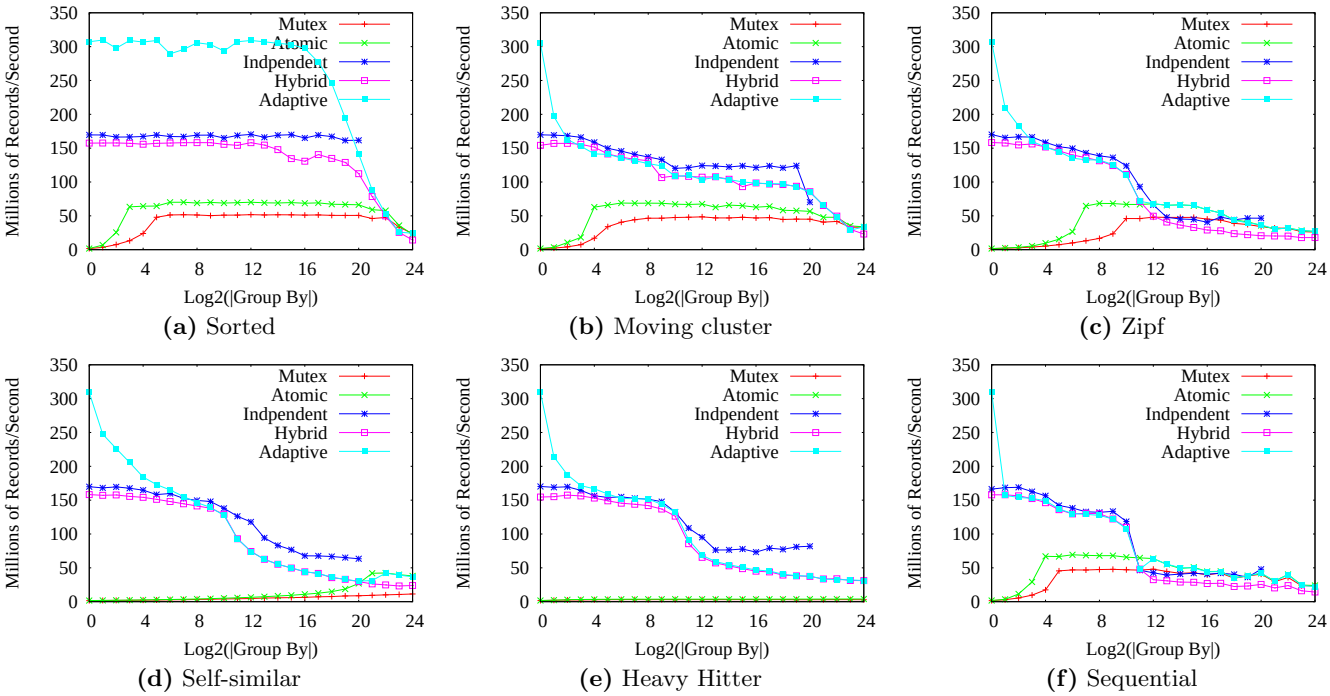


Figure 8: Throughput for select distributions for query `Q1`. Note the logarithmic scale on the x-axis.

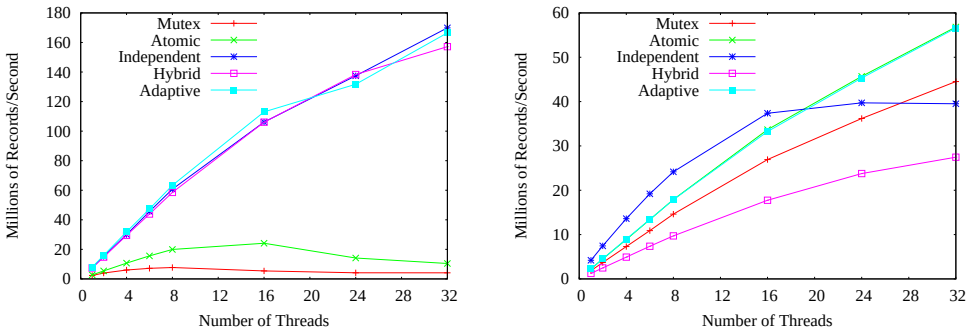


Figure 9: Scaling of the various aggregation methods on uniform input data.

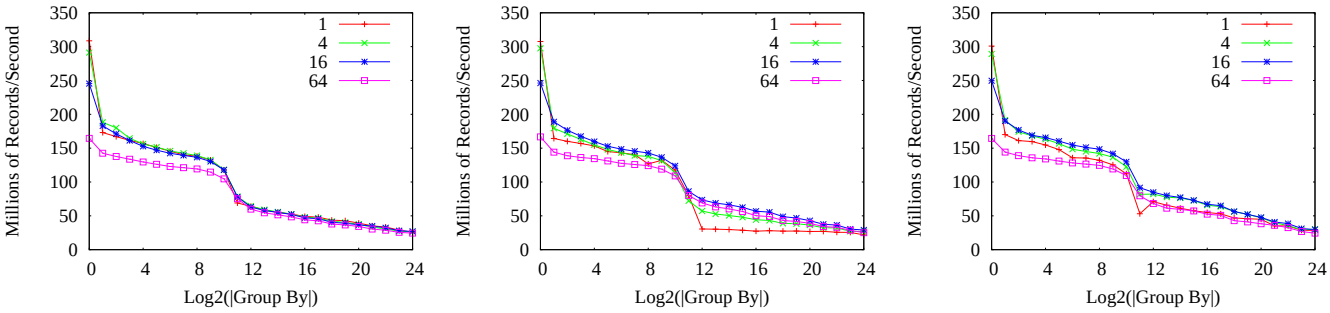
contention model, we expect a *decrease* in performance for small group-by cardinalities on the shared table strategies when more threads are added to the computation. Figure 9(a) confirms this expectation.

6.4 Sampling

In this section we quantify the overhead of sampling at various frequencies, and measure how well the adaptive algorithm performs on inputs whose underlying distribution is changing. We form mixed distributions by taking N records in turn from each of the seven previous distributions. At $N = 5,000$, the distribution is changing rapidly, on a scale that is comparable to the sampling window. At $N = 50,000$ the distribution is changing less frequently. At $N = 2^{19} \approx 500,000$, the distribution changes precisely 32 times within the input.

We consider several resampling frequencies. With a sample size of 3,500 records (warm-up and sampled tuples as described in Section 4.2), one sample per thread corresponds to sampling 0.667% of the data. At this setting, each thread processes $\frac{1}{32}$ of the data, sampling once. With α samples per thread, we are sampling 0.667α% of the data. A thread processes $\frac{1}{32\alpha}$ of the data before asking for more work. When α is large, fast threads (say those working on sorted data) may process more records overall than other threads.

The performance of the adaptive algorithm is shown in Figure 10 for various values of α. The best performance appears to be at $\alpha = 16$. For $N = 2^{19}$ where each thread sees a single distribution, the suboptimality of $\alpha = 1$ is due to the inability of fast threads to do extra work once they have finished their current work. For $N = 50,000$, the suboptimality of $\alpha = 1$ is due to running the "wrong" algorithm for the data because the sample is not representative of the current data. Sampling overhead causes $\alpha = 64$ to be suboptimal—42% of the data is sampled at this setting. Additional sampling does not seem to help for $N = 5,000$,



(a) $N = 5,000$ (b) $N = 50,000$ (c) $N = 2^{19}$

Figure 10: Resampling performance on a mixed input distribution.

since the distributions are changing too frequently.

We now compare the performance on the mixed distributions with the average performance over the individual single distributions. Since the mixed distributions contain an equal number of records from each distribution, a comparison with the average time taken over all distributions is fair. Figure 11 shows the results for $\alpha = 16$, normalized so that the average time of the single distributions is 100%. Figure 11 also shows the performance of the adaptive method on uniform data, for comparison.

When $N = 2^{19}$, the performance equals or exceeds the average performance of the single distributions. This improvement happens primarily because some threads observe locality and choose a local computation. This removes those threads from the pool of possible contenders for shared data, thus reducing the impact of contention on the global table. When $N = 50,000$, the performance is close to the average performance of the single distributions and always beats the performance of the uniform distribution. When $N = 5,000$, sampling is not very effective, and the performance is noticeably worse than the performance on the individual distributions. The spike at a group-by of 8,192 is because this group-by value is large enough to make most access patterns non-local. However, if the sampling is done over a window that exhibits locality, the (usually incorrect) decision to use the hybrid approach will result in many more cache misses.

6.5 Other Results

Startup and Finalization Costs. We measured the startup and finalization costs for each of the algorithms, and found that in most cases they were negligible compared with the cost of one time-slice. The one case where startup and finalization costs were significant was the independent tables approach, where 32 copies of the hash table need to be initialized and merged. At the very highest group-by cardinalities, the startup and finalization costs were each comparable to the cost of processing one time-slice.

More Aggregate Functions. We also implemented queries with more input columns and more computed aggregates. The results were similar to those already presented, except that the lock-based implementation begins to outperform the atomic implementation in contention-free regions once there are at least six aggregate functions being computed. As previously mentioned, the cost of locking is fixed, while the cost of atomic operations is proportional to the number of aggregate functions, so a transition of this kind was expected. In a complete system, a calibration experiment would identify this trade-off threshold, and use a lock-based implementation for queries with more than the threshold number of aggregate functions.

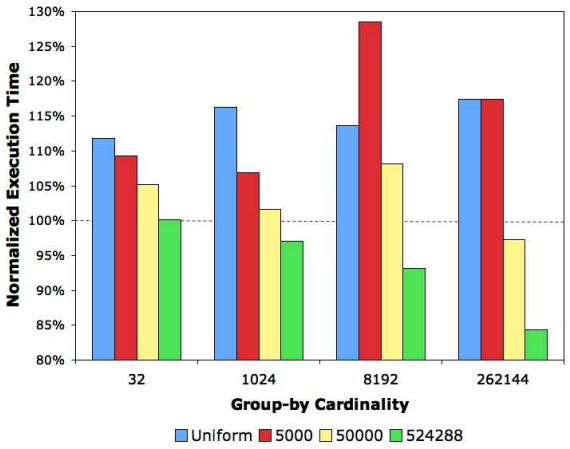


Figure 11: Mixed versus Single Distributions. Resampling rate $\alpha = 16$.

Sensitivity to the Group-By Estimate. The aggregation performance was reasonably sensitive to the initial group-by estimate provided by the query optimizer. If this estimate was too small, the hash table has many overflow chains that cause a significant slowdown. On the other hand, estimates that were too large had a minimal impact on performance. As a result, when the optimizer is uncertain about the group-by cardinality, it is a good idea to provide an overestimate to the aggregation algorithm.

7. DISCUSSION

The speed of aggregation is higher when the input stream displays temporal locality. For uniform data, Figure 4 shows that a group-by cardinality of 500 allows for a tuple to be processed every 8 cycles, while for a cardinality of 100,000 a tuple is processed every 20 cycles. It might be possible to preprocess the input stream to improve temporal locality by partitioning the input according to the group-by values. If the partitioning can be done using fewer than 12 cycles per record, then preprocessing would be worthwhile.

Locality enhancement of this kind has been studied in the form of the radix-cluster algorithm of [17]. On various architectures, an optimized 2000-way radix-cluster algorithm

took between 60 and 120 cycles per 8-byte record [17]. With 8-way parallelism on the T1 (recall that the threads are interleaved rather than truly parallel), a parallel radix-cluster algorithm might be expected to take roughly 8–15 cycles per 8-byte record, or slightly more per 16-byte record (as used in Figure 4). This performance may be less than the 12 cycle threshold mentioned above, and therefore radix-clustering could potentially improve performance by a small amount. Since implementing a parallel version of radix-cluster and accurately modeling when to use it are both complex problems, we leave such improvements to future work.

Zukowski et. al. [25] perform a query similar to our duplicate elimination query `Q3` on a 3GHz Pentium 4 machine, using a variant of cuckoo hashing. At one million group-by values, their measured performance is about 27 million four-byte records per second, while they obtain about 81 million records per second for a group-by cardinality of 256.

Our numbers for this scenario (measured over eight-byte records) are 54 million and 250 million records per second, respectively. Our results are therefore roughly consistent with our 8:3 total cycle advantage, and our keys are larger. The fact that we are cycle-comparable to Pentium-class machines is good, because the T1 is more energy-efficient. Also, our algorithms naturally scale with the additional parallelism that will be available on future processors.

8. CONCLUSION

In this paper we have examined aggregation on a chip multiprocessor. In addition to a novel performance model for locality and contention, we propose and evaluate an adaptive aggregation operator that provides good performance regardless of the input distribution.

Even though our target architecture, the UltraSPARC T1, is a single-issue, in-order processor, the overall CPU utilization is very high because of hardware support for 32 concurrent threads. Our performance evaluation also confirms that the aggregation work is well balanced between the various threads. Keeping all available threads busy with useful work is key to extracting maximum performance from parallel architectures such as CMPs.

Future multi-core architectures may cause contention and locality issues to change. For instance, additional cores or hardware threads will increase contention, resulting in contention for shared algorithms at higher group-by cardinalities. Observations from [12] suggest that larger L2 caches will have longer hit latencies, depressing the performance of L2 cache resident aggregation. If the amount of cache per core (or thread) decreases, the locality transition point between independent and global strategies will shift toward lower group-by cardinalities. For example, on the UltraSPARC T2, which will have 64 threads, the amount of L2 cache per thread will be 33% lower than on the UltraSPARC T1, even though the size of the L2 cache will increase [9].

The chip multiprocessor landscape continues to evolve. To get the most out of these processors, database algorithms must continue to adapt.

9. REFERENCES

[1] A. Ailamaki et al. Query co-processing on commodity processors. In *VLDB*, page 1267, 2006.

[2] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, pages 261–272, 2000.

[3] S. Chen et al. Inspector joins. In *VLDB*, pages 817–828, 2005.

[4] J. Cieslewicz et al. Realizing parallelism in database operations: Insights from a massively multithreaded architecture. In *DaMoN*, page 4, 2006.

[5] J. Cieslewicz et al. Parallel buffers for chip multiprocessors. In *DaMoN*, pages 9–18, 2007.

[6] A. Deshpande et al. Adaptive query processing: Why, how, when, what next. In *SIGMOD*, pages 806–807, 2006.

[7] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Comm. ACM*, 35(6):85–98, 1992.

[8] B. T. Gold et al. Accelerating database operations using a network processor. In *DaMoN*, page 1, 2005.

[9] R. Golla. Niagara2: A highly threaded server-on-a-chip, October 2006. http://www.opensparc.net/publications/.

[10] N. K. Govindaraju et al. Fast computation of database operations using graphics processors. In *SIGMOD*, pages 215–226, 2004.

[11] J. Gray et al. Quickly generating billion-record synthetic databases. In *SIGMOD*, pages 243–252, 1994.

[12] N. Hardavellas et al. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, pages 79–87, 2007.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture*. Morgan Kaufman, 4th edition, 2007.

[14] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, pages 106–117, 1998.

[15] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1998.

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

[17] S. Manegold et al. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[18] V. Markl et al. Robust query processing through progressive optimization. In *SIGMOD*, pages 659–670, 2004.

[19] N. Nagarajayya. Improving application efficiency through chip multi-threading. Technical report, Sun Microsystems, 2005.

[20] K. A. Ross. Efficient hash probes on modern processors. In *ICDE*, pages 1297–1301, 2007.

[21] K. A. Ross et al. Architecture sensitive database design: Examples from the Columbia group. *IEEE Data Eng. Bull.*, 28(2):5–10, 2005.

[22] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *SIGMOD*, pages 104–114, 1995.

[23] Sun Microsystems. *OpenSPARC T1 Microarchitecture Specification*, August 2006.

[24] Sun Microsystems. *UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005*, March 2006.

[25] M. Zukowski et al. Architecture-conscious hashing. In *DaMoN*, page 6, 2006.