

Data Partitioning on Chip Multiprocessors

John Cieslewicz^{* †}
Columbia University
New York, NY
johnc@cs.columbia.edu

Kenneth A. Ross[†]
Columbia University
New York, NY
kar@cs.columbia.edu

ABSTRACT

Partitioning is a key database task. In this paper we explore partitioning performance on a chip multiprocessor (CMP) that provides a relatively high degree of on-chip thread-level parallelism. It is therefore important to implement the partitioning algorithm to take advantage of the CMP's parallel execution resources. We identify the coordination of writing partition output as the main challenge in a parallel partitioning implementation and evaluate four techniques for enabling parallel partitioning. We confirm previous work in single threaded partitioning that finds L2 cache misses and translation lookaside buffer misses to be important performance issues, but we now add the management of concurrent threads to this analysis.

1. INTRODUCTION

Partitioning is a core database task used for many purposes. Partitioning can divide a larger task into constituent smaller subtasks that can be processed more quickly than the overall task taken as a whole. An example of this is when a complete task, done all at once, would not be entirely cache resident and could therefore suffer from a high number of cache misses. Smaller subtasks may fit within a cache, thereby experiencing good cache performance. Partitioning is also important because of its ability to group like values. In the case of a hash join, both relations are partitioned using the same hash function and only tuples from equivalent partitions can join to form part of the result. Similarly, range partitioning based on a sort key can improve the performance of sorting, which itself is a core database operation. In the context of parallelism, partitioning assists load balancing by producing similarly sized subtasks.

A chip multiprocessor (CMP) is a single chip that supports multiple concurrent threads of execution with multiple

processor cores per chip, multiple threads per core, or both. This paper uses a specific CMP, the Sun UltraSPARC T1, which has eight cores and four threads per core for a total of 32 threads on a single chip [11].

In this paper we examine in-memory hash-based partitioning performance on a chip multiprocessor. Though other types of partitioning (such as range partitioning) are common, we identify the process of writing partitioning output, rather than the method of computing a tuple's partition assignment, to be key to partitioning performance. Therefore, we focus on hash-based partitioning and explore different means of coordinating the writing of output.

High performance partitioning on a CMP requires balancing parallelism with interthread coordination and on-chip resource sharing. Because partitioning requires writing output to many different locations, shared cache and translation lookaside buffer (TLB) resources can become a source of contention between threads, causing lower performance. Cache and TLB pressure is a problem for single-threaded partitioning implementations [8]. The problem is compounded by the presence of multiple concurrent threads for two reasons. First, each additional thread may increase the number of output locations that are active at any one time. And second, the interthread coordination required to effectively manage the shared on-chip resources can become a bottleneck itself. In this paper we will describe the impact of these issues and present techniques that overcome them to achieve high performance partitioning on CMPs.

The rest of the paper is organized as follows. In Section 2 we will present work related to both partitioning and database operations on chip multiprocessors. Section 3 describes partitioning techniques and implementation options. Our experimental platform and setup is described in Section 4 and experiments are presented in Section 5. We present future work in Section 6 and conclude in Section 7.

2. RELATED WORK

Partitioning has been studied in a number of contexts, both parallel and not. It is central to many database operations, including joins and aggregates, and it is also important for load balancing [4, 3, 8, 10]. Many variants of parallel sorting include an initial partitioning step [5, 6]. As parallelism is now available on-chip, we revisit partitioning to investigate techniques that provide the best parallel partitioning performance on new CMPs.

Manegold et al. introduce a clustering (partitioning) algorithm that performs well on modern architectures by optimizing cache and TLB usage [8]. Though their analysis fo-

^{*}Supported by a U.S. Department of Homeland Security Graduate Research Fellowship

[†]Supported by NSF grant IIS-0534389

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.

Proceedings of the Fourth International Workshop on Data Management on New Hardware (DaMoN 2008) June 13, 2008, Vancouver, Canada.
Copyright 2008 ACM 978-1-60558-184-2 ...\$5.00.

Technique	Contiguous Output	Contention
Independent	No	No
Concurrent	Yes, per partition	Yes
Count-then-move	Yes	No
Parallel Buffers	Mostly per partition	No

Table 1: CMP Partitioning Techniques

cused on single threaded execution, it provides an excellent starting point for exploring the issues associated with in-memory parallel partitioning on chip multiprocessors. Our work differs in two main ways. First, our goal is high performance parallel partitioning on a CMP rather than a uniprocessor. And second, our target platform uses very simple cores optimized for thread-level parallelism rather than single threaded performance.

A motivation for examining parallel partitioning arose from observations made during an analysis of aggregation on CMPs where it was found that contention between threads for shared resources as well as interthread communication were key factors affecting performance [2]. Because locality of reference is important to hash aggregation performance, parallel partitioning may be able to improve aggregation throughput by clustering the input. This paper explores the characteristics of such a parallel partitioning. As mentioned in Section 1, coordinating the writing of output between threads is an important aspect of a parallel partitioning implementation. The parallel buffer data structure proposed in [1] provides a framework for parallel, contention free access to a mostly contiguous shared buffer. Using the parallel buffer structure is one of the options for implementing parallel partitioning described next in Section 3.

3. PARTITIONING TECHNIQUES

Partitioning data is a two step process. In the first step, the output partition to which a tuple belongs is identified. In the second step, the tuple is copied to its output location. The choice of the partitioning function in the first step depends on the properties desired from the partitioning. In this paper we focus on hash partitioning. Identifying a tuple’s partition using hashing is an $O(1)$ operation. We will use multiplicative hashing and describe the number of output partitions in terms of b , the number of hash bits used to partition the input. The use of b hash bits results in 2^b output partitions. Because the input is read only, it can easily be divided among t threads for parallelism. Writing the input to different partitions, however, involves trade-offs in thread coordination and resource sharing. We examine four methods of enabling parallelism in the writing of tuples to output partitions. The methods are summarized in Table 1 and described below.

3.1 Independent Output

In the *independent output* technique, each thread has its own output buffers for each output partition, i.e. $t * 2^b$ output buffers. There is no sharing of output space between threads and therefore no thread coordination required aside from assigning input tuples to each thread. Each buffer requires the storage of metadata, such as the current writing index and the size of the buffer. As the number of threads or hash bits increases, the number of buffers required also increases. At the same time, the expected size of each buffer,

$\frac{N}{t * 2^b}$, decreases¹, which means that the storage overhead associated with metadata increases.

The complete independence of each thread helps enable parallelism by avoiding any contention between threads. Unintentional *false sharing* in the cache can be avoided by ensuring that each thread’s buffer metadata does not share a cache line with the metadata from another thread.

There are two main disadvantages of this approach. First, the metadata overhead increases as additional threads or partitions are used. And second, each partition is fragmented into t separate buffers. The operator that next processes the partition must either accept fragmented input or a further consolidation step is needed.

3.2 Concurrent Output

The *concurrent output* technique uses a single buffer for each output partition. The one buffer is shared among all threads, which coordinate writing through the use of atomic instructions or locks. Specifically, the current writing index must be atomically incremented before each write. In contrast to the independent technique, the number of output buffers no longer depends on the number of threads, t . In terms of storage overhead, therefore, the concurrent output method scales better as more threads are used. Also, using this method, each partition’s output is stored contiguously, which may make further processing of a partition easier.

The atomic instructions or locks required for the correctness of this technique are expensive and susceptible to contention. In the independent approach, incrementing a write index takes one cycle, whereas performing the atomic increment required for the concurrent technique takes 22 cycles² on the Sun T1 [12]. Not only do atomic instructions have longer latency, but they can also cause contention. When many threads attempt to atomically increment the same variable, only one can proceed, forcing all other threads to wait. In the worst case, all threads serialize due to atomic increments to the same variable(s), which severely degrades performance because no parallel computation occurs.

3.3 Count-Then-Move

The *count-then-move* technique uses two passes over the input to partition the data into a single contiguous buffer in which consecutive partitions reside within consecutive ranges in the buffer. In the first pass, each thread processes an assigned range of input tuples, counting the number of tuples it would have placed in each partition. This step requires 2^b counters per thread. Each thread works independently, so a high degree of parallelism is possible. As with the independent technique above, care should be taken to avoid false sharing in the cache by ensuring that no two threads’ counters share a cache line.

Following the first pass, all of the threads must synchronize to signal that counting is complete. Using the counts supplied by all threads, each thread can then compute the exact offset at which it will start writing output for each partition. Each thread must therefore store 2^b writing offsets.

¹Assuming that the partitioning keys are unique and uniformly distributed.

²The Sun UltraSPARC T1 may not be used in an SMP configuration as it does not support off-chip cache coherency and atomic operations. This atomic latency would be higher in a comparable processor that supported off-chip cache coherency and was used in an SMP configuration.

Once these offsets are computed, the second pass begins. Each thread processes the same assigned range of tuples from the first pass, but this time, once a tuple’s assigned partition is determined, it is written using the thread’s offset for that partition. The offset is then incremented. The second pass also requires no interthread coordination.

A drawback of this approach is that it requires two passes over the input to accomplish what other techniques accomplish in one pass. On the other hand, the entire result is contiguous, which may be more useful than the fragmented results produced by the other methods. The other methods require further processing to produce contiguous output.

3.4 Parallel Buffers

The partitioning technique using *parallel buffers* [1] is very similar to the concurrent output approach except that instead of coordinating between threads on every write, coordination occurs at the coarser granularity of a “chunk” of tuples. Each thread atomically obtains a contiguous chunk of one or more tuples from the buffer. It then has exclusive access to those tuples and only needs to engage in interthread coordination when it acquires a new chunk. The cost of atomic operations is amortized over many writes and the chance of contention for shared data structures is reduced. An appropriately sized chunk can eliminate all coordination contention, as shown in [1]. The parallel buffer is designed so that all but the first t chunks are either completely full or completely empty and that there are no holes, i.e., all data not in the first t chunks is contiguous.

Using parallel buffers has the advantage of avoiding interthread contention while using a mostly contiguous shared buffer. Output written into a parallel buffer is also ready to be read in parallel by subsequent tasks. A disadvantage is that each parallel buffer requires more metadata than the other techniques in order to support the management of chunks and parallel access by multiple threads. Additionally, each buffer must have at least one chunk for each thread. Even when the chunk size is set to just one tuple, the parallel buffer must be at least t tuples in size.

4. EXPERIMENTAL SETUP

All experiments were conducted on real hardware, a Sun UltraSPARC T1, the details of which may be found in in Table 2. We chose this platform because the T1 and its recently introduced successor, the T2, are the commodity CMPs with the most on-chip thread level parallelism.

Input Characteristics

The input to all experiments consists of 16 byte tuples with an 8 byte partitioning key and an 8 byte payload. The partitioning keys are unique and uniformly distributed. This input is similar to the input used in [8], but updated for a 64-bit processor. Due to space limitations we do not explore the implications of non-uniform and non-unique input here, but we do discuss some of these issues in Section 6.

³The miss latency varies with the workload and with the load on the various processors [7].

⁴The TLB is shared among the 4 threads on the core, but each thread’s entries are kept mutually exclusive [11].

Clock rate	1 GHz
Cores (Threads/core)	8 (4)
RAM	8GB
Shared L2 Cache	3MB, 12-way associative 64B Cache Line Hit latency: 21 cycles Miss latency: 90–155 cycles ³
L1 Data Cache	8KB per core 16B Cache Line Shared by 4 threads
L1 Instruction Cache	16KB per core Shared by 4 threads
TLB	64 Entries per core ⁴
Supported Page Sizes	8KB, 64KB, 4MB, 256MB
On-chip bandwidth	132GB/s
Off-chip bandwidth	25GB/s over 4 DDR2
Operating System	Solaris 10
Compiler	Sun C 5.9
Flags	-fast -xtarget=native64 -mt

Table 2: Specifications of the Sun UltraSPARC T1.

Implementation Details

We used the pthreads library for all multithreading. Where possible we implement atomic operations with atomic intrinsics provided by the compiler rather than by using a mutex available in the threading library. This is advantageous because atomic instructions have lower latency than acquiring and releasing a lock.

Physical memory frame allocation by the operating system must be done atomically. Frame allocation may significantly reduce parallelism because threads must serialize when handling page faults that cause frame allocation from the operating system. To avoid this overhead, our code touches all of the pages to be used for writing output before collecting profiling or timing information. This is a reasonable modification because a long running database process could avoid this frame allocation bottleneck by reusing allocated buffers for partitioning and other operations.

To avoid the issue of growing an output buffer, the independent, concurrent, and parallel buffers use buffers that are allocated to be more than 50% larger than their expected size. This leads to some space overhead but simplifies the implementation and analysis. Issues such as needing to grow an output buffer in a thread safe manner are discussed in Section 6.

For all techniques, variables used for counting purposes are 32 bit integers, which helps lower the metadata overhead of each technique compared with using a 64 bit integer. For our experimental input, 32 bits is sufficient for this purpose, but in the future significantly larger partitions may require the use of 64 bit integers. The parallel buffers used in these experiments are a slightly improved version of the buffers used in [1]. The amount of metadata required per buffer has been reduced and code for writing and reading tuples has been made more modular, but the overall operation of the parallel buffer remains the same.

In the count-then-move technique, during the first pass one could record each tuple’s assigned partition in a parallel array. During the second pass, this parallel array could be read instead of recomputing each tuple’s partition. We tried both options and found that for hash partitioning the

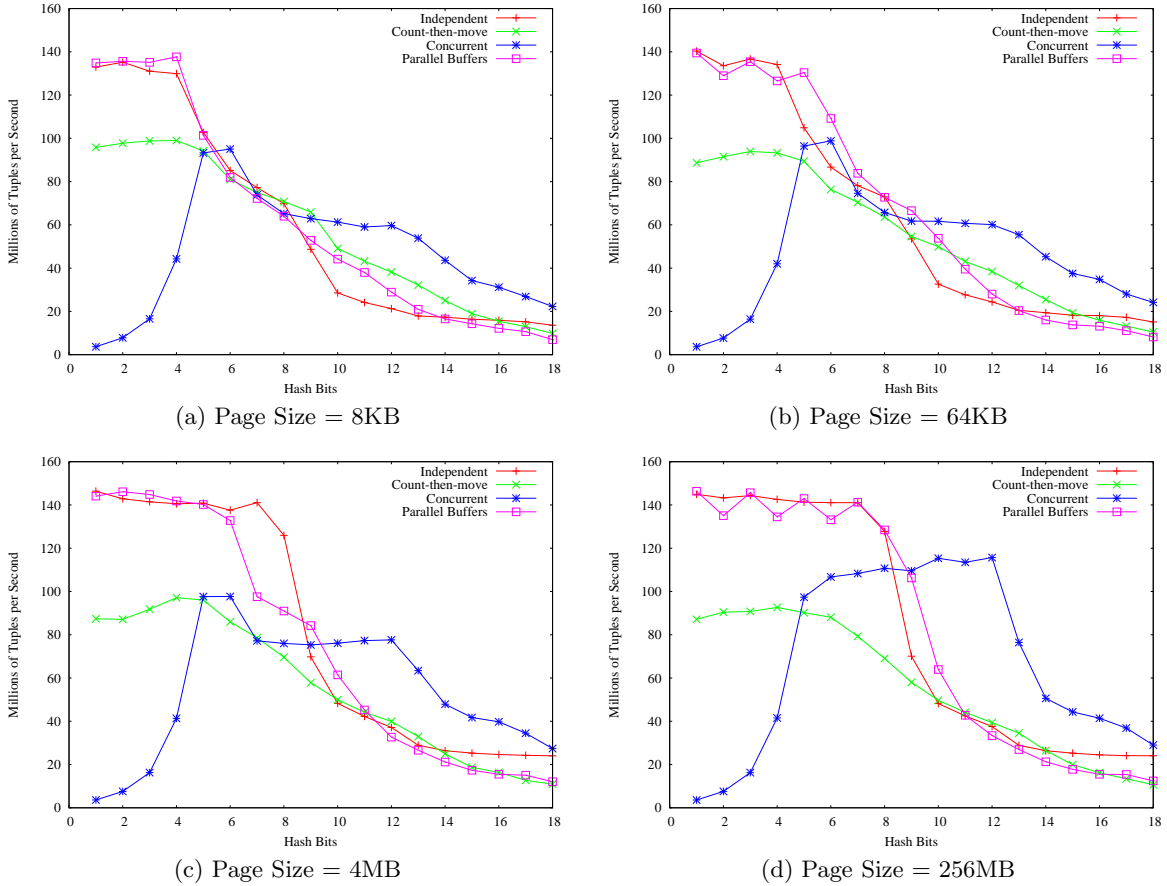


Figure 1: Comparing the throughput of four partitioning techniques using 32 threads and all available page sizes. The input cardinality was 2^{24} tuples.

performance was comparable so for the experiments in this paper we recompute the hash function.

5. EXPERIMENTS

In this section we present an experimental evaluation of the four partitioning techniques described in Section 3. We also present data collected using the T1’s hardware performance counters. All throughput values reported are averaged over eight trials of the same experiment.

5.1 Performance and Contention

The performance of the four partitioning techniques is shown in Figure 1. At best, partitioning can only equal the performance of copying the same quantity of data, i.e. a `memcpy`. As a comparison point, we present the performance of a multithreaded `memcpy` operation in Figure 2.

The performance of `memcpy` represents an extreme upper bound on partitioning performance because the C standard library `memcpy` uses a number of optimizations that our generic partitioning code cannot leverage. If a store instruction writes to a cache line that is not in the cache, normally a cache miss is first triggered to load the cache line and then the store proceeds. This is important because the store may only modify part of the cache line. However, it also means that writes can cause cache misses. Because `memcpy` knows that it is writing many full cache lines of data, it is able to use a block initializing store instruction that does not read

the cache line from memory, but rather allocates a new zeroed cache line directly in the cache. Thus `memcpy` is able to avoid a large number of cache misses and reduce the amount of memory bandwidth that it requires. A generic partitioning algorithm has difficulty using such a store instruction because writes to the same output partition may be separated in time and across threads, therefore having an entire cache line to write at once may not be practical, but we examine using such an instruction in Section 5.6. Figure 2 also shows a naïve, user implemented multithreaded `memcpy` to give a sense of the best performance we might expect from any of our partitioning techniques. The independent and parallel buffer techniques actually slightly exceed the performance of this `memcpy` implementation for small numbers of partitions.

The only technique to suffer from contention is the concurrent output technique (Section 3.2). Figure 1 shows that when fewer than six hash bits are used, performance of the concurrent buffer technique suffers due to contention. This makes sense given that there were 32 threads used and five or fewer hash bits means that there were 32 or fewer output locations. Because the input data is uniform and unique, once the number of partitions exceeds the number of threads each thread will likely be writing to a different partition. Even though the parallel buffer technique shares a buffer, its use of chunks of tuples successfully avoids contention. In these experiments the maximum chunk size was 128 tuples, but

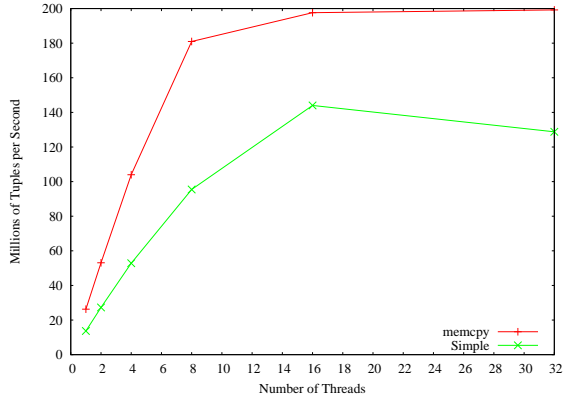


Figure 2: Throughput for copying 40,000 tuples using C Standard Library `memcpy` and using a simple user implemented copying loop.

that was scaled down to a size of 1 as the number of partitions increased and the expected partition size decreased.

The following sections explore in detail the factors influencing the performance reported in Figure 1, including TLB misses, cache misses, and scaling.

5.2 TLB Misses

TLB misses are known to be a significant performance issue during partitioning [8] and our experiments on a CMP confirm this observation. The TLB is a special cache that the processor uses to quickly translate virtual addresses to physical addresses. Because partitioning requires writing output to different partitions, TLB misses may become frequent if the TLB is not large enough to hold all of the pages to which output needs to be written. The size of a page of memory, therefore, influences the TLB’s ability to contain all output pages. The Sun T1 has one 64-entry TLB per core that supports four different page sizes. Partitioning performance using those page sizes is shown in Figure 1. The difference in performance between different page sizes is due in large part to the impact of TLB misses.

TLB miss data is presented in Figure 3 and for each technique, two graphs are shown: those on the left plot only user level TLB misses, and those on the right plot both user and system level TLB misses.

Figure 3(a) shows the user level TLB misses incurred per tuple partitioned by the independent technique. A page size of 256MB results in zero TLB misses because the 64 entry TLB has a reach of 16GB, which is double the size of the machine’s RAM! Each TLB is shared by four threads on one core. Because each thread has its own output location for each partition, the number of output locations the TLB must cover is $2^b * 4 = 2^{b+2}$. For small pages, the 64 TLB entry limit is reached when $b = 4$ as shown in Figure 3(a). Because the parallel buffers share a contiguous buffer, they share some output pages causing TLB misses to begin increasing at $b = 5$ instead (Figure 3(e)). The concurrent buffer has only one output location per partition regardless of the number of threads. It therefore sees no increase until $b = 6$ when the number of partitions equals the number of TLB entries (Figure 3(c)).

The plateau around one TLB miss per tuple seen for most techniques when using 8KB or 64KB pages is explained by requiring a TLB miss for the output location alone. This

happens as soon as there are more than 64 output locations active per core. The number of misses then grows to two per tuple for both the independent and concurrent techniques, Figures 3(a) and 3(c), respectively. The second miss is explained by needing another miss to access the metadata associated with the output buffer. As the number of partitions increases, the storage overhead and, therefore, the number of pages required by the metadata also increases.

Figures 4(a) and 4(b) show the number of pages of metadata required for the independent and concurrent techniques, respectively. Each output buffer requires 16 bytes of state and there is one buffer per partition in the case of concurrent output or one buffer per thread per partition in the case of independent output. As Figure 4 demonstrates, the TLB cannot hold entries for all of the metadata pages when smaller page sizes are used. Because metadata pages also compete with input and output pages in the TLB, two or more TLB misses per tuple partitioned becomes increasingly likely for larger numbers of hash bits when smaller pages are used. Comparing Figures 4(a) and 4(b) with Figures 3(a) and 3(c) one can see the rise in TLB misses from one to two per tuple processed roughly coincides with the point at which the number of metadata pages exceeds the TLB capacity. This makes sense, because given our uniform input distribution, metadata pages, which contain the metadata for many output locations, are more frequently accessed than output pages. Therefore, metadata pages are more likely to be in the cache even when the number of output pages greatly exceeds the TLB capacity. It is only when the metadata pages also exceed the TLB capacity that we begin to see TLB misses for metadata as well. The impact of metadata on cache misses is discussed in Section 5.3.

In general, when using smaller page sizes, increasing the number of output partitions beyond the reach of the TLB increases the number of TLB misses, which decreases performance. Large page sizes, such as those available on the T1, can mitigate this TLB reach issue.

The count-then-move technique’s TLB misses (Figure 3(g)) also plateau around one miss per tuple, but then increase to at least three misses per tuple when more hash bits are used. This is because each pass incurs a TLB miss while reading the meta data, plus one miss when writing the output. On the first pass, the metadata miss occurs when the counter is incremented and on the second pass it happens when the write offset is read and incremented.

Similarly, parallel buffers (Figure 3(e)) incur more than two misses per tuple. This is because pieces of its metadata are stored separately to avoid false sharing between threads. More sophisticated metadata allocation could eliminate some of these TLB misses by placing all metadata for a buffer on the same page, albeit on separate cache lines to avoid false sharing.

The recorded user level misses make sense given the pattern of memory accesses caused by the different partitioning techniques. What is less obvious is the total misses – user and system level (the right column of Figure 3). In all cases except the count-then-move technique, the total TLB misses exceed the user level misses at high partitioning cardinalities.

We suspect that the system is incurring additional system TLB misses while servicing user TLB misses. When a large number of partitions are used, the amount of metadata required is quite large and requires many pages when small

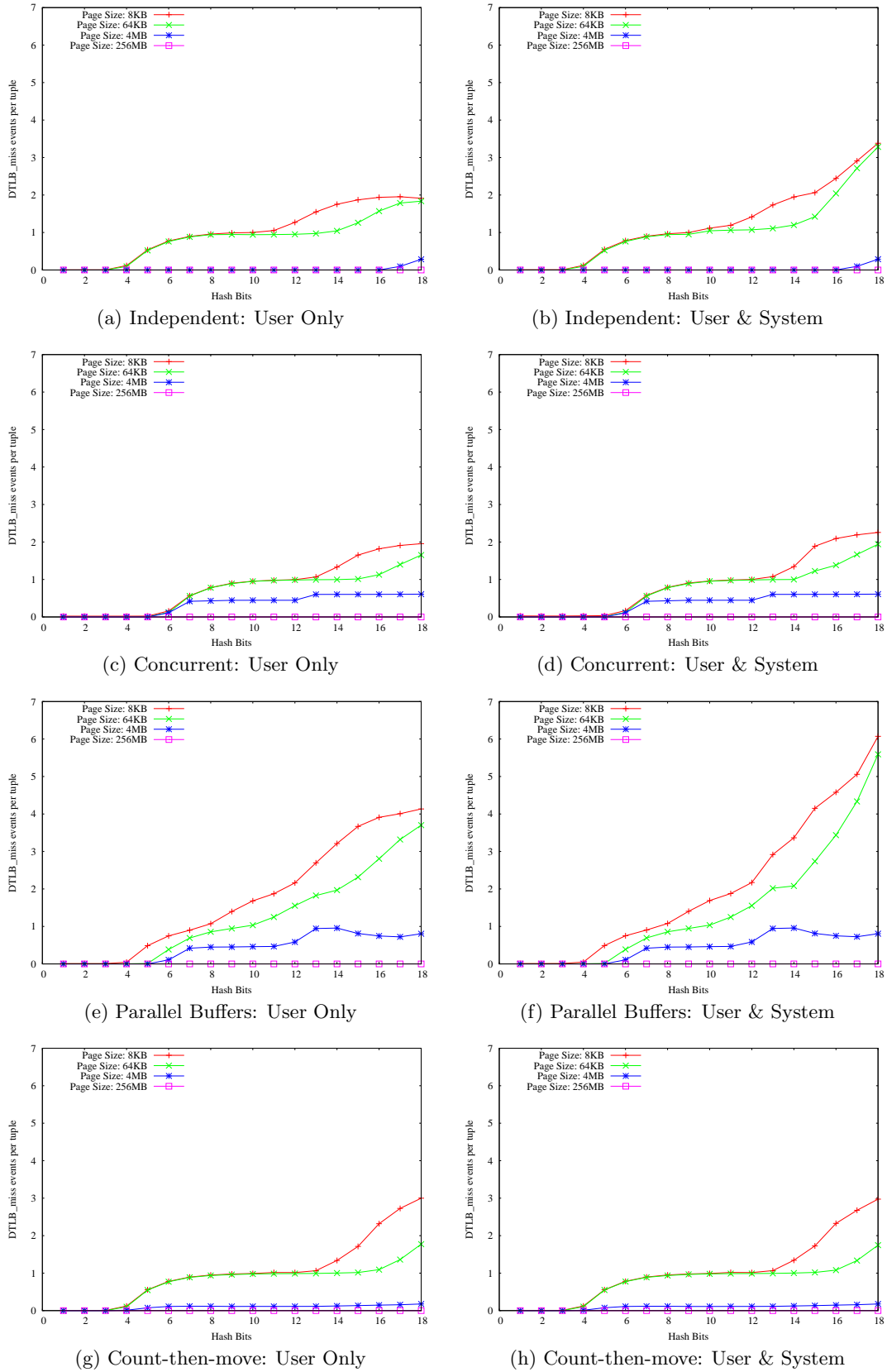


Figure 3: TLB misses incurred by the user and system while partitioning. The input cardinality was 2^{24} tuples and 32 threads were used.

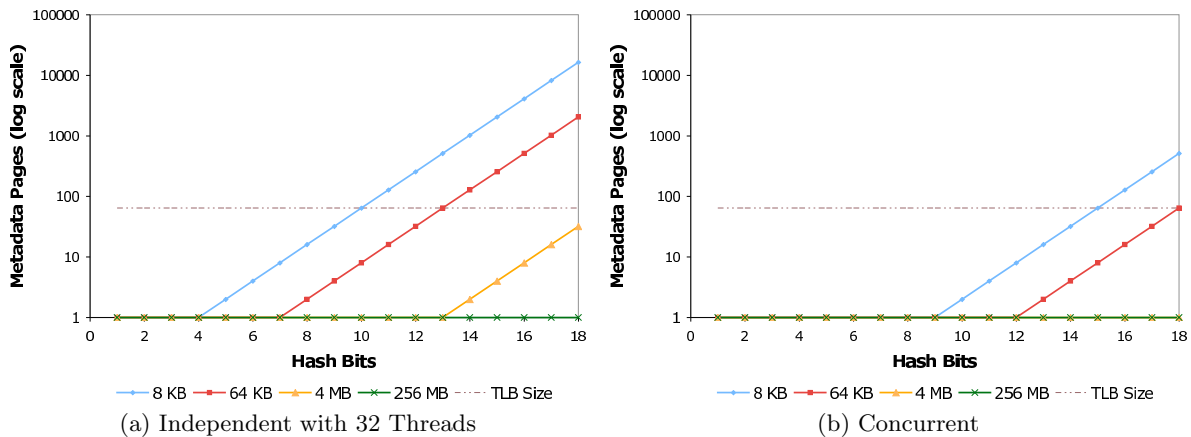


Figure 4: The number of pages required to hold partitioning metadata for different page sizes.

pages are used. This is in addition to the pages required for the 2^b output locations. Caching and searching all of the pages required for both writing output and managing metadata may require the operating system to use multiple levels of complex data structures⁵, straining the system in such a way that it incurs TLB misses while servicing TLB misses. We advocate using large pages for partitioning as this bad TLB behavior is then avoided.

5.3 Cache Misses

The experiments using 256MB pages (Figure 1(d)), where there are no TLB misses, show the impact of cache misses. With large pages, the performance of the four techniques drops off at a higher number of hash bits (more partitions) than with smaller pages. This is because the TLB misses are avoided, but cache misses still impact performance.

Using hardware performance counters, we measured the number of L2 *read* misses per tuple for each tuple processed. Unfortunately, cache misses triggered by *writes* are not recorded by the hardware performance counters. The results however, do show the increasing impact of the space occupied by metadata used by the partitioning techniques. Parallel buffers, which require the most metadata per partition, incur the most L2 read misses for metadata as the number of partitions increases, while concurrent output requires the least metadata per partition and subsequently incurs the fewest L2 read misses.

Even though the performance counters do not count L2 cache write misses, we can still reason about them. To do so, we introduce the concept of an *active cache line*. A cache line is *active* during partitioning from the time that data is first written to it until it has been completely filled. Because our tuples (16B) are smaller than an L2 cache line (64B), a large amount of time may pass between the first and last write. If the time between writes is too great, the cache line may be evicted, resulting in a cache miss on every write.

Ignoring the metadata, we can calculate the number of active cache lines for each partitioning method. If the number of active cache lines exceeds the 3MB L2 cache capacity, then the thrashing described above will occur, causing a cache miss on every write. The L2 cache contains 49152 cache lines. In the concurrent method, threads are shar-

ing an output buffer and incrementing the output location one tuple at a time so there is just one active cache line per partition. Because our input is unique and uniform, we would therefore expect to see an increase in cache misses and therefore a drop in performance when the number of partitions exceeds the number of cache lines in the cache. Based on this analysis we predict that the performance of the concurrent technique would begin to drop off at 15 to 16 hash bits. However, Figure 1(d) shows that the performance drops off earlier. In our analysis we also need to include the metadata stored in the cache and the fact that input is also being continuously loaded into the cache. Each output buffer requires 16B of metadata so the number of cache lines needed for the metadata is $\frac{2^b * 16}{64}$. Additionally, because the input is uniform, the expected number of input tuples read between accesses to a partition is the number of partitions, 2^b , which results in $\frac{2^b * 16}{64}$ cache lines of data added to the cache. With this amount of pressure on the cache capacity, not to mention the chance of a conflict miss, the drop in performance at 13 or 14 hash bits instead of the predicted 15 makes sense. Also, because output buffers are shared one tuple at a time, the point at which cache thrashing starts should be the same, regardless of the number of threads used as shown in Figure 5(b).

In contrast, the number of active cache lines in the other methods, increases as more threads are used because each thread has its own output location for each partition. Even in the case of parallel buffer output, although the buffer is shared, if the chunk size is equal to or greater than a cache line of tuples, each thread will write to unique cache lines within that buffer. Therefore, by using an analysis similar to the one above, we expect performance to worsen due to cache thrashing when fewer hash bits are used than with the concurrent output. This is confirmed in Figure 1(d). Furthermore, the inflection point should move left as more threads are used, which is confirmed in Figures 5(a) and 5(c) for the independent and parallel buffer output techniques, respectively. When few threads are used, there is little cache pressure for any number of partitions.

5.4 Scaling

All techniques stop scaling well once cache thrashing occurs as described above and shown in Figure 5. When cache misses are not an issue, additional threads improve perfor-

⁵A complete discussion of TLB miss handling in Solaris can be found in [9].

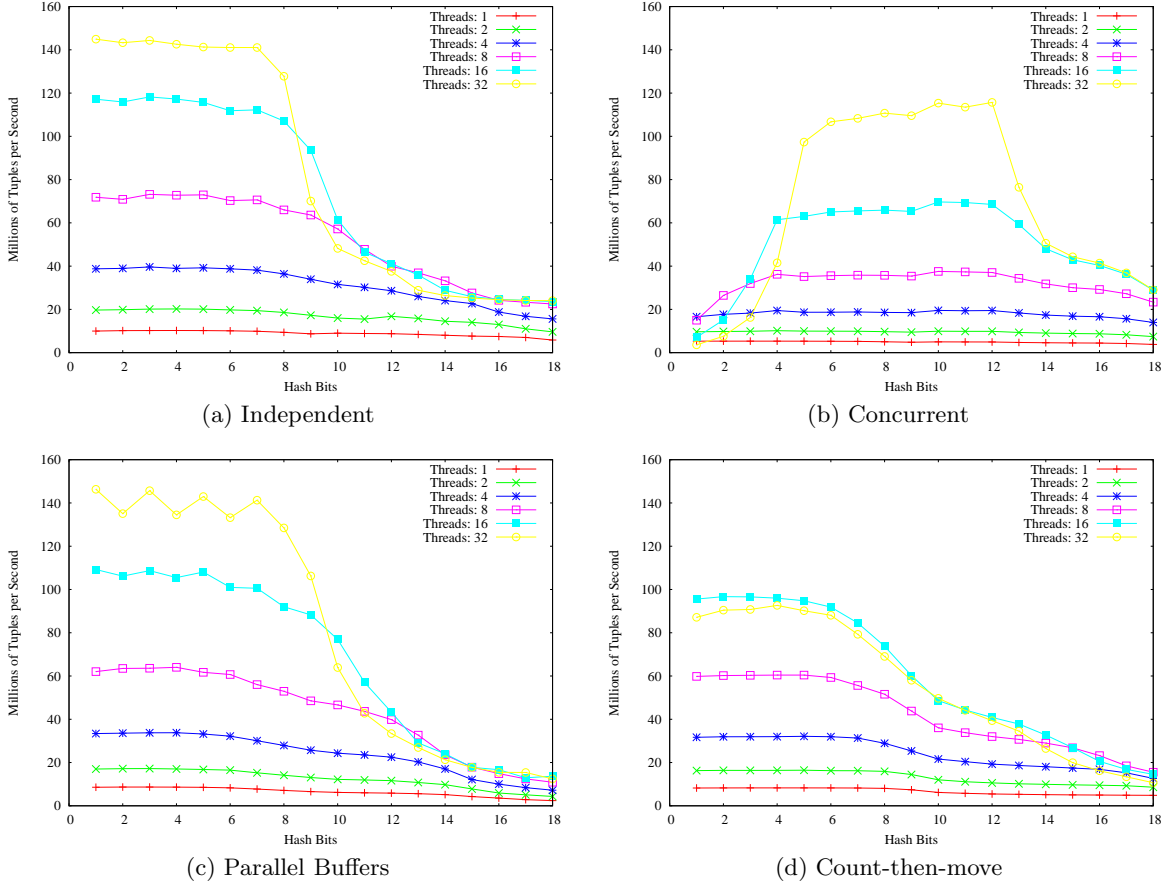


Figure 5: Comparing the scaling of partitioning techniques as the number of threads is varied. The input cardinality was 2^{24} tuples and the page size was 256MB.

mance, but the scaling is not perfect, i.e., a doubling of threads does not double the partitioning throughput. Also, as noted above, for all techniques other than the concurrent buffer, increasing the number of threads increases the cache pressure. This means that cache thrashing will occur at fewer and fewer output partitions as more threads are used. That cache thrashing is such an impediment to good scaling confirms the observation by [8] that suggests multi-pass partitioning, where each pass fits within the cache and the TLB, will give better performance than a single pass using a larger number of output partitions. For CMPs, this analysis is complicated slightly by the fact that output data structures that help to avoid thread contention also scale in terms of the number of active cache lines as the number of threads used increases. Future CMPs with more threads, all other characteristics being equal, will be limited to fewer and fewer cache resident output partitions per pass assuming that all threads are applied to the partitioning.

5.5 Multiple Passes

Figure 6 shows the performance of the parallel and concurrent buffer techniques using one and two passes. In the two pass versions, the first and second passes partition using $\lceil \frac{b}{2} \rceil$ and $\lfloor \frac{b}{2} \rfloor$ hash bits, respectively. As in [8], for very large numbers of partitions, making two passes using fewer hash bits performs better than one pass using all hash bits. A new

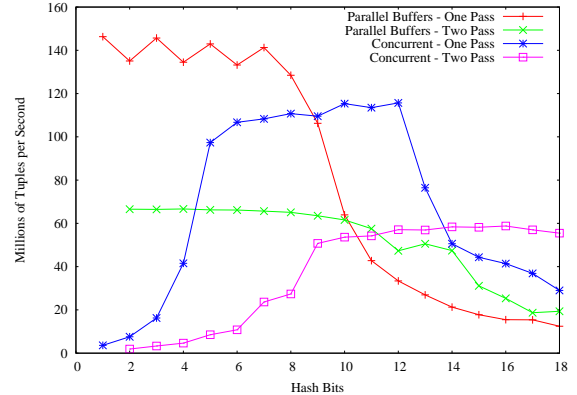


Figure 6: One vs. two pass partitioning using 32 threads and 256MB pages to partition 2^{24} tuples.

result, however, is the balancing of parallelism and the size of data structures that enable parallelism. Although parallel buffers avoid contention and have high performance when less than nine hash bits are used, two pass parallel buffer partitioning performance is worse than single pass concurrent buffer performance when nine or more hash bits are used. This is because the concurrent technique requires less metadata and fewer active cache lines per thread, therefore stay-

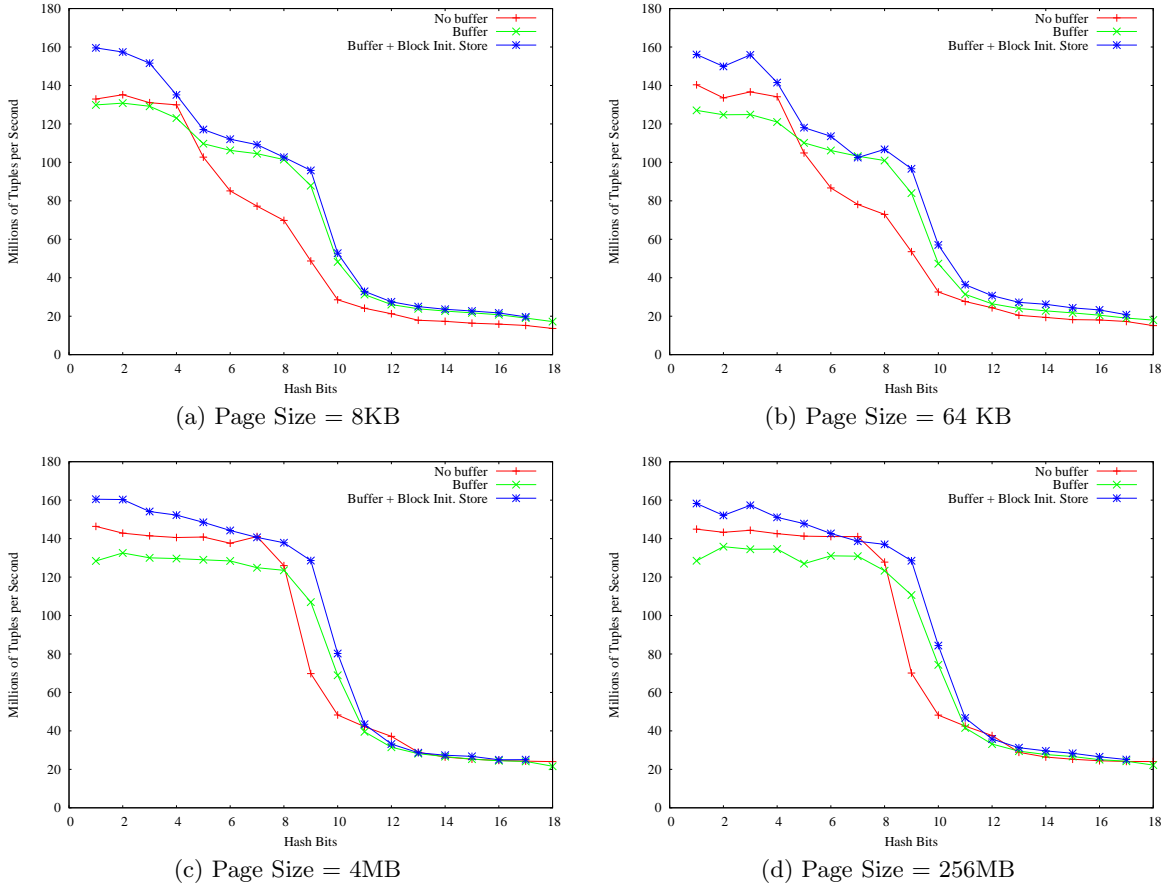


Figure 7: A throughput comparison of the independent output technique with different page sizes when using buffering with and without block initializing stores.

ing cache resident to larger numbers of partitions. Even once one-pass concurrent partitioning performance drops off, two pass concurrent partitioning still performs better than two pass parallel buffer partitioning. Note also, though, that using two-pass concurrent partitioning results in a larger range of hash bits in which it experiences contention.

5.6 Buffering and Block Initializing Stores

Buffering tuples for writing to each output location may improve partitioning performance in some circumstances. One performance problem identified in Section 5.2 is that for high numbers of output partitions, each write results in a TLB miss. By collocating output buffers for all partitions, the buffers take up fewer pages than the corresponding active output partitions. Therefore, placing a tuple into a buffer is more likely to be a TLB hit. Further, when a buffer is flushed to the actual output location, only one TLB miss is required for the entire buffer, thus amortizing the cost of the TLB miss over many tuples. A comparison of the independent technique with and without a buffer is shown in Figure 7. Figure 8 demonstrates that buffering significantly lowers the number of TLB misses per tuple. Buffering alone, however, introduces additional overheads, both in terms of storage and computation, that make it perform worse than no buffering in some cases. This performance can be improved by using a block initializing store instruction when

flushing a full buffer to its corresponding output partition.

When a valid block initializing store instruction is executed, rather than taking a cache miss to load the requested line from memory, a zeroed cache line is allocated directly in the L2 cache [12]. Using such an instruction when writing a buffer of tuples to an output partition, we avoid the latency of a cache miss and save some memory bandwidth. Block initializing stores on the UltraSPARC T1 must start at an offset aligned to the start of a cache line and should write a full cache line to memory. Therefore, buffering of tuples smaller than a cache line in size is required to use block initializing stores. Combining buffering with block initializing stores results in better performance as shown in Figure 7.

Because block initializing stores require the writing of entire cache lines, tuples that do not fit evenly within a cache line or span multiple cache lines may be more challenging to properly buffer and write to a destination. In our experiment, not only are the tuples smaller than a cache line, but they also fit evenly within a cache line. Buffering is useful for our data, but may not be appropriate in all situations. A buffered partitioning implementation for tuples of arbitrary size would need to balance total buffer size with the benefit of amortizing TLB misses. If block initializing stores are to be used, one must also ensure correct writing of whole cache lines, which may be non-trivial if tuples span cache lines and the output buffer is shared among threads.

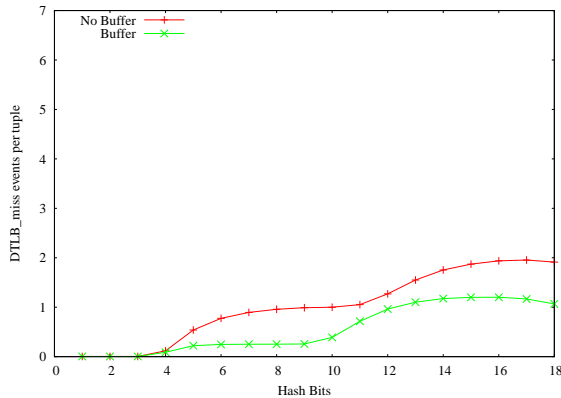


Figure 8: TLB user level misses for the independent output technique using 8KB pages with and without buffering.

6. FUTURE WORK

We realize that not all input is uniform and that uniform input, as used in these experiments, may not represent a worst case scenario for CMP partitioning. Future work includes examining CMP partitioning performance on non-unique, non-uniform input. Non-uniform distributions will have some “heavy hitter” values that may cause some partitions to grow much larger than the expected size given an assumption of uniform input.

In such a case, partition cardinality estimation may be required and output buffers must be able to grow efficiently and concurrently. Concurrent growth, in particular, may be a challenge for the techniques described in this paper that use shared buffers. The two-pass algorithm would remain unchanged for non-uniform input because the initial counting phase handles any amount of skew in actual partition size. The presence of heavy hitters in the input may actually improve performance by improving the locality of reference to some partitions, thereby reducing TLB and cache misses. On the other hand, frequently accessed partitions could become a source of contention if shared between threads.

Other future work includes using CMP partitioning to aid in the parallelizing or improving the performance of other CMP database operations. For instance, it was observed in [2] that an efficient means of clustering group by keys in the input to multithreaded hash aggregation could improve the overall performance of the aggregation operation. Unfortunately, the performance of CMP partitioning on this particular machine does not seem high enough to be able to improve the aggregation performance reported in [2].

CMP partitioning should also be investigated on CMPs with “fatter” cores than the UltraSPARC T1. These cores may have different performance characteristics due to more sophisticated support for prefetching and out-of-order execution and different capabilities in terms of atomic operations. As of publication, however, no commodity “fat” core CMP chips have on-chip thread-level parallelism to match the T1.

7. CONCLUSION

In this paper we have identified output coordination to be a key component of CMP partitioning performance and have presented four techniques for enabling parallel partitioning.

As described in Section 3, all of the techniques have different characteristics in terms of the format of their output and the means with which they enable concurrent partitioning on a CMP. Our results confirm the work of Manegold et al. [8] in that reducing TLB and cache misses is paramount to achieving good partitioning performance. In our study, large pages and sharing output pages among threads eliminate or reduce TLB misses. We also find that cache misses are influenced by the number of partitions, whether the number of active cache lines scales with the number of threads used, and how much per thread metadata is required to manage the output buffers. Finally, an analysis of two pass partitioning shows the importance of balancing space efficiency with enabling contention free parallelism. For smaller numbers of partitions, it is important to use data structures that enable contention-free partitioning. As the number of partitions increases, one must switch to more compact, shared buffer strategies and employ more than one pass for maximum performance. In conclusion, high performance CMP partitioning requires a careful balancing of parallelism, contention, active cache lines, and metadata size.

8. ACKNOWLEDGMENTS

Thanks to Peter Boncz and Marcin Żukowski for the suggestion of exploring partitioning as a means to improve the performance of our adaptive aggregation technique presented at VLDB 2007. We also thank the anonymous reviewers for their helpful suggestions.

9. REFERENCES

- [1] J. Cieslewicz et al. Parallel buffers for chip multiprocessors. In *DaMoN*, June 2007.
- [2] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pages 339–350, 2007.
- [3] D. J. DeWitt et al. Implementation techniques for main memory database systems. In *SIGMOD*, pages 1–8, 1984.
- [4] D. J. Dewitt et al. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, 1990.
- [5] D. J. DeWitt et al. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Parallel and Distributed Information Systems*, pages 280–291, 1991.
- [6] G. Graefe. Parallel external sorting in volcano. Technical Report CU-CS-459-90, University of Colorado, 1990.
- [7] J. L. Hennessy and D. A. Patterson. *Computer Architecture*. Morgan Kaufman, 4th edition, 2007.
- [8] S. Manegold et al. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.
- [9] R. McDougall and J. Mauro. *Solaris Internals*, chapter 8-13. Prentice Hall, 2nd edition, 2007.
- [10] A. Shatdal et al. Cache conscious algorithms for relational query processing. In *VLDB*, pages 510–521, 1994.
- [11] Sun Microsystems, Inc. OpenSPARC T1 microarchitecture specification, August 2006.
- [12] Sun Microsystems, Inc. UltraSPARC T1 supplement to the UltraSPARC architecture 2005, March 2006.