Algorithms for Sorting on PlayStation®Vita: Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Table of Contents

About This Document

Purpose

This document describes the implementation and performance of a number of sorting algorithms as implemented on the PSP2 platform.

Audience

This document is intended for engineers who are interested in how commonly encountered algorithms perform on the PSP2 platform. A large degree of knowledge about sorting is assumed.

Organization of This Document

This document provides the following major sections:

- Chapter 1, Background – briefly discusses some supporting systems that were implemented and usage of the PSP2 SDK that was required in order to support the experiments we carried out.
- Chapter 2, Serial Sorting – examines single-threaded sorting, specifically Quick Sort and Radix Sort and their performance characteristics.
- Chapter 3, Parallel Sorting – explores the world of parallelism and measures the performance available to developers by exploiting PSP2's multicore architecture.
- Chapter 4, Performance Conclusions and Recommendations – provides a brief comparison to similar experiments carried out on the PlayStation®3, and makes some suggestions about how you can achieve optimal performance from PSP2's CPU.

Typographic Conventions

The typographic conventions used in this guide are explained in this section.

Text

- Names of keyboard functions or keys are formatted in a bold serif font. For example, **Ctrl**, **Delete**, **F9**.
- File names, source code, and command-line text are formatted in a fixed-width font. For example:

```
const float A = delta * delta;
```

Hyperlinks

Hyperlinks (underlined and in blue) are available to help you to navigate around the document. To return to where you clicked a hyperlink, select **View** > **Toolbars** > **More Tools** from the Adobe® Reader® main menu, and then enable the **Previous View** and **Next View** buttons.

Related Documentation and Other Resources

You can find any updates or amendments to this guide in the release notes that accompany the SDK release packages.

Source code for all implemented algorithms is available at:

```
SDK/target/samples/sample_code/system/tutorial_sorting
```

Note: All sorts were compiled with the SNC compiler as of SDK version 0.93.1.

[1] T. Albrecht, "A Question of Sorts", http://seven-degrees-of-freedom.blogspot.com/2010/07/question-of-sorts.html, July 23, 2010.

[2] S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Design* (3rd edition), Pearson Education, Inc., 2005. ISBN: 0-321-33487-6.

[3] D. R. Martin, "Quick Sort – Sorting Algorithm Animations", http://www.sorting-algorithms.com/quick-sort, 2007.

[4] P.Terdiman, "Radix Sort Revisited", http://www.codercorner.com/RadixSortRevisited.htm, July 1, 2000.

[5] C. Vickery, "Interactive IEEE Floating-Point Calculators" (includes reference material), http://babbage.cs.qc.cuny.edu/IEEE-754, September 25, 2008.

1 Background

Introduction

This document discusses the performance and implementation of a number of sorting algorithms on the PSP2 platform. A familiarity with these algorithms is highly recommended. Unless otherwise stated, all tests were performed on arrays of 65,536 elements.

Parallelism

To get the best performance from the PSP2 platform's multicore design, it is best to exploit the available parallelism whenever possible. To this end, a simple job system was implemented whose source code is provided in the samples that accompany the SDK release (see Related Documentation and Other Resources in the About This Document chapter). The job system allows arbitrary functions to be queued up to run on worker threads from the application's main thread. When the worker thread is available it will then consume the tasks in the order they are placed into the FIFO by the main thread. Implementation details of the system are not part of this document's scope.

Razor and libPerf

The PSP2 SDK exposes a number of functions that allow you to interrogate hardware counters on the ARM Cortex A9. Each of the four cores has its own set of hardware counters keeping track of the number of cycles spent performing certain actions. These counters form the basis of our measurements for the algorithms in this document, together with the Razor profiling tool. The C code in Example 1 shows how to initialize the libPerf and Razor libraries in order to carry out CPU profiling.

Example 1 Initializing Razor and libPerf

```
const int32_t initPerf(const void* captureBuffer, const uint32_t bufferSize) {

    sceSysmoduleLoadModule(SCE_SYSMODULE_PERF);
    scePerfArmPmonReset(SCE_PERF_ARM_PMON_THREAD_ID_SELF);

    SceUInt8  eventCode[SCE_PERF_ARM_PMON_PMCNT_NUM] = {
        PERF_COUNTER_0,
        PERF_COUNTER_1,
        PERF_COUNTER_2,
        PERF_COUNTER_3,
        PERF_COUNTER_4,
        PERF_COUNTER_5
    };

    for(uint32_t i = 0; i<SCE_PERF_ARM_PMON_PMCNT_NUM; ++i) {
        scePerfArmPmonSelectEvent(SCE_PERF_ARM_PMON_THREAD_ID_SELF,
                                  i,
                                  eventCode[i]);
    }

    scePerfArmPmonStart(SCE_PERF_ARM_PMON_THREAD_ID_SELF);
    sceRazorCpuInit(captureBuffer,
                    bufferSize,
                    SCE_PERF_ARM_PMON_PMCNT_NUM);
    return SCE_OK;
}
```

Razor markers were placed around sorting code using the appropriate API. For information about using the Razor profiler, refer to the appropriate documentation in the SDK.

2 Serial Sorting

This section discusses serial sorting algorithms that exhibit both linear and logarithmic time complexity. Commonly used sorting functions in the C and C++ standard libraries are the first to be examined and serve as performance benchmarks against which the other sorting algorithms and implementations on PSP2 can be measured.

Standard Library Sorts

Implementation

Both C and C++ offer functions or function templates for sorting arbitrarily typed data as part of their standard libraries. The C and C++ standards do not specify a particular sorting algorithm that standard library implementers must utilize, but they do specify that any such algorithm should exhibit average-case runtime complexity of $O(n \log_n)$, with worst-case performance being no more than $O(n^2)$. This provides standard library implementers some degree of flexibility with respect to their choice of algorithm. The sorting algorithm of choice for most C++ standard library implementers is a hybrid of Heap Sort and Quick Sort known as Intro Sort.

Both functions sort a buffer of objects of a given size using a comparator function specified by either a function pointer or a function object. The C code shown in Example 2 shows an example of the integer comparators used for our tests for the C standard library function.

Example 2 Integer Comparator Function Used with qsort

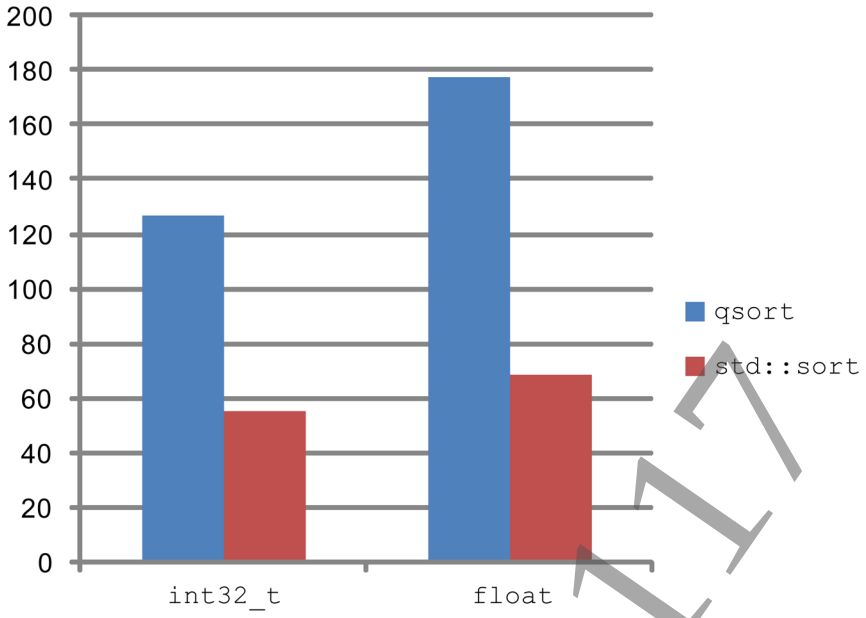
```
int32_t cmpInt(const void* a, const void* b) {

    int32_t aValue = *(int32_t*)a;
    int32_t bValue = *(int32_t*)b;
    if(aValue<bValue)
        return -1;
    else if (aValue>bValue)
        return 1;
    else
        return 0;
}
```

Because both functions form part of the C and C++ standard libraries, no implementation details are discussed. On other platforms, the C++ standard library's sort algorithm is known to offer superior performance to its libC counterpart [2].

Measurements

Similar to other platforms, the libC `qsort` function performs significantly worse than its C++ standard library counterpart in our test cases for both integer and floating-point data. Integer sorting vastly outperformed its floating-point counterpart in both cases, with significantly improved performance observed when using an integer comparator function for floating-point values in the case of `qsort`; this was a common theme in many of our experiments and is discussed in greater detail in Chapter 4, Performance Conclusions and Recommendations. Figure 1 shows the performance observed for `qsort` and `std::sort`.

Figure 1 Performance Measurements for the Standard Library Sorts



Quick Sort

Implementation

Our implementation of Quick Sort is *in place*. This means that it requires no extra temporary storage for intermediate results. This is a departure from traditional implementations of Quick Sort and was a conscious choice on our part to improve the cache and memory efficiency of our implementation. Moreover, our implementation (like most implementations of Quick Sort) is not a *stable* one. The code shown in Example 3 is our floating-point implementation. The source code for the integer version, and also our iterative version (making use of its own private stack) is also available at:

```
SDK/target/samples/sample_code/system/tutorial_sorting
```

For additional details about Quick Sort, refer to [3] or your favorite algorithms textbook.

Example 3 Floating-Point Version of the Quick Sort Algorithm Written in C

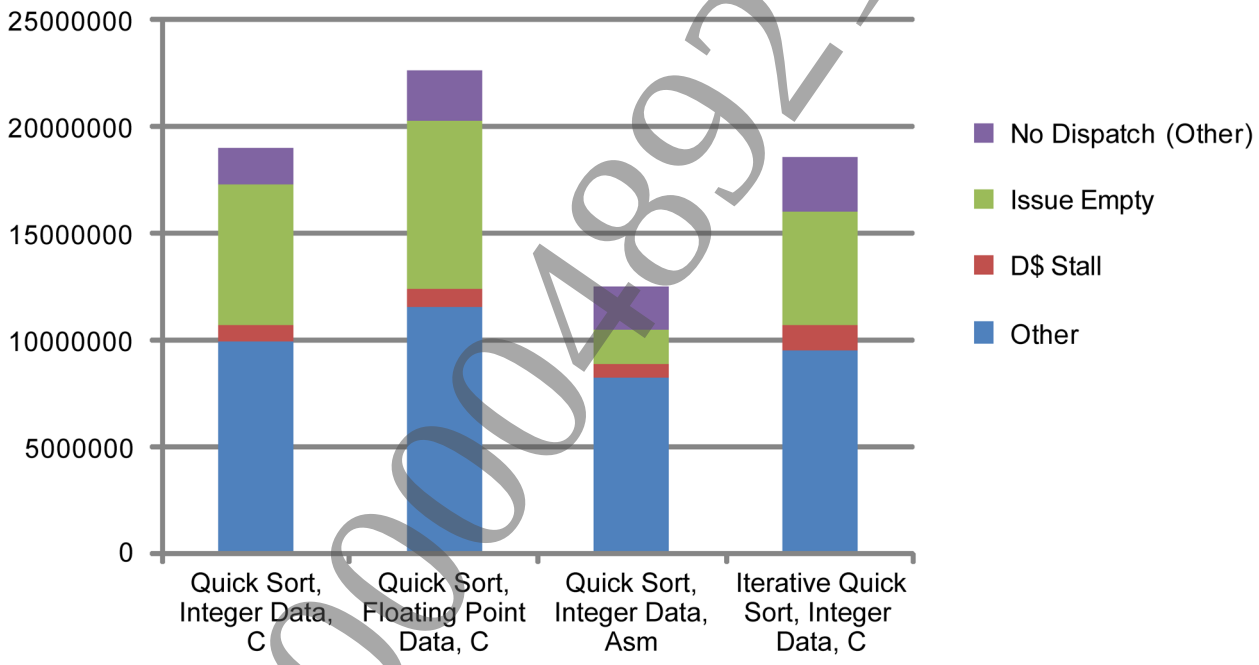
```
void quickSortFloatInternal(float* __restrict a,
                            const int32_t s,
                            const int32_t e) {

    SORT_ASSERT(a);

    if (s < e) {
        float p = a[e];
        uint32_t i = s;
        uint32_t j = e;
        while(i != j) {
            if(a[i] < p)
                ++i;
            else {
                a[j] = a[i];
                a[i] = a[j-1];
                --j;
            }
        }
```

```
            a[j] = p;
            quickSortFloatInternal(a, s, j-1);
            quickSortFloatInternal(a, j+1, e);
        }
        return;
    }
```

Measurements

The results of the various implementations of Quick Sort that we attempted may at first seem a little surprising. Figure 2 shows a breakdown of the notable areas of cycle utilization in the ARM Cortex A9. Perhaps the first thing to note is that performance is not limited by cache utilization. There is a relatively small overhead for data cache misses, which is very similar across each of the implementations of the algorithm. The `SCE_PERF_ARM_PMON_ISSUE_EMPTY` counter (shown as a proportion of the overall runtime in green) records the number of cycles where the instruction issue unit in the ARM Cortex A9 was empty, and thus unable to issue any instructions to the instruction pipelines. For a sufficiently large data set (such as the data targeted for sorting) the `if` statement immediately following the `while` loop will almost always have no relevant history on which the branch prediction unit can base its prediction. This causes (depending on the data) high frequency branch misprediction, which can have a very negative impact on performance; see Figure 2.

Figure 2 Breakdown of Notable Areas of Cycle Utilization



Example 4 Main Loop Written in ARMv7a Assembly

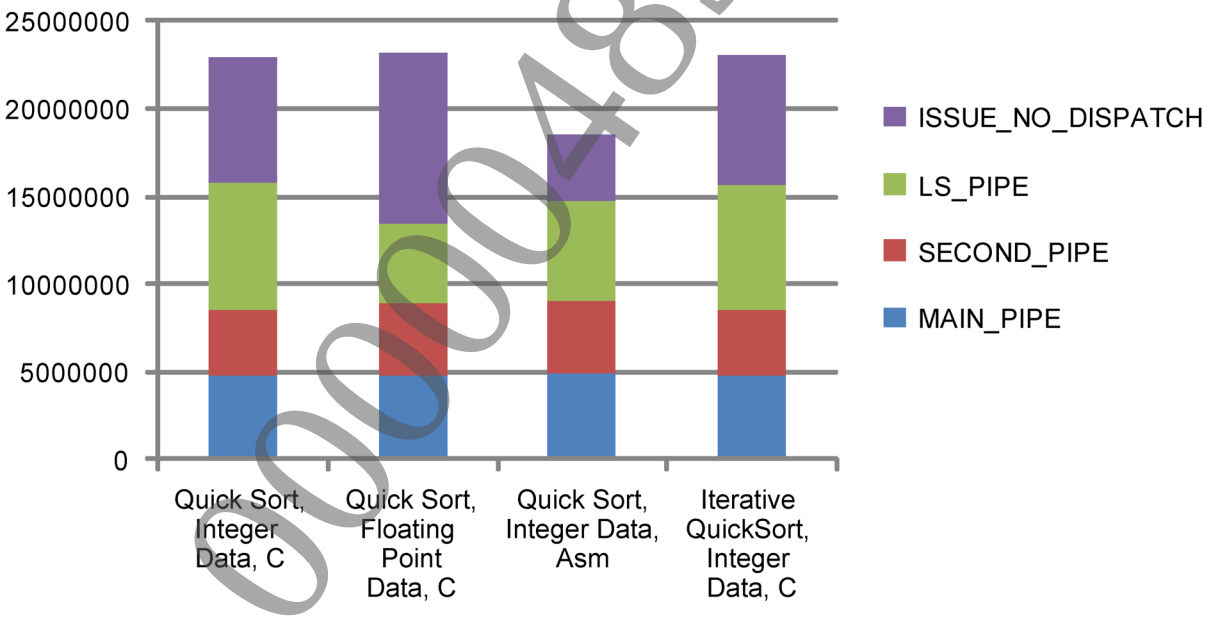
```
qsorti_asm_loop:
    sub    r10, r8, #4
    ldr    r9, [r0, r7]
    ldr    r10, [r0, r10]
    cmp    r7, r8
    beq    qsorti_asm_end_loop
    cmp    r9, r6              /* which side of pivot? */
    ittte    ge
    strge    r9, [r0, r8]
    strge    r10, [r0, r7]
    subge    r8, #4            /* j--, 4 bytes down */
    addlt    r7, #4            /* i++, 4 bytes up */
```

```
    b    qsorti_asm_loop
qsorti_asm_end_loop:
```

One of the more interesting features of the ARM Cortex A9 ISA is the ability for it to conditionally issue any instruction. There are currently no intrinsics that allow an application programmer to target this hardware feature directly, so we are left with little choice but to drop to the assembly level. It is worth pointing out that with some degree of skill we were successful in eliciting the compiler's cooperation in emitting one conditional instruction (although for the simpler case only), but we were unsuccessful in writing C code in such a way that the compiler generated the desired output; this is something that is currently undergoing investigation by the SCE toolchain team. Example 4, shown above, features the relevant assembly mnemonics that form the core of the Quick Sort algorithm. On line 8 of this listing, you can see the `IT` instruction, which sets up the processor for conditional instruction issue of up to four subsequent instructions. In the case shown in Example 4, four was enough, but if you require more, simply issue multiple `IT` instructions between the blocks of conditional instructions. It should be noted that the `IT` instruction takes the form `ITxxx <pred>`, where the `x` is either `T`, `E` or empty (standing for "then" or "else"). Each of these map one-to-one to each of up to four subsequent instructions. `<pred>` is the comparison predicate to use in conjunction with the most recent preceding comparison instruction. The first instruction of the `IT` block must always execute when the result of associated comparison is true. With the branch-free implementation there is a clearly observable performance improvement. This shows that while the ARM Cortex A9 performs fairly well with code containing branches, it is not a magic bullet, and thus cases can be trivially contrived that make branch avoidance techniques very relevant on the PSP2 platform.

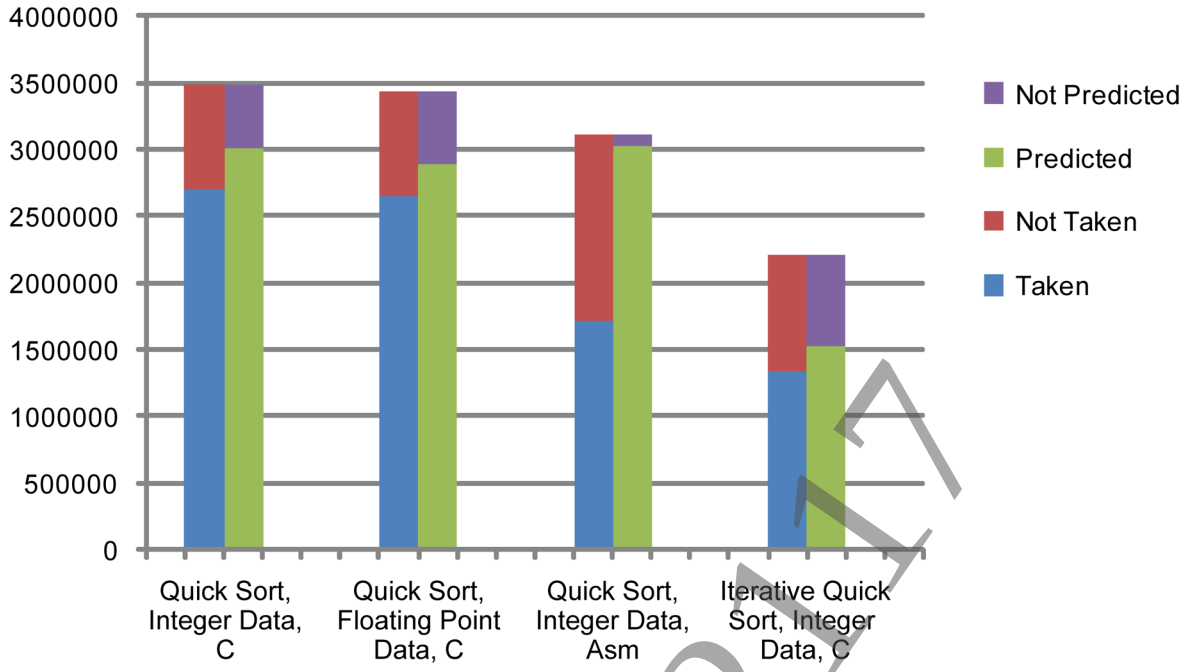
Note: We do not support GNU assembler except for its use to assemble the compiler's output. Therefore, we recommend you avoid writing assembly language and write code in C and intrinsics. This will avoid compatibility problems when the GNU assembler is replaced in the future.

Figure 3 Breakdown of Instruction Issue for Quick Sort Implementations



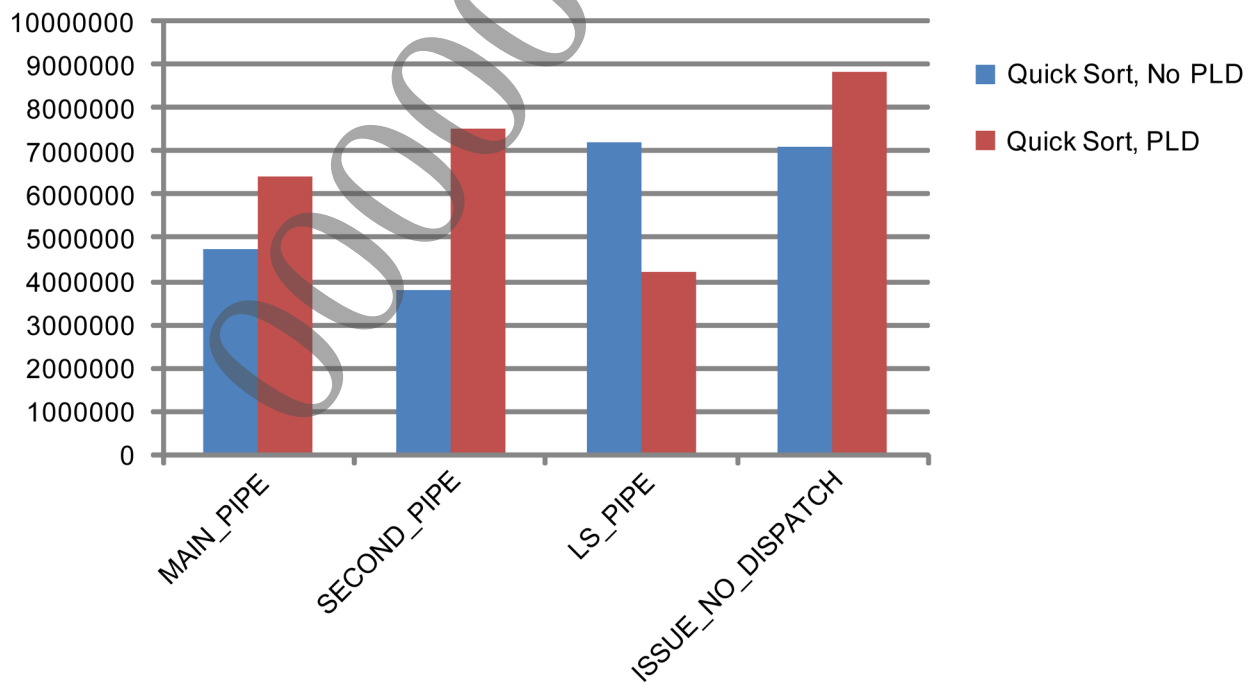
One interesting change from the PlayStation®3 is the unremarkable performance of the iterative version of Quick Sort relative to its recursive counterpart. Albrecht noted in [1] that on the PlayStation®3 Cell Broadband Engine™, the performance of the iterative version of Quick Sort was far superior to that of its recursive counterpart due to the presence of the Load-Hit-Store hazard on the PlayStation®3 PPU in the functions prologue and epilogue. In the iterative version of the algorithm, the hazard can be mitigated through careful use of a user-managed stack. However, as the penalty for snooping, the store queue on PSP2 is much less prohibitive than that of the PlayStation®3. This means that there are fewer significant gains to be made by adopting this technique, although due to the number of function calls the overall number of branches is greatly reduced as shown in Figure 4.

Figure 4 Breakdown of Branch Prediction Performance and Direction



During the course of optimizing this sorting routine (and in the absence of an operational preload engine) numerous attempts were made to utilize the `pld` instruction. The `pld` instruction is able to load the L1 cache with a specific cache line from main memory. In theory, this instruction accelerates subsequent accesses to that cache line if there is enough latency between issuing the `pld` instruction and the instruction requesting data from this cache line. Unfortunately, every attempt to utilize the `pld` instructions in Quick Sort ended in failure. Figure 5 shows the breakdown of instruction issue for the C version of Quick Sort, sorting integer data, both with and without the extra prefetch instructions. This is discussed in greater detail in the Quick Sort section of Chapter 2, Serial Sorting.

Figure 5 Breakdown of Instruction Issue With and Without PLD



Radix Sort

Implementation

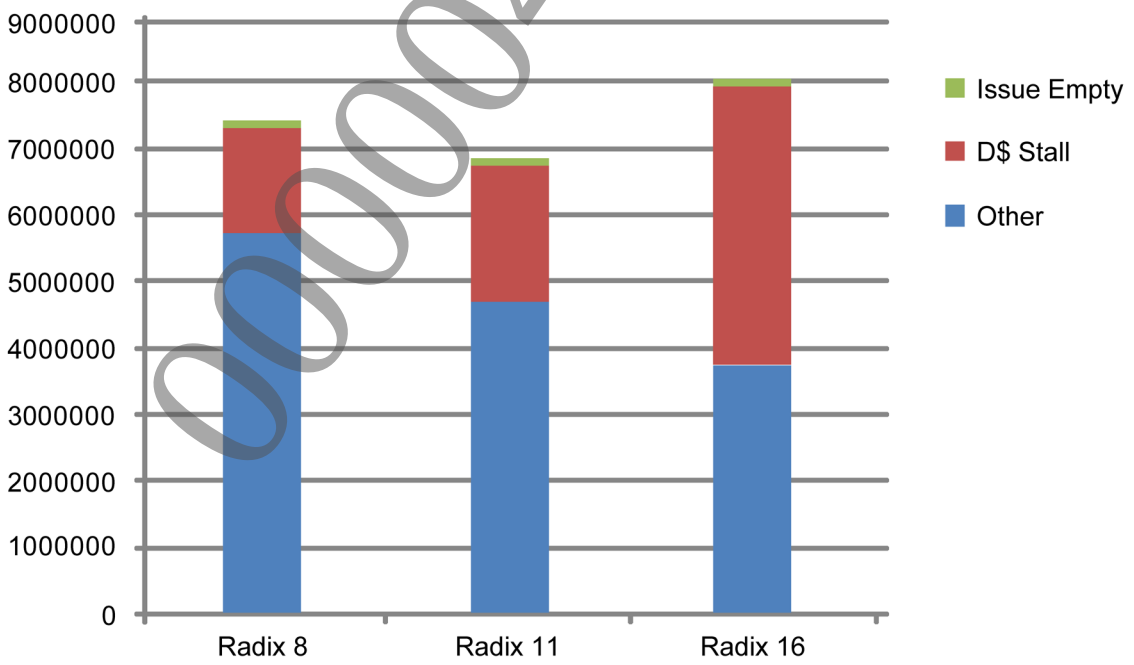
Radix Sort is one of the oldest known sorting algorithms. The algorithm is still of interest because it fits into a family of sorting algorithms that do not rely on comparisons in order to sort a list of values. Because it does not rely on comparisons, it is able to break free of the $O(n \log_n)$ complexity constraint that is intrinsically placed on comparison-based techniques, such as those encountered thus far. Radix Sort is guided by the simple principle that values should be placed in the correct position in the output array at the point they are encountered. However, for the algorithm to operate within realistic memory constraints, multiple passes are usually carried out with a radix value that is of less precision than the values being sorted.

In order to investigate the performance of Radix Sort, versions with different size radices were created; this had the effect of reducing the number of passes over the data at the expense of memory used to store histograms and offset tables. In addition to the issue of radix size, there is another implementation detail of Radix Sort that can drastically alter the properties of the algorithm: the sort direction. Most serial implementations of Radix Sort are done by starting with the least significant bit (backward sorting), due to simplicity of implementation and to avoid recursion. Although beginning with the most significant bit, forward sorting has significant benefits when parallelism is considered, as is discussed in Chapter 3, Parallel Sorting. Details about how our Radix Sort implementation works with floating-point values are provided in the appendix.

Measurements

Our initial implementations of Radix Sort used 8-bit, 11-bit and 16-bit radices and thus four, three, and two histograms respectively. All versions began by sorting the data from the LSB through the MSB. The performance of all three implementations was a significant improvement over all of the comparison-based techniques discussed so far, with the algorithms being limited largely by their patterns of data access (to varying degrees). Figure 6 shows a breakdown of the processor cycles expended for sorting 65,536 floating-point values with these three versions of Radix Sort.

Figure 6 Breakdown of Cycle Expenditure for Radix Sort Implementation



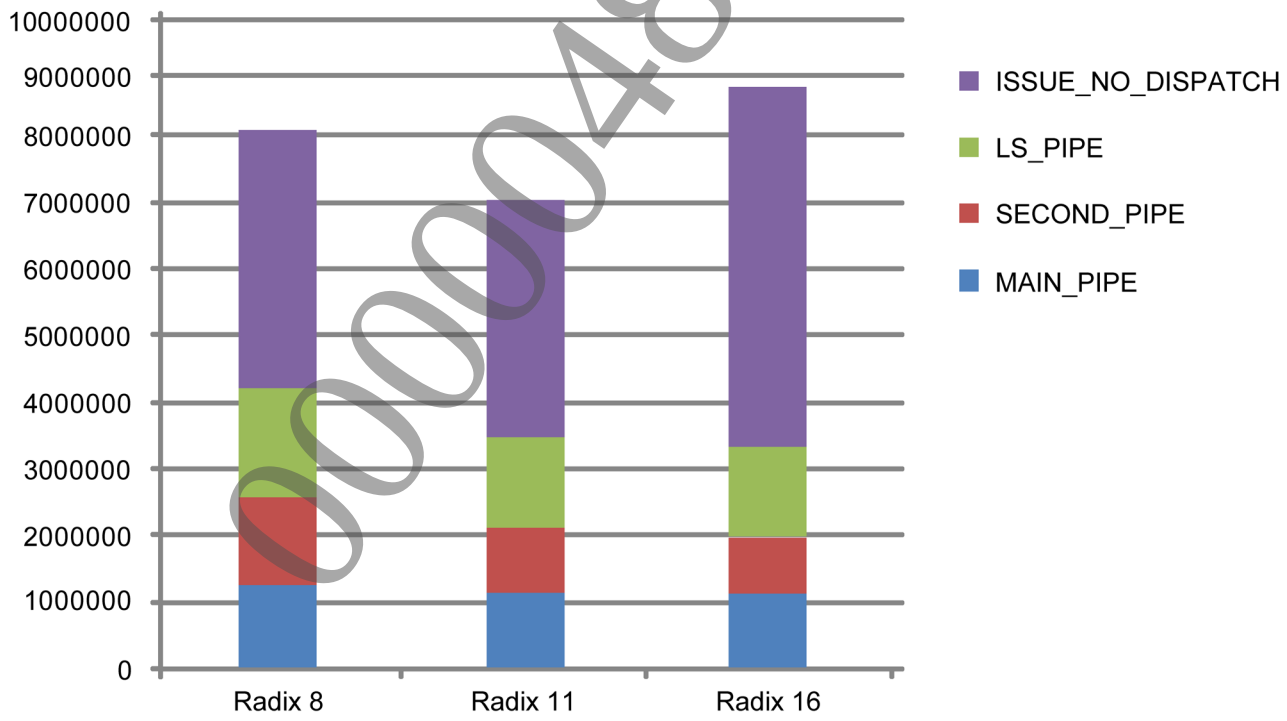
The absence of data-sensitive branches in the algorithm means that the number of cycles where an instruction is not issued, due to an empty instruction dispatch unit, is low. This is a significant change from the comparison-based approach. However, there are a large number of cycles where instruction

issue is stalled, with most attributable to stalls in the cache hierarchy or to register dependencies between instructions (that is, the latency between dependent instructions). For our dataset, the number of mispredicted branches attributable to each of the implementations was between 100 and 300 in total, a small number indeed considering the values being seen thus far. This is reflected in the low empty issue rate.

As mentioned, the data access pattern contributes greatly to the overall runtime of the algorithm, with the cost largely coming from non-linear access to the offset table during the final passes over the data. Access to the unsorted array itself is predictable and cache-friendly, meaning that the implementation with the smallest histogram, required only to accommodate 2^8 values, spends the least amount of time waiting for data to be retrieved from main memory. Indeed, Figure 6 shows that the data cache stalls increase almost linearly with the assumable ranges of the radices and thus the size of the histogram in memory. Unfortunately, because we increase the range of the radix by a power of two, the memory requirement for histograms also increases at an exponential rate, which leads to significantly worse cache performance. In our experiments, Radix11 was well-positioned because the degradation in cache performance was comprehensively outweighed by the gains made through the decrease in the overall number of instructions issued.

You may recall that the algorithmic complexity of Radix Sort is expressed as $O(kn)$, where k is the number of radices that can split a single sortable value, which corresponds to the number of passes required over the unsorted array in order to sort it. The reduced value of the constant k in this expression gives rise to a drop in the number of operations that must be performed in order to correctly sort the dataset. Again, this can be seen in Figure 6 because the blue bar, which represents the number of cycles spent not attributable to data cache misses, becomes smaller as the size of the radix increases with each implementation. Figure 7 further demonstrates this by showing the breakdown of issued instructions over the available pipelines in the ARM Cortex A9; the large purple sections in each of the bars show the instruction dispatch unit being stalled, largely by the data cache.

Figure 7 Breakdown of Pipeline Issue for Radix Sort Implementations

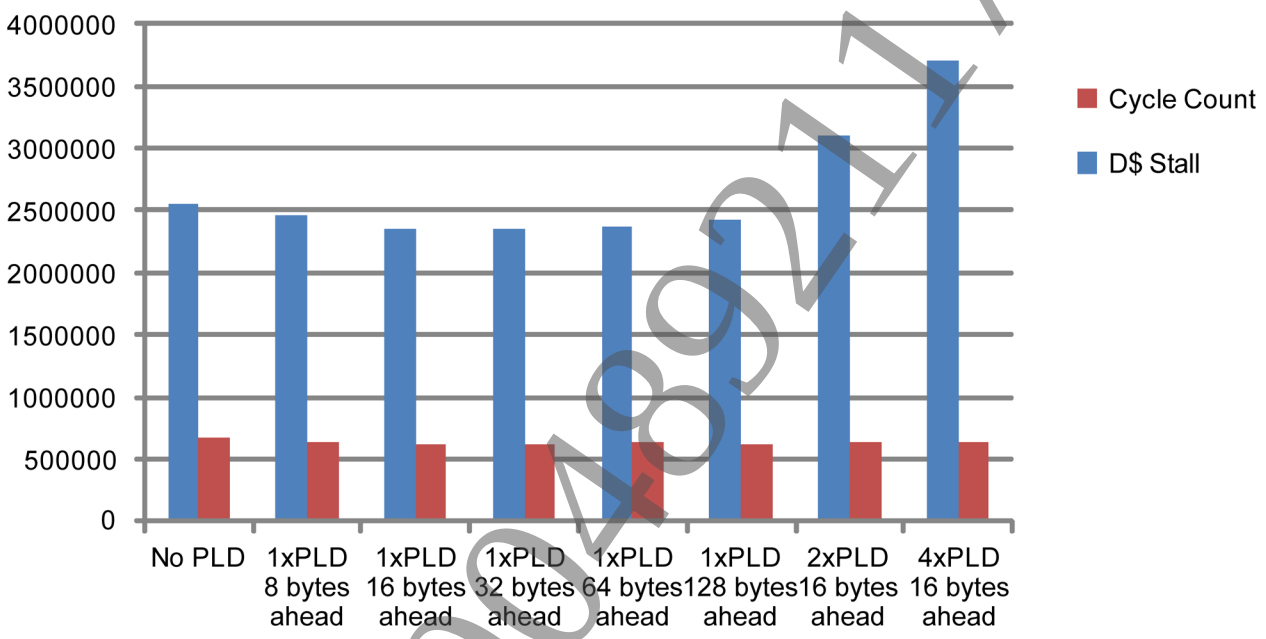


Because all implementations of backward Sort have linear access patterns for unsorted and partially sorted data (with interleaved scatter/gather memory access for the histogram), attempts were made to speed up the implementation through the use of the `pld` instruction. On the ARM Cortex A9, `pld` instructions are handled in a dedicated unit containing a four element FIFO, which loads the L1 data cache with the 32-byte cache line containing the address requested. If the PLD unit becomes saturated, this

can introduce a PLD stall. In a short loop, the cost of issuing the `pld` instructions (and the instructions associated with prefetch address calculations) can often outweigh the advantages afforded by the use of `pld`. Care must be taken to ensure this is not the case. Additionally, we must also ensure that there is the correct amount of time between the prefetch request and the attempt to access that location from the LS pipe. Too little, and the request will not have been completed (although this should reduce the cost of the associated cache miss); too much and the cache line may have already been evicted from the L1 cache.

For the purposes of our experiments we considered the histogram calculation in isolation. The calculation of the histograms constitutes the greatest single cost to our Radix Sort's performance on PSP2. The greatest improvements to performance were achieved through the use of a single `pld` instruction, which prefetched data 1-2 cache lines ahead of the current location in the linear traversal. Figure 8 shows the performance of different prefetch strategies when issued in the initial loop responsible for histogram calculation. Note that even by issuing four `pld` instructions per loop we were unable to saturate the PLD unit, which resulted in a PLD stall.

Figure 8 Performance Differences with Different L1 Data Cache Prefetch Strategies



In our case, the caching of the modified floating-point values back to the original data stream is likely to have inhibited the effectiveness of cache prefetching by causing our issued `pld` instructions to be later nullified in the prefetch unit. This is because the prefetch targets touched similar cache lines in main memory to those that are already the target of stores. We noticed that separating, into two separate loops, (a) the application of the floating-point transformation function to the unsorted data stream and (b) the histogram calculation gave us a very small increase in speed without the need for any prefetching. Additionally by separating loads and stores for all histograms, we were able to again observe a small increase in overall runtime performance.

A Final Word on Serial Sorting

It would appear that Radix Sort is a clear winner in the serial-sorting performance stakes. Radix Sort is discussed again in the next chapter, which deals with sorting algorithms that exploit parallelism, but first this chapter concludes with a brief summary of how the different serial algorithms investigated in this chapter perform over differently sized datasets. All tests were conducted with 32-bit integer values.

Figure 9 Graph Showing Sort Algorithm Performance over Differently Sized Data Sets

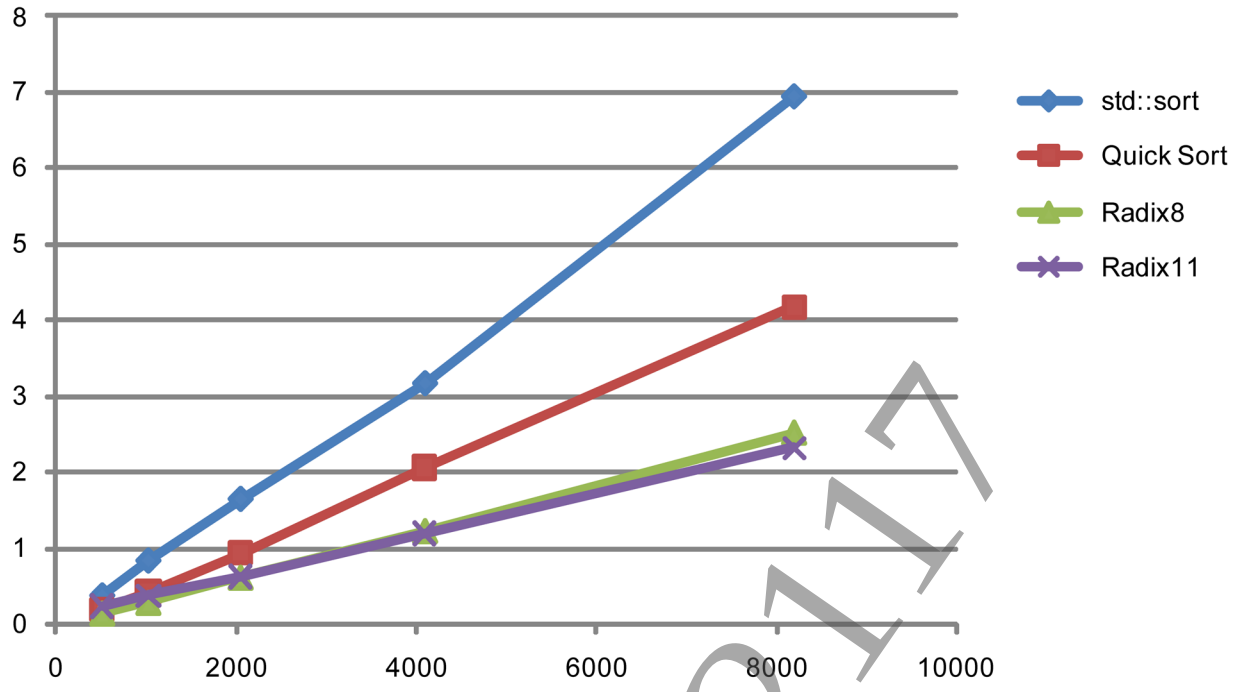


Figure 9 shows that for sufficiently large datasets our comparison-based sorting algorithms perform significantly worse than their linear counterparts. However, as the dataset becomes smaller the setup overhead of calculating histograms and offsets tables as required by Radix Sort becomes more expensive relative to the cost of sorting the data. The swing in favor to the logarithmic algorithms occurred at datasets of approximately 128 elements in size. The two Radix Sorts measured with different datasets also show that the benefit of having one less pass over the data (as is the case with Radix11) are lost for datasets smaller than 4096 elements; this is in accordance with our expectations.

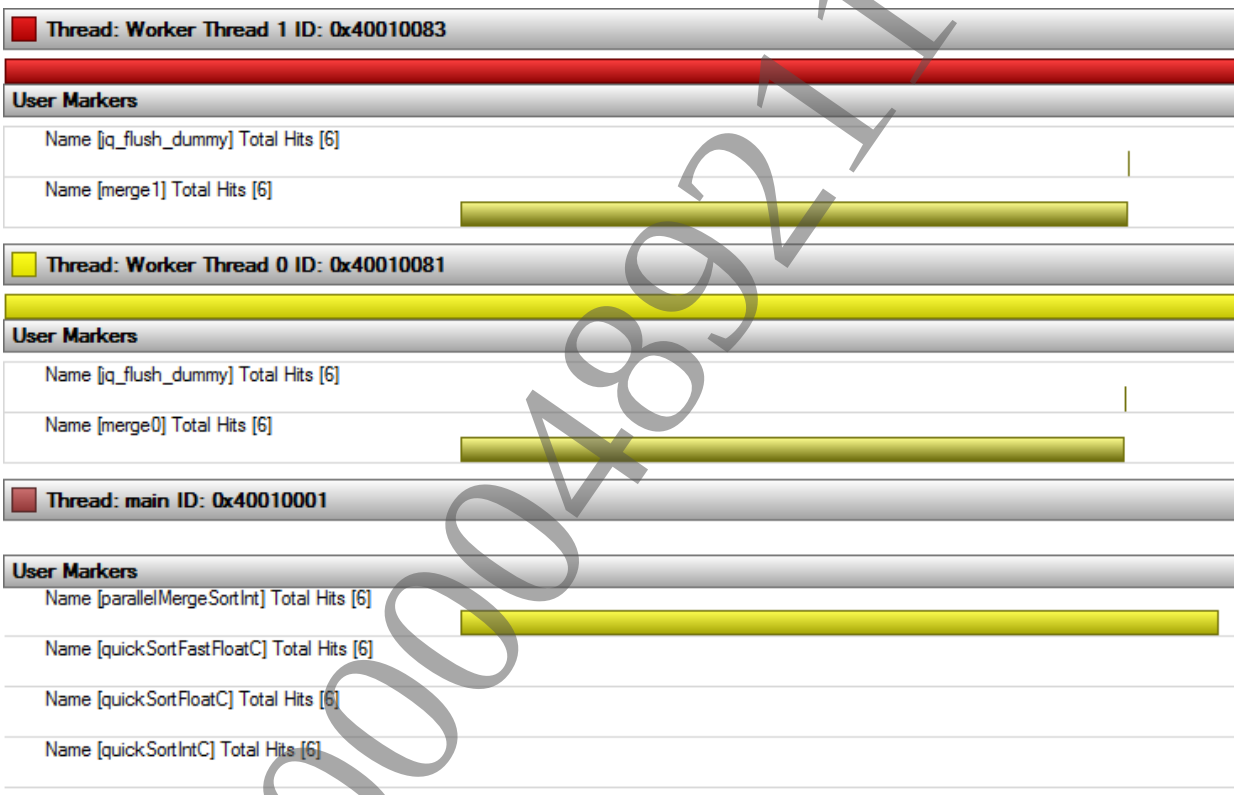
3 Parallel Sorting

Parallel Merge Sort

Implementation

Merge Sort works by recursively splitting a dataset into two or more pieces, recursively sorting them, and then merging the results back into a single array. This approach gives rise to an interesting property of Merge Sort; all but the last merge step are computable in parallel. Our implementation for PSP2 simply splits the dataset into two or three chunks and performs a serial Merge Sort on each chunk of the dataset on separate worker threads simultaneously. In our implementation, the main thread performs the final merge step, which produces a correctly sorted array. In Figure 10, you can see a Razor capture that shows merge tasks running simultaneously on all three available processing elements.

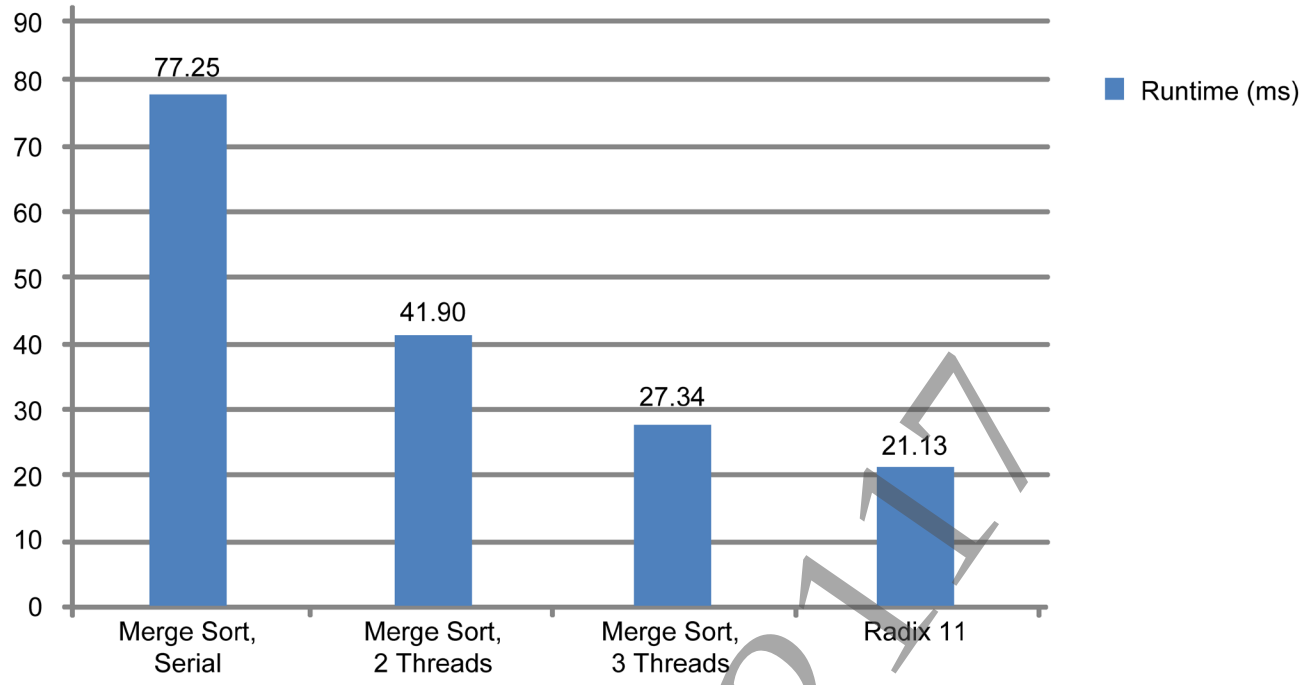
Figure 10 A Razor Capture Showing Merge Sort Running on Three Worker Threads



Measurements

The hardware counters obtained for parallel Merge Sort do not highlight anything particularly new in terms of hardware resource utilization. Because all cores are operating on independent data sets as mandated by the algorithm, there is little or no performance increase from PSP2's L1 cache snooping hardware. The interesting thing here is the net speedup we were able to obtain from the available parallelism. Figure 11 shows the performance of the parallel versions of Merge Sort compared with their serial counterpart. The speedup is certainly noteworthy, almost a linear gain for each additional core added, but the overall performance still leaves something to be desired when compared to any of the Radix Sort implementations from Chapter 2, Serial Sorting. This emphasizes the point that although there are significant speedups to be found through parallelism, it is important to choose the right algorithm in the first place.

Figure 11 Runtime Performance of Serial Merge Sort, Parallel Merge Sorts, and 11-Bit Radix Sort

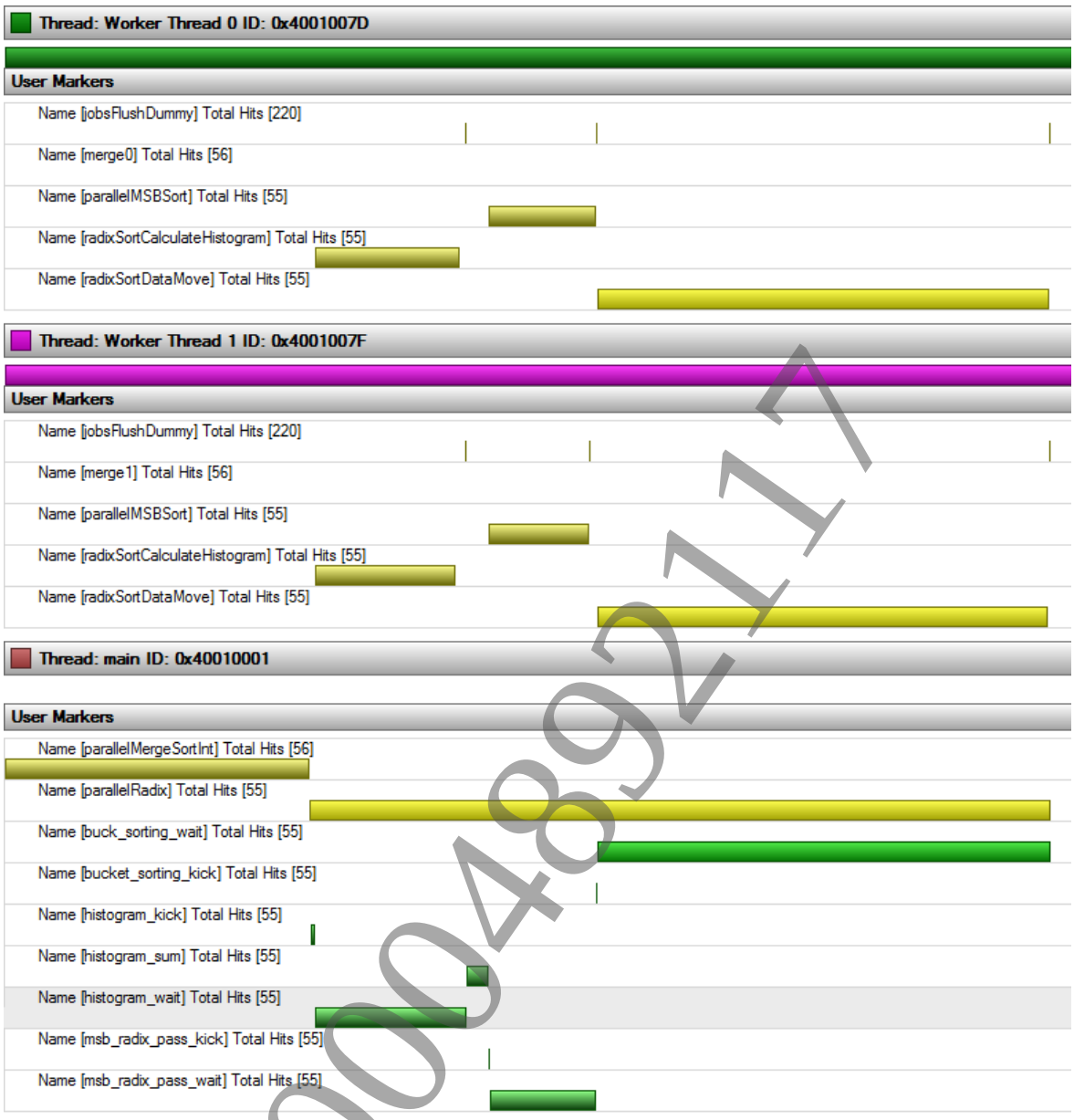


Parallel Radix Sort

Implementation

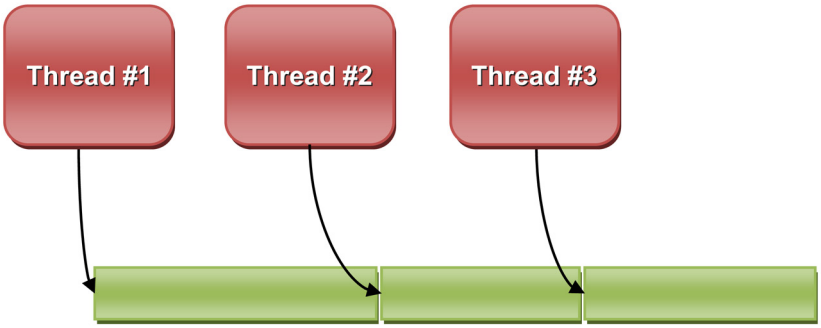
A traditional backward implementation of Radix Sort (that is, LSB first) does not provide much scope for parallelism. Each pass over the data reorders it with respect to the radix being considered, and each pass depends on the previous pass being fully completed. However, by sorting an array with its most significant byte first, we are essentially creating numerous *unsorted* buckets, each of which can be sorted completely independently of one another. This results in a significant amount of exploitable parallelism; see Figure 12.

Figure 12 A Razor Capture Showing Parallelism in Radix Sort



Each step in the algorithm is made parallel, beginning with the initial calculation of the histograms for the dataset. The dataset is split into three roughly equal chunks and each processing element calculates a histogram for that chunk of the dataset. When this step is completed, the histograms are then summed to produce a single histogram for the entire dataset. During this process, three offset tables are calculated, which allows the initial bucketing pass over the data to be carried out in parallel, as shown in Figure 13, with each processing element filling the same bucket.

Figure 13 Three Processing Elements Fill the Same Bucket, But Begin at Different Offsets



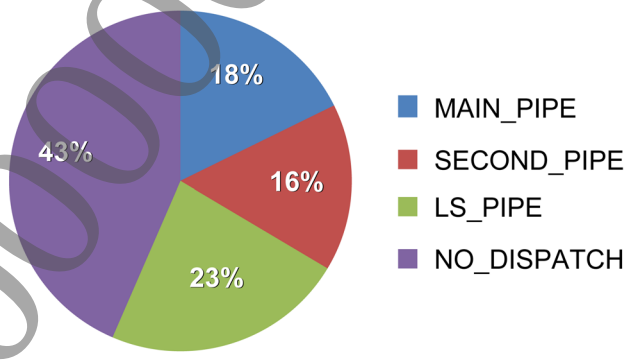
The data is now coarsely bucketed with each bucket containing elements with the same most significant 11 bits (meaning each bucket can be sorted independently as its final position in the array is somewhere within that bucket). An interesting side effect of independently sortable buckets is that the choice of sorting algorithm for those buckets can be arbitrary; we were not tied to any particular approach, however, we found the best approach to be to switch to a backward Radix Sort and carry out two additional passes over each bucket that contained more than a single element.

There are of course some problems that still need to be addressed with this approach, as inevitably the distribution of data into these independent buckets is non-uniform, especially when considering floating-point values, which even for most "real-world" values tend to congregate in relatively few buckets. To combat this, we used a simple, dynamic load balancing mechanism implemented using atomic operations instead of equally splitting up the sort jobs by index across the available processing elements.

Measurements

The performance characteristics of our implementation of parallel Radix Sort are similar to those of the serial version. The total runtime for the algorithm applied to the same dataset as previous experiments is in the region of 8 ms, by far the fastest sorting algorithm we were able to implement, although almost half of processor cycles were still failing to dispatch instructions from the instruction unit, as shown in Figure 14.

Figure 14 Instruction Pipelines Are Starved 50% of the Time



The reason for instruction pipeline starvation is attributable to stalls in the cache hierarchy as the processor waits for data to arrive from main memory. The different stages of the algorithm exhibit slightly different performance characteristics, but are all heavily limited by data cache performance.

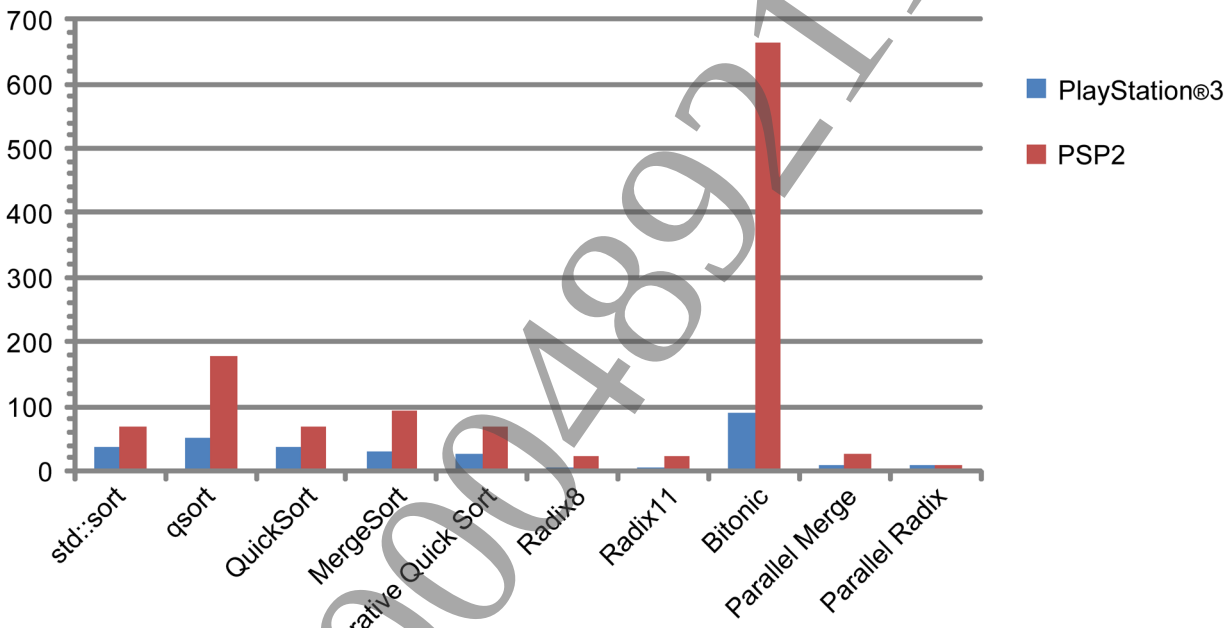
4 Performance Conclusions and Recommendations

Brief Comparison with PlayStation®3

A short time ago similar sorting experiments (for floating-point values only) were carried out on the PlayStation®3[1]. This section offers a brief comparison of two platforms' performance for similar sorting algorithms on similar datasets. For all tests an array of 65,536 32-bit floating-point values was used as the sole input to the sorting algorithms; some algorithms made use of an extra auxiliary memory buffer for temporary value storage.

It is important to bear in mind when comparing these results that the clock speed of the PlayStation®3 PPU is approximately ten times that of a single PSP2 core at the time of writing. Additionally, it is worth noting that for those sorts that exploit the parallel nature of modern hardware, the PlayStation®3 has significantly more computational horsepower due to the presence of six Synergistic Processing Elements. All measurements are given in milliseconds (ms).

Figure 15 Graph Showing the Measured Speeds of Similar Sorts on PSP2 and PlayStation®3



Recommendations for PSP2

This section provides SCE's recommendations for optimizing the performance of PSP2's CPU. These recommendations were informed by the experiments carried out during the course of authoring this document.

Target PSP2's Multicore Architecture

PSP2's CPU is the ARM Cortex A9. It has four cores, each with its own register file, L1 instruction and data caches, and instruction pipelines. Although one core is reserved for the operating system, three cores are available for applications to make use of. In order to get the most from PSP2 it is important to write code that takes advantage of this architecture and targets all three cores available to your application. A complete discussion of multithreaded programming is beyond the scope of this document; suffice it to say that good multithreaded code should have a minimal number of synchronization points and ideally not be sensitive to latency: the faster you need the results, the slower your system as a whole will be.

Data Layout

When you sort more complex data structures, using the traditional "Array of Structures" data layout means that the `ldm` and `stm` instructions can be used to move data at relatively low extra cost. However, the suitability of using these instructions is dependent on register pressure and structure size.

Branch Avoidance

In general the branch prediction unit in PSP2's CPU performs fairly well. However, there are cases which arise (not only in sorting) where algorithms will require branches to be taken based on some data that is intrinsicallydifficult for the CPU to predict. In these cases, the cost of stalls introduced through branch misprediction can quickly increase, causing your runtime performance to degrade significantly. If you notice that your algorithm is performing slower than you would expect, make sure that the CPU's instruction unit is not being throttled due to a large number of mispredicted branches. You can do this by monitoring both `SCE_PERF_ARM_PMON_ISSUE_EMPTY` and `SCE_PERF_ARM_PMON_BRANCH_MISPREDICT` through libPerf. If you find that you are indeed suffering from a large number of mispredicted branches it may be worth using the ARM Cortex A9's ability to conditionally issue instructions (which is currently directly targetable through assembly only) to eliminate some of these branches.

Be Careful When Using Cache Prefetching

The L1 cache prefetching mechanism on PSP2 is quite tricky to use in order to gain any significant performance advantage. It is all too easy to actually make your algorithm run considerably slower if you are not careful when using these instructions. We found the most successful strategy to be to issue a single `pld` instruction that prefetches data 1-2 cache lines ahead of the current access; of course, depending on the complexity of your code and access pattern, your experience might be different. We also found that using `pld` to accelerate one part of an algorithm may in some cases impact subsequent parts negatively, due to additional cache trashing in the first part, which evict data required during the second. It is important to take a holistic approach to profiling when attempting to use the `pld` instructions to optimize your code.

Read-Modify-Write Pattern

In many cases, we were able to improve the performance of our algorithms by avoiding a read-modify-store pattern. It can sometimes be tricky to ensure that the compiler generates the desired code when your goal is the separation of loads from stores. Be sure to verify the compiler output manually when attempting these kinds of optimizations.

A Note on Floating Point

During our experiments, both floating-point and integer data were used to measure the differences in performance between these two fundamental data types. Interestingly enough, in most cases we noticed that integer implementation substantially outperformed the floating-point version of an identical algorithm, when sorting identical data. Because the size of the data types are identical, and the input arrays for both runs of the algorithm are identical in both size and formulation, both versions of the algorithms produce similar memory access patterns. Consequently, the observed leap in performance is likely attributable to the ARM Cortex A9's out-of-order execution, which it employs to seemingly great effect on the integer instruction pipelines. Thus, in many cases using the integer instructions to perform sorting of floating-point values can give notable performance gains for some datasets. In the case of our Quick Sort implementation (described in chapter 2) we measured an approximate 15% gain in runtime performance.

Example 5 Generate a Signed Integer Value from a Floating-Point Number

```
static const int32_t floatFlip(const int32_t flt) {

    int32_t msbMask = flt>>31;
    uint32_t x = ((uint32_t)msbMask)>>1;
    uint32_t result = flt ^ x;
    return result;
}
```

In our experiments, we found that the cost of applying the transformation shown in Example 5 during the sorting phase generally caused the algorithms in question to perform *worse* than using standard floating-point comparisons – this was true in all but the worst performing sorting algorithms. (Of course in the case of Radix Sort a similar transformation is intrinsically necessary in order to sort floating-point values.) Therefore, you should measure the cost of applying the above transformation during your algorithms to see if simply targeting the floating-point instruction pipelines would result in better performance. Alternatively, if your dataset is amenable, it could be a worthwhile exercise to bias input values so that they are all in the positive domain, although this would be at the expense of some numerical robustness. For datasets for which the constituent values reside firmly in either the positive or negative domains to begin with, using integer comparisons could yield significant performance benefits and should be targeted. For more information about floating-point to integer conversion, refer to the appendix of this document.

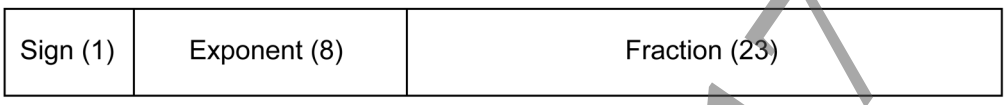
The Standard Library

Similar to many other platforms, the PSP2 implementation of the C++ standard library algorithm `std::sort` should be preferred to its C counterpart, `qsort`. Both algorithms can be outperformed by hand-rolling more sophisticated sorts and tailoring data design to the processing that will be performed on it.

Appendix: Converting IEEE754 Floating Point into Ordered Unsigned Integers

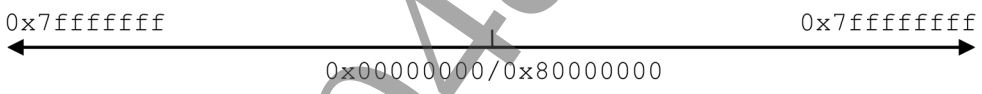
For floating-point numbers, the PSP2 CPU supports the format mandated by the IEEE754[5] standard. So simply interpreting a 32-bit value as an integer (unsigned or otherwise) would not yield correctly sorted values. To implement a Radix Sort capable of sorting floating-point values correctly, it is desirable to construct unsigned integers that can be used as indices into the arrays used for histograms and ultimately used as indices into a table of output array offsets. Thus we need to derive a transformation that will take an IEEE754 floating-point value and produce an unsigned integer. The key to this derivation is held in the format used for our floating-point numbers on PSP2. Figure 16 shows this format:

Figure 16 IEEE754 Floating-Point Representation



As with signed integers, the sign bit resides in the most significant bit of the 32 bit value. This means that if we were to simply interpret the floating-point value as an unsigned integer, negative values would be larger than their positive counterparts! To correct this, the first thing we must do is to flip the sign bit. This is much more correct, and indeed if we were to use signed comparisons this is all we need to do. However, there is one additional problem that must be solved for successful unsigned conversion. If you were to plot both positive and negative real numbers on a number line, the magnitude of the numbers would increase with distance from the origin, as in Figure 17. However, because the values we are considering are unsigned, this results in our negative values having a backward ordering. To correct this, we must flip all the remaining bits, but only for numbers that were originally negative. Example 6 shows a C implementation of this technique and is used heavily throughout our Radix Sort implementations.

Figure 17 Large Bit Patterns Tend Toward the Extremities of the Number Line



Example 6 Convert a Floating-Point Value into a Monotonic Unsigned Integer

```
/** Transform a floating-point value to a sortable unsigned integer.
    @param flt [in] Floating-point value as an unsigned integer.
    @retval A sortable unsigned integer.
 */
static inline uint32_t floatFlip(uint32_t flt) {
    int32_t msb    = (int32_t)(flt>>31);
    uint32_t msbMask = -msb;
    uint32_t maskSignFlip = msbMask | 0x80000000;
    return flt ^ maskSignFlip;
}
```