Chapter 6

Experimental Results

We will now apply our sorting algorithms in an experimental study comparing it to other sorting algorithms. The goal is to establish whether cache-oblivious sorting algorithms can compete with simpler sorting algorithms designed to be efficient in the RAM model and with algorithms tuned to be efficient in the use of caches.

We start with an overview of how the study is conducted and then present the results.

6.1 Methodology

6.1.1 The Sorting Problem

When comparing different algorithms that solve the same problem, we need to take care that they are treated equally. To do this we will clearly define the problem we wish to solve, and do it in a way, which does not favor any algorithm. The problem we will look at in the next to sections is as follows.

We are given the name of a file stored on a local disk on a native file system. The file contains a number of contiguous elements. No part of the file can be in memory beforehand. The problem is solved, when a there exists a file in the file system with the same elements but in non-decreasing order stored contiguously. We do not require the file to be physically stored on disk, nor do we require the elements to be in the original file.

The reason we use the file system and state explicitly that no part of the file may be in memory when the algorithm is started is that different algorithms access elements in different order. Mergesort typically access elements from the first to the last, while partition based sorting algorithms access elements from the both ends of the file. If we were to generate problem instances by writing elements to disk, in a streaming fashion, the last M elements of the file would likely be cached in memory. This means that partition-based algorithms that access the last elements early will have an advantage, because these elements can be accessed without causing memory transfers. In general, we do not believe such an advantage to be present in real life problems.

To ensure that no elements are in memory, when the sorting begins, we run a small program, named fillmem, after the input file has been generated. fillmem allocates an array exactly the size of main memory. It then writes values in all entries of the array, forcing the operating system to allocate pages for the array and evicting pages already in memory. Typically an operating system prefer to evict pages occupied by the file

system cache before evicting pages used by user processes [Tan01], so we expect this process to make sure no part of the input is in memory.

In the Datamation Benchmark, the input is also in a file on disk [DB03]. However, they require the output to be stored in a different file. If we were to require this from all algorithms, we would have to add an artificial copy phase to all algorithms that sort in-place, thus giving the merge-based sorting algorithms an artificial advantage. We expect most applications of sorting algorithms to want only the sorted elements and not to care how or where they are stored.

6.1.2 Competitors

There are several different classes of competitors to choose from, when comparing sorting algorithms. We wish to compare with algorithms known to be efficient due to low instruction count and with algorithms that are efficient due to efficient use of the memory hierarchy.

Cache Optimized Algorithms

LaMarca and Ladner implement sorting algorithms optimized for L1 or L2 cache [LL99]. Improving on their effort, Wickremesinghe *et al.* implements sorting algorithms, called R-merge and R-distribution, which utilizes registers and lower level caches better [ACV+00]. Kubricht *et al.* implements variants of the algorithms of LaMarca and Ladner that also takes the translation look-aside buffer and low associativity into account [XZK00].

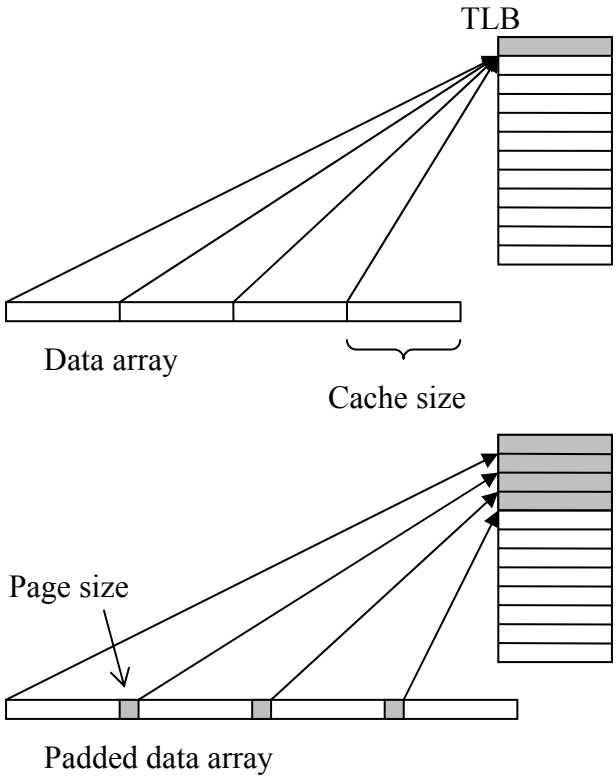

Figure 6-1. Multiway merging with and without padded inputs.

We would have liked to compare our algorithm to those of [ACV+00] and [XZK00]. However, the source code for R-merge and R-distribution is not publicly available and we are still waiting for a reply to the request to obtain it. The source code used in [XZK00] is publicly available. However, we did encounter problems using it.

The algorithms are based on LaMarca and Ladner's multi-mergesort and tiled mergesort. Multi-mergesort is essentially the multiway mergesort. It forms runs the size of the cache, sorts them while they are in cache, and merges all runs in a single pass using a heap. Tiled mergesort also use a run formation phase; however, it merges the runs formed using binary merging over several passes. The observation made in [XZK00] is that elements that are roughly one cache size apart are often mapped to the same TLB entry, since the TLB cover the same range of virtual address as the L2 cache (see Figure 6-1 above). This is in turn likely to cause a conflict miss on every element merged. For example, the Pentium 3 has 64 entries in its TLB, each covering 4KB for a total of 256KB, which is exactly the size of the L2 cache. The solution consists of introducing holes in the array containing the elements to be merged. These holes pad the runs formed during the run formation phase of the algorithms, so runs are separated by one page. If elements are read from each run at the same rate, this will insure that no conflict misses will occur.

In the implementation of padded tiled mergesort, the runs formed in each pass remain padded and the algorithm stops with the holes are still in the array. This means that this implementation does not solve the problem stated above. We do not consider an algorithm not generating an output of contiguous non-decreasing elements a valid sorting algorithm and thus exclude it from our benchmarks. The padded multi mergesort merges in a single pass and thus does not need to maintain the holes. However, we cannot confirm that it generates correctly sorted output and we believe the implantation to be buggy. The non-padded variants of tiled mergesort (`msort-c`) and multi mergesort (`msort-m`) seem to function fine, albeit only on input sizes of a power-of-two.

For our tests, we have changed their implementation slightly. We have changed the type of elements sorted from `long long` to a template parameter. Furthermore, the tiled mergesort did an explicit copy of the elements back to the original array, in case the array was not the output of the final merge pass. Since we do not require the elements to be in their original array, we have removed this final copy pass.

The algorithms are cache-aware and thus needs to know the size of the cache. For the Pentium builds, we specify a cache size of 256KB and for the MIPS build, we specify a cache size of 1024KB.

External Memory Sorting Algorithms

For algorithms designed for external memory, there is the LEDA-SM [CM99] and TPIE [TPI02].

The LEDA-SM is build on top of the commercially available LEDA library. We were able to get a copy of the LEDA library and the source code for the LEDA-SM is freely available. However, our copy of the LEDA library is designed for the GCC version 3 and LEDA-SM is written in a pre-standard dialect of C++ that GCC version 3 does not understand. We succeeded in translating most of the LEDA-SM into standard C++, only to realize that the namespaces used in LEDA made linking impossible. We

then got an older version of GCC and a matching version of the LEDA library, but when LEDA-SM still would not compile, we had to abandon the effort.

TPIE is also written in a pre-standard dialect of C++; however, unlike LEDA-SM, our old GCC compiler was able to build the TPIE library. TPIE includes a sorting algorithm, called ami_sort, that sorts a given input stream into a given output stream. These streams have to be managed by TPIE and cannot consist of files on disk. However, TPIE allows the streams to use explicitly named files, so we have created a program that generates TPIE streams as files on disk. These files can then be used in an actual sorting program as streams for the input of the sorting algorithm.

We only had the old version of GCC for Linux, so we will only be benchmarking AMI_sort on the Pentium computers. AMI_sort is also cache-aware and needs to know the amount of available RAM. The manual suggests specifying slightly less than the amount of physical memory, so we specified 192MB.

Sorting Algorithms for the RAM Model

The main competitor among classical sorting algorithms is the std::sort, available from the Standard Template Library accompanying any C++ compiler. The GCC compiler comes with the SGI implementation of the STL. This implementation uses the introsort by Musser for the std::sort function [Mus97]. Introsort is based on quicksort, but unlike earlier variants like [Sed78], introsort achieves a worst-case complexity of $\mathcal{O}(N\log N)$ and does so without sacrificing performance. This is achieved by detecting if the recursion becomes too deep, and if so switch to heapsort. Furthermore, the implementation uses insertion sort to handle problems of size less than 16, as suggested in [Sed78], however it is done at the bottom of the recursion, as suggested in [LL99], to preserve locality. For partitioning, it uses the fastest possible approach, not separating elements that are equal to the pivot element. We expect this implementation to very efficient and fully optimized.

Cache-oblivious Sorting Algorithms

For funnelsort, we use a variant of the funnel that is laid out according to the mixed van Emde Boas layout and uses the manually inlined four_merger as basic merger ($z = 4$). For computing buffer sizes, we use $\alpha = 16$ and $d = 2$. According to the study conducted in the previous chapter, these choices constitute a high performing funnel on all platforms. The LOWSCOSA uses the same funnel.

We have implemented a special output stream that uses the write system call (WriteFile in Windows) to generate the output directly on disk. It is implemented as a class containing a buffer with a capacity of 4096 elements. When the elements are written to this stream by funnelsort, the stream puts them in the buffer. When the stream iterator is incremented beyond the capacity of the buffer, it is written to disk and the iterator set to the beginning of the buffer. Using this stream, we avoid having to hold the entire output array in the address space, thus allowing to potentially sort up to 2GB of data using funnelsort. The variant using this stream is called ffunnelsort, whereas the one writing to an array in memory is called funnelsort.

6.1.3 Benchmark Procedure

Except the ami_sort, none of the algorithms is designed to work with files on disk. Indeed, the algorithms of [XZK00] expect the elements to be in an array in memory. To

solve the problem as stated, we use the memory mapping functionality of the operating systems. Memory mapping works like ordinary paged memory except a specified file is used as backing store, not the swap file.

A program, sortgen, that generates inputs of a given type of elements, a given distribution, and a given size has been build. In addition, for each of the sorting algorithms, a separate executable has been build that takes the name of the input file as an argument and the name of an optional output file. All but the TPIE executable then maps these files into memory using memory mapping.

The procedure for benchmarking is then as follows. For each algorithm we the use sortgen (or the program using TPIE streams in case of ami_sort) to generate an input file. We then use fillmem to force the operating system to flush its file cache and run the executable containing the sorting algorithm.

A list of all algorithms used in the benchmarks can be found in Table 6-1.

Algorithm	Source	File access	Cache-aware	I/O optimal
ami_sort	[TPI02]	read/write	Yes	RAM-Disk
msort-c	[XZK00]	Memory map	Yes	No
msort-m	[XZK00]	Memory map	Yes	L2 cache-RAM
std::sort	SGI/GCC	Memory map	No	No
funnelsort	This thesis	Memory map	No	Yes
ffunnelsort	This thesis	Memory map/write	No	Yes
lowscosa	This thesis	Memory map	No	Yes

Table 6-1. Sorting algorithms used in comparative benchmarking.

6.2 Straight Sorting

For the first collection of benchmarks, we focus on uniformly distributed data and study the performance on different data types. We will use the three data types described in Section 5.1.3. A sample of 1,024 key values can be seen in Figure 6-2.

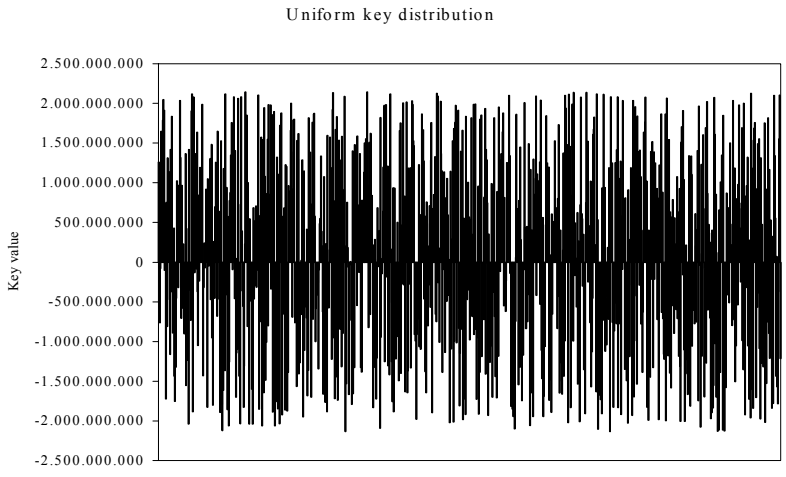

Figure 6-2. Uniform key distribution.

6.2.1 Key/Pointer pairs

The results of measuring wall clock time when sorting pairs are as follows:

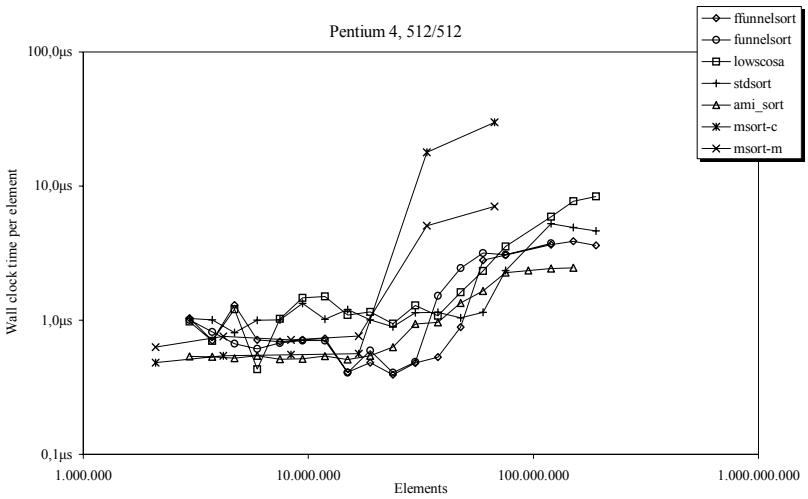

Chart C-61. Wall clock time sorting uniformly distributed pairs on Pentium 4.

The internal memory sorting algorithm used in TPIE is the fastest of the algorithms when the datasets fit in RAM. As a close second comes the cache tuned tiled mergesort. Relative to the tiled mergesort, the multi mergesort performs slightly worse. The reason for this is likely the overhead of managing the heap. While the cache-oblivious sorting algorithms cannot keep up with the memory tuned variants within RAM, do they outperform std::sort and perform on par with multi mergesort. As expected, the LOWSCOSA performs rather poorly.

The picture changes when the dataset takes up half the memory. This is when the merge-based sorts begin to cause page faults, because their output cannot also fit in RAM. The tiled mergesort suddenly performs a factor 30 slower, due to the many passes it makes over the data. We see that both funnelsort and TPIE begins to take longer time, however the LOWSCOSA and ffunnelsort does not loose momentum until the input cannot fit in RAM. Writing the output directly to disk instead of storing it, really helps in this region. The LRU replacement strategy of the operating system does not know that the input is more important to keep in memory than the output, so it will start evicting pages from the input to keep pages from the output in memory. When writing the output directly to disk, the output takes up virtually no space so the input need not be paged out.

When the input does not fit in memory, TPIE again has the superior sorting algorithm. This is indeed what it was designed for. It is interesting to see that it is so much faster than funnelsort, even though funnelsort incurs an optimal number of page faults. One explanation for this could be that TPIE uses double-buffering and overlaps the sorting of one part of the data set with the reading or writing of another, thus essentially sorting for free. Another explanation could be that is reads in many more blocks at a time. During the merge phase, usually no more than 8 or 16 streams are merged. Instead of reading in one block from each stream, utilizing only $16B$ of the

memory, a cache aware sorting algorithm could read in a much larger part of the stream, up to $M/16$ elements at a time. Both funnelsort and ffunnelsort outperform std::sort when the input cannot fit in RAM. This must be attributed to the optimal number of page faults incurred by the funnelsorts. Even though std::sort is, in some sense, close to being optimal, it is clear that it is not. The LOWSCOSA, unlike the funnelsorts and TPIE, does not seem to reach a plateau. This is because it is it keeps incurring a significant number of page faults due to it only sorting half the dataset per pass.

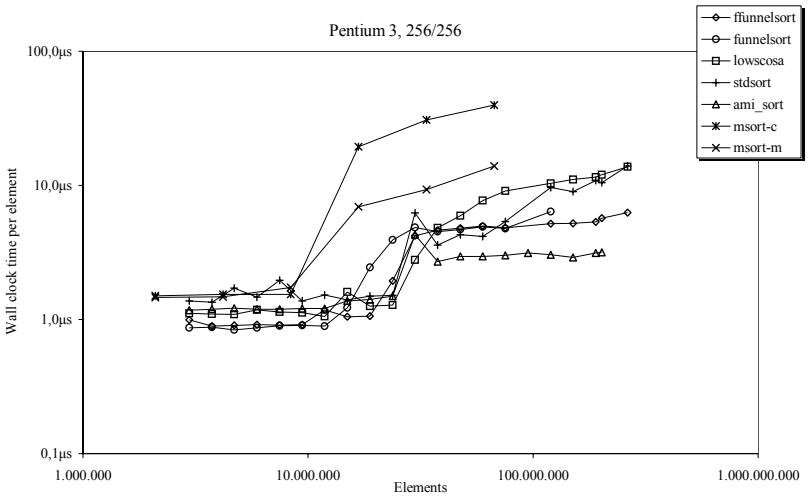

Chart C-62. Wall clock time sorting uniformly distributed pairs on Pentium 3.

On the Pentium 3, things are turned around a bit. Here, the funnelsorts are the fastest sorting algorithms when dataset fits in RAM. They outperform both the cache tuned algorithms and std::sort. Even the LOWSCOSA can compete with TPIE. Beyond RAM, we again see TPIE as the fastest sorter. As the dataset becomes much larger than RAM, we can see that funnelsort holds its performance level, while std::sort becomes slower and slower per element sorted. We can also see that the running time per element of the LOWSCOSA almost becomes constant, and that it will eventually outperform std::sort.

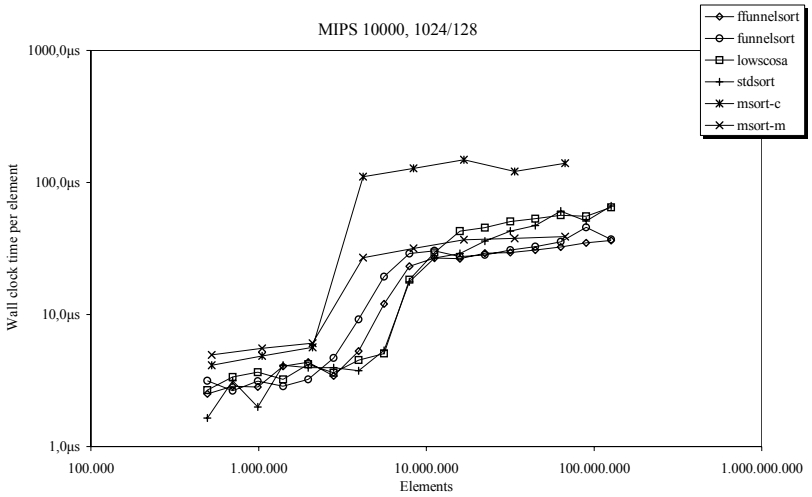

Chart C-63. Wall clock time sorting uniformly distributed pairs on MIPS 10000.

On the MIPS, the picture is not that clear, when looking at the time for sorting datasets that fit in RAM. However, we can see that the cache-tuned algorithms perform rather poorly. This is likely to be because of the many TLB misses they incur. The MIPS uses software TLB miss handlers, so the cost of a TLB miss is greater here than on the Pentiums. We see the same trend of the performance of std::sort, ffunnelsort and the LOWSCOSA not degrading until the input cannot fit in RAM. Then, we see funnelsort as the fastest sorting algorithm and the performance of std::sort continuing to degrade. As on the Pentium 3, the LOWSCOSA settles in with a somewhat higher running time than the funnelsorts but is eventually faster than std::sort.

Let us see, if we can locate the cause of these performance characteristics in the number of page faults incurred by the algorithms.

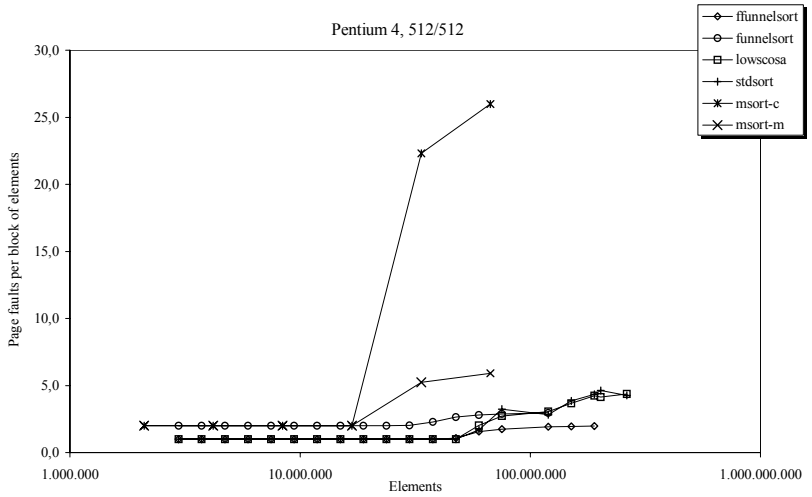

Chart C-64. Page faults sorting uniformly distributed pairs on Pentium 4.

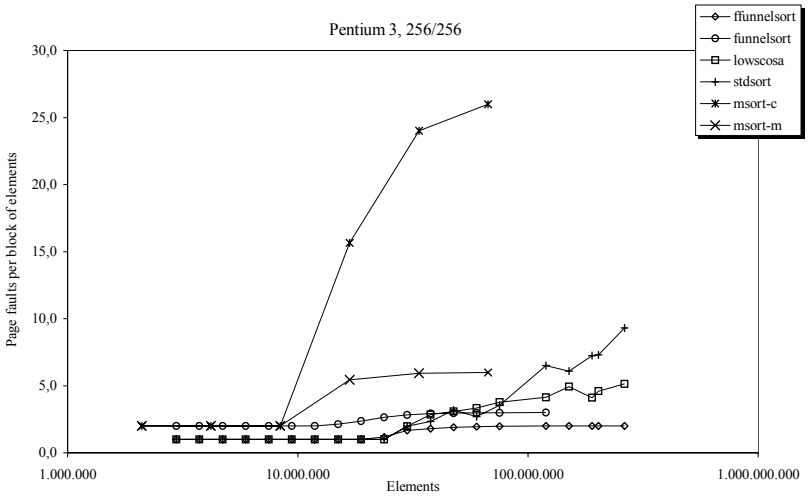

Chart C-65. Page faults sorting uniformly distributed pairs on Pentium 3.

The picture is identical on the two Pentium architectures. We can see that with very high accuracy, all in-place algorithms and the ffunnelsort incur N/B page faults and all merge-based algorithms incur $2N/B$ page faults. When input cannot fit in RAM, we also see that funnelsort incurs exactly $3N/B$ page faults and the ffunnelsort incurs $2N/B$. This is exactly as expected, since the cost of writing of the sorted result is not included in this measure.

The LOWSCOSA settles in at about 4-5 complete scans, while std::sort again continues to incur an increasing number of page.

We see that the tiled mergesort incurs a lot more page faults than any of the other algorithm does. Again, this is due to the many passes over the input. Multi mergesort incurs up to $6N/B$ page faults. It forms runs by scanning the entire input and generating the runs in an equal sized array. These runs are then scanned and the output written back in the original array. This accounts for only $4N/B$ page faults. We cannot account for the remaining $2N/B$. We suspect it is an unneeded pass, like the copying of all elements from the output to the input, in the middle of the algorithm.

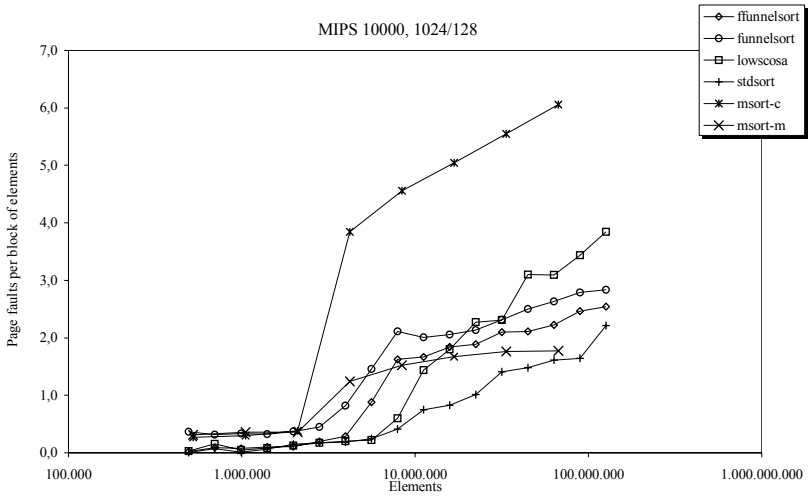

Chart C-66. Page faults sorting uniformly distributed pairs on MIPS 10000.

The numbers are a lot different on the MIPS computer. Here we are unable to explain the page fault count by the number of passes over the data, the algorithms make.

IRIX supports using several different page sizes. The getpagesize system call reveals that the page size of this particular system is 8KB. The values in the chart are based on this value. However, getpagesize may return a lower number to indicate that the allocation granularity is 8KB and not necessarily the actual page size used by the operating system, which may then be up to 64KB.

Another explanation is that this is the effect of the operating system prefetching pages. That is, if it can detect that a process is streaming through data, it may read in 8 or 16 pages per page fault. According the manual pages, the number of page faults reported by the getrusage system call is the number of memory operations that has caused an I/O. If indeed the operating system chooses to prefetch pages, this number will be significantly lower. An indication that this may be the case is that as the datasets get very large, the number of page faults caused by funnelsort and ffunnelsort comes closer to the expected values. This may then be because funnelsort is accessing so many streams at once that the operating system is unable to detect a streaming behavior and thus opts to not prefetch pages.

The next chart shows the number of cache misses incurred on the MIPS computer.

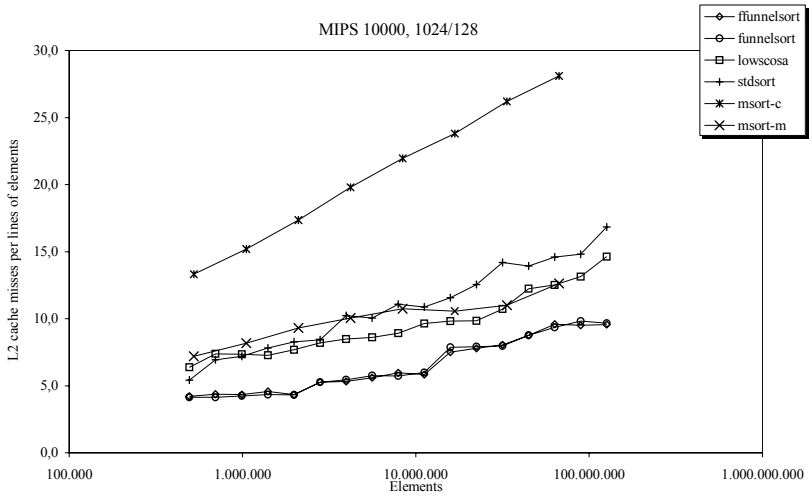

Chart C-67. Cache misses sorting uniformly distributed pairs on MIPS 10000.

This is indeed an interesting result. It clearly shows that funnelsort is able to maintain a very high degree of cache utilization, even on lower level caches, where the assumptions of the ideal cache model, such as full associativity and optimal replacement, most certainly does not hold. Even the LOWSCOSA incurs fewer cache misses than the other algorithms.

It is interesting to see that even for such small datasets as less than one million pairs, the high number of passes done by tiled mergesort causes a significant number of cache misses. It is also interesting to see that multi mergesort is not able to keep up with funnelsort. This is most likely due to a high number of conflict misses.

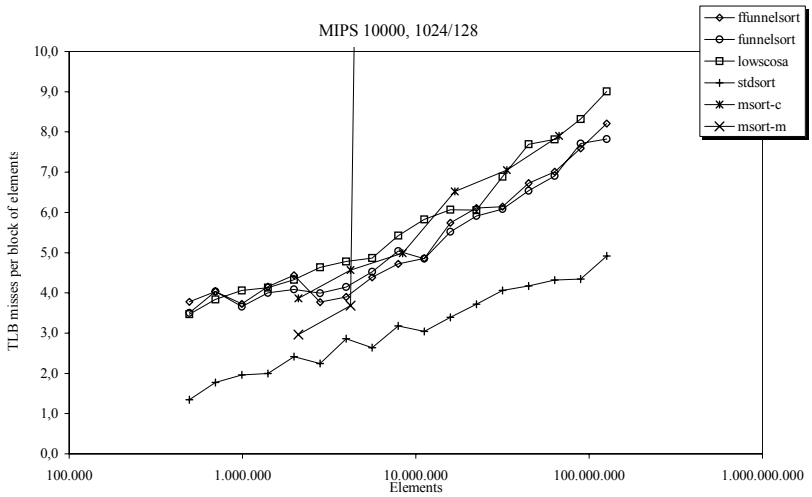

Chart C-68. TLB misses sorting uniformly distributed pairs on MIPS 10000.

The std::sort incurs the fewest TLB misses. The reason funnelsort is not as dominating on the TLB level of the hierarchy is likely because the TLB is not as tall as

the L2 cache or RAM. With only 64 TLB entries, the output of at most a 64-funnel can be merged with B^{-1} TLB misses per element, before another sub-funnel is loaded. This is likely significantly less than what is possible on the other levels of the hierarchy, where the order of the j-funnel is likely bounded by the capacity of the cache, rather than the number of blocks it contains.

The multi mergesort incurs more that 100 times more TLB misses and goes off the scale, when it sorts 2^{22} or more elements. 2^{22} elements correspond exactly to 64 cache-loads, which again corresponds to one cache-load per TLB entry. Merging more streams than there are TLB entries will cause thrashing.

It is interesting to note that neither the L2 cache misses nor the TLB misses is reflected significantly in the wall clock time, except perhaps for the extreme cases of the multi and tiled mergesorts.

6.2.2 Integer Keys

For the remainder of this chapter, we will look for changes in performance characteristics when the algorithms are applied to different data types. Unfortunately, due to time constraints not all algorithms and architectures could participate in the remaining the benchmarks. Let us first look at the performance when sorting only integers.

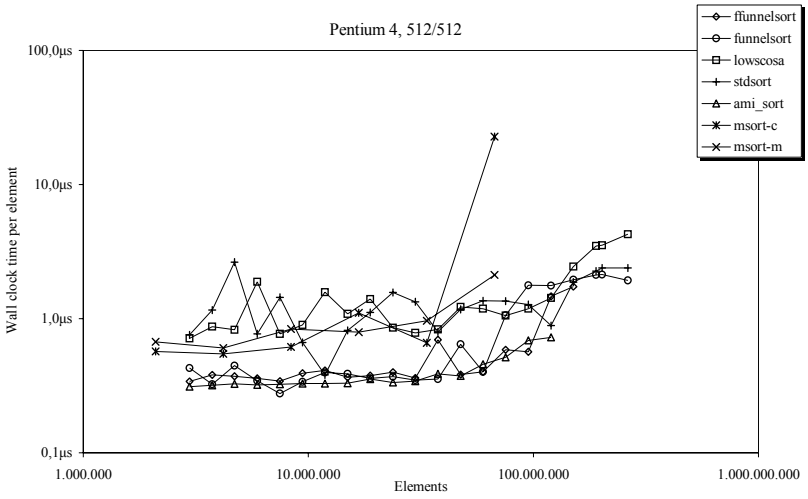

Chart C-69. Wall clock time sorting uniformly distributed integers on Pentium 4.

On the Pentium 4, the ami_sort still dominates, albeit not by as much as when sorting pairs. We see some large fluctuations in the input-sensitive algorithms based on partitioning. This could be due to unlucky pivot selection; however, it may also simply be caused by external influences. Funnelsort is very competitive and std::sort fluctuates a lot. Other than that there is not much new.

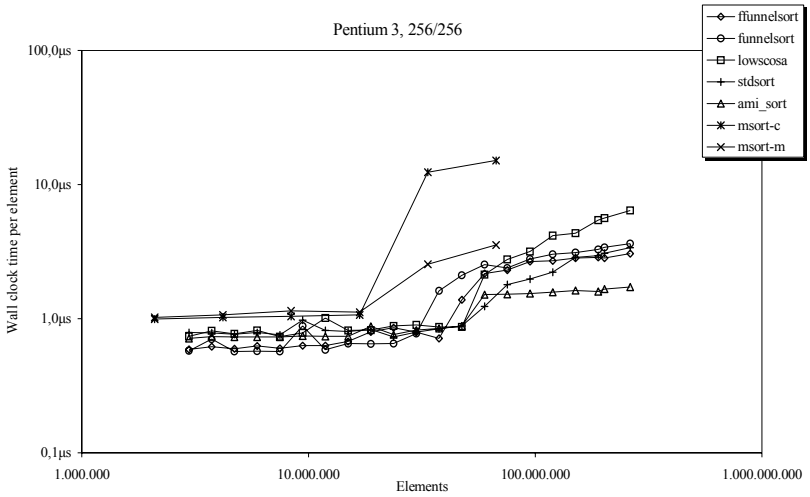

Chart C-70. Wall clock time sorting uniformly distributed integers on Pentium 3.

On the Pentium 3, the picture is largely unchanged from when sorting pairs.

6.2.3 Records

In this section, we sort uniformly distributed records of size 100 bytes each. Using such large elements reduces the number of elements within a given part of memory, thus fewer comparisons are done to sort the elements within the same amount of space. This should in turn downplay the cost of a comparison compared to cache effects.

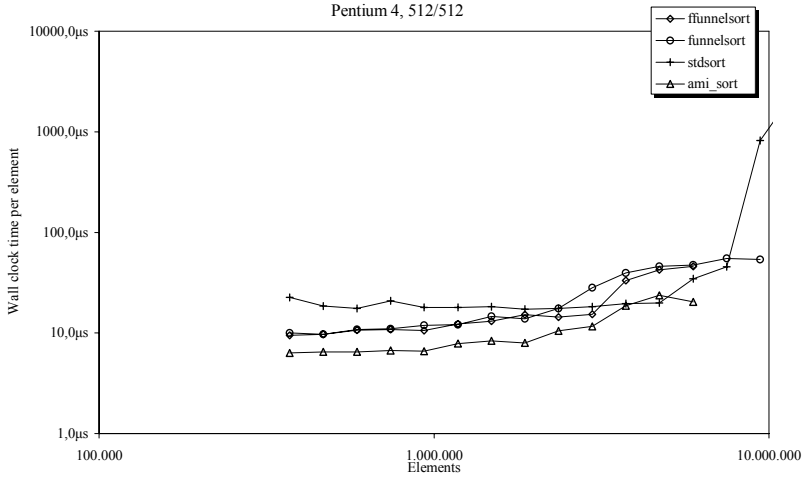

Chart C-73. Wall clock time sorting uniformly distributed records on Pentium 4.

Again, we see ami_sort dominate on the Pentium 4. Close second and third are the funnelsorts and slowest is std::sort, exactly as when sorting pairs. This leads us to believe that cache effects rather than just comparisons are also very important when sorting small elements.

std::sort exhibits a dramatic jump in running time when input can no longer fit in RAM. We can see from the page fault count that it is accompanied by an increase in total incurred page faults:

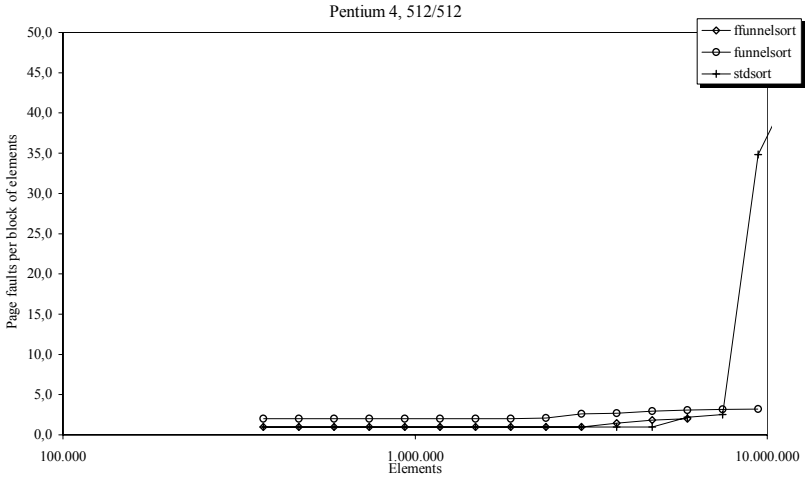

Chart C-75. Page faults sorting uniformly distributed records on Pentium 4.

The reason for this jump is not entirely clear.

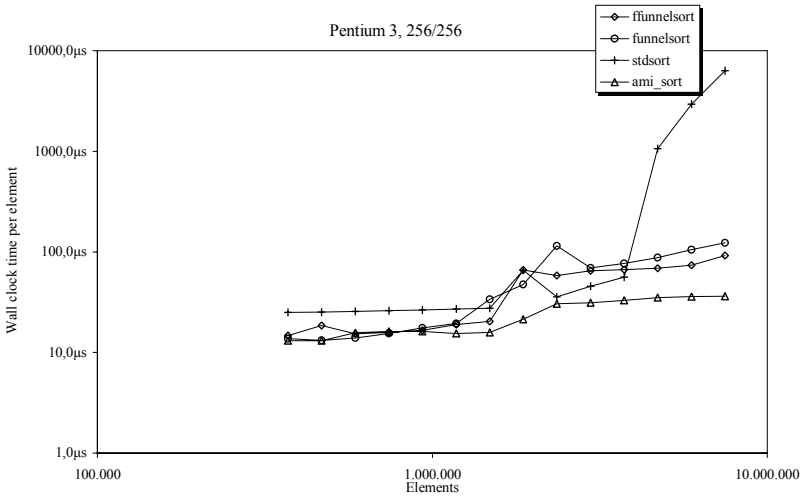

Chart C-74. Wall clock time sorting uniformly distributed records on Pentium 3.

On the Pentium 3, the ami_sort now performs on par with the funnelsorts. std::sort is significantly slower and exhibits the same increase in running time as seen on the Pentium 4.

6.3 Special Cases

In this section, we investigate the performance of the algorithms when sorting special distributions of elements. std::sort and the LOWSCOSA are both input sensitive and may thus react differently to different distributions. Even though merge-based sorting algorithms (as implemented here) are not considered input-insensitive, certain distributions may make branches in the code more predictable, and thus influence merge-based sorting algorithms as well.

6.3.1 Almost Sorted

The first distribution we will look at mimics an almost sorted dataset that needs to be completely sorted. A sample distribution of 1024 elements can be seen in Figure 6-3, where there is six buckets of elements in a given range, such that all elements in one bucket is smaller than any element in the next. In general, there will be $\ln(n)$ buckets in a distribution of n elements.

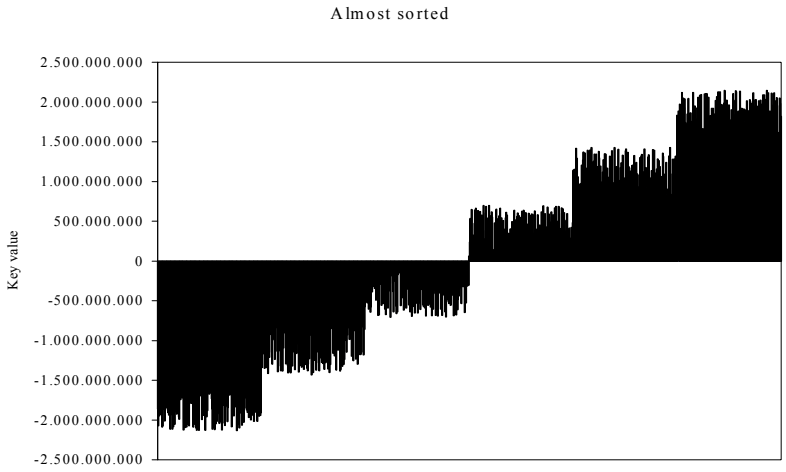

Figure 6-3. Almost sorted key distribution.

The results are as follows:

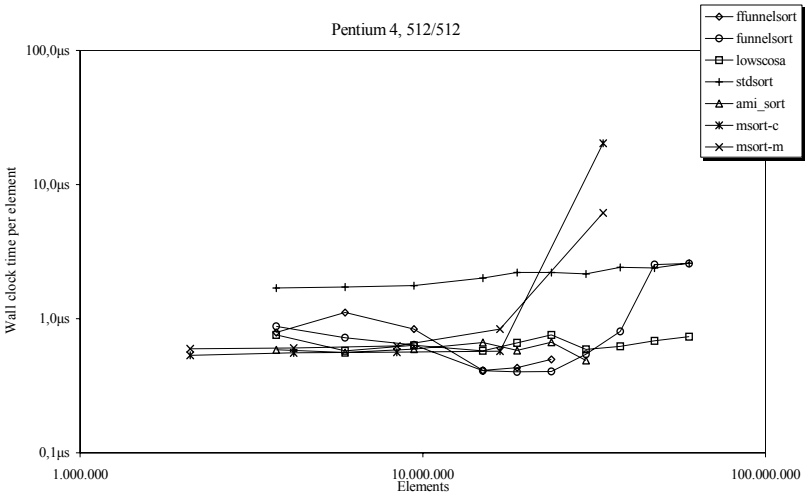

Chart C-77. Wall clock time sorting almost sorted pairs on Pentium 4.

Comparing to Chart C-61, we can see that std::sort performs slightly worse on almost sorted data. This is indeed unexpected and cannot immediately be explained. We would expect a partitioning operation on almost sorted data not to swap many elements, in turn not to cause the referenced bit to be set and thus save many expensive writes to disk. Other than that, no significant changes can be seen.

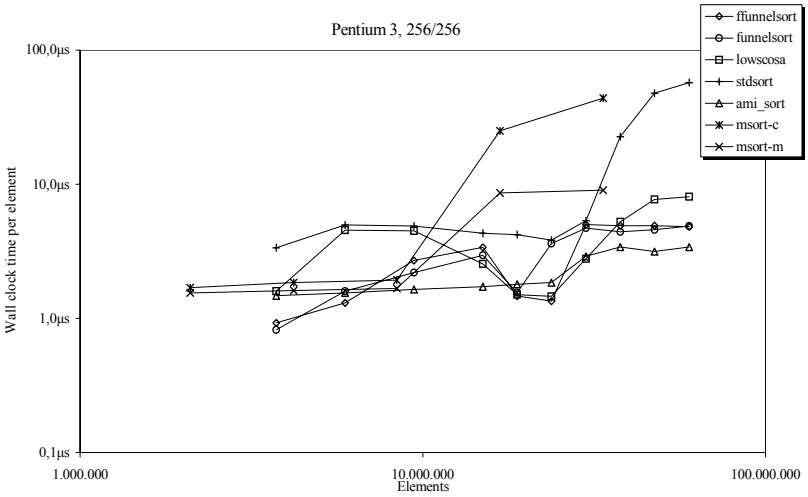

Chart C-78. Wall clock time sorting almost sorted pairs on Pentium 3.

Comparing to Chart C-62, we can again see an increase in the running time of std::sort. We can also see that the funnelsorts increase their per element running time as the size of the dataset increases up to about 15,000,000 elements.

6.3.2 Few Distinct Elements

The other distribution we will look at contains many elements but only few distinct ones. A sample distribution of 1,024 elements with six distinct keys can be seen in Figure 6-4. In general, there will be $\ln(n)$ distinct keys in a distribution of n elements.

6.3.2 Few Distinct Elements

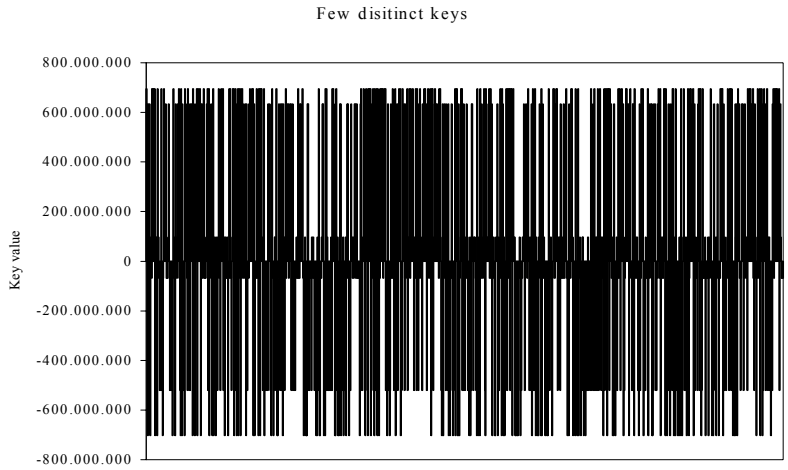

Figure 6-4. Few distinct keys distribution.

The results are as follows:

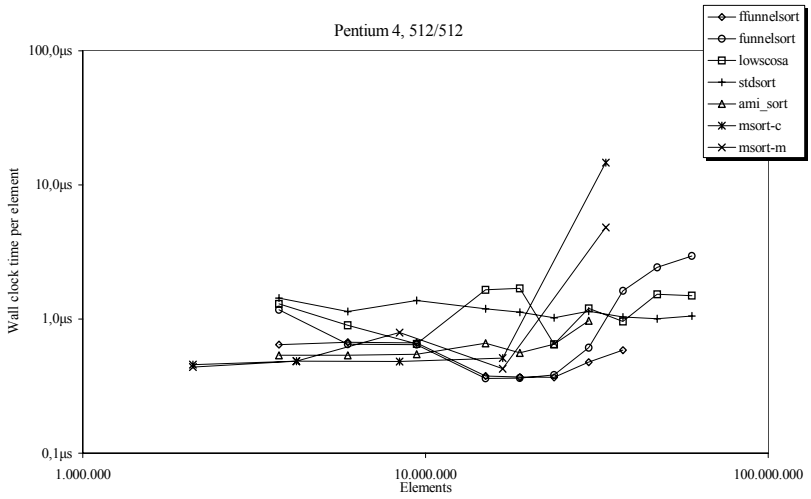

Chart C-81. Wall clock time sorting few distinct pairs on Pentium 4.

This result looks like the result for uniform distribution, with the running time per element of the funnelsorts now being more constant.

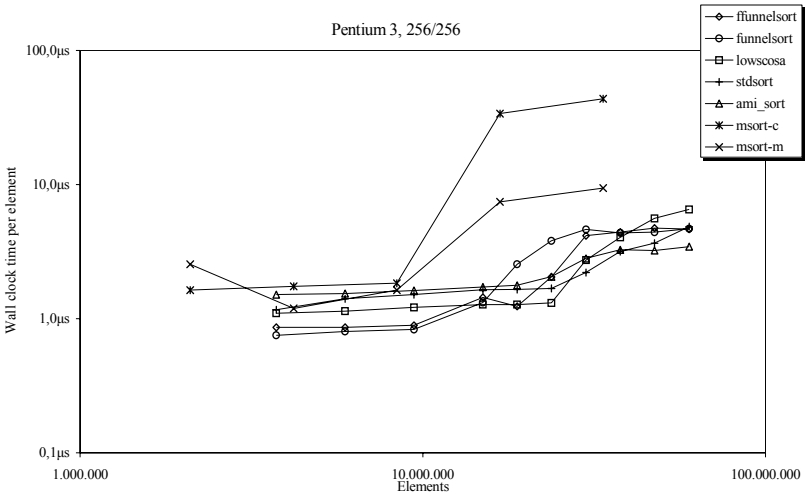

Chart C-82. Wall clock time sorting few distinct pairs on Pentium 3.

As does this to a high degree. If anything, the funnelsorts appear to perform slightly better, relative to std::sort, than when sorting uniformly distributed elements. Had std::sort been using a Dutch flag partitioning, it would have probably been faster.