Automatic Contention Detection and Amelioration for Data-Intensive Operations

John Cieslewicz*, Kenneth A. Ross†, Kyoho Satsumi, Yang Ye
Department of Computer Science, Columbia University, New York NY
(johnc,kar,yeyang)@cs.columbia.edu, ks2677@columbia.edu

ABSTRACT

To take full advantage of the parallelism offered by a multi-core machine, one must write parallel code. Writing parallel code is difficult. Even when one writes correct code, there are numerous performance pitfalls. For example, an unrecognized data hotspot could mean that all threads effectively serialize their access to the hotspot, and throughput is dramatically reduced. Previous work has demonstrated that database operations suffer from such hotspots when naively implemented to run in parallel on a multi-core processor.

In this paper, we aim to provide a generic framework for performing certain kinds of concurrent database operations in parallel. The formalism is similar to user-defined aggregates and Google's MapReduce in that users specify certain functions for parts of the computation that need to be performed over large volumes of data. We provide infrastructure that allows multiple threads on a multi-core machine to concurrently perform read and write operations on shared data structures, automatically mitigating hotspots and other performance hazards.

Our goal is not to squeeze the last drop of performance out of a particular platform. Rather, we aim to provide a framework within which a programmer can, without detailed knowledge of concurrent and parallel programming, develop code that efficiently utilizes a multi-core machine.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*concurrency, query processing*

General Terms

Performance

*The work of John Cieslewicz was supported by a Department of Homeland Security Fellowship. Current affiliation: Aster Data.

†This work was supported by the National Science Foundation under awards IIS-0534389 and IIS-0915956.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0032-2/10/06 ...$10.00.

1. INTRODUCTION

Thread level contention is a new bottleneck facing developers of data-intensive applications, including databases. Thread level contention occurs when tightly coupled, concurrently executing threads serialize due to shared access to a data structure. The contention may arise from an explicit desire to achieve atomicity, for instance a lock protecting a critical section, or implicitly when updates to a shared structure cause expensive cache coherence that slows access to the same data by other threads. When only a few threads are involved, contention is rarely detected, but with the increasing number of threads available on chip multiprocessors, the likelihood of contention also increases.

Contention is an important problem for databases because recent developments in computer architecture favor a database execution model with tightly coupled, concurrently executing threads for in-memory database workloads. We will first cover two processor architecture issues that inform this discussion and then discuss the implications for database implementation.

1.1 Hardware Architecture Issues

Chip designers are finding it harder to increase clock frequencies, because further increases in clock speed consume too much power and generate too much heat. Instead of going faster, chips are becoming more parallel. Chip multiprocessors that contain multiple processor cores allow hardware architects to improve computing performance per unit chip area with lower power and heat demands. As these chips become common in the marketplace, application programmers (including implementors of database systems) are faced with the new challenge of utilizing the parallel resources provided by the new hardware. This is not an easy task as decades of legacy code was developed for a uniprocessor architecture.

At the same time, main memory access has become a performance bottleneck for many computer applications, including database systems [1, 3]. We assume that the data needed for queries is memory-resident. In disk-resident databases with sufficient I/O bandwidth and adequate indexing, I/O is not a performance bottleneck [1, 13].

Advances in the speed of commodity CPUs have far outpaced advances in memory latency. Modern machines have *caches* that enable them to store relatively small amounts of data in higher-speed memory. When data is not found in the cache, a *cache miss* results, and main memory must be accessed. In response to the growing memory latency problem, the database community has explored a variety of techniques to perform database operations in a cache-conscious way, to

reduce the number of cache misses, or to hide their latency (see [7] for a survey).

1.2 Implications for Databases

When many concurrent threads are operating in parallel, they could be working on independent tasks, or they could be cooperating with each other on common tasks. When each task has a small amount of state, the independent-tasks model is attractive. Each thread can have its working-set reside in cache memory. Because the tasks are independent, there is no explicit coordination required between threads, reducing overheads for coordination, such as locking.

The independent-tasks model has some shortcomings, however. When the working set of multiple tasks exceeds the cache memory, cache interference can result in both the data and instruction caches. For example, in [23] it was shown that running two database operations in parallel was noticeably slower than running them in sequence due to this interference effect. Another example is partitioning. Using many independent partitioning threads performs worse than concurrent partitioning using a shared data structure when there are many partitions; when each thread has its own partitions, the number of active cache lines is effectively multiplied by the number of threads, leading to cache thrashing [6]. To avoid cache interference on a machine running n independent tasks, each task can (on average) use only $1/n$th of the cache and memory capacity.

An additional shortcoming of the independent tasks model is the need to have many independent tasks available. Today, the Sun UltraSparc T2 provides 64 thread contexts, and future machines will likely have many more. It may not be realistic to assume that the number of available independent units of work will always exceed the number of threads. Load balancing in the presence of tasks that differ in size also becomes a problem.

Instead of the independent-tasks model, we advocate the opposite approach, namely devoting many threads to performing a single operation in parallel. The advantage of this approach is that all threads use a common data structure that we can design to be cache efficient, meaning that unnecessary cache interference will be avoided. All threads will be sharing, and fully utilizing, the entire cache.

The difficulty of this many-threads approach is that the threads are not independent. They may update shared data elements. In order to guarantee correct execution, these data elements need to be protected using locks or atomic operations. The use of such protection can lead to two kinds of performance problems: (a) the overhead of guaranteeing exclusive access to data, and (b) the risk that popular data items will cause threads to queue waiting for access to the item, drastically reducing the effective parallelism due to contention.

The difficulties of parallel programming are well-known, and tuning algorithm behavior to avoid hotspots and other performance hazards is extremely challenging. Further, if there were to be a change in the hardware platform, such as an upgrade to a machine with more cores, the old tuning parameters may no longer prevent the hotspot behavior.

1.3 Our Contributions

We present ways to support generic operations that need to access and update shared data structures in parallel. We provide *abstractions* that allow a programmer to specify a computation without worrying about parallelism or contention. We also supply *mechanisms* that, without explicit control from the programmer, manage the concurrent access while avoiding or mitigating the contention bottleneck

Abstractions. We define a set of abstractions for use by programmers who wish to enable concurrent updates to a shared data structure. A programmer is required to write four simple functions that describe how the state of the data structure changes in response to new data elements. Our abstractions resemble user-defined aggregates in SQL, and the MapReduce framework [9], which we will discuss in more detail in Section 2.2.

Mechanisms. Our contention detection mechanism responds to explicit instances of contention, where a thread tries to update a resource and fails because some other thread is updating the resource. We demonstrate that obtaining this failure information can be done with low overhead. In response to contention, we *clone* the resource, so that future accesses to the resource are spread among the various copies, reducing contention. We propose two cloning methodologies: a local cloning approach that lets each thread manage its own clones, and a global cloning approach in which a shared pool of clones is globally managed.

Performance Results. We have implemented our proposed system, and have measured its performance. Our system allows a contention-naive programmer to avoid contention-related performance pitfalls that would otherwise cause an order of magnitude drop in performance. We compare the global and local approaches to clone management, and explain how important performance problems can be avoided.

Our goal is not to squeeze the last drop of performance out of a particular platform. Rather, we aim to provide a framework within which a programmer can, without detailed knowledge of concurrent and parallel programming, develop code that efficiently utilizes a multi-core machine.

2. RELATED WORK

2.1 Parallelism and Contention

Modern commercial database systems typically employ some form of shared-nothing or shared-memory parallelism [10]. A chip multiprocessor (CMP) is a shared-memory processor, but differs from previous shared-memory systems: in a symmetric multiprocessor (SMP), accesses to common data elements in RAM must be mediated using a cache coherency protocol. Such protocols incur significant latencies and use resources such as bus bandwidth. In a CMP, coherency is maintained on-chip, using fast, dedicated hardware. As a result, parallel algorithms that would be impractical on an SMP due to cache coherency overhead may be efficient on a CMP.

There have been many investigations of the performance of database systems on multicore platforms. Hardavellas et al. have profiled the performance of commercial database systems on various multicore platforms [16], and show that L1 data cache misses can be an important performance issue. Qiao et al. show how to batch a collection of aggregate operations running on a multicore machine so as to avoid cache thrashing [22]. Héman et al. describe ways to use the vector processing capabilities of the Cell processor to improve the performance of the MonetDB/X100 system [17].

There are numerous proposals for language constructs to facilitate parallel programming. Examples include MPI [15],

OpenCL [19], Cilk [2], Pthreads [21], and StreamIt [12]. These language constructs enable programmers to specify parallel tasks such as the spawning of threads in a generic, platform-independent way. Cilk, for example, provides runtime support for managing units of work so that a programmer does not have to explicitly schedule tasks. This line of research is orthogonal to the present work. Rather than focusing on explicit coordination of threads, most of whose work is independent, we focus on detecting and ameliorating contention on individual data elements.

There is also a significant body of work on non-blocking data structures for concurrently accessed data items. Many of these algorithms use compare-and-swap primitives. Massalin and Pu [20] implement stacks, queues and linked lists using double compare and swap (DCAS) primitives that are more powerful, but not typically available on modern architectures. In general, writing correct non-blocking algorithms using these primitives is difficult, and bugs in published algorithms have been observed [11]. Our proposed abstractions limit the computations we consider to simple commutative operations for which such correctness concerns are of less importance.

Another proposed method for dealing with concurrent access to data is transactional memory [18]. Transactional memory uses the traditional database abstraction of a transaction to control access to data items. In a typical transactional memory implementation (in either hardware or software) the system keeps track of the values read and written, and will abort and restart a transaction if it appears that there were updates that could have conflicted with another thread's updates.

Transactional memory is effective at resolving rare contention. When accesses are mostly independent, there will be few aborted transactions and performance will be high while still guaranteeing correctness. Unfortunately, when accesses are often contentious, transactional memory will, like related optimistic concurrency-control methods, result in a very large number of aborted transactions. Throughput will be limited by the serialization of threads accessing the popular data elements. Our work is thus complementary to transactional memory approaches. One can use transactional memory for shared data items that are known to be accessed infrequently, and our cloning approach for popular data items.

2.2 User-Defined Aggregates and MapReduce

Our proposed infrastructure uses abstractions that are motivated by abstractions provided by user-defined aggregates in SQL, and by the MapReduce framework [9]. In this section, we briefly describe these abstractions.

All major database systems today allow users to define their own aggregate functions to be used within SQL queries. The following code fragment shows how one might define a user-defined aggregate to compute sums of complex numbers in Postgres.[1]

```
CREATE AGGREGATE complex_sum (
    sfunc = complex_add,
    basetype = complex,
    stype = complex,
    initcond = '(0,0)'
);
```

[1]http://www.postgresql.org/docs/8.4/static/xaggr.html

The `initcond` parameter specifies that the initial value of the aggregate is $(0, 0)$. The `sfunc` parameter specifies a function that takes the current aggregate and a new value, and combines them to yield a new aggregate value.

By allowing user-defined aggregates, database system vendors allow users and programmers to leverage the infrastructure that already exists within SQL to compute aggregates. The alternative, namely writing procedural code to compute an aggregate, is far less satisfactory.

MapReduce [9] provides an abstraction for distributed large-scale computation. A programmer specifies two relatively simple basic functions, depending on the kind of computation that is desired. The *Map* function specifies how records are mapped to partitions or groups. The *Reduce* function specifies how a derived value is computed from the records within a partition. An example *Reduce* function that is used by Google during inverted index construction takes a (word,document-id) pair, and appends the document-id to a list of document-ids for that word [9]. Other examples can be found in [9]. The system manages the complex infrastructure needed to effectively manage the computation. Issues of data placement, data movement, data sorting, scheduling of computation, load balancing, and management of node failures are handled automatically. The programmer does not need to write code to address these issues.

Not all computations can be abstracted into a MapReduce framework. Nevertheless, many useful computations can be abstracted in this way. For those computations, the big advantage of a MapReduce-style abstraction is that the effort of developing a robust infrastructure for scalable computation can be shared by a variety of computational tasks. One does not have to re-code the infrastructure elements for each new problem. As a result, the time taken to write new code for computations that match the MapReduce abstraction is dramatically reduced, while scalable performance is retained.

2.3 Contention on Multicore Processors

We briefly describe prior work in which contention hazards for databases have been investigated.

2.3.1 Shared Buffers

A parallel join operation on the Cray MTA-2 multiprocessor [4] exhibited a performance bottleneck caused by threads writing to a common output array. The threads had to wait for access to the index counter for that array, causing them to serialize their execution. Once the bottleneck was identified using performance counters and other diagnostic tools, it was mitigated by giving each thread its own output array. This was a specific case of the general problem of sharing input or output buffers among concurrently executing threads. Though fraught with potential thread-level contention, concurrent buffer sharing was addressed in a more generic manner in [8].

2.3.2 Hash-based Aggregation

Contention for performing aggregates on the Sun T1 architecture, which has 32 thread contexts, was described in [5]. Two kinds of concurrency control for parallel access to a shared hash table were used for keeping running aggregate values. The first employs locking primitives provided by the operating system. The second takes advantage of the pres-

ence of operations that are guaranteed by the hardware to be atomic.

When many threads attempt to modify the same hash cell, a contention bottleneck can occur. There is a time-window during which all but one of the competing threads will be forced to retry an unsuccessful update. Such contention bottlenecks can decrease throughput by an order of magnitude [5]. These effects are particularly apparent when the group-by cardinality is small, or when there are a few heavy hitters among the grouping values [5].

3. DETECTING AND MEASURING CONTENTION

Atomic instructions are often exposed to programmers through intrinsics, short pieces of code that look like `C` functions, but are implemented in assembly language and inlined by the compiler. To enable contention detection, we have written our own atomic intrinsics for the T2.

When performing an atomic update a developer will use our atomic intrinsics, which also provide light-weight contention measurement during the atomic operation. Two atomic intrinsics used in this paper are 64-bit atomic increment and 64-bit atomic add, named `my_atomic_inc_64` and `my_atomic_add_64`, respectively. The `C` language usage of these intrinsics can be found in Figure 2. We also defined slight variants of these intrinsics that return the new value of the data element.

The code for the `my_atomic_add_64` intrinsic is shown in Figure 1 in Sun's assembly language. (`my_atomic_inc_64` is similar.) The most important instruction in this code is the compare-and-swap instruction `casx`, which is guaranteed to be performed atomically, without interference from other threads. The meaning of `casx [L],A,B` is:

> Compare the old value in location `L` with `A`, the expected old value. If they are the same, then exchange `B`, the new value, with the value in location `L`.
>
> Otherwise do not modify the value at location `L` because some other thread has changed the value at location `L` since `A` was read. Return the current value of location `L` in `B`.

After a compare-and-swap operation, one can determine whether the location `L` was successfully updated by comparing the contents of `A` and `B`.

The code in Figure 1 is based on Sun's atomic add intrinsic. It has only three additional instructions, for loading, incrementing, and storing the update-counter. Of these three instructions, two actually occupy otherwise-unused instruction slots. As a result, there is only one instruction's worth of visible overhead. We measured the performance of the alternative intrinsic, and found that it took about 7% percent longer than the original on the T2. The bulk of the time is spent on the relatively expensive `casx` instruction, so the relative cost of the update-counter is small.

An important advantage of our direct approach to measuring contention is that we get an *exact* and *current* measure of actual contention. Alternative methods (such as the sampling and estimation methods described in [5]) are approximate at best, are not responsive to short-range fluctuations in access patterns, and have sample collection/analysis overheads of their own. Further, such estimation techniques need to be aware of the algorithm's behavior. In contrast, our measurement does not have to know anything about the algorithmic behavior generating the reference patterns.

4. CONTENTION MANAGEMENT

We aim to address contention bottlenecks apparent for multicore machines. The primary hazard we wish to avoid is situations where many threads attempt to concurrently update a small number of data elements.

Unlike [5], we do not strive to design the best overall algorithm for a single computational task, to get the best possible performance out of the machine. To achieve the level of performance in [5], parallel code specific to the problem at hand was needed, code that could not be reused without much effort.

For algorithms of central importance (such as aggregation in a database system) this effort may be justified. However, the effort would have to be repeated for other similar tasks such as partitioning [6]. Further, the performance tuning aspects of the algorithms, including the various thresholds for switching between algorithms, would need to be recalibrated on each candidate platform. For many tasks, it would be a major advantage to be able to quickly write code that performs in a scalable fashion on a multicore machine, avoiding the contention hazards.

4.1 Abstractions

In what follows, we talk about a *data structure element*, which may be any component of a data structure that is referenced using a pointer. Thus, an element may be a leaf of a tree data structure, a hash bucket within a hash table, the head of a linked list, a simple counter, or any other suitable structure. The data structure element is encapsulated into an abstract data type (ADT). The ADT is aware that the element may have multiple internal versions of data. Nevertheless, the ADT provides a simple interface that allows the programmer to imagine that there is a single data item being manipulated.

Our generic solution to contention on a data structure element x is to create an additional version of x. For example, suppose we are accumulating a sum in x, and determine that we have contention on x. We *clone* x to create an additional element x_1 that is initialized to zero. With two copies of x, each will be accessed half as often as before. Our provided infrastructure takes care of determining when to clone, and which clone is accessed by which threads. At the end of the computation, the two data elements are *combined* (summed together in this example) to give the final result. If contention continues after the cloning step, additional clones can be created.

For such an approach to be meaningful, the computation on x must, like the `sfunc` function of a user-defined aggregate and the Reduce operation of MapReduce, be commutative up to equivalence.[2]

In a manner similar to user-defined aggregates or MapReduce discussed in Section 2.2, we expect a user to provide four template functions for a computation. The *create-clone* function specifies how a new version of a data element is created. The *combine* function specifies how multiple versions of a data element are merged into a single result. The

[2]For example, in partitioning different orders of elements within a partition are equivalent.

```
.inline my_atomic_add_64,0          ! %o1 contains update value
        ldx     [%o0], %o4          ! load current sum into %o4;
        ld      [%o2], %o5          ! load update-counter into %o5
1:
        inc     1, %o5              ! increment update-counter
        add     %o4, %o1, %o3       ! add value to current sum; put in %o3
        casx    [%o0], %o4, %o3     ! compare-and-swap %o3 into memory location of sum;
                                    !   %o4 contains the value seen
        cmp     %o4, %o3            ! check if compare-and-swap succeeded
                                    !   i.e., if %o4 is equal to %o3
        bne,a,pn %xcc, 1b           ! if not, retry loop starting at 1:
          mov   %o3, %o4            ! statement executed even when branch taken; %o4 now
                                    !   has a more recent value of the current sum and
                                    !   we have to add %o1 over again
        st      %o5, [%o2]          ! store the update-counter
.end
```

Figure 1: The `my_atomic_add_64` intrinsic.

```
bool AggregatorAtomicUpdate(Aggregator *agg, const uint64_t value){
  int32_t cas_counter = 0;

  my_atomic_inc_64(&agg->count, &cas_counter); /* atomic increment */
  my_atomic_add_64(&agg->sum, value, &cas_counter); /* atomic add */
  my_atomic_add_64(&agg->sum_squares, value*value, &cas_counter); /* atomic add */

  return (3 < cas_counter);
}
```

Figure 2: The atomic update operation.

simple-update operation specifies how the new value of a data element is obtained from the current value and an update. In the sum example above, the function would simply be "`x+=v;`" where `v` is the update.

The fourth (and most interesting) function is an *atomic-update* operation. We provide a set of atomic intrinsics based on compare-and-swap primitives provided by the architecture. The code given in Figure 2 is the *atomic-update* routine to compute three aggregates: a count, sum, and sum-of-squares.

To understand this code fragment, one needs to understand the meaning of `my_atomic_inc_64` and `my_atomic_add_64` presented in Section 3. The first argument of these intrinsics gives a pointer to the data item being modified. The second argument of `my_atomic_add_64` gives the value of the update being applied. The final argument of both intrinsics is the address of a counter. This counter is incremented within the intrinsic each time an update is attempted. An increment of 1 means that the update happened without contention. When contention happens, the intrinsic will make multiple update attempts. Contention may occur several (say n) times within a single intrinsic, meaning that the counter returned will be $n+1$ bigger than its previous value. For example, suppose the first intrinsic encountered contention twice, the second encountered no contention, and the third encountered contention once. At the end of the function, the `cas_counter` variable will have value 6; there were three successful updates, and three contention events that required the compare-and-swap update to be retried.

The final step of the *atomic-update* operation is to return a boolean indicating whether there was contention encountered during the computation. In the example above, the programmer has chosen to report contention to the system when more than three update attempts were detected altogether. Thus, with a `cas_counter` value of 6, the function returns *true*, informing the system that contention was detected. It is then the responsibility of the system to determine how to respond to the contention, which will be presented next in Section 4.2.

The use of these abstractions is not limited to aggregate functions. Another important application within database systems is data partitioning, which is used in many contexts, including sorting and joins. When data is being partitioned in parallel, there may be hotspots in the form of popular partitions [6]. We have implemented partitioning using these abstractions. In our implementation, a partition consists of a linked list of buckets that can each hold a fixed number of records. Writers to a partition atomically increment an index into the current bucket. This index corresponds to the slot in which to insert the record. When a thread reaches the end of the bucket, it obtains a lock on the partition and allocates[3] a new bucket that is prepended to the list. The *combine* operation simply concatenates two lists. While concatenation of lists may lead to buckets that are only partially full within the linked list, we believe that it is not worth the overhead of copying data between buckets;

[3]To avoid the time overhead of using `malloc` within a critical section, we manage bucket memory ourselves by preallocating a large chunk of memory and allocating buckets from that chunk as needed.

consumers of the data in the partitions are likely to remain efficient even when buckets are not completely filled.

4.2 Techniques for managing contention

Given that we can accurately and efficiently measure contention, what do we do about it when we find it? Our primary response to contention is to clone the contentious data element, and manage future accesses to the element so that they are divided among the available clones. To support this kind of policy, we need to maintain information about the current number of clones and to map threads to clones in a balanced fashion. Since this information will be consulted on every access, we need to minimize the number of instructions needed for a thread to get access to a version.

We consider two broad approaches to managing clones. We describe them below.

4.2.1 *Managing Clones Globally*

Under a global approach, we respond to a contention event by creating one or more new clones in a shared address space. Threads are partitioned among the clones in a deterministic fashion. For example, if there are four clones and 64 threads, then thread i will access clone number ($i \bmod 4$).

Clone allocation may happen in response to a single contention event, or in response to a threshold number of contention events. Using a threshold ensures that we only create clones when contention is sufficiently frequent. However, such an approach requires the maintenance of a contention counter for each data element. Responding to every contention event means that the system will respond more quickly to contention events, and no contention counters are needed. However, more memory may be consumed, leading to additional cache misses. (An extreme example is described in Section 5.2.)

In our implementation, we clone a resource in response to a single contention event. If the contention is on a resource that has already been cloned, then the number of clones is doubled. Thus, each contention event leads to an increasing number of new clones in the system. In this way, we can respond to contention quickly. For example, we can get to 64 clones of a heavy-hitter element after 6 contention steps. Had we chosen to simply add one more clone for each contention event, 63 steps would be required.

In the special case where cloning has happened sufficiently often that each thread maps to its own clone, we can avoid synchronization altogether. That thread will have exclusive access to a single version of the data item. Since atomic operations are significantly more expensive than their non-atomic counterparts, utilizing this many clones could be a useful optimization for a few popular data elements.

4.2.2 *Managing Clones Locally*

An important observation based on [5] is that there can be only a relatively small number of truly contentious data items at any point in time. On the experimental platform of [5], contention is noticeable only when a data element is responsible for more than one eighth of the data accesses. In such a scenario, there can only be eight contentious elements at any given moment. The identity of the contentious elements may vary over time as the reference pattern of the data changes, but the total number of currently contentious elements remains bounded.

The precise number (eight in [5]) depends on the computational task and the number of available threads. On a faster machine with more threads, the number would be higher. Nevertheless, even if it was an order of magnitude higher, it would still be a small number in absolute terms.

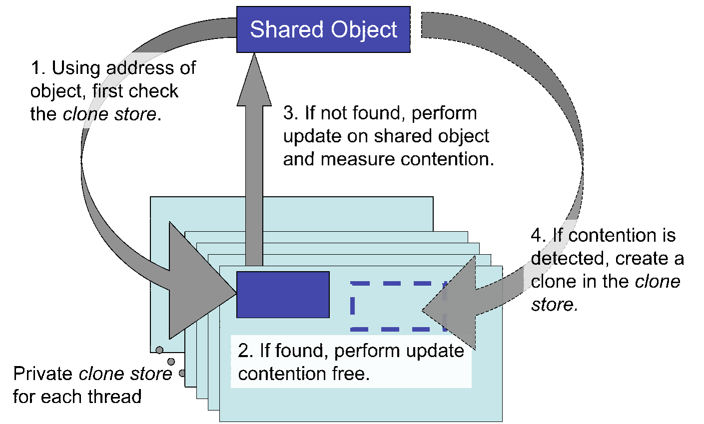


Figure 3: The process involved in managing the clones locally.

With this size bound in mind, we propose an alternative local design for clone management, which is depicted in Figure 3. When a thread sees contention on a data element, it creates a clone in a local table used by that thread alone. The size of the local table is kept relatively small, e.g., smaller than the thread's share of the L1 data cache. When the table is full, new insertions are accomplished by spilling an existing value into the global data element.

Each new access is handled by first looking in the local table. If a clone of the data element is found, the data value is updated there using a fast non-atomic update. Heavy hitters that initially cause contention should become resident in this table, after which further contention is avoided. Since the local table is L1-resident, the overhead of this lookup should be small for values that are not found in the table, and are subsequently updated atomically in the global data element.

4.2.3 *Contention Thresholds*

In Figure 2, the contention threshold for aggregation is 3. If more than three attempts to atomically update three data values are made, we report contention to the contention manager. For partitioning, there is a single atomic update to the bucket index, and so we could analogously report contention if more than one attempt is needed. However, we chose to use a threshold of two rather than one. In partitioning using local clone management, if swapping happens too often, one may end up with many nearly-empty buckets, wasting space and time for memory allocation. The higher threshold helps distinguish between true contention and spurious events in which two threads happened to be updating the same element at the same time without broader contention.

5. EXPERIMENTS

5.1 Experimental Setup

All experiments were conducted on a 1.2GHz Sun Fire T2000 server with an UltraSparc T2 processor (Table 1).

Operating System	Solaris 10
Cores (Threads/core)	8 (8)
RAM	32GB
Shared L2 Cache	4MB, 16-way associative
L1 Data Cache	8KB per core Shared by 8 threads
L1 Instruction Cache	16KB per core Shared by 8 threads
Off-chip bandwidth	60GB/s, 4 memory controllers

Table 1: Specifications of the Sun UltraSparc T2.

Unless otherwise stated, we use all 64 available hardware threads.

In this section we describe some of our initial experiments. Our goal is to demonstrate that it is possible to overcome contention hotspots using the proposed framework. We emphasize that we do not aim to find the very best parallel algorithm for a specific task, such as aggregation. Our goal is to enable a quick implementation of more generic tasks that need to update a shared data structure in parallel, while retaining high performance on a parallel machine.

5.1.1 Implementation Details

The first workload we consider is an aggregation workload similar to that in [5]. 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 group-by value `G` and an aggregated value `V`. Though the query is identical, the implementation is very different from [5]. [5] was highly optimized and hard-coding of the aggregate functions allowed the compiler to generate very tight inner loops and use optimizations such as loop unrolling to reduce the instruction count.

In contrast, the aggregation implementation in this paper uses the framework presented in Section 4. We have built an aggregator ADT that implements the four functions described in Section 4.1, including the atomic update function shown in Figure 2. Not only do we use function calls (whereas [5] avoided them), we also use function pointers. Clearly, the compiler will not be able to optimize this code as successfully, and we expect that our absolute performance will not match that of the algorithms described in [5]. Nevertheless, the point of the present work is not to match [5], but to investigate frameworks that can be used to avoid contention in a more general way than the careful coding and domain-specific analysis used in [5].

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. All comparison and arithmetic operations use 64-bit integer instructions.

The second workload we consider is data partitioning. We use the same input as for the aggregation experiments, and partition the data into a fixed number of partitions based on a hash function of the grouping attribute. Both workloads use a common infrastructure. Only the abstract functions discussed in Section 4.1 are different.

For data-intensive applications such as databases, it is important to use a large page size. If pages were small, such as 4 kilobytes, then a 64-element TLB can only cover 256 kilobytes of RAM; data structures much larger than that would likely cause TLB thrashing. On the T1 and T2, it is possible to have memory regions with different page sizes, and pages can be as big as 256 megabytes. Our implementation makes particular effort to allocate large data structures on large pages to avoid TLB thrashing. (For a more detailed discussion of how page size choices influence TLB behavior, see [6].)

5.1.2 Input Distributions

We consider multiple input distributions, where the distribution refers to the characteristics of the group-by key in the input relation. For ease of comparison, use the synthetic distribution generation code from [5] (which was based on the work of Gray et al. [14]) and used the same distributions. For each input distribution, we vary the number of distinct group-by keys in the distribution. The distributions are: (1) uniform, (2) sorted, (3) heavy hitter, (4) repeated-run, (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 repeated-run 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 $1, 2, \ldots, 10000, 1, 2, \ldots$. The self-similar distribution uses an 80-20 proportion, and the Zipf distribution uses an exponent of 0.5. In the moving-cluster distribution, there is a window of data locality that gradually shifts as the input sequence progresses.

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 always met, especially when the requested group-by cardinality approached the size of the input.

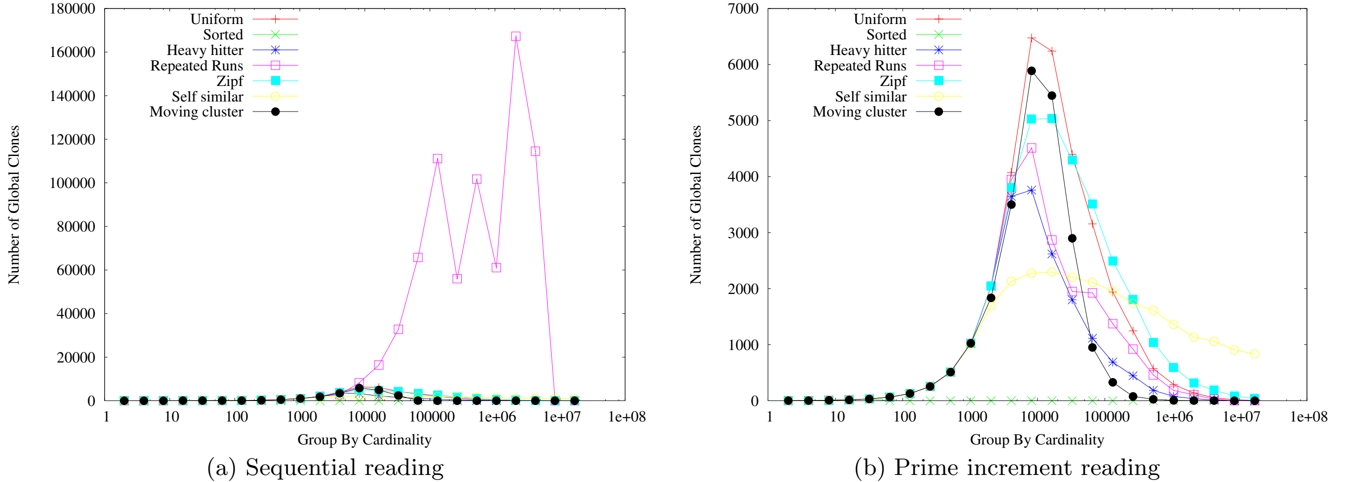
The input is divided into 64 equal chunks to match the number of available threads. Each thread processes its chunk in its entirety. Considering other methods of dividing the input, such as using a parallel buffer [8], is left to future work.

5.2 Cache and Memory Issues

On modern architectures, cache misses can cause significant latency. Accessing an item that is not in the L2 cache can take hundreds of CPU cycles. As a result, it pays to design algorithms and data structures so that they have good spatial and temporal locality.

While implementing our prototype of the global clone management scheme described above, we encountered the following scenario that illustrates the difficulties in developing a robust generic system. Imagine that we are computing a multi-threaded aggregation in which, in every thread's input stream, the group-by values cycle in the following pattern according to our repeated-run distribution:

$$1, 2, \ldots, n, 1, 2, \ldots, n, 1, 2, \ldots$$



(a) Sequential reading (b) Prime increment reading

Figure 4: Number of group by values where contention has been detected and at least one clone constructed.

When n is large, we would hope to encounter relatively little contention, because there are no elements that have a frequency of more than $1/n$.

Unfortunately, significant contention is observable. After some analysis, we realized that every so often, there will be a contention event caused by the random synchronization of two threads. When this happens, a new clone will be created, and processing will continue. The problem is that because the input group-by sequences are the same, a cascade of cloning events happens as element after element experiences momentary contention. Even worse, the cloning and contention slows down the contending threads, allowing the remaining threads to "catch up" and ultimately compound the contention problem by updating the same data elements in lockstep, a phenomenon known as convoying.

Eventually, the contention itself is resolved, as all elements reach t clones, where t is the number of threads. By that time, however, we have used space nte, where e is the size of a data element, rather than ne. On the T2 where $t = 64$, that means we exceed the cache size 64 times earlier than with a single copy of each data element. As a result, we suffer many more cache misses whenever $c/64 \leq ne \leq c$, where c is the cache size. This is a large range of sizes.

Figure 4(a) shows the number of elements that have at least one clone after processing 2^{24} tuples. Note that at a group by cardinality of around 100000 the number of elements in the repeated runs distribution that have clones is also close to 100000. The number of clones drops to zero at the far right of the figure as the repeated runs distribution degenerates to a sorted distribution with all unique values so contention is not possible.

Since our goal is to create a *robust* infrastructure, we cannot ignore such problems. We should have confidence that there is no "degenerate" distribution of inputs that causes a drop in performance that would surprise the programmer using the system.

Our first attempt at a solution was to force each thread to cycle through its input records using a prime increment, wrapping around to the start of the array until all records are consumed. If each thread uses a different prime increment, then the threads will have different effective access patterns and not end up in lockstep. (If one is worried about an adversary designing a degenerate workload, one could even choose the prime increments at random.)

While this attempt solved the contention artifact described above, it introduced a new cache miss problem. Each cache line was being read for just one input record. By the time we cycled around to get another record from that cache line, the cache line was no longer cache-resident, and we suffered an additional cache miss. If there are r records in a cache line, we suffer one miss per record rather than $1/r$ for a sequential scan.

We addressed this second cache miss problem by reformulating the traversal so that we use all of the records in a cache line, and cycle through the cache lines (rather than the records) using a prime increment. The results are shown in Figure 4(b). Note that the y-axis scale is significantly smaller than in in Figure 4(a). Although clones are being created, the total number is much lower.

5.3 Global Clone Management

The performance of global clone management on various input distributions is shown in Figure 5. Global clone management helps aggregation avoid the contention that is present at low group by cardinalities or in input distributions with frequently occurring values, e.g., zipf, self similar, and heavy hitter. For many points on the spectrum, the performance improvement is more than an order of magnitude.

It is also important to consider the memory footprint of this technique. Given an input whose distribution is unchanging, one would expect the number of contentious elements to be some small factor of the number of threads. But as Figure 4 shows, the number of group by values that experience at least one contentious update is quite a bit higher than the number of threads. This phenomenon is demonstrated in Figure 6, which plots the percent of group by values encountering at least one instance of contention and therefore having at least one clone. The first clone causes the most memory allocation overhead as the contentious element is transformed to enable support for clones.

Why do 100% of group by values experience contention even after there are many more values than threads? The answer is that random collisions happen more often than one might think. With a uniform input of N elements and

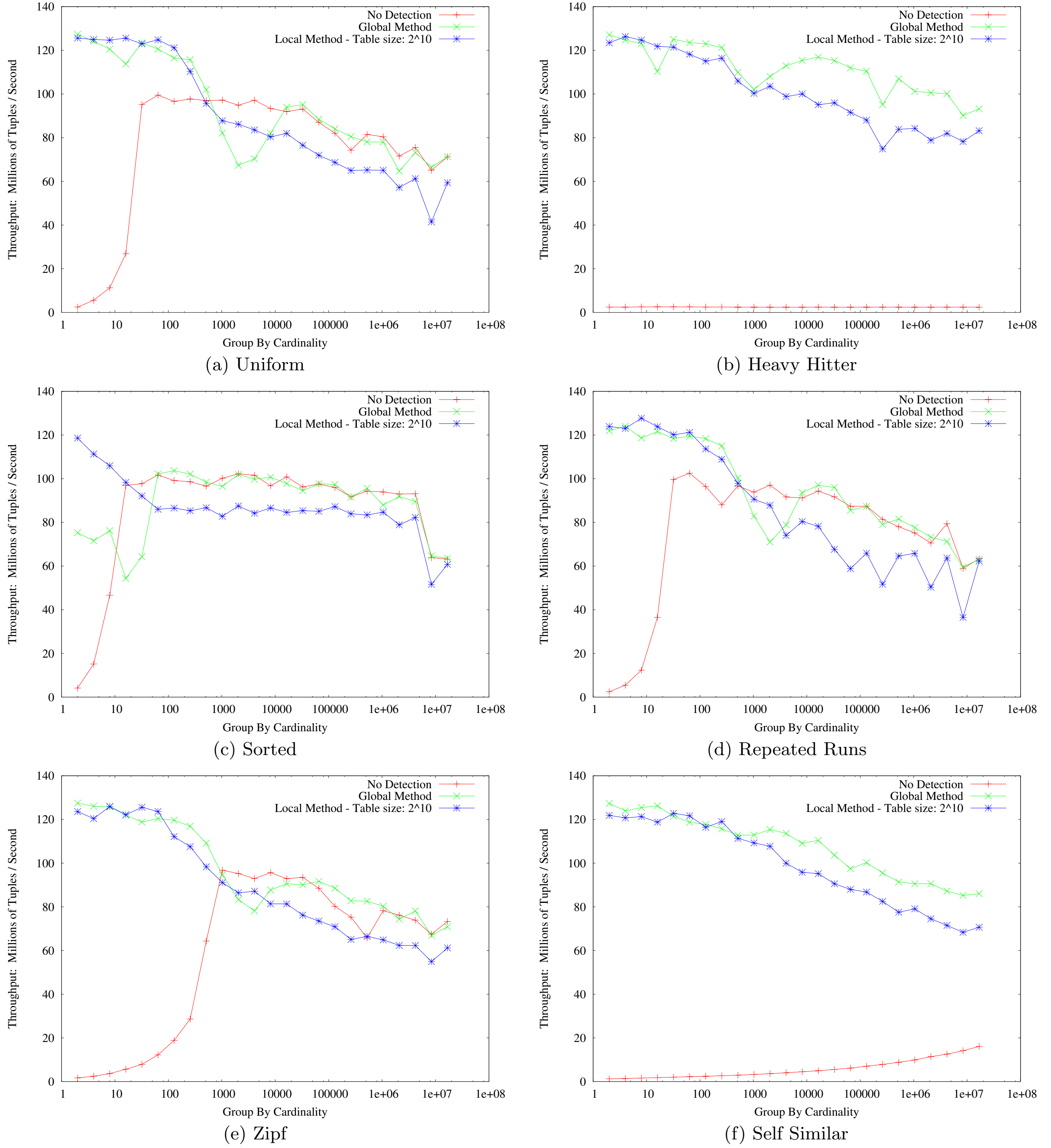
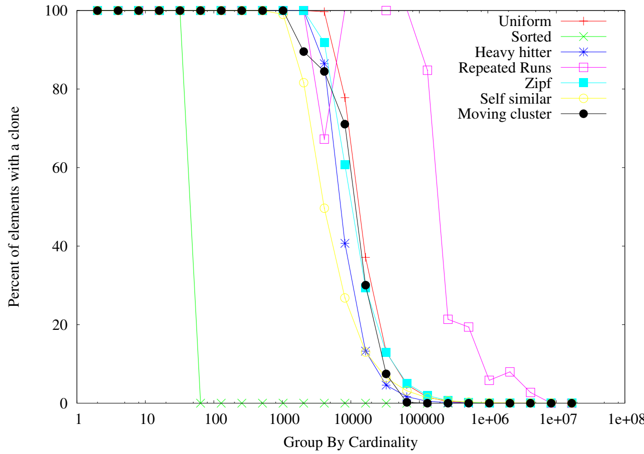


Figure 5: The proposed techniques, global and local clones, on six representative data distributions.

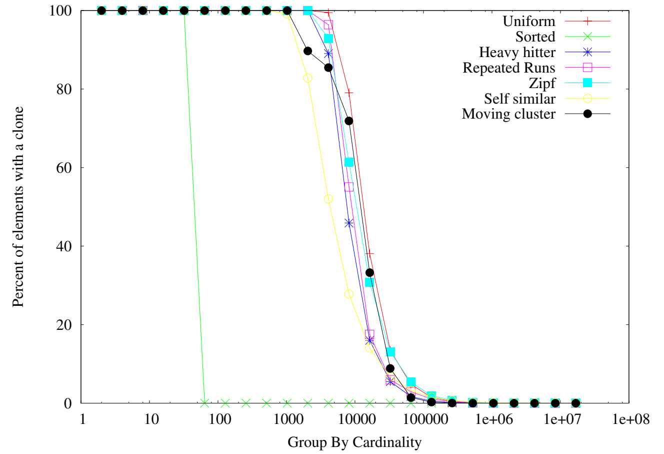
a group-by cardinality of g, the total number of random contentions per group-by element is $N\left(1-(1-\frac{1}{g})^{63}\right)/2g$. (We divide by $2g$ rather than g to avoid counting a collision twice, once for each thread.) At $N = 2^{24}$ and $g = 1,000$, we expect about 500 random contentions per group-by element. At $g = 10,000$, this number drops to about 5, and by $g = 100,000$, the number drops further to about 0.05. These expectations are consistent with the uniform curve in Figure 6.

5.4 Local Clone Management

The performance of local clone management, described in Section 4.2.2, is also shown in Figure 5. In all of these per-



(a) Sequential reading



(b) Prime increment reading

Figure 6: Percent of group by values for which at least one clone has been created.

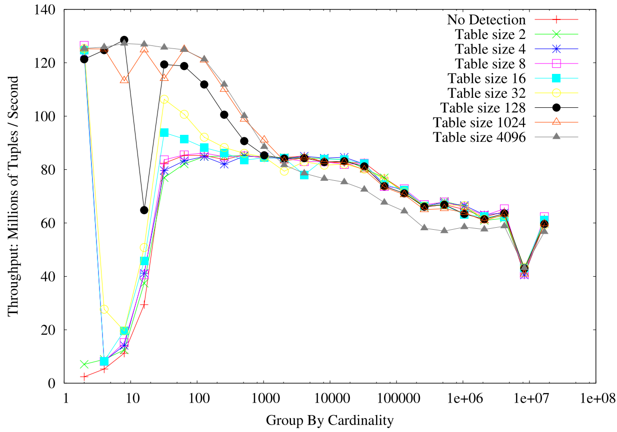


Figure 7: Varying the local table size. Size is the number of clones.

formance graphs the size of the local table is 1024 elements. Like the global clones, using local clones clearly mitigates the contention bottleneck.

This 1024 element table size exceeds the L1 cache design principle proposed in Section 4.2.2. Although a smaller table size did perform well, a larger table size was found to be better because of the higher than expected thrashing of elements in the local table due to the infrequent random contention events described in the previous section. The effect of table size can be seen in Figure 7. As expected, table sizes that are too small do not fully alleviate contention. As the table size increases to 4096 elements, the cache performance causes a reduction in throughput for larger group-by cardinalities.

Because the local clone table size is fixed, the run-away memory allocation that can occur with global clones is not possible. When the table is full, every addition of a contentious element must evict an element from the table whose value is then pushed back into the global table. Therefore, applications seeking greater control over memory usage should consider the local tables approach.

At high group by cardinalities, Figure 5 shows that the local tables approach performs consistently worse than the global tables approach. This is because checking for an element in the local table becomes pure overhead: almost all elements will not be found in the local table. This suggests a potential optimization to disable local table lookups automatically when the system determines that finding an element is unlikely. The benefits of such an optimization would have to outweigh the cost of determining the rate of hits to the local table.

5.5 Partitioning

We present performance results for partitioning with local clone management. The size of each local table is fixed at 1024 partitions. Figure 8(a) shows the partitioning rate for a uniformly distributed input with 2^{24} distinct group-by values. When contention management is turned off, we see poor performance when the number of partitions is small, due to contention among the 64 threads for the partition buckets. In contrast, when contention management is turned on, local clones are created, and very high throughput is obtained in this range because most partitions are L1 cache resident and most updates can be achieved without using atomic instructions. These results show that our contention detection framework is effective at identifying and fixing contention-related performance problems.

Eventually, the number of partitions becomes too large to fit in the L2 cache, and performance drops. The blue line in Figure 8(a) shows the L2 cache misses per record for the contention-management-on code; the scale for this curve is on the right of the graph. There is a potential interaction between cache misses and contention. If a cache miss occurs during an atomic operation, then the time window during which contention is possible becomes wider.

Figure 8(b) shows the performance of 8192-way partitioning on the repeated-runs distribution. When the group-by cardinality hits 4096, performance drops dramatically. (The performance returns to “normal” when the cardinality is high, where the repeated runs are very long.) We traced this problem to the same convoy effect described previously for aggregation. When there are many partitions and all threads have identical access patterns, threads degenerate into a lockstep pattern in which there is thrashing on an

associativity set within the L2 cache. We applied the same prime cache-line increment method described earlier, which eliminates the sudden performance drop, with only a modest overhead for for smaller group-by cardinalities.

Figure 8(c) shows the performance of 2048-way partitioning for all distributions when contention detection is enabled and the prime cache-line increment is employed. Performance is high for all distributions. Once the group-by cardinality exceeds 2048, the performance levels off because there are only 2048 partitions being used.

5.6 Absolute Performance

We ran the special-purpose algorithms from [5, 6] on the T2 architecture to see how close the generic implementation comes in terms of performance. As discussed in Section 5.1.1, the system cannot employ the same level of code optimization because our implementation is generic. Further, the present paper considers just one algorithm, with no programmer-controlled parameters to tune (apart from the threshold of Section 4.2.3). In contrast, [5] switches between several algorithms depending on domain-specific knowledge of sampled data and tuned thresholds.

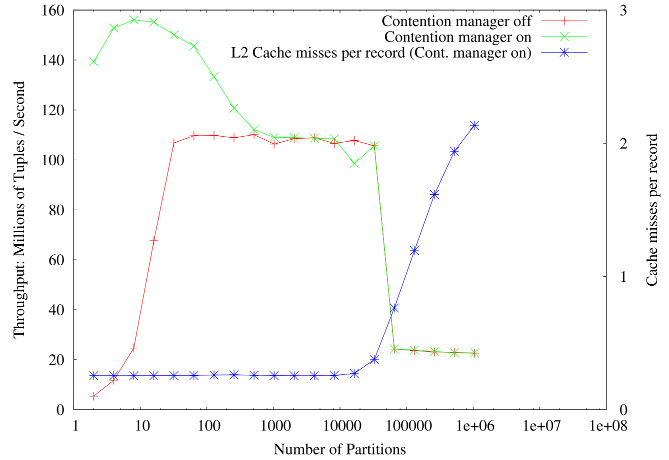
On the T2, the atomic algorithm of [5] hits a performance plateau of about 130 million records per second for large group-by cardinalities in the absence of contention. Our "no detection" method, reaches about 100 million records per second. The only significant difference between these methods is that the "no detection" method is implemented generically. Thus, we can estimate that the overhead of a generic implementation is a factor of about 1.3.

On the T2, the hybrid algorithm of [5] achieves a performance of about 250 million records per second for small to moderate group-by cardinalities for which most data fits in the local table. Our local and global methods reach about 125 million records per second in comparable ranges. Thus, we can estimate that the overhead of a generic implementation of aggregation relative to a special-purpose implementation is about a factor of 2. For partitioning, we get about 110 million records per second for 2048-way partitioning, a region where the special-purpose implementation achieves 239 million records per second, suggesting a factor of 2.2 overhead.

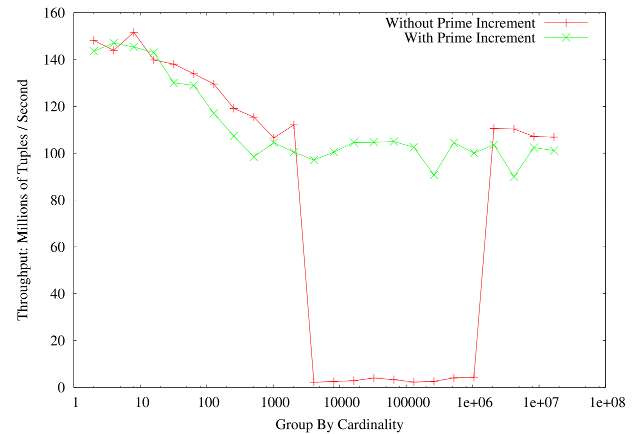
We argue that this is a good trade-off point, because (a) the performance was achieved with minimal performance tuning knowledge on the part of the programmer, and (b) the dramatic decrease in performance that occurs due to contention is avoided and the parallel resource is being used effectively.

5.7 Global vs. Local

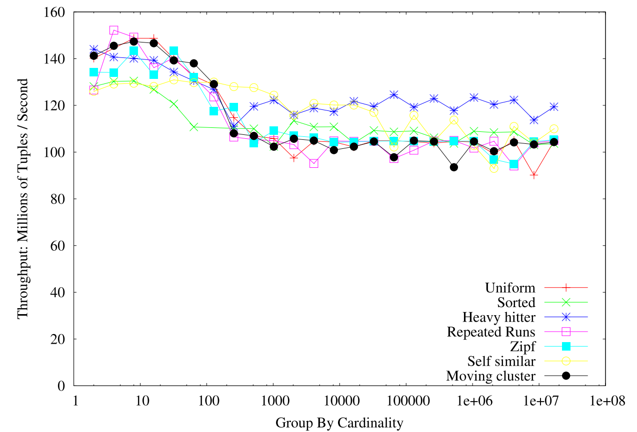
Based on our experiments, both local and global clone management are effective at eliminating contention bottlenecks. Apart from the sorted distribution on small group-by cardinalities, the global method performs slightly better. However, it may consume more memory as discussed above. To avoid convoys, both methods need to employ the prime cache-line increment technique. This technique is straightforward to implement when the input is an array of fixed-length records. For variable-length records (or for dynamically generated inputs), such a technique would need additional bookkeeping to properly process the input a few cache lines at a time.



(a) Contention management on vs. off.



(b) Prime cache line increment.



(c) 2048-way partitioning, various distributions.

Figure 8: Partitioning with local clones.

6. EXTENSIONS AND FUTURE WORK

While we have focused on atomic operations here, one can also come up with mechanisms to manage contention via locking. Contention can be detected as a failed attempt to obtain a lock. While lock-based mechanisms may be more

expensive than atomic operations, there are computations that are not easily expressed via atomic updates, and would require a lock-based implementation. The same abstract functions would be usable, with the system implicitly placing locks around the non-atomic update function.

One could further improve performance of the prime-increment method by using software prefetching to overlap some of the cache miss latencies for the various cache lines being read.

An advantage of implementing contention detection and amelioration generically is that the framework could be useful for a variety of applications. We have focused on aggregation and partitioning, which can be expressed naturally using our abstractions. In future work, we plan to investigate how broadly our abstractions can be applied to parallel data-intensive computations.

The Sun Niagara T1 and T2 machines are convenient for studying contention because they have so many parallel thread contexts. Future machines from other vendors will soon match or exceed this level of parallelism.

We plan to implement our framework on multiple hardware platforms, and investigate the trade-offs necessary to get good performance on each of them. The abstractions will not change across platforms, meaning that programmers can write code that is portable from one supported architecture to another. The "intelligence" of efficiently managing contention on each architecture will have been factored out into code libraries.

7. CONCLUSION

Our goal in this paper was to enable a programmer to avoid contention hotspots without requiring sophisticated parallel programming or performance tuning. We provided abstractions that allow a class of data-intensive computations to be formulated in a simple way, using four basic functions.

We have built a system that uses these four functions to perform the computation in a multithreaded way. Contention is detected in a low-overhead fashion. When contention is detected, we clone the contentious resource so that future references are distributed among the clones. We have implemented a global clone management scheme, and a local clone management scheme, and measured their performance. We have shown that contention hazards that could cause an order-of-magnitude drop in performance are avoided, and that the computation makes efficient use of the parallel resources available.

8. REFERENCES

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

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *In Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, 1994.

[3] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of VLDB Conference*, 1999.

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

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

[6] John Cieslewicz and Kenneth A. Ross. Data partitioning on chip multiprocessors. In *DaMoN*, 2008.

[7] John Cieslewicz and Kenneth A. Ross. Database optimizations for modern hardware. *Proceedings of the IEEE*, 96(5), 2008.

[8] John Cieslewicz, Kenneth A. Ross, and I. Giannakakis. Parallel buffers for chip multiprocessors. In *DaMoN*, pages 9–18, 2007.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the Symposium on Operating Systems Design & Implementation*, 2004.

[10] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Comm. ACM*, 35(6), 1992.

[11] Simon Doherty et al. DCAS is not a silver bullet for nonblocking algorithm design. In *SPAA*, 2004.

[12] Michael Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[13] Goetz Graefe and Per-Ake Larson. B-tree indexes and cpu caches. In *Proceedings of the 17th International Conference on Data Engineering*, pages 349–358, Washington, DC, USA, 2001. IEEE Computer Society.

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

[15] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 2nd edition, 1999.

[16] Nikos Hardavellas et al. Servers on Chip Multiprocessors: Limitations and Opportunities. In *CIDR*, pages 79–87, 2007.

[17] Sándor Héman, Niels Nes, Marcin Zukowski, and Peter Boncz. Vectorized data processing on the cell broadband engine. In *DaMoN*, pages 1–6, 2007.

[18] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[19] Khronos Group. OpenCL overview. http://www.khronos.org/opencl/.

[20] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-9, Columbia University, 1991.

[21] Bradford Nichols, Dick Buttlar, and Proulx Farrell. *Pthreads Programming.* O'Reilly, 1996.

[22] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman. Main-memory scan sharing for multi-core CPUs. *Proc. VLDB Endow.*, 1(1):610–621, 2008.

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