Cache and TLB-aware Parallel Sorting

Kynan Shook

1. Abstract

Sorting is an important function for many applications, and is used as part of many other functions, such as searching or merging data. In this paper, a study is presented of optimizing an existing sorting algorithm while paying special attention to the behavior of the cache and TLB, which can significantly limit performance. The code is also ported to a cross-platform C implementation, and tested on three platforms, which each have different amounts of hardware parallelism. A number of interesting notes on performance and behavior of the sorting algorithms are also presented.

2. Introduction

As computational power increases, memory latencies dominate processor performance. An L2 cache miss or TLB miss can cause a thread to stall for hundreds or thousands of cycles while the request is handled. For this reason, optimization of an algorithm should include investigation of the patterns and causes of these expensive events, and try to reduce them as much as possible.

This paper presents two different areas of work performed. In the first half of the paper, code previously developed in [10] and [11] is improved to reduce the number of cache and TLB misses. Four different strategies were attempted to improve the behavior of the program, though only one area of optimization was successful in obtaining a speedup. In the second half of this work, this code was ported from Objective-C to C, and tested on two other architectures. Another sorting algorithm, a parallel Quicksort-Mergesort, was also implemented, to see if the locality of Quicksort would improve upon the performance of Radix Sort. This process reinforced some of the lessons discovered in the first stage, and yielded some interesting results.

Finally, there are some interesting behaviors discovered in the execution of this code. Several cases of performance that differs from the expected results are presented, along with possible causes. However, due to the complexity of real hardware (as opposed to a simulated system), it is impossible to determine definite cause and effect with any certainty.

2.1 Related Work

The main algorithm used in this paper is Parallel Radix Sort, described in [1]. There are a number of papers that address various improvements to similar Radix Sort algorithms. [9] focuses on improving communication between processors and appropriate load balancing if data is not evenly distributed. CC-Radix [6] is an improvement to Radix Sort that increases cache locality. AlphaSort [7] was an early paper to recognize that optimizing for the memory hierarchy is an important consideration for performance. Additionally, [8] finds that reducing TLB misses in a Radix Sort can significantly improve the speed of the algorithm.

Not all architectures can efficiently support a Radix Sort, so other algorithms can be used, such as that in [4]. Given the parallel speedups achieved by the work in this paper, Parallel Radix Sort is likely not the best algorithm for a massively parallel architecture, however it has performed the best out of a variety of parallel sorts tested in this paper and prior work with up to tens of threads [10].

3. Methodology

In this section, the hardware and software used is presented. Each of the three hardware platforms used has a slightly different mechanism to support multiple concurrent threads. The first three parts of this section describe this hardware. Next, the sorting algorithms used in these evaluations is described in the following two sections. Finally, the last section describes the performance tools used to analyze the algorithms.

3.1 PowerPC

The primary architecture used for testing is an Apple PowerMac G5. It consists of two PowerPC 970FX processors at 2.7 GHz. Each processor has a 1.35 GHz bus to the memory controller, with 32 bits in each direction. The DDR memory bus is 128 bits wide, and runs at 400 MHz. The processor has a 64 KB direct-mapped L1 instruction cache, a 32 KB 2-way set associative L1 data cache, and a 512 KB L2 cache [3]. The TLB is 4-way set associative with 1024 entries, and supports hardware reload [5]. Although the hardware supports 16 MB pages in addition to the default 4 KB size, the operating system used (Mac OS X 10.5) does not, so large pages were not used. The built-in performance counters of this processor were used many times for this work. Each processor can count up to 8 events simultaneously, and about 400 different events can be monitored.

3.2 Clovertown

The Clovertown architecture used is a pair of Intel Xeon E5345 processors at 2.33 GHz. Each processor has 4 cores, and each pair of cores shares a 4 MB L2 cache. Intel does not appear to have publicly released detailed information on the cache and TLB of this processor model.

3.3 Niagara

The Niagara machine used is a Sun UltraSPARC T1 processor in a Sun Fire T2000 server. It has a single processor with 8 cores. Each core supports 4 threads simultaneously, allowing 32 threads to run concurrently, though each core is only single-issue. It has a single 3 MB 12-way set associative L2 cache, and each core has an 8 KB 4-way set associative L1 data cache. There is one fully associative 64-entry TLB per core for data references, and a second identically-structured one for instruction references [12]. The CPU supports reloading the TLB from memory without software intervention [13]. It supports page sizes of 8 KB, 64 KB, 4 MB, and 256 MB.

3.4 Parallel Radix Sort

Parallel Radix Sort is a form of Least Significant Digit radix sort, where the data is sorted first by the least significant bits, and then iterates to sort data by the more significant bits, until the output is completely sorted. It is a stable sort, meaning that keys with identical values stay in the same order with respect to each other. Parallel Radix Sort also uses an underlying sorting algorithm for the local sorts, typically either counting sort or bucket sort. In this implementation, counting sort was used. The parallelization is achieved by performing the counting sorts in parallel.

Counting sort works by first counting the number of keys for each possible value, then using these counts to reposition each key in the data to its appropriate location, determined by how many preceding keys there are. The parallel version works by splitting the data evenly among the available processors and counting the frequency of each of the keys. This process results in a sum for each key on each processor, so then the processors split the keys evenly, and compute sums for each one, generating a global total frequency of each key. This is used to create prefix sums, giving the number of keys that occur before any particular key of any thread. Then, each key can be repositioned in the data array using these prefix sums. As data is shuffled, the prefix sums are adjusted to point to new locations; for example, if data with the key 5 begins at offset 10 in the final data array, the first data with the key 5 is written to offset 10, the second (which is not necessarily encountered immediately after the first) is written to offset 11, and so on.

Radix sort enables counting sort by limiting the number of keys at any given time. When sorting a 32-bit number without radix sort, the computer would have to count how many occurrences of each of the 2^{32} possible numbers there are. This would require 16 GB of memory, one word for each possible integer. Radix sorting masks out a set number of bits during each iteration. It then repeatedly runs the counting sort masking a different portion of the numbers to be sorted in each iteration. Because the sort is stable, later iterations with the most significant bits still retain the ordering of the least significant bits when all higher-order bits are identical.

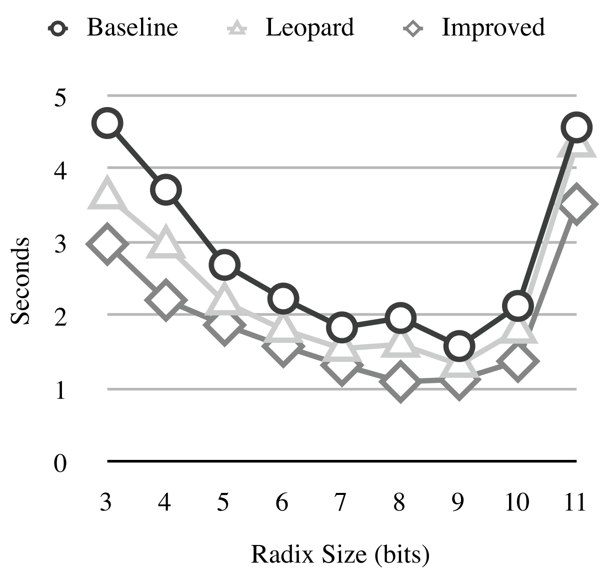

Figure 1: Finding the optimal radix size of Parallel Radix Sort. The original implementation is compared against the speedup from a recent OS update and the algorithm with the improvements described here.

Each iteration of the radix sort has 3 communication points, when the threads must synchronize or communicate with each other. These break each iteration of the algorithm into three phases that are referred to later in the paper; counting, summation, and shuffling. During the counting phase, each thread counts how many occurrences of each digit there are at the current radix offset. The summation phase involves summing all these digit occurrences over all processors. Finally, the shuffling phase moves data based on offsets calculated in the summation phase.

Balancing the number of iterations with the size of the counting sort data structures is important to performance. For a larger radix, fewer iterations may be needed, but more individual counts need to be maintained (2^n for a radix of n bits). A smaller radix, on the other hand, requires more frequent communication between threads, slowing the sorting process as threads wait for each other.

Figure 1 shows a graph of the radix size for Parallel Radix Sort versus execution time. The radix size is compared against the optimal masks determined in prior work. In this work, the radix size used is 8, while in [10] the optimal radix size was found to be 9. This is due to an off-by-one error causing an extra loop iteration if the remainder of an integer division was zero, which was corrected in the improved version. This graph also includes a comparison against the old code running in Mac OS X 10.5 Leopard. This version of the OS improves thread affinity, leading to a performance boost without any changes to the code.

3.5 Quicksort-Mergesort

To test the Parallel Radix Sort against an algorithm with better cache locality, a parallel Quicksort-Mergesort was developed. It divides the input into some number of chunks specified at runtime, performs a quicksort on each chunk, then creates a tree of parallel mergesorts to produce a single sorted output. The idea behind this algorithm was to choose a method such that each working set (here, the individual Quicksorts) would fit into the cache, hopefully reducing the time spent in cache or TLB misses. The Quicksort used performs in-place partitioning, so it uses minimal extra space to sort a large data set. This differs greatly from the Parallel Radix Sort, which cannot copy the data in-place to the same data array, and hence requires approximately twice as much RAM compared to the size of the data.

3.6 Performance Monitoring

The main tool used to gain insight into the behavior of these programs is called Shark, and is part of the free Apple Developer Tools suite. Several methods of statistical sampling were used, and are described here. One is a simple time-based sampling, which interrupts the computer periodically, typically on the order of about 1 ms, but adjustable to meet the granularity needed. This is useful to show where a program is spending most of its time, as these instructions will be flagged in the results of the profile. It can also include threads that are not running, which was used to observe where a thread might be spending a lot of time waiting, such as for a lock or barrier.

Shark can also be used to access performance counters in the CPU or memory controller. These can be used with a time profile, collecting the rate of events, or a single event can be used as a trigger. The latter mode was useful to find where the most cache and TLB misses occurred; by triggering periodically on cache and TLB misses, it can determine which instructions are the most common causes of these expensive events.. Combining information from several performance counters and timed sampling, Shark can produce graphs of more complex events, such as the TLB miss rate over time, or the memory read and write bandwidth used.

For measuring execution times, each platform uses a different high-resolution time library function. On PowerPC, this is mach_absolute_time(). Clovertown used the rdtsc assembly instruction, while Niagara has gethrtime(). Another utility used on Niagara was trapstat, which allowed investigation of TLB miss rates (along with other types of traps) on this platform.

4. Techniques

In this section, several different techniques are described that were implemented as attempts to reduce the number of cache and TLB misses. Their success or failure is also determined, as well as an analysis of what makes them good or bad.

4.1 Segmented Counting Sort

The first modification attempted was to operate with only a subset of digits in the radix at once during the counting sorts. This would, for example, count only half of the digits in the radix at a time. The benefit is that it only uses a fraction of the count array at a time, however it requires iterating through the unsorted data more to make up for this. This modification also only moved data with a subset of the digits in the radix. This reduces the number of locations that the data shuffle step needs to write to, reducing the number of cache and TLB misses in the shuffle phase. However, this modification also increases the work done, as the source data must be iterated through several times.

The result of this modification was a significant slowdown. Because the array of counting sort values is only 1 KB per thread with an 8-bit radix, it easily fits in the L1 cache. Hence, the counting phase of the program is slowed down significantly because a thread must read its entire chunk of data (tens or hundreds of megabytes) multiple times, requiring significant memory bandwidth. The shuffle phase suffers a similar penalty, though it may see a slight benefit because it writes to fewer points in the destination array at once. However, this is not enough to overcome the need to iterate through the data multiple times.

This technique is quite similar to adjusting the radix size, as presented in section 3.4. One advantage this modification could have over adjusting the radix size is a reduction of communication between threads; a smaller radix requires more iterations, and hence communicates more often. However, unlike reducing the radix, running the counting sort on a subset of digits does not reduce the total number of counts maintained, so more memory is used.

4.2 Bucket Sort

Bucket sort is a sorting algorithm that is sometimes used instead of counting sort to implement parallel radix sorts. Instead of counting the number of occurrences of a digit, bucket sort simply moves the data to a particular bucket based on the key's value at the radix currently in use. The output of the bucket sort is a concatenation of the buckets, which can be performed reasonably quickly, since the data access pattern is very predictable.

However, just like counting sort, bucket sort still requires writing to an unpredictable address. Instead of this step happening at the end of each iteration, this happens at the beginning, as the data is appended to the bucket.

Additionally, bucket sort has some additional overhead that counting sort does not. First, the data must be copied from the buckets back into the data array. Although this copy can be performed quickly, the time it takes is not insignificant. Second, the buckets may need to be resized dynamically. Because the algorithm does not know in advance how big any particular bucket will grow, the algorithm can only take a guess each time a bucket is resized. If the realloc() call is unable to expand the bucket in place, it may have to copy the data elsewhere so the bucket can continue to grow. This algorithm also requires more memory, as the total size of all buckets will be somewhat larger than the amount of data being sorted to accommodate these variations in bucket size. Because of all these issues, using bucket sort in place of counting sort also resulted in a significant slowdown.

4.3 Early Counting Sort

In a single-threaded counting sort-based radix sort, the algorithm can generate several arrays of counts at once. For example, with an 8-bit radix size and 32-bit keys, the algorithm can generate 4 arrays with a single pass through the data, using a different 8-bit mask for each. This then allows for only a single counting phase, while all four shuffling phases are still performed. Unfortunately, this does not work with Parallel Radix Sort, which needs the additional information of the frequency of each digit on each thread in order to determine the correct destination and remain a stable sorting algorithm. Because of this, the earliest that the count can be generated is during the previous shuffle step. Because the data is split evenly among processors, the destination of a key implies the next thread that will be moving the data.

So, in an attempt to reduce the total amount of data read (and hence cache misses, as the data is brought in from memory), the algorithm was modified to increment a count array of the appropriate thread when shuffling data to new locations in memory. This nearly halves the amount of data read from memory, since only the first iteration must read the whole input data twice, and subsequent iterations only read it once, while moving it to its next location.

However, this attempt was also unsuccessful because it takes the one structure that easily fits in cache, the count array, and makes it significantly larger. It also requires more communication between threads, which reduces the performance during the summation phase.

4.4 Other Optimizations

Along with the prior three optimization attempts, some more traditional optimizations were made. These had the most success of any of the changes made, giving a speedup of at least 25%, depending on the data size. The optimizations focused on in this section are avoiding the use of globals, reducing the use of library functions that could be avoided, and using appropriate locking libraries to reduce unnecessary work.

The first change made, avoiding globals, was suggested by Shark as part of the static code analysis it performs while viewing performance statistics of a section of assembly or source code. Because the compiler can't determine whether the global will be changed by another thread, a constant load from a global variable inside a loop can't be optimized to outside of a loop. In this case, the globals were used as pointers to the source and destination arrays of data. Although they change after each iteration of the algorithm, they were constant during the innermost loops. To avoid using globals, a local variable is used to shadow the global, and is updated after the synchronization points in the algorithm where the globals change. This way, the compiler can optimize out two instructions from the most time-consuming loops in the algorithm, about a 10% reduction in the number of assembly instructions.

The next change made was to find more appropriate locking mechanisms than the ones that were initially implemented. Due to the limited timeframe available during initial development of the

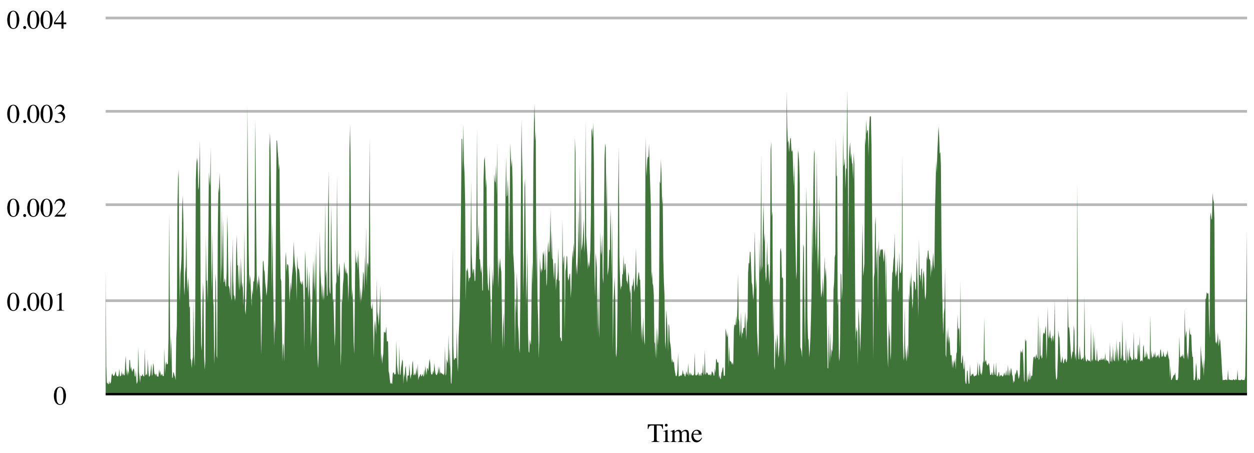

Figure 2: A graph of the TLB miss rate per instruction completed in the initial algorithm. The four shuffle phases are easily seen by the 4 groups of large spikes, and the 4 counting phases are the quiet periods just beforehand.

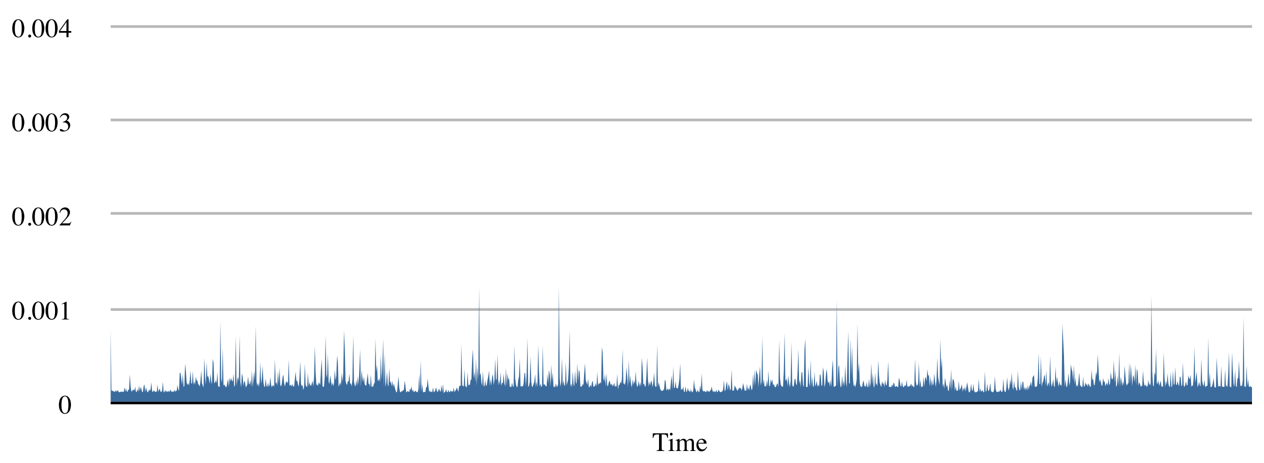

Figure 3: A graph of the TLB miss rate per instruction completed in the improved algorithm, using the same vertical scale as Figure 2. The shuffle phases are still visible, but are much less pronounced.

algorithm (and the limited parallel programming experience of the author), a poor choice was made for the locking implementation. Instead of using locks, the program used a sort of semaphore; the worker threads would set a boolean variable to indicate they had finished running. When the main thread saw that all threads had stopped, it would perform any necessary work, then reset the boolean variable. The worker threads would observe this, and continue running. When a thread was waiting for the semaphore value to change, it would sleep for several tens of milliseconds, yielding the processor to another thread. This mechanism meant that a thread with no useful work would still take CPU time. Although this was known during the initial design, it was a choice made to reduce the development time. It was also initially believed to be an acceptable solution, since the number of threads running was expected to be equal to the number of available processor cores.

Now, with more time to improve the code, several appropriate locking mechanisms were investigated. The one chosen for use is a class provided by the OS called NSConditionLock. A thread can use it as a simple lock, or can conditionally lock based on its value. In this implementation, the synchronizing thread asks to acquire the lock when the condition is zero. Then, the last thread to finish a phase sets the condition to zero. The condition lock guarantees that it will only wake the synchronizing thread when the condition is met. To wait for the next phase, the worker threads each request to acquire the lock when it was, for example, one. To avoid a race condition caused when one thread finishes a phase before all threads have started the phase, the worker threads wait for a different condition value at each phase of an iteration. Using this class, threads are not scheduled on the processor until the condition is set appropriately.

This change had two distinct benefits. First, it reduced the amount of idle work done to check the semaphores while other threads could be doing useful work. This improvement allowed 4-threaded operation to be slightly faster than 2-threaded operation on the dual processor PowerPC. It also gives the processors an opportunity to handle background system tasks without interrupting the slowest thread in the program. Second, it reduced the number of library calls. In the original implementation, 40% of TLB misses were caused by library calls, especially those related to sleeping and waking the threads. With the improved algorithm, only 6% of TLB misses are caused by operations other than data access during the counting or shuffling phases.

4.5 Improved Performance

In the final version of the algorithm, the total TLB misses were reduced by about a factor of 4. The TLB miss rate (data TLB misses per instruction) for a 256 MB data size went from 0.10% to less than 0.03%. The number of instructions completed was also reduced by about 21%. In the L2 cache, although the miss rate is 40% higher in the improved algorithm, the total number of misses is about 10% lower. A similar trend is seen in the L1 cache, where the miss rate is 30% higher, but the total misses are 20% lower.

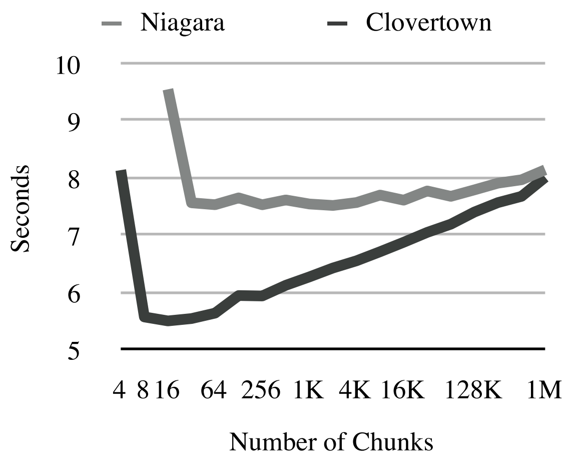

Figure 4: This graph shows the overall execution time of Quicksort-Mergesort as a function of the number of chunks. The total data size is 256 MB. The low chunk measurements of Niagara are omitted to show more detail. That portion of the graph continues nearly linearly to the left from the jump shown between 16 and 32 chunks. Data points were taken at every power of 2 from 4 to 1M (1,048,576).

Additionally, the hardware prefetching is slightly higher for the improved algorithm, helping prevent cache misses during sequential access. In all, a 25% speedup was attained when sorting 256 MB of data in 2 threads.

5. Architecture Comparison

This section discusses the performance of both algorithms across the different architectures used. The first part of this section focuses on Quicksort-Mergesort, including the optimal Quicksort chunk size. The second part is directed at the behavior of Parallel Radix Sort.

5.1 Quicksort-Mergesort

The performance of Niagara was quite level over a large range of chunk sizes. With 256 MB of data, the sorting time was between 7 and 8 seconds for 32 to 512K chunks, representing sizes of 8 MB to 512 bytes. Below 32 chunks, the performance rapidly drops. This is because the parallelism in this algorithm is obtained by having many chunks; with fewer than 32 chunks, not all 32 threads of Niagara can be used. The best performance on Niagara is with 2048 chunks, after which increasing the number of chunks results in a slow decrease in performance. With 2048 chunks, the chunk size is 128 KB. This means that the total working sets for the Quicksort phase at any given time is 4 MB (128 KB for each of 32 threads). Although these chunks can't fit in the L1 cache, they nearly fit in the L2 cache. This is probably optimal for the Niagara since, during a cache miss in one thread, the other three threads on the core can get more work done. This data size provides a good compromise between too many cache misses, which could cause all 4 threads in a core to stall, and too few, which then requires more time to merge into a single sorted list. This algorithm was also run with 4 MB or 256 MB page sizes, but this did not have a significant impact on performance.

Clovertown, on the other hand, performed best with 16 chunks of 16 MB each. It is more difficult to explain this result, however it is likely a balance between the number of cache misses and the number of merge steps needed. Because Quicksort works recursively, only the outermost partition works with the full 16 MB data chunk. With this data size, the call stack will get up to 22 calls deep, but only the outermost 3 calls will be working with data that is larger than the amount of L2 cache per core. Merging, on the other hand, does not make good use of the cache, since each thread moves linearly through the lists to be merged. Because of this, Clovertown works best when the number of merge steps is fairly low.

Figure 4 shows how these two architectures behave as the chunk size is varied. Although Niagara slowly approaches its best performance at 2048 chunks and then slowly retreats as chunk size decreases, Clovertown rapidly reaches its peak at 16 chunks, then loses performance as the chunk size is reduced.

The PowerPC machine has significantly different performance from the other two architectures. Its peak performance is achieved with 4,194,304 chunks, so each Quicksort only operates on 16 values. This drastic difference in behavior is likely because it has significantly less hardware parallelism than the other architectures, so it does not suffer as much from serialization caused by communication and synchronization between threads.

5.2 Parallel Radix Sort

Parallel Radix Sort outperformed the Quicksort-Mergesort on both Niagara and Clovertown by about a factor of two, while on PowerPC, Parallel Radix Sort was about 3 times faster. Unlike Quicksort-Mergesort, adjusting the page size made a difference in the performance of the radix sort. A comparison of execution time, page size, and number of threads can be seen in Figure 5. It can be seen from the shrinking difference between the two lines that as the number of threads increases, the penalty of a TLB miss is reduced. This is because the Niagara can only issue 8 instructions at once (one from each of 8 threads), so it only suffers a penalty if all 4 threads on a core are stalled at the same time.

During this testing, it was found that Sun Solaris will somewhat intelligently choose page sizes based on the amount of data requested. When several hundred megabytes were allocated at once, the allocated memory consisted of 8 KB pages up to a 256 MB boundary, after which the remainder was 256 MB pages. Changing the preferred page size to 4 MB yielded much better results, however, as many fewer 8 KB pages are needed before reaching a 4 MB boundary, compared to reaching a 256 MB boundary. Although this is better than the other OSes used for this paper, it could still be improved by adding further flexibility to automatically use the 64 KB or 4 MB page sizes that the UltraSPARC T1 supports.

Although the Quicksort-Mergesort only reached a speedup of about 8 on Niagara when going from single-threaded to parallel, Parallel Radix Sort got speedups of up to 16. This is likely because a single-threaded Quicksort has fewer cache misses, and can keep a single core reasonably busy. However, because the radix sort has much less predictable data access patterns, running 4 threads on a single core can get a significant speedup, since some threads will be stalled while others can continue executing.

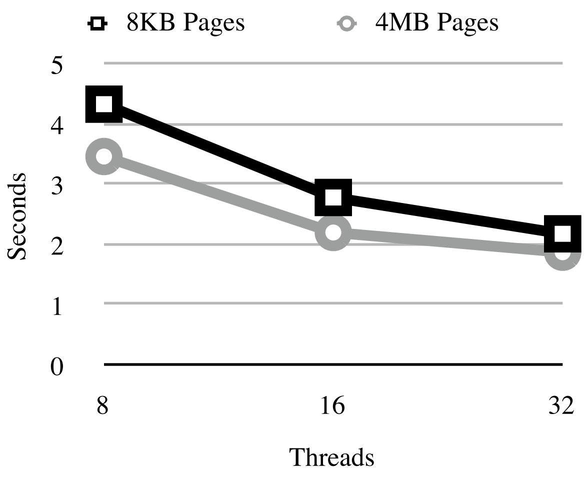

Figure 5: This shows the effect of large pages on Parallel Radix Sort. The total data size is 256 MB.

Also interesting about Niagara's performance on this algorithm is that it easily keeps pace with the more recent Clovertown architecture. The ability to extract significant parallelism on Niagara helps it overcome the lack of a superscalar out-of-order core, like those present in the other two architectures used. In past work by the author that involved programming both Clovertown and Niagara, the Clovertown's superior single-threaded performance helped it outperform Niagara, often by a factor of at least 5. Surprisingly, this was not the case on Parallel Radix Sort. This is probably an indication that the prior work had much higher rates of expensive misses, allowing the out-of-

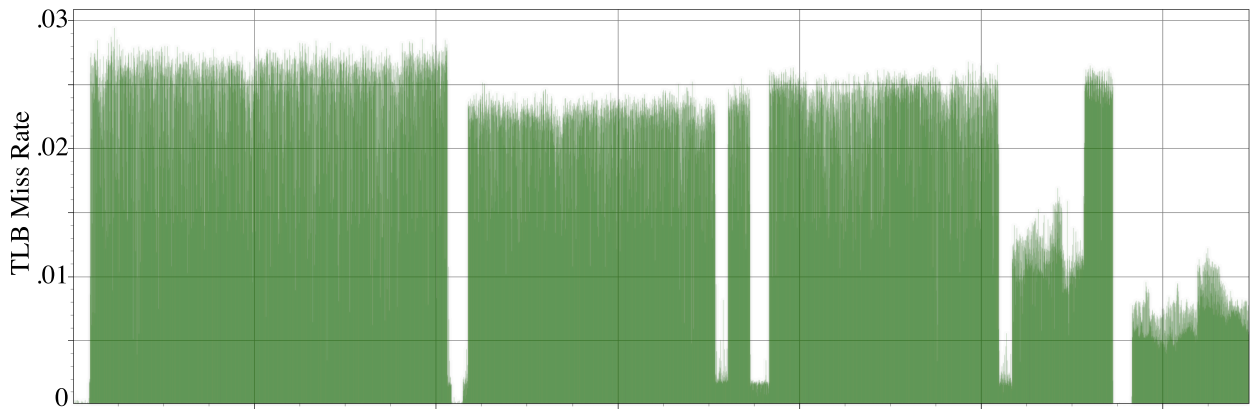

Figure 6: TLB Miss Rate as a function of time when sorting a data set exactly 256 MB in size. The four counting phases show as the periods of low misses, however the third and fourth phases do not line up due to sampling error between the two CPUs.

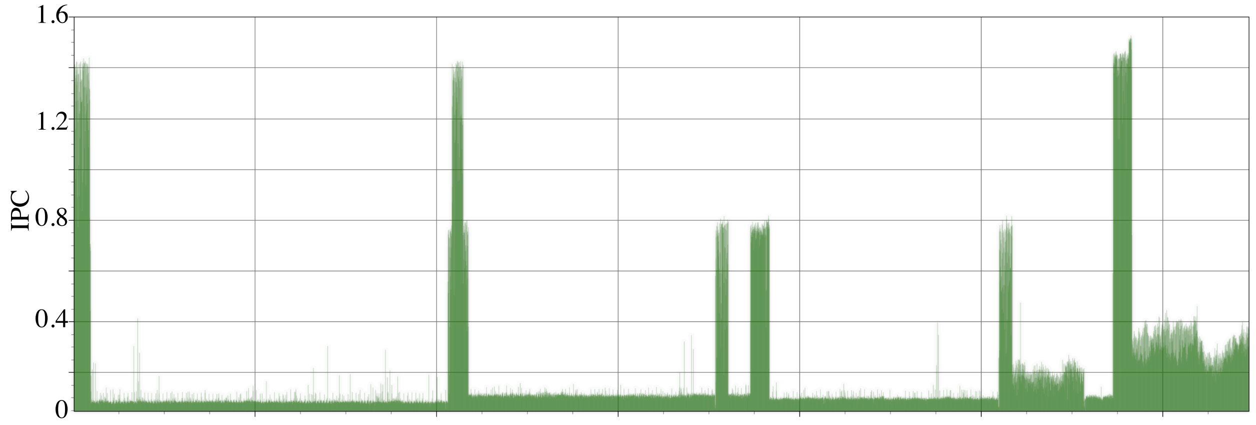

Figure 7: IPC for the same data as in Figure 6. A typical IPC for a different data size is about 1.4 at the peaks (during the counting phase), and 0.5 at the valleys (during the shuffle phase). The peaks here are not affected, however the performance in the shuffle phase is significantly degraded.

order execution to speculatively execute instructions where Niagara would have to stall all 4 threads on a core.

However, the poor performance of Niagara in a single thread did affect the algorithm outside the timed sorting section. Both the creation of random data and the verification of correct sorting operate in only a single thread on all platforms, in order to simplify implementation. While the PowerPC and Clovertown architectures are designed to run single-threaded code very quickly, Niagara can only compete with these chips on highly parallel code. Because these functions were not parallelized, the total runtime of the programs on Niagara was around 5 times longer than on the other platforms, as most of the execution time was spent in data creation and verification, and not on sorting.

5.3 Observations on Parallel Radix Sort

There was a particularly unusual effect discovered in the Parallel Radix Sort that depended on sorting exactly 256 MB of data. Conceptually, the data is treated as a square matrix, a requirement of one of the sorting algorithms used in [10] that is not described here. When sorting a matrix that was exactly 8192 by 8192 elements, performance was significantly slower than on a matrix of 8190 by 8190, or 8194 by 8194. On PowerPC, the difference was as great as a factor of 4, while on Niagara, the difference was significantly smaller, but still noticeable.

The TLB miss rate of this data size is significantly higher than with slightly different data sizes. The average TLB miss rate was 1.1%, compared to 0.02% with a slightly different data size. The pathological miss rate is shown in Figure 6. Almost all the TLB misses occur during the shuffle phase. In addition, the L2 cache has a miss rate about five times higher than normal. Due to this high rate of misses in both the TLB and cache, the IPC is about five times lower during the shuffle phase, causing its runtime to be significantly longer than the counting phase. Figure 7 shows the IPC for the same case, where the effects mentioned can be clearly seen. Typical IPC for the shuffle phase on other data sizes is about 0.5, compared to 0.1 as shown here.

It appears that the more parallelism available, the less pathology this affects the system. As mentioned, relative to a slightly different data size, Niagara handled this case much better than Clovertown, which performed much better than PowerPC. Especially on Niagara, having more threads appears to be able to mitigate these performance hits.

After further investigation, it was found that the prefetch rate was four times higher in the normal case. Obviously, this will increase the cache miss rate, as less data is fetched in advance. This could also be causing the higher TLB misses, as a prefetch could trigger a hardware TLB reload. The PowerPC 970 manual does not specify whether a speculative prefetch will reload the TLB, so this is only speculation, however no other explanation has been found.

However, this doesn't explain the problem completely, since the reason for reduced prefetching is still unknown. In testing on Niagara, the problem disappeared with various combinations of other activity on the system and a different compiler. On all systems, GCC was used as the primary compiler. However, using the Sun Studio 10 cc compiler instead appeared to reduce the effects of this problem. A GCC-created executable also seemed to function normally at this data size when there were no other active users on the system; since Niagara is a shared system, it can be difficult to find a time when it is not running any other processes. This indicates that it could be an artifact caused by both the compiler's optimized assembly as well as interference from other processes running on the system.

This theory hasn't been fully tested, however, as GCC was the only compiler available on the Clovertown and PowerPC machines. Also, the PowerPC machine was running some background system tasks, while the Clovertown machine is also shared with many users.

The improvements to the algorithm also yielded another interesting change in the cache behavior of the program. The initial version would rapidly fluctuate between miss rates of approximately 2%, 4%, and 9%. These changes were not caused by any particular portion of the algorithm, unlike the behavior shown here in earlier graphs. However, the final implementation did not have these variations. To investigate this, a system trace was performed, sampling system calls, interrupts, and context switches. Comparison of this type of trace found that the improved algorithm was typically running for much longer quanta, and was rescheduled on the same CPU more often than the initial algorithm. It is believed that, because the improved algorithm has fewer TLB misses, and some of these probably require OS intervention, that the OS will therefore increase the quanta size. This allows the process to get more useful work done, since it is not spending as much time waiting for misses to be handled. Furthermore, since the quanta are longer, the thread affinity appears to improve, because there are fewer chances for the thread to switch processors.

There is one other unusual effect that was noticed with Parallel Radix Sort. This implementation was originally written in Objective-C, and later ported to C for cross-platform compatibility. Although these two versions can only be compared directly on the PowerPC platform used here, it was found that the Objective-C version ran faster than the C version. However, Objective-C is generally considered a slower language than C, with Apple's documentation suggesting an Objective-C method invocation is about 1.7 times slower than a C function call [2].

Upon further investigation, it was noted that the TLB miss rate of the C implementation was over an order of magnitude higher than the Objective-C implementation, 0.55% compared with 0.02%. Since GCC is used to compile both, it would not appear to be an effect of the compiler.

The main difference between these implementations could also be the cause of the performance drop. The Objective-C implementation uses Apple's Cocoa libraries, which are well-optimized on the platform, as most applications make significant use of them. The C implementation, on the other hand, uses the pthread library, which may not be as heavily optimized, since fewer native applications take advantage of it. There could also be differences in the Objective-C runtime that cause this, however there is no published information on the runtime that would suggest that it could improve TLB or other memory performance over C code.

6. Conclusion

This paper details work done in a variety of areas on two sorting algorithms. The first, Parallel Radix Sort, is by far the fastest sorting algorithm of those that have been tested in the prior work that led to this paper. In addition to its good performance, I have shown that it also scales well, and its parallel speedups are slightly better than those from other work the author has done on the Niagara and Clovertown architectures.

Also, a wide variety of optimizations were attempted with a Parallel Radix Sort. The difficulty of improving locality of one part of the algorithm without degrading locality elsewhere was shown through a series of modifications to the original algorithm. Of the modifications made, the reduction of use of global variables, improved locking and inter-thread communication, and reduction of library calls were the most beneficial.

Comparing performance of Parallel Radix Sort and Quicksort-Mergesort on several architectures showed the flexibility and scalability of the radix sort. Although the Quicksort may be able to fit the whole working set into cache at once, the radix sort can still complete faster. The design of both of these shows that many optimizations are possible that do not require modification of the underlying algorithm. Some of these optimizations were detailed, however, because every program has different behavior, these optimizations may not be necessary, and other optimizations may be more important. For this reason, it is always important to perform optimization with the aid of detailed profiling, rather than attempting to optimize code that may not be in need of optimization.

Several pathologies were discussed that were discovered in the work performed. Although possible causes are given, it is almost impossible to determine the cause with certainty on a realistic system. In a simulation environment, these would be easier to track down; for example, speedup due to reduced TLB misses can be gauged by simulating an infinitely large TLB. However, the performance hit of simulation is significant, so the scope of a simulated program must be quite limited.

The work here shows that cache and TLB performance can be significantly improved without even having to modify the basic algorithm in use. Performing profiling of the code, including observing a variety of events, can yield valuable information on where performance can be improved. The work done in this paper resulted in the time spent in code that is directly necessary for sorting data being increased from about 80% of execution time to 99%. Additionally, 96% of TLB and cache misses during execution of the program occur during the counting and shuffling phases. It is clear from these statistics that any further optimization would have to modify the algorithm itself in ways that were not attempted here.

7. References

1. N. Amato, R. Iyer, S. Sundaresan and Y. Wu, "A Comparison of Parallel Sorting Algorithms on Different Architectures."
2. Apple Inc., *The Objective-C 2.0 Programming Language,* December 2007.
3. Apple Inc., *Power Mac G5 Developer Note*, April 2005.
4. N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, “GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management,” *SIGMOD ’06: Proceeding of the 2006 ACM SIGMOD International Conference on Management of Data,* pp. 325-336, 2006.
5. International Business Machines Corporation, *IBM PowerPC 970FX RISC Microprocessor User’s Manual,* IBM, December *2005.*
6. D. Jiménez-González, J. Navarro, and J. Larriba-Pey, “CC-Radix: a Cache Conscious Sorting Based on Radix sort,” *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed, and Network-Based Processing,* pp. 101-108, February 2003.
7. C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet, “AlphaSort: A Cache-Sensitive Parallel External Sort,” *VLDB Journal,* vol. 4, pp. 603-627, March 1995.
8. N. Rahman and R. Raman, “Adapting radix sort to the memory hierarchy,” *Workshop on Algorithm Engineering and Experimentation*, 2000.
9. M. Schmollinger, “Improving Communication Sensitive Parallel Radix Sort for Unbalanced Data,” *Euro-Par 2003 Parallel Processing,* vol. 2790, pp. 885-893, August 2003.
10. K. Shook, “A sort-did tale of multiprocessor sorting,” unpublished, 2007.
11. K. Shook, “Experimentation into Parallel Sorting Algorithms using Mac OS X,” presented at the 2007 Apple Worldwide Developers Conference (WWDC), San Francisco, California, June 2007.
12. Sun Microsystems, *Developing and Tuning Applications on UltraSPARC T1 Chip Multithreading Systems,* October 2007.
13. Sun Microsystems, *UltraSPARC Architecture 2005,* May 2007.