Parallel Buffers for Chip Multiprocessors

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

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

Ioannis Giannakakis[†]
Columbia University
giannis@cs.columbia.edu

ABSTRACT

Chip multiprocessors (CMPs) present new opportunities for improving database performance on large queries. Because CMPs often share execution, cache, or bandwidth resources among many hardware threads, implementing parallel database operators that efficiently share these resources is key to maximizing performance. A crucial aspect of this parallelism is managing concurrent, shared input and output to the parallel operators. In this paper we propose and evaluate a parallel buffer that enables intra-operator parallelism on CMPs by avoiding contention between hardware threads that need to concurrently read or write to the same buffer. The parallel buffer handles parallel input and output coordination as well as load balancing so individual operators do not need to reimplement that functionality.

1. INTRODUCTION

Modern database systems process queries by constructing query plans. A plan consists of a collection of operators connected to each other by data buffers. The output from one operator is fed as input to another operator.

Recently, microprocessor designs have shifted from fast uniprocessors that exploit instruction level parallelism to chip multiprocessors that exploit thread level parallelism. Because of power and design issues that reduce the performance improvement obtainable from faster uniprocessors, improved performance now depends on taking advantage of on-chip parallelism by writing applications with a high degree of thread level parallelism [11].

In this paper, we focus on query plans running on a chip multiprocessor. On such a machine, many concurrent threads may cooperate to collaboratively perform a single database operation [20, 6]. The advantage of using the available parallelism in this way is improved locality. Instructions and some data structures are shared, leading to good cache behavior. In contrast, treating each thread as a parallel processor that performs independent tasks can lead to cache interference [20].

[*]Supported by a U.S. Department of Homeland Security Graduate Research Fellowship
[†]Supported by NSF grant IIS-0534389

Proceedings of the Third International Workshop on Data Management on New Hardware (DaMoN 2007), June 15, 2007, Beijing, China.
Copyright 2007 ACM 978-1-59593-772-8 ...$5.00.

In a collaborative design, an operator would be executed by all threads for a certain time-slice. The time-slice needs to be long enough to amortize the initial compulsory misses on the instruction and data caches, as well as the context-switch costs. A time-slice might end when either (a) a time-window expires, (b) the input is fully consumed, or (c) the output buffer becomes full.

A critical question for a system employing parallel operators is the design of buffers for passing data between operators. A naïve choice can lead to hot-spot points of contention that prevent the operator from working at its full capacity [3]. For example, if all threads write output records to a common output array, then there will be contention for the mutex on the pointer to the current position within the array.

One way to avoid output contention is to give each thread its own output array [3, 20]. While this choice avoids contention, it has other disadvantages. The next operator that consumes the data has to be aware of the partitioned nature of its input. This can lead to a relatively complex implementation for all operators, because they have to take into account run-time information such as the number of available threads, which may vary during the course of query execution. An output partition could, in the worst case, grow much faster than the others. For instance, consider a parallel range selection operator where each thread runs on a portion of the data that has been partitioned by the attribute being selected. As a result, each output thread must be pessimistic in allocating output space consistent with worst-case behavior. Such allocation can waste memory resources, particularly when there are many threads.

There is no guarantee that the separate output partitions will be balanced. Therefore, the consuming operator also becomes responsible for load balancing. If the operator does not balance the load, then it is possible for a single slow thread to cause all other threads to stall for long periods, substantially reducing the effective data parallelism. We take a closer look at load balancing in Section 5.4.

Simply put, a challenge for multithreaded database and other similar data intensive workloads is that each thread may consume different amounts of input, generate different amounts of output, and take significantly different amounts of time to run. In this paper, we propose a buffer structure that avoids these pitfalls, while still minimizing contention for a shared buffer. Our solution has the following desirable properties:

- The output and input structures are the same, so that arbitrary operators can be composed.
- The buffer is allocated from a single array, so that memory can be allocated in proportion to the *expected output of all threads* rather than the worst-case output.
- Data records are processed in chunks. Mutex or atomic operations are required only for whole chunks. By choosing sufficiently large chunks, contention can be minimized.
- Parallel utilization is high. In particular, no thread is stalled for longer than the smaller of the input-chunk processing time and the time to generate one output-chunk. Utilization is high even when there is an imbalance in the rate of progress made by the various threads.
- Operators use simple `get_chunk` and `put_chunk` abstractions, and do not have to re-implement locking or load-balancing functions.

We evaluate our parallel buffer data structure on real hardware, the Sun Microsystems UltraSPARC T1. The T1 is a chip multiprocessor with eight cores and four hardware threads per core for a total of 32 threads on one chip.

2. RELATED WORK

Parallelism in databases has been well studied, however most research, and therefore the lessons learned, predate chip multiprocessors. DeWitt and Gray [5] and DeWitt et al. [4] advocate *shared-nothing* parallelism for database operations. A key part of this argument is that interference limits the performance of *shared-memory* systems. Modern commodity chip multiprocessors exhibit shared-memory parallelism, sharing some levels of the memory hierarchy. Therefore, managing interference between hardware threads will be paramount to achieving good query performance.

Graefe [8, 7] advocates creating intra-operator parallelism via partitioning. On a shared memory system, static partitioning makes sense when the coordination overhead between processors is high, but this coordination overhead, such as cache coherency, is lower on chip multiprocessor because all communication is done on chip. A problem with data partitioning is that it is static and can be sensitive to skew in the data, resulting in sub-optimal load balancing. Whereas a conventional shared memory multiprocessor system avoids close coordination between threads or processes because of the high cost of coordination and data sharing, chip multiprocessors benefit from it because cooperative threads can better share on-chip resources and any coordination is done at on-chip speeds.

A recent study by Hardavellas et al. [10] explored database systems on chip multiprocessors. This work found that OLAP workloads on chips similar to the UltraSPARC T1 exhibit good throughput. In this study, parallelism in database operations was achieved by increasing the number of concurrent clients accessing the database (inter-query parallelism). The good throughput was found when the number of clients saturated the system. When few concurrent clients were connected, throughput was poor because some hardware threads were idled. This highlights a significant performance pitfall of parallel architectures: not taking advantage of the parallelism [11].

Our proposal to exploit intra-operator parallelism for OLAP aims to keep all hardware threads busy, regardless of the number of concurrent clients, thus yielding high system utilization and throughput. Additionally, by exploiting intra-operator parallelism, it is easier to manage resource sharing and thus improve performance because all hardware threads are working on the same task. In contrast, in an inter-query parallelism model, threads sharing memory, cache, and execution resources may conflict. Understanding and managing inter-operator or inter-query interference is fraught with complications.

Parallel queue data structures on shared memory systems have been studied in the past [16, 15, 14, 18]. These investigations focus on creating general purpose queue structures that allow concurrent enqueue and dequeue operations. This is most useful in situations with multiple concurrent producers and consumers. In our parallel buffer structure described in Section 3, we leverage the semantics of database processing to guarantee that only concurrent enqueue or dequeue operations to a buffer occur during a time slice, but not both.

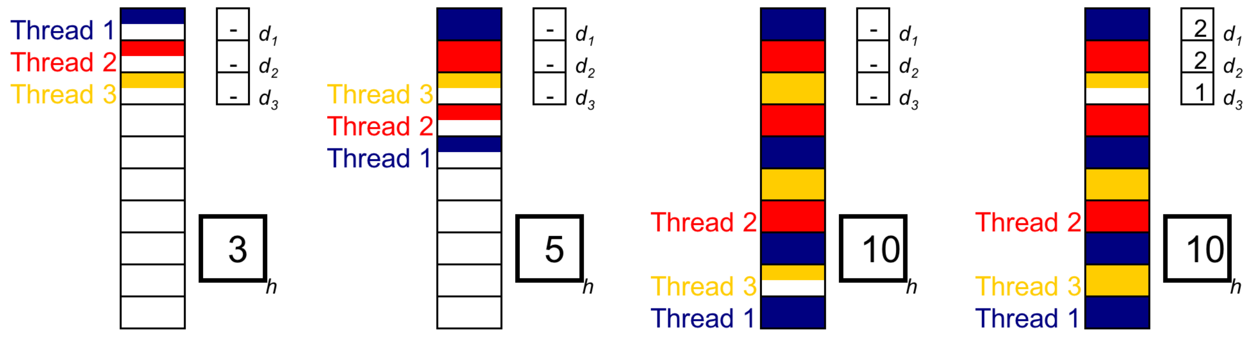
Other work has suggested inserting buffers between operators [21] and processing blocks of input at a time and materializing the intermediate results instead of pipelining [12, 17]. Advantages of this block processing approach include the efficient reuse of instructions and data structures, such as an index. With block processing, fewer instruction and data cache misses occur because the operator's instructions and data structures remain cache resident. Work by Boncz et al. [2] shows that processing a vector of tuples can lead to more than an order of magnitude performance improvement compared to previous Volcano-style query execution. In pipelined query execution, an operator may process only one tuple, yet must pay the cost of many cache misses for both its data structures and instructions. On a chip multiprocessor with shared cache and execution resources, block processing is even more important because it allows for easier concurrent management of these shared resources. The parallel buffer proposed in this paper will help enable multi-threaded block processing on chip multiprocessors.

3. PARALLEL BUFFER

We assume records have fixed length, and allocate an array that is capable of holding a large number M of records. We divide the array into chunks of size c, and assume M is a multiple of c. The buffer structure maintains a count h of the number of chunks in the array that are in use. Additionally, the structure contains p additional count variables $d_1, \ldots, d_p$ where p is the maximum number of thread contexts available on the system. The use of these counters will be described below.

The buffer is in a *stable* state when every chunk from $p + 1$ to h is fully occupied. In a stable state, the variable d_i denotes the number of records present in chunk i; chunks 1 through p may be partially occupied.

To write data to a buffer, a thread atomically increments the chunk counter h, and uses chunk number h as the destination for output records. Each thread will be accessing a different chunk, and so actual data output does not need to be regulated by locks or mutexes. Only accesses to h are controlled, using the atomic increment instruction. Once an



(a) Each thread starts writing to its first chunk.

(b) Chunk counter incremented as threads write to new chunks.

(c) Threads exhaust output buffer space.

(d) Threads *top-up* and update counters to reflect first chunks' actual counts.

Figure 1: Parallel buffer example with three threads where threads terminate because the output buffer capacity is reached. Each chunk can hold two tuples.

output chunk is full, the thread tries to obtain a new chunk in the same fashion. When no more chunks are available, i.e., $h = M/c$, a flag is returned to the operator to signal this fact. A thread in this state is said to be *finalized*, and will stall until all other threads are also finalized.

Reading data from a buffer proceeds in a similar fashion to writing. When reading a stable buffer, a thread will be told how full its chunk is based on the d_i values. A thread that requests a new input chunk and finds none available also enters a *finalized* state. Such a thread (say it is thread number j) performs a *top-up* operation, in which its current output chunk is topped up using data from chunk j. The count d_j in the output buffer is set according to how many records remain in chunk j. The top-up operations restore the buffer to a stable state.

Figure 1 demonstrates the filling of an output buffer when threads terminate because the capacity of the output buffer is reached. Note that finalization begins when one thread fails to obtain a new output chunk. Finalization prevents threads from obtaining new input chunks, thereby guaranteeing termination of all threads quickly. This can leave holes in the buffer, as shown in Figure 1, requiring a *top-up* operation to return the buffer to a stable state. Figure 2 demonstrates the filling of an output buffer when threads terminate because the input is exhausted.

Finalization can happen due to a full output buffer or an empty input buffer. Once one thread becomes finalized, we *induce* finalization in all other threads by preventing them from obtaining new input or output chunks. For example, if one thread has found the output buffer full, then no other threads can get new input chunks. When another thread tries to get a new input chunk, it sees that the finalization flag has been set, and instead enters the top-up phase and finalizes itself.

The advantage of coordinating finalization between input and output is that we can bound the idle time of all threads to the smaller of the time taken to process one input chunk, and the time taken to generate one output chunk. Without coordination, it might be possible for one thread to continue operating on the input for a very long time, even after all other threads have finalized due to running out of output chunks. If the thread's operator is very selective, for example, many input records would need to be consumed to generate an output record. While this remaining thread is making progress, it harms overall utilization because $p-1$ threads remain idle.

A dual problem can occur in the absence of input/output coordination. Suppose $p-1$ threads have finalized due to running out of input, but one remaining thread is generating a lot of output for each record in its chunk. This one thread would keep working, even though it is forcing $p-1$ threads to remain idle.

Finalization can also be externally induced, for example by an interrupt at the end of an operator's time-slice. Since we schedule operators one at a time, a buffer is used either for input or for output, but not both at the same time. Once a buffer has begun to be used as input, it cannot be used for output (by an upstream operator) until it has been fully emptied. Double-buffering can be used if upstream operators need to be rescheduled before downstream operators have consumed all of the previous records. Double-buffering is actually desirable, because our coordination mechanism can leave a buffer in a state with just a few remaining records.

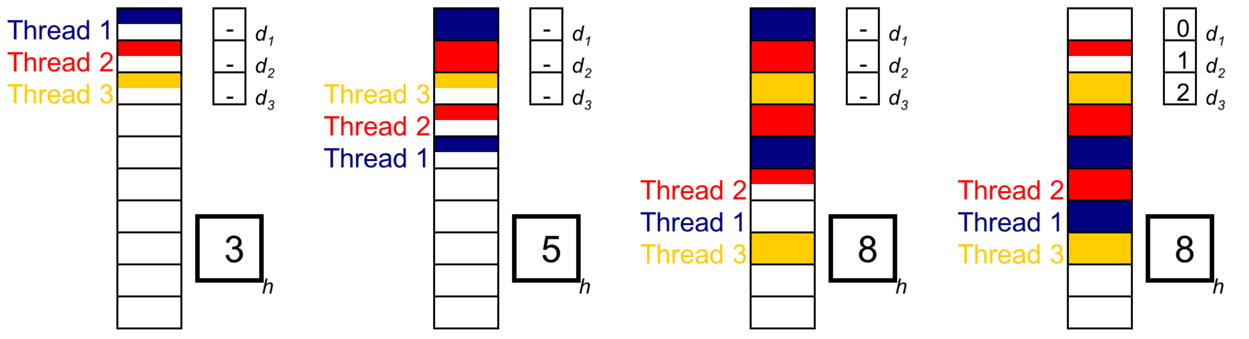
The contiguous nature of the parallel buffer data structure is a natural fit for the output of table scan operations often found at the leaves of a query plan, with the exception that the final chunks may need an inexpensive top-up operation.

4. MODELING BUFFER CONTENTION

The appropriate chunk size can be determined theoretically. We will first present a simple model that provides insight into issues related to contention and chunk size. Then we present a probabilistic model that provides a better estimation of the necessary chunk size to eliminate contention.

The total time to process a chunk is $T = cr$ where c is the size of a chunk in tuples, and r is the time to process a tuple. If the time to perform the necessary mutually exclusive operations for each chunk processed is L, then

$$(p-1)L \leq T - L \tag{1}$$



(a) Each thread starts writing to its first chunk. **(b)** Chunk counter incremented as threads write to new chunks. **(c)** Threads exhaust input and must terminate in current state. **(d)** Threads *top-up* and update counters to reflect first chunks' actual counts.

Figure 2: Parallel buffer example with three threads where threads terminate because the input is exhausted. The output is left unfilled, but in a *stable* state. Each chunk can hold two tuples.

where p is the number of hardware threads. As more hardware threads are added, the potential contention for locking or atomic operations increases. We find the size of a chunk that avoids contention is

$$c = \frac{T}{r} \geq \frac{pL}{r} \tag{2}$$

Equation 2 shows that the chunk size must increase as more hardware threads are used or the time per tuple processed decreased. Simply stated, both more threads or shorter tuple processing time results in a shorter amount of time until some hardware thread must execute the critical section of the parallel buffer code. Therefore more tuples must be processed by each thread between subsequent executions of the critical section, hence the larger chunk size.

This simple model could underestimate the necessary chunk size. The solution to Equation 2 assumes the best case that the concurrent accesses to the critical section are spread out evenly in time. The problem is that even uniformly distributed accesses might not be equally spread out. When some accesses cluster, all of the participating threads slow down significantly. The period during which more accesses cause contention also extends. We need to model the probability that two accesses to the critical section will conflict. We model this as a statistical process.

Consider the locked resource to be a line extending to infinity, indexed by time. When a thread needs to execute the critical section, it reserves a line segment of size L starting at the current point in time. If a thread finds the lock already taken, it must reserve a line segment of size L starting at the end of the last reservation on the line. The cost of contention is the overlap of requests on this line. We can approximate this effect discretely by dividing the line into N buckets, where $N = t/L$. That is, we discretize the number of locking slots available over the duration of our experiment (time slice), t. If one thread acquires the lock f times per second and p threads are used, a total of pft locks will be acquired during the experiment. The discrete probabilistic model, therefore, is to place balls in a uniformly random manner into the N buckets and count how many have two or more balls (contention).

$$E[0 \text{ Ball buckets}] = N\left(1 - \frac{1}{N}\right)^{pft} \tag{3}$$

$$E[1 \text{ Ball buckets}] = pft\left(1 - \frac{1}{N}\right)^{pft-1} \tag{4}$$

We can use Equations 3 and 4 to estimate the number of buckets with two or more balls and therefore the amount of contention. If we want to limit the amount of contention to a fraction b of the buckets, we need to solve:

$$N - N\left(1 - \frac{1}{N}\right)^{pft} - pft\left(1 - \frac{1}{N}\right)^{pft-1} = bN \tag{5}$$

We can simplify by introducing a variable X:

$$X = \left(1 - \frac{1}{N}\right)^{pft-1} = \left(1 - \frac{L}{t}\right)^{pft-1} \tag{6}$$

Thus, Equation 5 becomes:

$$(1-b) = X\left(1 - \frac{L}{t} - pfL\right) \tag{7}$$

We will assume that $N \gg pft$, which means that there are many more locking slots available than lock requests. This is reasonable since we are trying to configure the system to have few slots with more than one request. With this assumption we can approximate X as

$$1 - (pft - 1)\frac{L}{t} \tag{8}$$

As t goes to infinity, Equation 7 can be rewritten as:

$$(1-b) = (1 - pfL)^2 \tag{9}$$

$$f = \frac{1 - \sqrt{1-b}}{pL} \tag{10}$$

Based on our earlier terminology, $f = \frac{1}{cr}$, so

$$c = \frac{pL}{r(1 - \sqrt{1-b})} \tag{11}$$

When $b = 0.1$, $(1 - \sqrt{1 - b}) \approx 0.05$. In this case, the chunk size estimate, c, is about 20 times greater than the estimate with the simpler model. The appropriate chunk size for a buffer is a chunk size that eliminates contention in the buffer's upstream and downstream operator. This is simply the maximum chunk size found by performing the above analysis on the operators that share a buffer. We will examine the two chunk size models empirically in Section 5.

5. EXPERIMENTS

To validate our chunk size model, we performed experiments on real hardware using a machine with a Sun UltraSPARC T1 processor. The specifications of our test platform can be found in Table 1. The T1 has some unique characteristics. For one, the cores are much simpler than those found on other commodity processors: the pipeline is a shallow six stages, instructions are issued in order, and no hardware prefetching occurs. This simpler core does, however, support four hardware threads that share the core in a fair manner. A context switch occurs on each clock cycle and an instruction is then issued from the least recently used, ready thread. This sharing has important implications for performance on the T1. When all threads are ready, each issues an instruction every fourth cycle, which means that the effective clockrate seen by each thread is one quarter that of the core's clockrate. In the event of longer latency instructions or events (e.g., a cache miss) having other threads ready to run keeps the core from becoming idle. The T1 also foregoes branch prediction, instead relying on the other threads to issue instructions to fill the pipeline until a branch is resolved. These characteristics suggest the importance of keeping all threads on all cores busy to achieve optimum performance, particularly for data dependent applications, such as databases, that often cause many long latency cache misses.

5.1 Setup

The parallel buffer data structure was implemented in `C` and is very lightweight, requiring less than 200 lines of code. To test the parallel performance of the buffer, an operator was created that reads from an input buffer and writes to an output buffer. This operator allowed many performance parameters to be specified, including the amount of work per tuple, size of the input and output tuples, chunk size in the input and output buffer, and the selectivity. The selectivity is the number of output tuples produced for each input tuple read. In our test operator, selectivity was simulated using a random number test, but in practice it would be data dependent.

For all experiments, the results are averages of four runs using the same parameters. We also implemented a version of the parallel buffer that did not use the *top-up* procedure, but instead kept a counter for each chunk in the buffer to keep track of the occupancy of each chunk. This design allows partially filled chunks anywhere in the array, but at the expense of additional storage for the counters. This data structure performed nearly identically to the data structure described in Section 3. Therefore we provide results using only the parallel buffer defined in Section 3 because it is more space efficient. Additionally, because all but the first chunk processed by each thread are a fixed size, compile time optimizations such as loop unrolling and instruction reordering can improve performance. Although we did

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[1]
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
Compiler	Sun C 5.8

Table 1: Specifications of the Sun UltraSPARC T1.

not find a significant performance difference on the Sun T1, other architectures with deeper pipelines, branch misprediction penalties, and out of order execution may benefit more from these optimizations.

To minimize the amount of time spent starting and stopping threads, the threads were created and joined recursively. For example, the initial thread would create two child threads, begin processing, and then join its two children when processing completes. Those children would also create additional threads, and so on in a recursive manner until the target number of threads have been created. When one thread created all of the threads in a linear fashion, the time until all threads were working was longer, which resulted in an uneven amount of processing by each thread. Getting all threads to work as quickly as possible is important to achieving good processor utilization and thus maximizing the parallelism available on the processor.

5.2 Buffer Performance

We performed a number of experiments, varying the parameters of our test operator described above. Figure 3 shows the throughput of the test configuration when all 32 hardware threads were used to copy every input tuple from the input buffer to the output buffer (selectivity of 1.0) using different chunk sizes. The tuple size was 16 bytes. This graph clearly shows that the cost of contention is over an order of magnitude in lost throughput. This graph also shows the estimates provided by the simple and probabilistic chunk size models. The simple model provides an estimate that is very close to a chunk size that provides the best possible performance. The probabilistic model with $b = 0.1$ estimates a chunk size that is well into the performance plateau of chunk sizes that provide the same operator throughput. The two models provide a range of good chunk sizes for parallelism without significant contention. In practice, one can expect to have low contention if a chunk size somewhat greater than the simple model estimate is chosen.

A closer analysis of the two models in the context of the experiments sheds some light on their accuracy. In the probabalistic model, our discretization of the experiment's running time into buckets means that we consider any two requests to the same bucket to be contentious. In reality, unless the requests arrive at the exact same time, the overlap is not total. On average, one might expect a contentious

[1]The miss latency varies with the workload and with the load on the memory controllers [11].

operation to overlap with half of another operation. This is one reason why the probabalistic model may produce an overestimate. The probabalistic model was also proposed because uniformly distributed accesses might not be evenly spread out. But in practice if the tuple processing times are uniformly distributed, one can expect the chunk processing finishing times to be evenly spread out. If each chunk takes roughly the same amount of time to finish, then the simple model does provide the chunk size necessary to avoid contention.

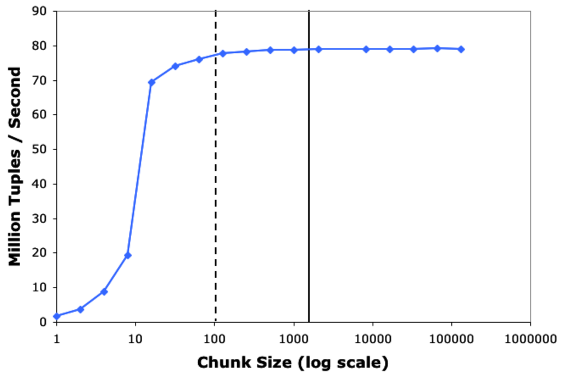


Figure 3: The dashed line is the simple estimate and the solid line is the probabilistic estimate with $b = 0.1$.

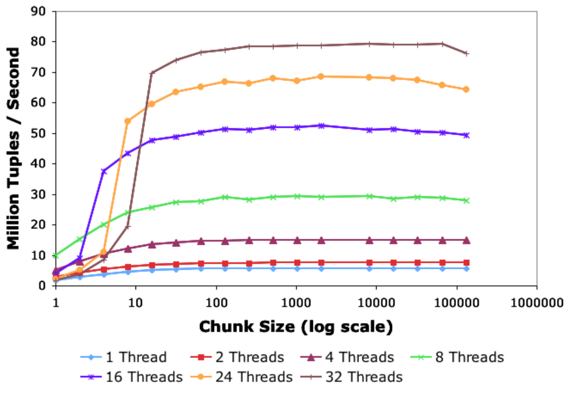


Figure 4: Scaling Performance.

The effects of contention are clearly demonstrated by Figure 4. As the number of threads concurrently accessing the parallel buffer increases, the performance penalty due to contention increases. This confirms predictions made by the chunk size model that the chunk size necessary to avoid contention increases as the amount of thread level parallelism increases. The penalty due to contention is so severe that fewer threads without contention outperform more threads that do have contention. This graph also shows that choosing a larger chunk size also helps amortize the cost of a more expensive atomic or locking operation over more tuples. In the case of one thread, there is no contention, but the throughput improves for larger chunk sizes because of this amortization.

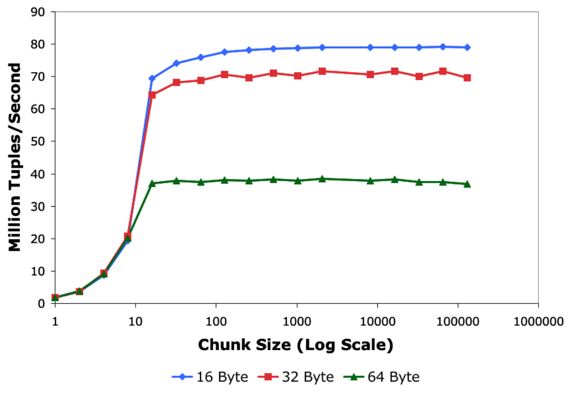


Figure 5: Performance based on tuple size. Larger tuples yield more cache misses and slower processing, but less contention.

As the size of the tuples stored in the parallel buffer increases, the number of cache misses incurred during processing each tuple increases. Figure 5 shows the results. Because the time to process each chunk increases, contention is reduced. For example, contention appears to be absent for 64-byte records at a chunk size of 16, while contention remains an issue for 16-byte records for chunks containing 100 records. The UltraSPARC T1 has 64 byte L2 cache lines, so 16, 32, and 64 byte tuples represent 1/4, 1/2, and 1 cache misses per tuple read or written, respectively.

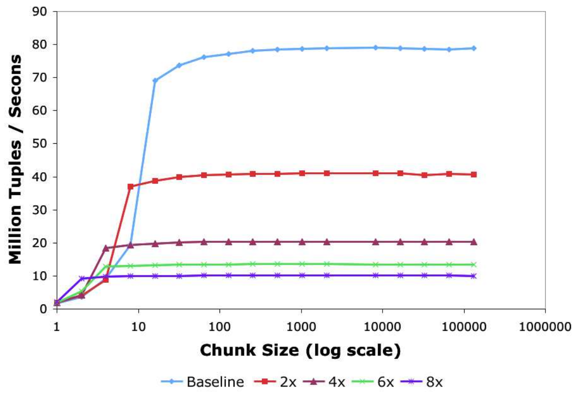


Figure 6: Throughput as work per tuple increases.

As the amount of work performed per tuple is increased, throughput naturally decreases as shown in Figure 6. The "baseline" in this experiment is 32 threads processing 16 byte tuples with a selectivity of 1.0. Extra work was then added to the processing of each tuple. The increased time per tuple and, therefore, increased time per chunk also results in less contention. Figure 6 shows that as the work per tuple increases, a smaller chunk size is necessary to eliminate contention.

5.3 Mutexes vs. Atomic Operations

Incrementing the chunk counter must be done atomically to ensure that each thread obtains a unique chunk to read from or write to. Two techniques may be used to achieve

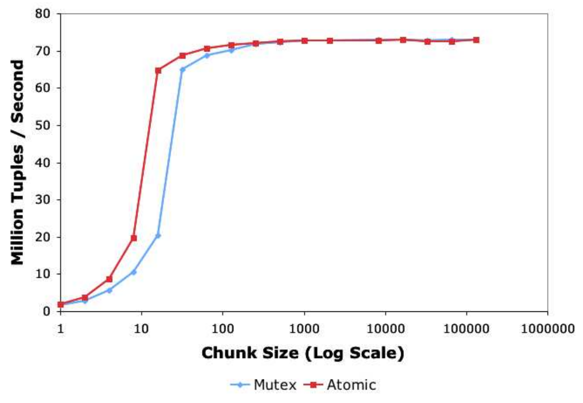


Figure 7: Incrementing the chunk counter atomically using a mutex vs. atomic operations.

this atomicity. First, threading libraries provide mutexes that can be used to provide exclusive access to particular variables and critical sections of code. When one thread has locked a mutex, all other threads that request that lock must wait for the first thread to release the mutex. In the case of incrementing the chunk counter, we acquire the mutex (it is shared among all of the threads), increment the counter remembering the new value, and then release the mutex.

Another way of atomically incrementing the chunk counter is to use atomic operations provided by the architecture's instruction set. Most microarchitectures provide some type of atomic operations on which synchronization objects, such as the mutexes described above, can be built. Some microarchitectures, including the Sun T1, provide more advanced atomic operations that can be used to perform atomic arithmetic and logical operations.[2] Using an atomic operation, if available, to increment the chunk counter has a number of advantages. First, acquiring and releasing a mutex requires using atomic operations anyway, so the atomic increment operation is unlikely to be slower. Second, using a mutex requires invoking the threading library, which at the very least means that more instructions will be executed, lowering performance. A third issue is that when a thread attempts to acquire a mutex, but fails, it may be put to sleep until the mutex becomes available. This means that the mutex implementation interacts with the system scheduler, incurring an even higher overhead. For a very lightweight operation such as incrementing the chunk counter, the overhead of acquiring a mutex can be significant, especially when there is contention between threads acquiring the mutex.

In all of our experiments we have used a lighter-weight atomic increment operation instead of mutexes. Figure 7 shows a performance comparison of buffer performance using the atomic increment and a mutex. The experimental parameters are the same as those in the experiment from Figure 3. In a simple experiment, we measure the single threaded latency incurred while performing the atomic increment using a mutex to be about 128 cycles compared to 88 cycles when using the atomic increment operation. Figure 7 demonstrates that a larger chunk size is required for the mutex approach to achieve a performance comparable to the implementation using atomic operations. This result follows from the mutex's higher latency, which increases the chance of contention at lower chunk sizes. For sufficiently large chunk sizes, the throughput obtained by both approaches is nearly identical. This is because the time to process a chunk dominates the time required to atomically increment the chunk counter.

[2]Though the T1 ISA does not have, for example, an atomic add instruction, such an operation can be built using provided atomic primitives.

5.4 Load Balancing

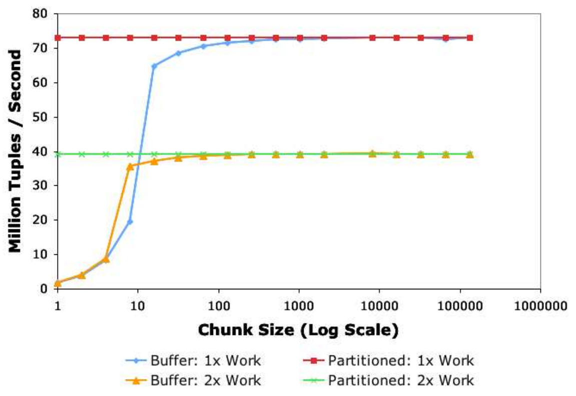
Achieving high performance on a chip multiprocessor requires keeping all of the hardware thread contexts busy so that the processor is fully utilized. In the context of database operations that exploit intra-operator parallelism, the key to keeping the processor fully utilized is load balancing. Threads that complete their work and then wait for other threads to also finish lower overall performance, whereas threads that complete their work and then find other work to complete help maximize overall performance. In this section we examine some examples of skew that can occur with other approaches to parallelism and demonstrate how the proposed parallel buffer structure achieves good load balancing even in the presence of significant skew.

A common method of partitioning data for parallel processing is to partition the tuples based on a hash of some combination of attribute values [4]. In the case of the Sun T1, we might want 32 partitions – one for each hardware thread. Partitioning, however, is sensitive to skew in the data that can cause some partitions to be much larger than others. In a simple experiment we used multiplicative hashing [13] to partition input into 32 partitions. The input distributions consisted of 2^{24} tuples and were generated using techniques similar to those found in Gray et al [9].

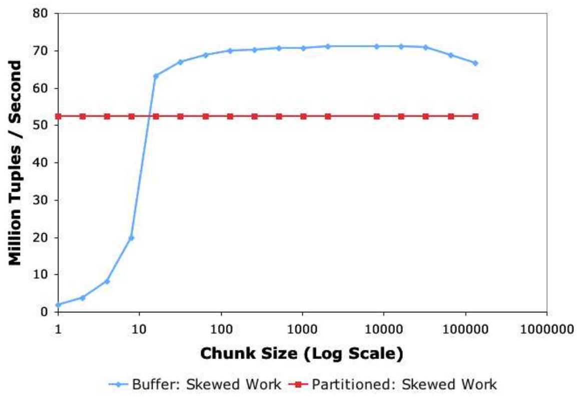
When the input values were distinct, the sizes of the partitions were very similar with a standard deviation of just 1.8 tuples. For distributions such as Zipf and self-similar, the amount of skew was higher. For Zipf, the measured standard deviation was about 1000 tuples when the values were chosen from a range as large as the size of the input, but increased significantly as the range was decreased causing values to repeat more frequently. For the self-similar distribution, the standard deviation was almost 300000 tuples and the largest partition was about five times greater than the smallest partition. Even in the presence of moderate amounts of skew, threads assigned to smaller partitions will finish early and wait for other threads to complete, underutilizing the processor. Using the proposed parallel buffer, threads continue to work on new chunks until a the buffer is exhausted, thus keeping all threads busy performing useful work.

Another significant problem with the partitioned approach is skew introduced during query processing. Even if an initial partitioning of the input is well balanced, some partitions may contain tuples that fail to pass a selection condition, while other partitions contain tuples that have large join products, bloating their join output relative to other partitions. Solutions to this problem include repartitioning and variable sized buffers for the partitions between operators, which we argue is much more complicated than the single, unified parallel buffer proposed in this paper.

A different form of skew involves the amount of time required to process a tuple. Some tuples take longer to process than others. Consider the example of a hash join. If a tu-



(a) Constant amount of work per tuple.



(b) Skewed amount of work per tuple.

Figure 8: Performance using parallel buffers vs. static partitioning.

ple hashes to an empty bucket, processing stops because the tuple does not participate in the join. In contrast, if a tuple hashes to an occupied bucket, then the values in that bucket must be interrogated along with any potential overflow buckets. In the partitioned approach, even if the partitions are of equal size the amount of processing time may be skewed. To compare partitioned processing with using the parallel buffer, we create a skewed scenario. Each tuple in the first 1/32 of the input requires twice as long to process as the rest of the input. In the partitioned approach, this first 1/32 of the input corresponds with the partition processed by the first thread. Using a parallel buffer, all of the threads share in processing this more expensive input.

Figure 8 shows the effects of processing time skew on performance of both the partitioned and parallel buffer approaches. The work is introduced in the same manner as in the experiment associated with Figure 6. When work per tuple is constant, as in Figure 8a, the partitioned and parallel buffer approaches perform similarly when the buffer chunk size is sufficiently large. This is good because it means that the buffer infrastructure has negligible overhead compared to course grained partitioning. The benefit of using a parallel buffer is considerable when significant skew is introduced in the manner described above. The difference in performance between the baseline buffer performance (Figure 8a) and the buffer performance with skew (Figure 8b) is less than 1/32, which is what we would expect since 1/32 of the work is twice as expensive. In contrast, Figure 8b shows that the difference between the parallel buffer and partitioned processing for the skewed workload or the partitioned approach is almost 30%, which means that many threads are idle, resulting in lower processor utilization.

The skewed performance might be expected to equal that of the partitioned approach with twice the work for each tuple. However, this does not happen because of the way that four threads share one core on the Sun T1. In the skewed case, one thread is doing more work and issuing more instructions, which may fill holes where the other threads cannot issue instructions because of delays. In the case where all threads have equal work, they compete evenly for execution resources. Also, once other threads terminate early, the slower thread that is processing the more expensive partition never conflicts with other threads and can always issue instructions when ready. Therefore the partitioned approach's skewed performance is somewhat better than might be expected, but still significantly worse than the parallel buffer.

The advantage of the parallel buffer is that each thread will process as many input chunks as it is able and write to as many output chunks as needed. No adjustment is needed if one thread produces more output than other threads. Similarly, no load balancing steps are required within the plan. Keeping threads busy with work is obviously important, but there are situations, such as the exhaustion of input tuples or space to write output that will require that a thread terminates. Ensuring that all threads terminate quickly when one thread is forced to finish is also important to maintaining high processor utilization and is the focus of the next section.

5.5 Thread Finalization

Efficiency during the finalization of threads using the parallel buffer data structure is also important to performance. If some threads take a much longer time to stop work, the processor could be underutilized while a majority of threads wait for all of the threads to terminate. Section 3 describes this condition and how our parallel buffer avoids this problem. Figure 9 shows the finishing times of all 32 hardware threads during an experiment adjusted to the time that the first thread finishes. The amount of time between the first and last thread termination is much less than 1% of the total execution time and represents the time to process about 30 chunks. Though this overhead is not as low as suggested in Section 3, it still ensures that the processor is fully utilized during almost all processing.

We suspect that the reason we observe a 30-chunk window rather than a smaller one is that there is contention on the timing counter used to perform the measurement for this experiment, forcing the threads to serialize their access to the counter. The true finishing times (in the absence of a measurement or other serialization point) would show an approximately cumulative normal distribution, something not apparent in Figure 9.

6. CONCLUSION AND FUTURE WORK

Achieving good performance on chip multiprocessors requires applications to exhibit sufficient thread level parallel-

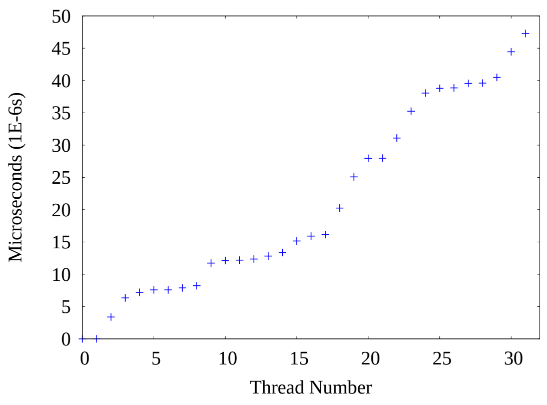


Figure 9: Difference in thread finishing times from first thread to finish.

ism to saturate the available hardware threads, while also managing shared resources efficiently. Database operations exhibit a high degree of parallelism, but a challenge is in coordinating the input and output to a parallel operator in a manner that avoids contention between threads. In this paper we present a new parallel buffer data structure that helps to enable intra-operator parallelism. This buffer provides unified input or output to a parallel data structure. Based on a theoretical analysis and experimental validation, processing portions of the input and generating the output in sufficiently large chunks can eliminate contention between threads. The appropriate chunk size can be determined via a theoretical model, which we have verified experimentally.

Another advantage of our data structure is that it also provides load balancing between threads running a parallel operator. Because every thread consumes and produces chunks of tuples, the amount of input processed or output generated by any one thread can adapt to the speed of that thread. This is in contrast to a per-thread buffer or static partitioning. Those techniques are sensitive to skew and may result in under utilization. An example of this under utilization occurs when some threads drain their input buffers quickly and others process input more slowly. With a unified parallel buffer, individual operators do not need to address this load balancing problem.

The parallel buffer is also compatible with row-wise or column-wise storage. Column-wise storage has been shown to be particularly beneficial for OLAP workloads [1, 19]. The parallel buffer only requires fixed size elements. Whether this is a full record or only a single attribute does not affect the load balancing and contention avoidance properties of the data structure. If multiple columns are required as input or output, multiple buffers may be used or a single buffer with multiple arrays could be used. In the later case, the columns in each chunk would represent values from the same records. We have not implemented the buffer data structure for a particular data layout, but as future work we will investigate the best way to use parallel buffers for column- and row-based data.

In future work, we plan to implement real database operators and validate complete system performance. The parallel buffers presented here form the core of the necessary infrastructure for managing load balancing and parallelism. Operator implementation can thus focus on achieving good threaded performance, by choosing efficient algorithms that share cache-resident data structures and avoid inter-thread interference.

7. ACKNOWLEDGEMENTS

We thank the anonymous peer reviewers for their constructive comments.

8. REFERENCES

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

[2] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[3] J. Cieslewicz, J. W. Berry, B. Hendrickson, and K. A. Ross. Realizing parallelism in database operations: insights from a massively multithreaded architecture. In *DaMoN*, 2006.

[4] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

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

[6] P. Garcia and H. F. Korth. Database hash-join algorithms on multithreaded computer architectures. In *Conf. Computing Frontiers*, pages 241–252, 2006.

[7] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[8] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[9] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, 1994.

[10] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, pages 79–87, 2007.

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

[12] M. L. Kersten, S. Manegold, P. A. Boncz, and N. Nes. Macro- and micro-parallelism in a dbms. In *Euro-Par*, pages 6–15, 2001.

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

[14] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free fifo queues. In *DISC*, pages 117–131, 2004.

[15] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[16] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, 1998.

[17] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block oriented processing of relational

database operations in modern computer architectures. In *ICDE*, pages 567–574, 2001.

[18] C.-H. Shann, T.-L. Huang, and C. Chen. A practical nonblocking queue algorithm using compare-and-swap. In *ICPADS*, pages 470–475, 2000.

[19] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[20] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. Improving database performance on simultaneous multithreading processors. In *VLDB*, pages 49–60, 2005.

[21] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *SIGMOD Conference*, pages 191–202, 2004.