The Influence of Caches on the Performance of Sorting *

Anthony LaMarca
Xerox PARC
3333 Coyote Hill Road
Palo Alto CA 94304

Richard E. Ladner
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

lamarca@parc.xerox.com   ladner@cs.washington.edu

Abstract

We investigate the effect that caches have on the performance of sorting algorithms both experimentally and analytically. To address the performance problems that high cache miss penalties introduce we restructure mergesort, quicksort, and heapsort in order to improve their cache locality. For all three algorithms the improvement in cache performance leads to a reduction in total execution time. We also investigate the performance of radix sort. Despite the extremely low instruction count incurred by this linear time sorting algorithm, its relatively poor cache performance results in worse overall performance than the efficient comparison based sorting algorithms. For each algorithm we provide an analysis that closely predicts the number of cache misses incurred by the algorithm.

1 Introduction

Since the introduction of caches, main memory has continued to grow slower relative to processor cycle times. The time to service a cache miss to memory has grown from 6 cycles for the Vax 11/780 to 120 for the AlphaServer 8400 [3, 6]. Cache miss penalties have grown to the point where good overall performance cannot be achieved without good cache performance. As a consequence of this change in computer architectures, algorithms that have been designed to minimize instruction count may not achieve the performance of algorithms that take into account both instruction count and cache performance.

One of the most common tasks computers perform is sorting a set of unordered keys. Sorting is a fundamental task and hundreds of sorting algorithms have been developed. In this paper we explore the potential performance gains that cache-conscious design offers in understanding and improving the performance of four popular sorting algorithms: mergesort[1], quicksort [11], heapsort [24], and

*A preliminary version of this paper appears in *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 370-379, 1997.*

[1]Knuth [15] traces mergesort back to card sorting machines of the 1930s.

radix sort[2]. Mergesort, quicksort, and heapsort are all comparison based sorting algorithms while radix sort is not.

For each of the four sorting algorithms we choose an implementation variant with potential for good overall performance and then heavily optimize this variant using traditional techniques to minimize the number of instructions executed. These heavily optimized algorithms form the baseline for comparison. For each of the comparison sort baseline algorithms we develop and apply memory optimizations in order to improve cache performance and, hopefully, overall performance. For radix sort we optimize cache performance by varying the radix.

In the process we develop some simple analytic techniques that enable us to predict the memory performance of these algorithms in terms of cache misses. Cache misses cannot be analyzed precisely due to a number of factors such as variations in process scheduling and the operating system's virtual to physical page-mapping policy. In addition, the memory behavior of an algorithm may be too complex to analyze completely. For these reasons the analyses we present are only approximate and must be validated empirically.

For comparison purposes we focus on sorting an array (4,000 to 4,096,000 keys) of 64 bit integers chosen uniformly at random. Our main study uses trace-driven simulations and actual executions to measure the impact that our memory optimizations have on performance. We concentrate on three performance measures: instruction count, cache misses, and overall performance (time) on machines with modern memory systems. Our results can be summarized as follows:

1. For the three comparison based sorting algorithms, memory optimizations improve both cache and overall performance. The improvements in overall performance for heapsort and mergesort are significant, while the improvement for quicksort is modest. Interestingly, memory optimizations to heapsort also reduce its instruction count. For radix sort the radix that minimizes cache misses also minimizes instruction count.

2. For large arrays, radix sort has the lowest instruction count, but because of its relatively poor cache performance, its overall performance is worse than the memory optimized versions of mergesort and quicksort.

3. Although our study was done on one machine, we demonstrate the robustness of the results by showing that comparable speedups due to improved cache performance can be achieved on several other machines.

4. There are effective analytic approaches to predicting the number of cache misses these sorting algorithms incur. In many cases the analysis is not difficult, yet highly predictive of actual performance.

The main general lesson to be learned from this study is that because cache miss penalties are large, and growing larger with each new generation of processor, selecting the fastest algorithm to solve a problem entails understanding cache performance. Improving an algorithm's overall performance may require increasing the number of instructions executed while, at the same time, reducing the number of cache misses. Consequently, cache-conscious design of algorithms is required to achieve the best performance.

[2] Knuth [15] traces the radix sorting method to the Hollerith sorting machine that was first used to assist the 1890 United States census.

2 Caches

In order to speed up memory accesses, small high speed memories called *caches* are placed between the processor and the main memory. Accessing the cache is typically much faster than accessing main memory. Unfortunately, since caches are smaller than main memory they can hold only a subset of its contents. Memory accesses first consult the cache to see if it contains the desired data. If the data is found in the cache, the main memory need not be consulted and the access is considered to be a cache *hit*. If the data is not in the cache it is considered a *miss*, and the data must be loaded from main memory. On a miss, the block containing the accessed data is loaded into the cache in the hope that data in the same block will be accessed again in the future. The *hit ratio* is a measure of cache performance and is the total number of hits divided by the total number of accesses.

The major design parameters of caches are:

- *Capacity*, which is the total number of bytes that the cache can hold.
- *Block size*, which is the number of bytes that are loaded from and written to memory at a time.
- *Associativity*, which indicates the number of different locations in the cache where a particular block can be loaded. In an *N-way set-associative* cache, a particular block can be loaded in N different cache locations. *Direct-mapped* caches have an associativity of one, and can load a particular block only in a single location. *Fully associative* caches are at the other extreme and can load blocks anywhere in the cache.
- *Replacement policy*, which indicates the policy of which block to remove from the cache when a new block is loaded. For the direct-mapped cache the replacement policy is simply to remove the block currently residing in the cache.

In most modern machines, more than one cache is placed between the processor and main memory. These hierarchies of caches are configured with the smallest, fastest cache next to the processor and the largest, slowest cache next to main memory. The largest miss penalty is typically incurred with the cache closest to main memory and this cache is usually direct-mapped. Consequently, our design and analysis techniques will focus on improving the performance of direct-mapped caches. We will assume that the cache parameters, block size and capacity, are known to the programmer.

High cache hit ratios depend on a program's stream of memory references exhibiting locality. A program exhibits *temporal locality* if there is a good chance that an accessed data item will be accessed again in the near future. A program exhibits *spatial locality* if there is good chance that subsequently accessed data items are located near each other in memory. Most programs tend to exhibit both kinds of locality and typical hit ratios are greater than 90% [18]. With a 90% hit ratio, cutting the number of cache misses in half has the effect of raising hit ratio to 95%. This may not seem like a big improvement, but with miss penalties on the order of 100 cycles, normal programs will exhibit speed-ups approaching 2:1 in execution time. Accordingly, our design techniques will attempt to improve both the temporal and spatial locality of the sorting algorithms.

Cache misses are often categorized into *compulsory*, *capacity*, and *conflict* misses [10]. Compulsory misses are those that occur when a block is first accessed and brought into the cache. Capacity misses are those caused by the fact that more blocks are accessed than can fit all at one time in the cache. Conflict misses are those that occur because two or more blocks that map to the same location in the cache are accessed. In this paper we address techniques to reduce the number of both capacity and conflict misses for sorting algorithms.

3 Design and Evaluation Methodology

Cache locality is a good thing. When spatial and temporal locality can be improved at no cost it should always be done. In this paper, however, we develop techniques for improving locality even when it results in an increase in the total number of executed instructions. This represents a significant departure from traditional design and optimization methodology. We take this approach in order to show how large an impact cache performance can have on overall performance. Interestingly, many of the design techniques are not particularly new. Some have already been used in optimizing compilers, in algorithms which use external storage devices, and in parallel algorithms. Similar techniques have also been used successfully in the development of the cache-efficient Alphasort algorithm [19].

As mentioned earlier we focus on three measures of performance: instruction count, cache misses, and overall performance in terms of execution time. All of the dynamic instruction counts and cache simulation results were measured using Atom [22]. Atom is a toolkit developed by DEC for instrumenting program executables on Alpha workstations. Dynamic instruction counts are obtained by inserting an increment to a counter after each instruction executed by the algorithm. Cache performance is determined by inserting calls after every load and store to maintain the state of a simulated cache and to keep track of hit and miss statistics. In all cases we configure the simulated cache's block size and capacity to be the same as the second level cache of the architecture used to measure execution time. Execution times are measured on a DEC Alphastation 250, and execution times in our study represent the median of 15 trials. The machine used in our study has 32 byte cache blocks with a direct-mapped second level cache of $2,097,152 = 2^{21}$ bytes. In our study, the set to be sorted is varied in size from 4,000 to 4,096,000 keys of 8 bytes each.

Finally, we provide analytic methods to predict cache performance in terms of cache misses. In some cases the analysis is quite simple. For example, traditional mergesort has a fairly oblivious pattern of access to memory, thereby making its analysis quite straightforward. However, the memory access patterns of the other algorithms, such as heapsort, are less oblivious requiring more sophisticated techniques and approximations to accomplish the analysis. It should be noted that due to unpredictable factors such as operating system scheduling, our analyses, while often extremely accurate, are all approximations.

In all our analyses we consistently use n to indicate the number of keys to be sorted, C to indicate the number of blocks in the cache and B to indicate the number of keys that fit in a cache block. When we evaluate our analyses on the DEC Alphastation 250 sorting 8 byte keys, $C = 2^{16}$ and $B = 4$.

4 Mergesort

Two sorted lists can be merged into a single sorted list by traversing the two lists at the same time in sorted order, repeatedly adding the smaller key to the single sorted list. By treating a set of unordered keys as a set of sorted lists of length one, the keys can be repeatedly merged together until a single sorted set of keys remains. Algorithms which sort in this manner are known as mergesort algorithms, and there are both recursive and iterative variants [12, 15].

4.1 Base Algorithm

For a base algorithm, we chose an iterative mergesort [15] since it is both easy to implement and is very amenable to traditional optimization techniques. The standard iterative mergesort makes

$\lceil log_2 n \rceil$ passes over the array, where the i-th pass merges sorted subarrays of length 2^{i-1} into sorted subarrays of length 2^i. The optimizations applied to the base mergesort include: alternating the merging process from one array to another to avoid unnecessary copying, making sure that subarrays to be merged are in opposite order to avoid unnecessary checking for end conditions, sorting subarrays of size 4 with a fast in-line sorting method, and loop unrolling. Thus, the number of merge passes is $\lceil \log_2(n/4) \rceil$. If $\lceil \log_2(n/4) \rceil$ is odd then an additional copy pass is needed to move the sorted array to the input array. Our base mergesort algorithm has very low instruction count, executing the fewest instruction of any of our comparison based sorting algorithms.

4.2 Memory Optimizations

While the base mergesort executes few instructions, it has the potential for terrible cache performance. The base mergesort uses each data item only once per pass, and if a pass is large enough to wrap around in the cache, keys will be ejected before they are used again. If the set of keys is of size $BC/2$ or smaller, the entire sort can be performed in the cache and only compulsory misses will be incurred. When the set size is larger than $BC/2$, however, temporal reuse drops off sharply and when the set size is larger than the cache, no temporal reuse occurs at all. To improve this inefficiency, we apply two memory optimizations to the base mergesort. Applying the first of these optimizations yields *tiled mergesort*, and applying both yields *multi-mergesort*.

Tiled mergesort employs an idea, called *tiling*, that is also used in some optimizing compilers [25]. Tiled mergesort has two phases. To improve temporal locality, in the first phase subarrays of length $BC/2$ are sorted using mergesort. The second phase returns to the base mergesort to complete the sorting of the entire array. In order to avoid the final copy if $\lceil \log_2(n/4) \rceil$ is odd, subarrays of size 2 are sorted in-line instead of size 4. Tiling the base mergesort drastically reduces the misses it incurs, and the added loop overhead increases instruction counts very little.

While tiling improves the cache performance of the first phase, the second phase still suffers from the same problem as the base mergesort. Each pass through the source array in the second phase needs to fault in all of the blocks, and no reuse is achieved across passes if the set size is larger than the cache. To fix this inefficiency in the second phase, we employ a multi-way merge similar to those used in external sorting (Knuth devotes a section of his book to techniques for multi-merging [15, Sec. 5.4.1]). In multi-mergesort we replace the final $\lceil \log_2(n/(BC/2)) \rceil$ merge passes of tiled mergesort with a single pass that merges all of the pieces together at once. This single pass makes use of a memory-optimized heap to hold the heads of the lists being multi-merged [16]. The multi-merge introduces several complications to the algorithm and significantly increases the dynamic instruction count. However, the resulting algorithm has excellent cache performance, incurring roughly a constant number of cache misses per key in our executions.

4.3 Performance

As mentioned earlier, the performance of the three mergesort algorithms is measured by sorting sets of uniformly distributed 64 bit integers. Figure 1 shows the number of instructions executed per key, the cache missed incurred and the execution times for each of the mergesort variants. The instruction count graph shows that as expected, the base mergesort and the tiled mergesort execute almost the same number of instructions. The wobble in the instruction count curve for the base mergesort is due to the final copy that may need to take place depending on whether the final merge wrote into the source array or the auxiliary array [15]. When the set size is smaller than the cache, the multi-mergesort executes the same number of instructions as the tiled mergesort. Beyond that size, the multimerge is performed and this graph shows the increase it causes in the instruction count. For 4,096,000 keys, the multimerge executes 70% more instructions than the other mergesorts.

The most striking feature of the cache performance graph is the sudden increase in cache misses for the base mergesort when the set size grows larger than the cache. This graph shows the large impact that tiling the base mergesort has on cache misses; for 4,096,000 keys, the tiled mergesort incurs 66% fewer cache misses that the base mergesort. This graph also shows that the multi-mergesort is a clear success from a cache miss perspective, incurring slightly more than 1 miss per key regardless of the input size.

The graph of execution times shows that up to the size of the second level cache, all of these algorithms perform the same. Beyond that size, the base mergesort performs the worst due to the large number of cache misses it incurs. The tiled mergesort executes up to 55% faster than the base mergesort, showing the significant impact the cache misses in the first phase have on execution time. When the multi-way merge is first performed, the multi-mergesort performs worse than the tiled mergesort due to the increase in instruction count. Due to lower cache misses, however, the multi-mergesort scales better and performs as well as the tiled mergesort for the largest set sizes.

4.4 Analysis

It is fairly easy to approximate the number of cache misses incurred by our mergesort variants as the memory reference patterns of these algorithms are fairly oblivious. We begin with our base algorithm. For $n \leq BC/2$ the number of misses per key is simply $2/B$, the compulsory misses. Since the other two algorithms employ the base algorithm for $n \leq BC/2$ then they all have the same number of misses per key in this range.

For $n > BC/2$, the number of misses per key in the iterative mergesort algorithm is approximately

$$\frac{2}{B}\lceil \log_2 \frac{n}{4} \rceil + \frac{1}{B} + \frac{2}{B}(\lceil \log_2 \frac{n}{4} \rceil \bmod 2). \tag{1}$$

The first term of expression (1) comes from the capacity misses incurred during the $\lceil \log_2 \frac{n}{4} \rceil$ merge passes. In each pass, each key is moved from a source array to a destination array. Every B-th key visited in the source array results in one cache miss and every B-th key written into the destination array results in one cache miss. Thus, there are $2/B$ cache misses per key per pass. The second term, $1/B$, is the compulsory misses per key incurred in the initial pass of sorting into groups of 4. The final term is the number of misses per key caused by the final copy, if there is one.

For $n > BC/2$ the number of misses per key in tiled mergesort is approximately

$$\frac{2}{B}\lceil \log_2 \frac{2n}{BC} \rceil + \frac{2}{B}. \tag{2}$$

The first term of expression (2) is the number of misses per key for the final $\lceil \log_2 \frac{n}{BC/2} \rceil$ merge passes. The second term is the number of misses per key in sorting into $BC/2$ size pieces. The number of passes is forced to be even so there is no additional cache misses caused by a copy. Figure 6 shows how well the analysis of base mergesort and tiled mergesort predict their actual performance.

Finally, for $n > BC/2$, the number of misses per key in multi-mergesort is approximately

$$\frac{4}{B}. \tag{3}$$

For $B = 4$, this closely matches the approximately 1 miss per key shown in cache misses graph of Figure 1. The first phase of multi-mergesort is tiled mergesort which incurs $2/B$ misses per key. In the algorithm we make sure that the number of passes in the first phase is odd so that the second phase multi-merges the auxiliary array into the input array. The second phase multi-merges the pieces stored in the auxiliary array back into the input array causing another $2/B$ misses per key. The multi-merge employs a heap-like array containing $k = 2\lceil 2n/BC \rceil$ keys. For practical purposes it is safe to assume that k is much smaller than BC. Every BC accesses to the input array removes each member of the heap-like array from the cache. Hence, there can be an additional k/B misses for every BC accesses to the input array. This can give an additional $k/(B^2C)$ misses per key incurred during the multi-merge phase. This number is negligible for any practical values of n, B, and C.

5 Quicksort

Quicksort is an in-place divide-and-conquer sorting algorithm considered by most to be the fastest comparison-based sorting algorithm when the set of keys fit in memory [11]. In quicksort, a key from the set is chosen as the *pivot*, and all other keys in the set are partitioned around this pivot. This is usually accomplished by walking through an array of keys from the outside in, swapping keys on the left that are greater than the pivot with keys on the right that are less than the pivot. At the end of the pass, the set of keys will be partitioned around the pivot and the pivot is guaranteed to be in its final position. The quicksort algorithm then recurses on the region to the left of the pivot and the region to the right. The simple recursive quicksort is a simple, elegant algorithm and can be expressed in less than twenty lines of code.

5.1 Base Algorithm

An excellent study of fast implementations of quicksort was conducted by Sedgewick, and we use the optimized quicksort he develops as our base algorithm [20]. Among the optimizations that Sedgewick suggests is one that sorts small subsets using a faster sorting method [20]. He suggests that, rather than sorting these in the natural course of the quicksort recursion, all the small unsorted subarrays be left unsorted until the very end, at which time they are sorted using insertion sort in a single final pass over the entire array. We employ all the optimizations recommended by Sedgewick in our base quicksort.

5.2 Memory Optimizations

In practice, quicksort generally exhibits excellent cache performance. Since the algorithm makes sequential passes through the source array, all keys in a block are always used and spatial locality is excellent. Quicksort's divide-and-conquer structure also gives it excellent temporal locality. If a subset to be sorted is small enough to fit in the cache, quicksort will incur at most one cache miss per block before the subset is fully sorted. Despite this, improvements can be made and we develop two memory optimized versions of quicksort, *memory-tuned quicksort* and *multi-quicksort*.

Our memory-tuned quicksort simply removes Sedgewick's elegant insertion sort at the end, and instead sorts small subarrays when they are first encountered using an unoptimized insertion sort. When a small subarray is encountered, it has just been part of a recent partitioning and this is an ideal time to sort it, since all of its keys should be in the cache. While saving small subarrays until the end makes sense from an instruction count perspective, it is exactly the wrong thing to do from a cache performance perspective.

Multi-quicksort employs a second memory optimization in similar spirit to that used in multi-mergesort. Although quicksort incurs only one cache miss per block when the set is cache-sized or smaller, larger sets incur a substantial number of misses. To fix this inefficiency, a single multi-partition pass can be used to divide the full set into a number of subsets which are likely to be cache sized or smaller.

Multi-partitioning is used in parallel sorting algorithms to divide a set into subsets for multiple processors [1, 13] in order to quickly balance the load. We choose the number of pivots so that the number of subsets larger than the cache is small with sufficiently high probability. Feller shows that if k points are placed randomly in a range of length 1, the chance of a resulting subrange being of size x or greater is exactly $(1-x)^k$ [5, Vol. 2, Pg. 22]. Let n be the total number of keys, B the number of keys per cache block, and C the capacity of the cache in blocks. In multi-quicksort we partition the input array into $3n/(BC)$ pieces, requiring $(3n/(BC))-1$ pivots. Feller's formula indicates that after the multi-partition, the chance that a subset is larger than BC is $(1-BC/n)^{(3n/(BC))-1}$. In the limit as n grows large, the percentage of subsets that are larger than the cache is e^{-3}, less than 5%.

There are a number of complications in the design of the multi-partition phase of multi-quicksort, not the least of which is that the algorithm cannot be done efficiently in-place and executes more instructions than the base quicksort. However, the resulting multi-quicksort is very efficient from a cache perspective.

5.3 Performance

Figure 2 shows the performance of the three quicksort algorithms sorting 64 bit uniformly distributed integers. The instruction count graph shows that the base quicksort executes the fewest instructions with the memory-tuned quicksort executing a constant number of additional instructions per key. This difference is due to the inefficiency of sorting the small subsets individually rather than at the end as suggested by Sedgewick. For large set sizes the multi-quicksort performs the multipartition which result in a significant increase in the number of instructions executed.

The cache performance graph shows that all of the quicksort algorithms incur very few cache misses. The base quicksort incurs fewer than two misses per key for 4,096,000 keys, lower than all of the other algorithms up to this point with the exception of the multi-mergesort. The cache miss curve for the memory-tuned quicksort shows that removing the instruction count optimization in the base quicksort improves cache performance by approximately .25 cache misses per key. The cache miss graph also shows that the multi-way partition produces a flat cache miss curve much the same as the curve for the multi-mergesort. The maximum number of misses incurred per key for the multi-quicksort is slightly larger than 1 miss per key, validating the conjecture that it is uncommon for the multipartition to produce subsets larger than the size of the cache.

The graph of execution times shows the execution times of the three quicksort algorithms. All three of these algorithms perform similarly on our DEC Alphastation 250. This graph shows that sorting small subsets early is a benefit, and the reduction in cache misses outweighs the increase in instruction cost. The multipartition initially hurts the performance of the multi-quicksort due to the increase in instruction cost, but the low number of cache misses makes it more competitive as the set size is increased. This graph suggests that if more memory were available and larger sets were sorted, the multi-quicksort would outperform both the base quicksort and the memory-tuned quicksort.

5.4 Analysis

We begin by analyzing memory-tuned quicksort which has slightly simplier behavior than base quicksort. If $n \leq BC$ then memory-tuned quicksort incurs $1/B$ misses per key for the compulsory misses. Because base quicksort makes an extra pass at the end to perform the insertion sort, the number of cache misses per key incurred by base quicksort is $1/B$ plus the number of cache misses per key incurred by memory-tuned quicksort. The cache miss graph of figure 2 clearly show the additional $1/B = .25$ misses per key for base quicksort in our simulation.

For $n > BC$ the expected number of misses per key in memory-tuned quicksort is approximately

$$\frac{2}{B}\ln(\frac{n}{BC}) + \frac{5}{8B} + \frac{3C}{8n}. \tag{4}$$

We analyze the algorithm in two parts. In the first part we assume that partitioning an array of size m costs m/B misses if $m > BC$ and 0 misses otherwise. In the second part we correct this by estimating undercounted and overcounted cache misses. Let $M(n)$ be the expected number of misses incurred in quicksort under our assumption of the first part of the analysis. We then have the recurrence

$$M(n) = \frac{n}{B} + \frac{1}{n}\sum_{i=0}^{n-1}(M(i) + M(n-i-1)) \text{ if } n > BC$$

and $M(n) = 0$ if $n \leq BC$. Using standard techniques [15] this recurrence solves to

$$M(n) = \frac{2(n+1)}{B}\ln(\frac{n+1}{BC+2}) + O(\frac{1}{n})$$

for $n > BC$.

The first correction we make is undercounting the misses that are incurred when the subproblem first reaches size $\leq BC$. In the analysis above we count this as zero misses, when in fact this subproblem may have no part in the cache. To account for this we add in n/B more misses since there are approximately n keys in all the subproblems that first reach size $\leq BC$.

In the very first partitioning in quicksort, there are n/B cache misses, but not necessarily for subsequent partitionings. At the end of partitioning some of the array in the left subproblem is still in the cache. Hence there are hits that we are counting as misses in the analysis above. Note that the right subproblem does not have these hits because by the time the algorithm reaches the right subproblem its data has been removed from the cache.

We first analyze the expected number of subproblems of size $> BC$. This is given by the recurrence

$$N(n) = 1 + \frac{1}{n}\sum_{i=0}^{n-1}(N(i) + N(n-i-1)) \text{ if } n > BC$$

and $N(n) = 0$ if $n \leq BC$. This recurrence solves exactly to

$$N(n) = \frac{n+1}{BC+2} - 1$$

for $n > BC$. For each of these subproblems there is a left subproblem. On average, $BC/2$ of the keys in this left subproblem are in the cache. Not all the accesses to these keys are hits. While the right pointer in the partitioning enjoys the benefit of access to these keys, the left pointer eventually accesses blocks that map to these blocks in the cache, thereby replacing them. Without loss of generality we can assume that the right pointer in the partitioning begins at the right end of the cache. There are two scenarios. In the first scenario, the right pointer progresses left and the left pointer starts i blocks to the left of the middle of the cache and progresses right. On average, there will be $\frac{C/2+i}{2}$ hits. In the second scenario, the left pointer starts i blocks to the right of the middle of the cache and progresses right. On average, there will be $i + \frac{C/2-i}{2} = \frac{C/2+i}{2}$ hits. If we assume that the left pointer can start on any block with equal probability then we can estimate the expected number of hits to be

$$\frac{2}{C}\sum_{i=1}^{C/2}\frac{C/2+i}{2} \approx \frac{3C}{8}.$$

Hence the expected number of hits not accounted for in the computation of $M(n)$ is approximately

$$\frac{3C}{8}N(n).$$

Adding up all the pieces, for $n > BC$, the expected number of misses per key is approximately

$$(M(n) + (n/B) - \frac{3C}{8}N(n))/n$$

which is approximated closely by expression (4) above. Figure 7 shows how well this approximation of cache misses predicts the actual performance of memory-tuned quicksort. In addition, Figure 7 shows approximation of $1/B$ more misses per key for base quicksort is predicted well.

For multi-quicksort the approximate analysis is quite simple. If $n \leq BC$ multi-quicksort the number of misses per key is simply $1/B$, the compulsory misses. For $n > BC$ then the algorithm partitions the input into $k = 3n/(BC)$ pieces, the vast majority of which are smaller than the cache. For the purposes of our analysis we assume they are are all smaller than the cache. In this multi-partition phase we move the partitioned keys into k linked lists one for each partition. Each node in a linked list has room for 100 keys. In this way can minimize storage waste at the same time as minimizing cache misses. We approximate the number of misses per key in the first phase as $2/B$, one miss per block in the input array and one miss per block in the linked list. In the second phase each partition is returned to the input array and sorted in place. This costs approximately another $2/B$ misses per key. The cache miss overhead of maintaining the additional pivots is negligible for practical values of n, B and C. The total is approximately

$$\frac{4}{B} \tag{5}$$

misses per key which closely matches the misses reported in Figure 2.

6 Heapsort

While heaps are used for a variety of purposes, they were first proposed by Williams as part of the heapsort algorithm [24]. The heapsort algorithm sorts by first building a heap containing all of the keys and then removing them one at a time in sorted order. Using an array implementation of a heap results in an straightforward in-place sorting algorithm. On a set of n keys, Williams' algorithm takes $O(n \log n)$ steps to build the heap and $O(n \log n)$ steps to remove the keys in sorted order. In 1965 Floyd proposed an improved technique for building a heap with better average case performance and a worst case of $O(n)$ steps [7]. Williams' base algorithm with Floyd's improvement is still the most prevalent heapsort variant in use.

6.1 Base Algorithm

As a base heapsort algorithm, we follow the recommendations of algorithm textbooks and use an array implementation of a binary heap constructed using Floyd's method. In addition, we employ a standard optimization of using a sentinel at the end of the heap to eliminate a comparison per level which reduces instruction count. The literature contains a number of other optimizations that reduce the number of comparisons performed for both adds and removes [4, 2, 9], but in practice these increase the total number of instructions executed and do not improve performance. For this reason, we do not include them in our base heapsort.

6.2 Memory Optimizations

To this base heapsort algorithm, we now apply memory optimizations in order to further improve performance. Our previous results [17, 16] show that William's repeated-adds algorithm [24] for building a binary heap incurs fewer cache misses than Floyd's method. In addition we have shown that two other optimizations reduce the number of cache misses incurred by the remove-min operation [17, 16]. The first optimization is to replace the traditional binary heap with a d-heap [14] where each non-leaf node has d children instead of two. The fanout d is chosen so that exactly d keys fit in a cache block. If d is relatively small, say 4 or 8, there is an added advantage that the number of instructions executed for both add and remove-min is also reduced. The second optimization is to align the heap array in memory so that all d children lie on the same cache block. This optimization reduces what Lebeck and Wood refer to as *alignment misses* [18]. Our memory-optimized heapsort dynamically chooses between Williams' repeated-adds method and Floyd's method for building a heap. If the heap is larger than the cache and Williams' method can offer a reduction in cache misses, it is chosen over Floyd's method. We call the base algorithm with these memory optimizations applied *memory-tuned heapsort.*

6.3 Performance

Since the DEC Alphastation 250 has a 32 byte block size and since we are sorting 8 byte keys, 4 keys fit in a block. As a consequence we choose $d = 4$ in our memory-tuned heapsort. Figure 3 shows the performance of both the base heapsort and the memory-tuned heapsort. The instruction count curves show that memory-tuned heapsort executes fewer instructions than base heapsort ($d = 2$).

The graph of cache performance shows that when the set to be sorted fit in the cache, the minimum 8 bytes / 32 bytes = 0.25 compulsory misses are incurred per key for both algorithms. For larger sets the number of cache misses incurred by memory-tuned heapsort is less than half the misses incurred by base heapsort.

The execution time graph shows that the memory-tuned heapsort outperforms the base heapsort for all set sizes. The memory-tuned heapsort initially outperforms the base heapsort due to lower instruction counts. When the set size reaches the cache size, the gap widens due to differences in the number of cache misses incurred. For 4,096,000 keys, the memory-tuned heapsort sorts 81% faster than the base heapsort.

6.4 Analysis

For $n \leq BC$ heapsort, as an in-place sorting algorithm takes $1/B$ misses per key. For $n > BC$ we adopt an analysis technique, *collective analysis* that we used in a previous paper [17]. Collective analysis is an analytical framework for predicting the cache performance of algorithms when the algorithm's memory access behavior can be approximated using independent stochastic processes. As part of the analysis, the cache is divided into regions that are assumed to be accessed uniformly. By stating the way in which each process accesses each region, a simple formula can be used to predict cache performance. While collective analysis makes a number of simplifications which limit the class of algorithms that can be accurately analyzed, it serves well for algorithms whose behavior is understood, but whose exact reference pattern varies.

As part of our collective analysis work, we analyzed the cache performance of d-heaps in the *hold model*. In the hold model, the number of elements in the heap is held constant as elements are alternately removed from and added to the heap. By adding some additional work between the removes and the adds, the hold model becomes a good approximation of a discrete event simulation with a static number of events. While the number of elements in the heap does not remain constant during heapsort, we can still use our collective analysis of heaps to approximate its cache performance.

Our heapsort algorithm goes through two phases: the build-heap phase and the remove phase. For the build-heap phase, recall that William's method for building the heap simply puts each key at the bottom of the heap and percolates it up to its proper place [24]. We pessimistically assume that all of these adds percolate to the root and that only the most recent leaf-to-root path is in the cache. In this case, the probability that the next leaf is not in the cache is $1/B$, that chance that the parent of that leaf is not in the cache is $1/(dB)$, the chance that the grandparent of that leaf is not in the cache is $1/(d^2B)$, and so on. This gives the simple approximation of the number of misses incurred per key during the build phase of $\sum_{i=0}^{\infty} 1/(d^iB) = d/((d-1)B)$. Thus we estimate the expected number of misses for the build-heap phase at $dn/((d-1)B)$. It is interesting to note that this is within a small constant factor of the n/B compulsory misses that must be incurred by any build-heap algorithm.

We divide the remove phase of heapsort into $n/(BC)$ subphases with each subphase removing BC keys. For $0 \leq i < n/(BC) - 1$ we model the removal of keys $BCi + 1$ to $BC(i+1)$ as BC steps on an array of size $n - BCi$ in the hold model. In essence, we are saying that the removal of BC keys from the d-heap is approximated by BC repeated remove-mins and adds in approximately the same size d-heap. Admittedly, this approximation was one of convenience because we already had a complete approximate analysis of the d-heap in the hold model. Combining this remove phase estimate with our build-heap predictions yields Figure 8. This graph shows our cache predictions for heapsort using both the traditional binary heap (base heapsort) and the 4-heap (memory-tuned heapsort). The predictions match the simulation results surprisingly well considering the simplifying assumptions made in the analysis.

7 Radix Sort

Radix sort is the most important non-comparison based sorting algorithm used today. Knuth [15] traces the radix sort suitable for sorting in the main memory of a computer to a Master's thesis of Seward, 1954 [21]. Seward pointed out that radix sort of n keys can be accomplished using two n key arrays together with a count array of size 2^r which can hold integers up to size n, where r is the "radix." Seward's method is still a standard method found in radix sorting programs. Radix sort is often called a "linear time" sort because for keys of fixed length and for a fixed radix a constant number of passes over the data is sufficient to accomplish the sort, independent of the number of keys. For example, if we are sorting b bit integers with a radix of r then Seward's method does $\lceil b/r \rceil$ iterations each with 2 passes over the source array. The first pass accumulates counts of the number of keys with each radix. The counts are used to determine the offsets in the keys of each radix in the destination array. The second pass moves the source array to the destination array according to the offsets. Friend [8] suggested an improvement to reduce the number of passes over the source array, by accumulating the counts for the $(i+1)$-st iteration at the same time as moving keys during the i-th iteration. This requires a second count array of size 2^r. This improvement has a positive effect on both instruction count and cache misses. Our radix sort is a highly optimized version of Seward's algorithm with Friend's improvement. The final task is to pick the radix which minimizes instruction count. This is done empirically since there is no universally best r which minimizes instruction count.

There is no obvious memory optimization for radix sort that is similar to those that we used for our comparison sorts. A simple memory optimization is to choose the radix which minimizes cache misses. As it happens, for our implementation of radixsort, a radix of 16 bits minimizes both cache misses and instruction count on an Alphastation 250. In the analysis subsection below we take a closer look at the cache misses incurred by radix sort.

7.1 Performance

For this study the keys are 64 bit integers and the counts can be restricted to 32 bit integers. With a 16 bit radix the two count arrays together are 1/4-th the size of the 2 Megabyte cache. Figure 4 shows the resulting performance. The instruction count graph shows radix sort's "linear time" behavior rather stunningly. The cache miss graph shows that when the size of the input array reaches the cache capacity the number of cache misses rapidly grows to a constant slightly more than 3 misses per key. The execution time graph clearly shows the effect that cache misses can have on overall performance. The execution time curve looks much like the instruction count curve until the input array exceeds the cache size at which time cycles per key increase according to the cache miss curve.

7.2 Analysis

The approximate cache miss analysis of radix sort is more complicated than our previous analyses for a number of reasons. First, there are a multitude of parameters to consider in this analysis: n, the number of keys; b, the number of bits per key; r, the radix; B, the number of keys per block; A, the number of counts per block; C, the capacity of the cache in blocks. Second, there are several cache effects to consider. We will focus on what we feel are the three most significant cache effects, the capacity misses in traversals over the source and destination arrays, the conflict misses between the traversal of the source array and accesses to the two count arrays, and the conflict misses between the traversal of the source array and accesses to the destination array.

We will focus on analyzing the number of cache misses for a fixed large number of keys, while varying the radix. In the case of radix 16 this analysis attempts to predict the flat part of the cache curve in Figure 4. We assume that the size of the two count arrays is less than the cache capacity, $2^{r+1} \leq AC$. In addition, we assume that $r \leq b$. For $n > BC$, the expected number of cache misses per key in radix sort is approximated by sum of three terms $M_{trav} + M_{count} + M_{dest}$ where M_{trav} is the number of misses per key incurred in traversing the source and destination arrays, M_{count} is the expected number of misses per key incurred in accesses to the count arrays, and M_{dest} is the expected number of misses per key in the destination array caused by conflicts with the traversal of the source array. We approximate M_{trav}, M_{count}, and M_{dest} by the following:

$$M_{trav} = \frac{1}{B}(2\lceil\frac{b}{r}\rceil + 1) + \frac{2}{B}(\lceil\frac{b}{r}\rceil \bmod 2) \tag{6}$$

$$M_{count} = \frac{2^{r+1}}{ABC}(1-(1-\min(1,\frac{A}{2^r}))^{BC})\lfloor\frac{b}{r}\rfloor + \frac{2^{(b \bmod r)+1}}{ABC}(1-(1-\min(1,\frac{A}{2^{b \bmod r}}))^{BC}) \tag{7}$$

$$M_{dest} = \frac{(B-1)2^r}{B^2C}(1-(1-\frac{1}{2^r})^{BC})\lfloor\frac{b}{r}\rfloor + \frac{(B-1)2^{b \bmod r}}{B^2C}(1-(1-\frac{1}{2^{b \bmod r}})^{BC}) \tag{8}$$

We start with the derivation of M_{trav}, equation (6). To implement Friend's improvement, there is an initial pass over the input array during which counts are accumulated in the first count array. Then there are $\lceil b/r \rceil$ iterations, where a source array is moved to the the destination array according to the offsets established by one of the count arrays in the previous iteration. If $\lceil b/r \rceil$ is odd then the final sorted array must be copied back into the input array. In total, there are $2\lceil b/r \rceil + 1 + 2(\lceil b/r \rceil \bmod 2)$ passes over either a source or destination array. Ignoring any conflicts between the source and destination array, the number of misses per key is $1/B$ times the number of passes.

The quantity M_{count} accounts for the misses in random accesses in the two count arrays caused by the traversal over the source array. There are a total of $\lceil b/r \rceil + 1$ traversals over the source array that conflict with random accesses to at least one count array. If r divides b then the first and last traversal conflict with random accesses to one count array of size 2^r and all the other traversals conflict with random accesses to two count arrays of size 2^r. If r does not divide b then the first traversal conflicts with random accesses to one count array of size 2^r, the last traversal conflicts with random accesses to one count array of size $2^{b \bmod r}$, the second to last traversal conflicts with random access to one count array of size 2^r and one count array of size $2^{b \bmod r}$, and the remaining traversals, if any, conflict with random accesses to two count arrays of size 2^r. Generally, let us consider a pass over a source array that is in conflict with a count array of size 2^m. Consider a specific block x in one of the count arrays. Every BC steps of the algorithm the traversal of the source array ejects the block x from the cache. If the block is accessed in the count array before the another BC steps of the algorithm then a miss is incurred in that access. The probability that block x is accessed in BC steps is $1 - (1 - \min(1, A/2^m))^{BC}$. The total expected number of misses is approximated by this quantity times the number of blocks in the array $(2^m/A)$ times the number of times x is visited in the traversal $(n/(BC))$. Thus, the expected number of misses per key incurred by a traversal over the source array in the count array of size 2^m is approximately

$$\frac{2^m}{ABC}(1 - (1 - \min(1, \frac{A}{2^m}))^{BC}). \tag{9}$$

If r divides b then the effect is $2b/r$ passes that conflict with a single count array of size 2^r. Expression (9) yields the first term of equation (7) (and the second term is negligible). If r does not divide b then the effect is $2\lfloor b/r \rfloor$ passes that conflict with a count array of size 2^r and two passes that conflict with a count array of size $2^{b \bmod r}$. This yields equation (7).

The quantity M_{dest} accounts for the conflict for the cache between the traversal over the source array and accesses into the destination array. Consider a specific block x in the destination array. In each iteration of the algorithm this block is visited exactly B times. The first visit is a cache miss that was accounted for in M_{trav} above. Each of the remaining $B-1$ visits is a cache hit unless the traversal in the source array visits the cache block of x before the next visit to x by the destination array. Assume that the number of pointers into the destination array is 2^m. As an approximation we assume that the $B-1$ remaining accesses to block x each occur independently with probability $1/2^m$. Under this assumption, the probability that the traversal will visit the cache block of x before the next visit to x by the destination array is

$$\frac{2^m}{BC}(1 - (1 - \frac{1}{2^m})^{BC}). \tag{10}$$

To see this, suppose that in exactly $i+1$ steps the traversal will visit the cache block of x. The probability that x will be accessed in the destination array before the traversal captures the cache block of x is approximately $(1 - 1/2^m)^i$. With probability $1/BC$ the traversal is $i+1$ steps from capturing the cache block of x. Thus, the probability that the traversal will capture the cache block of x before the access to block x is

$$\frac{1}{BC}\sum_{i=0}^{BC-1}(1-\frac{1}{2^m})^i$$

which is equal to expression (10). To complete the derivation of equation (8) there are two cases to consider, depending on whether r divides b or not. If r divides b then there are b/r traversals of the source array that conflict with a traversal of the destination array. There are 2^r pointers into the destination array. The expected number of misses per key is then the number of passes, b/r, times $\frac{2^r}{BC}(1-(1-1/2^r)^{BC})$ times $(B-1)/B$, which totals to the first term of M_{dest}. In this case the second term of M_{dest} is negiligible. If r does not divide b then there are $\lfloor b/r \rfloor$ traversals with 2^r pointers and one traversal with $2^{b \bmod r}$ pointers. Using an analysis similar to that above yields the equation for M_{dest}. We should note that a more accurate formula for M_{dest}, that does not treat the $B-1$ accesses to block x as independent, can be derived. However, the more accurate formula is actually a complex recurrence that does not yield itself to a simple closed form solution nor to a tractable numerical solution. Our independence assumption seems to yield fairly accurate results.

Figure 9 compares this approximation with the actual number of cache misses incurred per key, as a function of r, by radix sort for $n = 4,096,000$, $b = 64$, $A = 8$, $B = 4$, and $C = 2^{16}$. Figure 9 only goes up to radix 18 because beyond radix 18 the two count arrays are together larger than the cache. The theoretical prediction and the simulation agree that for this parameter set the optimal radix is 16.

8 Discussion

In this section we compare the performance of the fastest variant of each of the four sorting algorithms we examined in our study. In addition, we examine the performance of our algorithms on four additional architectures, in order to explore the robustness of our memory optimizations.

8.1 Performance Comparison

To compare performance across sorting algorithms, Figure 5 shows the instruction count, cache misses and cycles executed per key for the fastest variants of heapsort, mergesort, quicksort and radix sort.

The instruction count graph shows that the memory-tuned heapsort executes the most instructions, while radix sort executes the least. The cache miss graph shows that radix sort has the most cache misses, incurring more than twice as many misses as memory-tuned quicksort. The execution time graph strikingly shows the effect of cache performance on overall performance. For the largest data set memory-tuned quicksort ran 24% faster than radix sort even though memory-tuned quicksort performed more than three times as many instructions. This graph also shows that the small differences in instruction count and cache misses between the memory-tuned mergesort and memory-tuned quicksort offset to yield two algorithms with very comparable execution times.

8.2 Robustness

In order to determine if our experimental results generalize beyond the DEC Alphastation 250, we ran our programs on four other platforms: an IBM Power PC, a Sparc 10, a DEC Alpha 3000/400 and a Pentium base PC. Figure 10 shows the speedup that the memory-tuned heapsort achieves over the base heapsort for the Alphastation 250 as well as these four additional machines. Despite the differences in architecture, all the platforms show similar speedups.

Figure 11 shows the speedup of our tiled mergesort over the traditional mergesort for the same five machines. Unlike the heapsort case, the speedups for mergesort differ substantially. This is partly due to differences in the page mapping policies of the different operating systems. Throughout this study we have assumed that a block of contiguous pages in the virtual address space map to a block of contiguous pages in the cache. This is only guaranteed to be true when caches are virtually indexed rather than physically indexed [10]. Unfortunately, the caches on all five of our test machines are physically indexed. Fortunately, some operating systems, such as Digital Unix, have virtual to physical page mapping polices that attempt to map pages so that blocks of memory nearby in the virtual address space do not conflict in the cache [23]. Unlike the heapsort algorithms, tiled mergesort relies heavily on the assumption that a cache-sized block of pages do not conflict in the cache. As a result, the speedup of tiled mergesort relies heavily on the quality of the operating system's page mapping decisions. While the operating systems for the Sparc and the Alphas (Solaris and Digital Unix) make cache-conscious decisions about page placement, the operating systems for the Power PC and the Pentium (AIX and Linux) appear not to be as careful.

9 Conclusions

We have explored the potential performance gains that cache-conscious design and analysis offers to classic sorting algorithms. The main conclusion of this work is that the effects of caching are extremely important and need to be considered if good performance is a goal. Despite its very low instruction count, radix sort is outperformed by both mergesort and quicksort due to its relatively poor locality. Despite the fact that the multi-mergesort executed 75% more instructions than the base mergesort, it sorts up to 70% faster. Neither the multi-mergesort nor the multi-quicksort are in-place or stable. Nevertheless, these two algorithms offer something that none of the others do. They both incur very few cache misses, which renders their overall performance far less sensitive to cache miss penalties than the others. As a result, these algorithms can be expected to outperform the others as relative cache miss penalties continue to increase.

In this paper, a number of memory optimizations are applied to algorithms in order to improve their overall performance. What follows is a summary of the design principles applied in this work:

- Improving cache performance even at the cost of an increased instruction count can improve overall performance.
- Knowing and using architectural constants such as cache size and block size can improve an algorithm's memory system performance beyond that of a generic algorithm.
- Spatial locality can be improved by adjusting an algorithm's structure to fully utilize cache blocks.
- Temporal locality can be improved by padding and adjusting data layout so that structures are aligned within cache blocks.
- Capacity misses can be reduced by processing large data sets in cache-sized pieces.
- Conflict misses can be reduced by processing data in cache-block-sized pieces.

This paper also shows that despite the complexities of caching, the cache performance of algorithms can be reasonably approximated with a modest amount of work. Figures 6-9 show that our approximate analysis gives good information. However, more work needs to be done to improve the analysis techniques to make them more accurate.

References

[1] G. Blelloch, C. Plaxton, C. Leiserson, S Smith, B. Maggs, and M. Zagha. A comparison of sorting algorithms for the connection machine. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms & Architecture*, pages 3–16, July 1991.

[2] S. Carlsson. An optimal algorithm for deleting the root of a heap. *Information Processing Letters*, 37(2):117–120, 1991.

[3] D. Clark. Cache performance of the VAX-11/780. *ACM Transactions on Computer Systems*, 1(1):24–37, 1983.

[4] J. De Graffe and W. Kosters. Expected heights in heaps. *BIT*, 32(4):570–579, 1992.

[5] W. Feller. *An Introduction to Probability Theory and its Applications.* Wiley, New York, NY, 1971.

[6] D. Fenwick, D. Foley, W. Gist, S. VanDoren, and D. Wissell. The AlphaServer 8000 series: High-end server platform development. *Digital Technical Journal*, 7(1):43–65, 1995.

[7] Robert W. Floyd. Treesort 3. *Communications of the ACM*, 7(12):701, 1964.

[8] E. H. Friend. *Journal of the ACM*, 3:152, 1956.

[9] G. Gonnet and J. Munro. Heaps on heaps. *SIAM Journal of Computing*, 15(4):964–971, 1986.

[10] J. Hennesey and D. Patterson. *Computer Architecture A Quantitative Approach, Second Edition.* Morgan Kaufman Publishers, Inc., San Mateo, CA, 1996.

[11] C. A. R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[12] F. E. Holberton. In *Symosium on Automatic Programming*, pages 34–39, 1952.

[13] Li Hui and K. C. Sevcik. Parallel sorting by overpartitioning. In *Proceedings of the 6th ACM Symposium on Parallel Algorithms & Architecture*, pages 46–56, June 1994.

[14] D. B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4, 1975.

[15] D. E. Knuth. *The Art of Computer Programming, vol III – Sorting and Searching.* Addison–Wesley, Reading, MA, 1973.

[16] A. LaMarca. Caches and algorithms. Ph.D. Dissertation, University of Washington, May 1996.

[17] A. LaMarca and R. E. Ladner. The influence of caches on the performance of heaps. *Journal of Experimental Algorithmics*, Vol 1, Article 4, 1996.

[18] A. Lebeck and D. Wood. Cache profiling and the spec benchmarks: a case study. *Computer*, 27(10):15–26, Oct 1994.

[19] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: a RISC machine sort. In *1994 ACM SIGMOD International Conference on Management of Data*, pages 233–242, May 1994.

[20] R. Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, October 1978.

[21] H. H. Seward. Masters Thesis, M.I.T. Digital Computer Laboratory Report R-232, 1954.

[22] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the 1994 ACM Symposium on Programming Languages Design and Implementation*, pages 196–205. ACM, 1994.

[23] G. Taylor, P. Davies, and M. Farmwald. The TBL slice:a low-cost high-speed address translation mechanism. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 355–363, 1990.

[24] J. W. Williams. Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[25] M. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, pages 655–664, 1989.

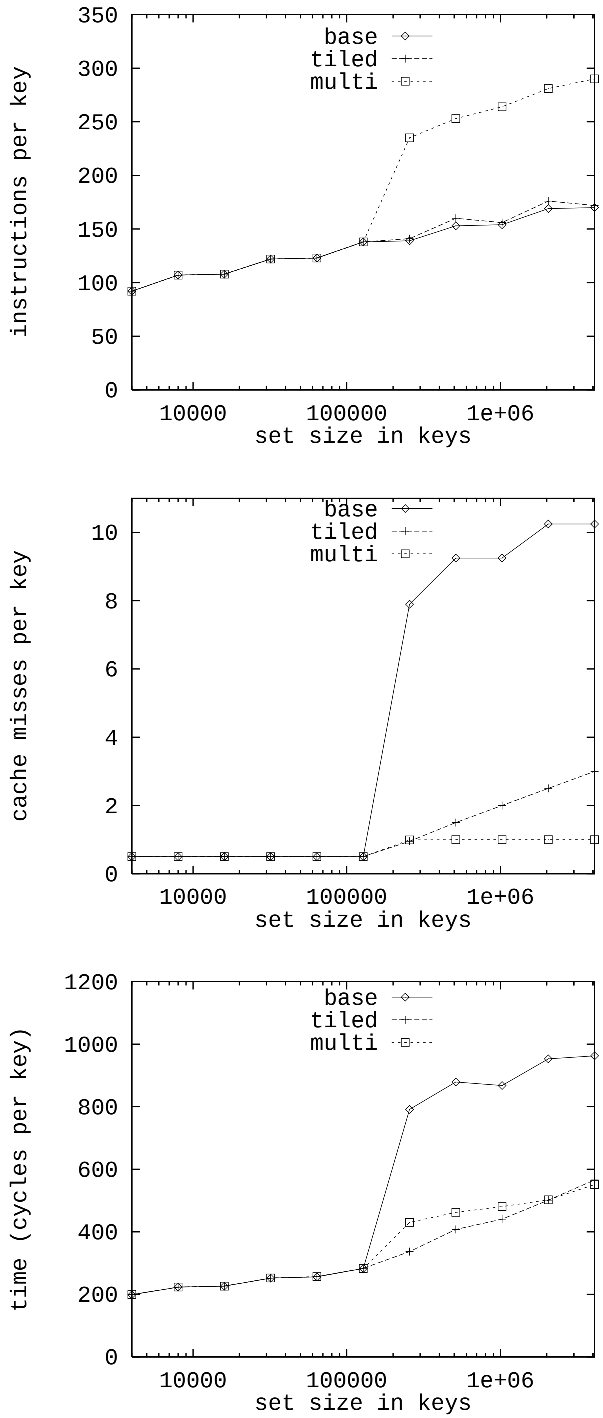

Figure 1: Performance of mergesort on sets between 4,000 and 4,096,000 keys. From top to bottom, the graphs show instruction counts per key, cache misses per key and the execution times per key. Executions were run on a DEC Alphastation 250 and simulated cache capacity is 2 megabytes with a 32 byte block size.

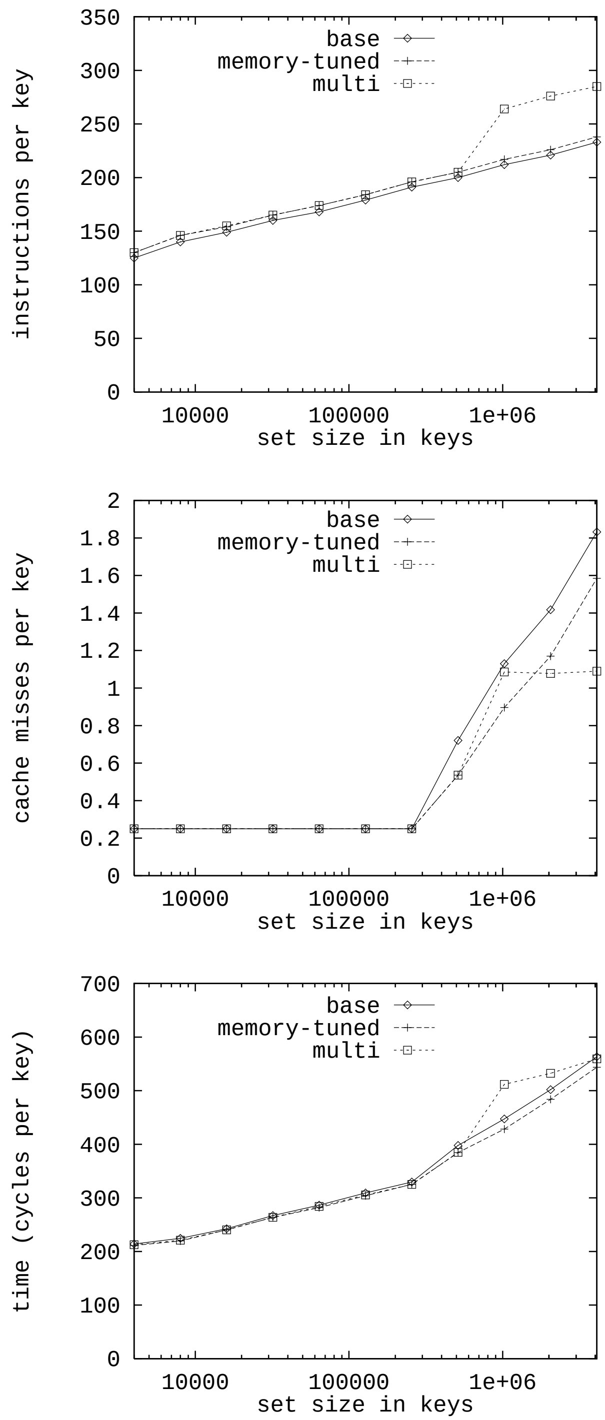

Figure 2: Performance of quicksort on sets between 4,000 and 4,096,000 keys. From top to bottom, the graphs show instruction counts per key, cache misses per key and the execution times per key. Executions were run on a DEC Alphastation 250 and simulated cache capacity is 2 megabytes with a 32 byte block size.

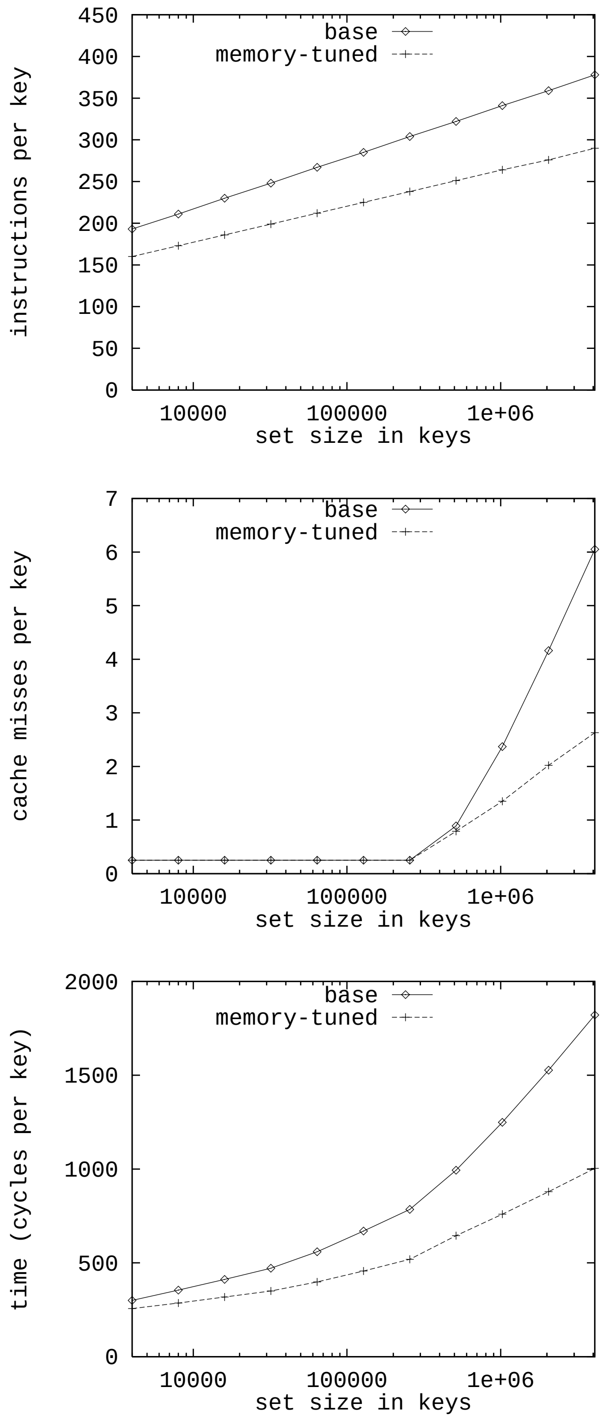

Figure 3: Performance of heapsort on sets between 4,000 and 4,096,000 keys. From top to bottom, the graphs show instruction counts per key, cache misses per key and the execution times per key. Executions were run on a DEC Alphastation 250 and simulated cache capacity is 2 megabytes with a 32 byte block size.

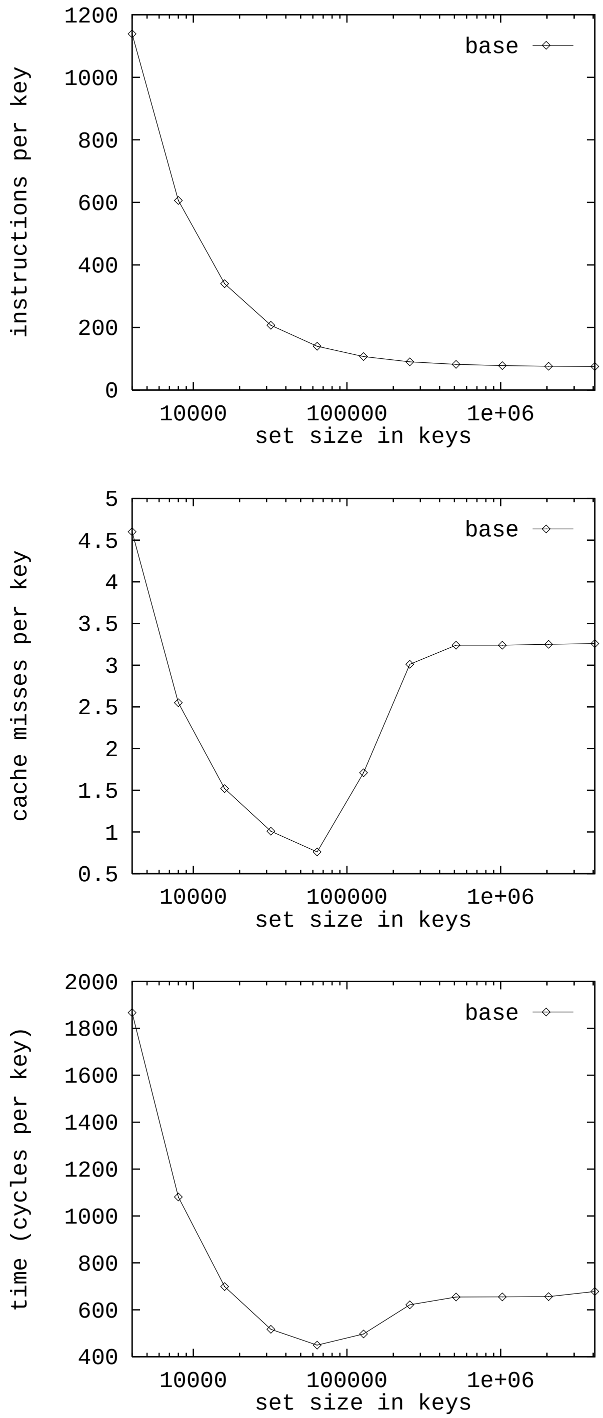

Figure 4: Performance of radixsort on sets between 4,000 and 4,096,000 keys. From top to bottom, the graphs show instruction counts per key, cache misses per key and the execution times per key. Executions were run on a DEC Alphastation 250 and simulated cache capacity is 2 megabytes with a 32 byte block size.

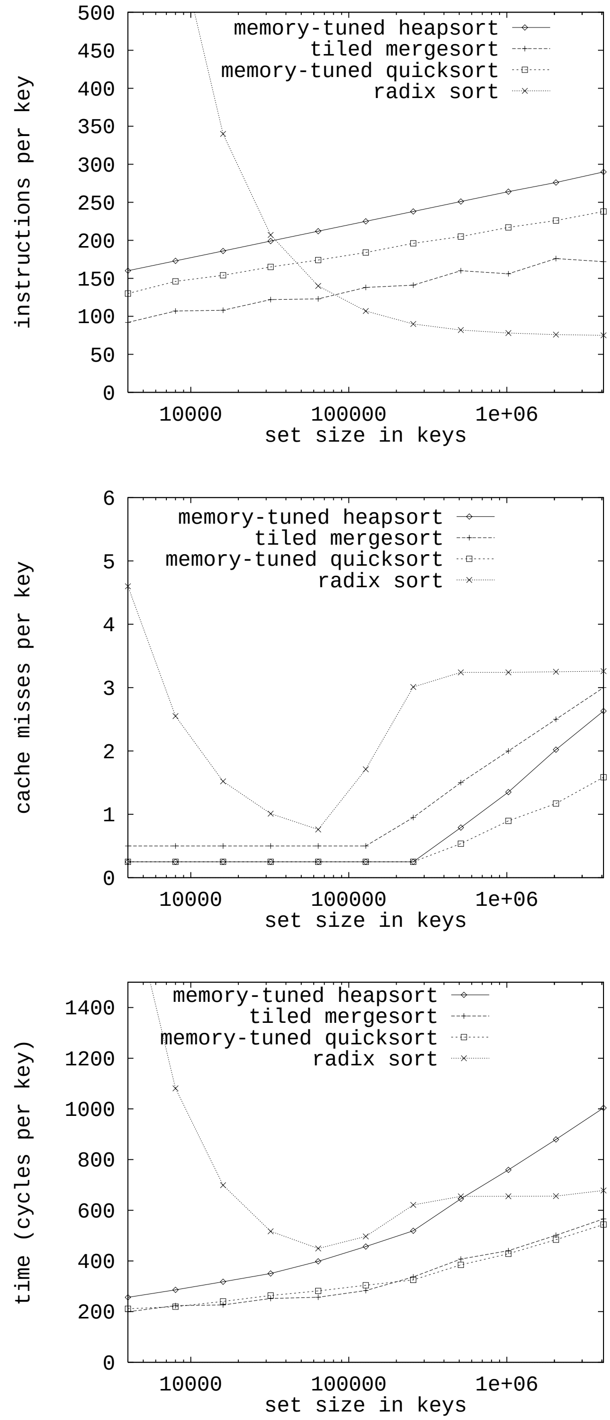

Figure 5: Instruction count, cache misses and execution time per key for the best heapsort, mergesort, quicksort and radix sort on a DEC Alphastation 250. Simulated cache capacity is 2 megabytes, block size is 32 bytes.

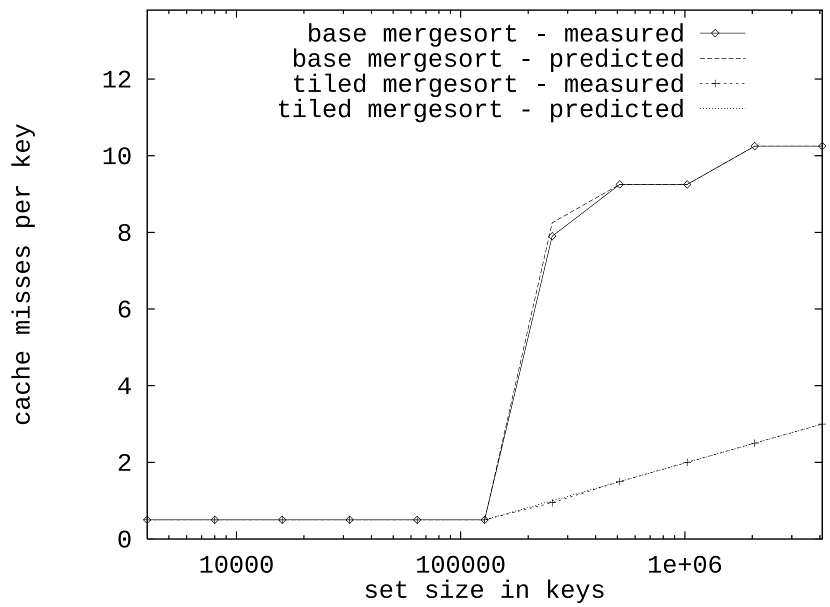

Figure 6: Cache misses incurred by mergesort, measured versus predicted.

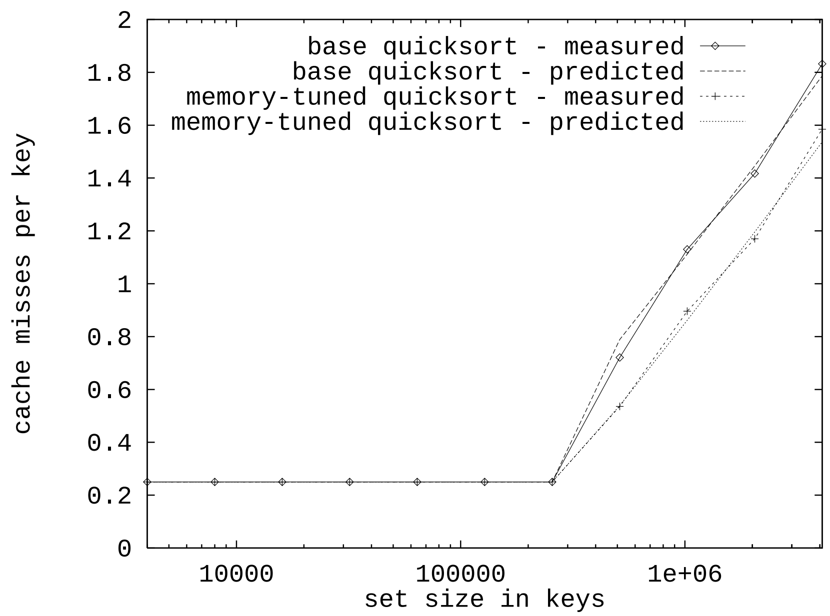

Figure 7: Cache misses incurred by quicksort, measured versus predicted.

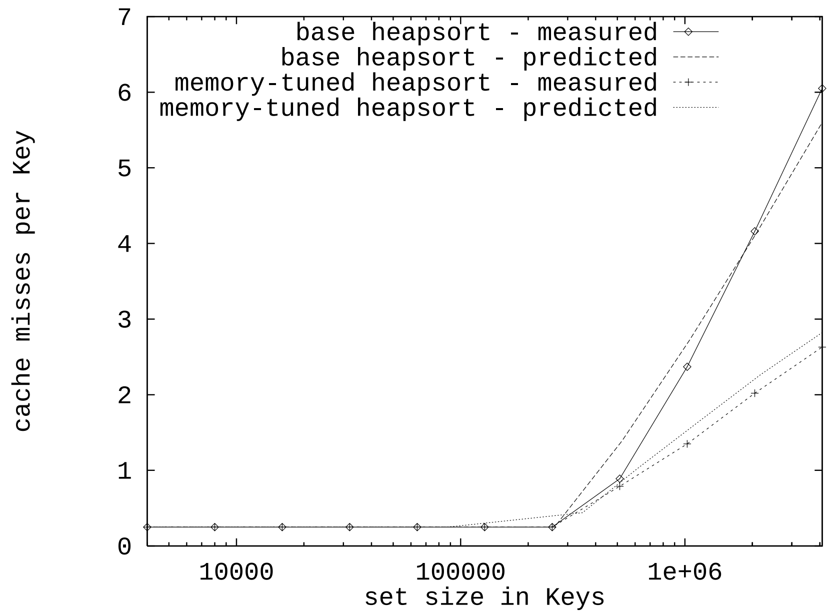

Figure 8: Cache misses incurred by heapsort, measured versus predicted.

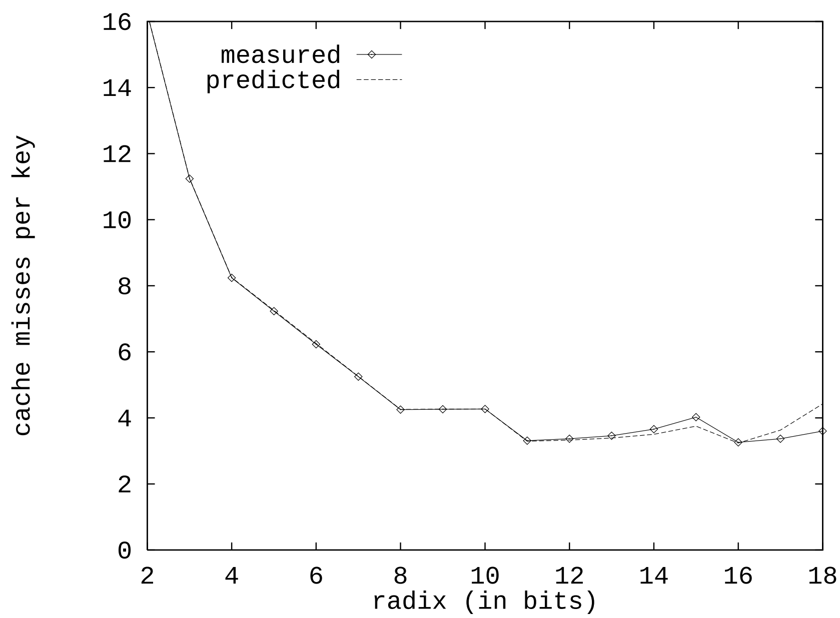

Figure 9: Cache misses incurred by radix sort, measured versus predicted.

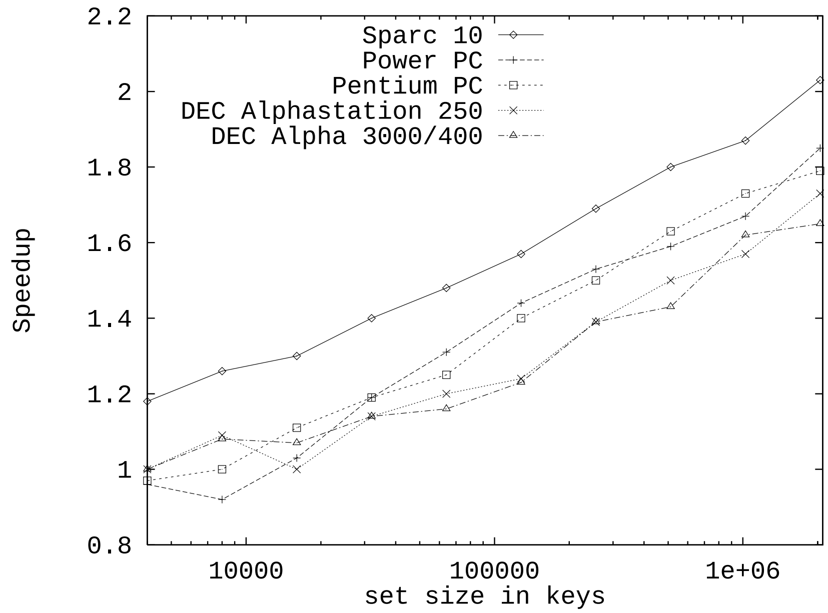

Figure 10: Speedup of memory-tuned heapsort over base heapsort on five architectures.

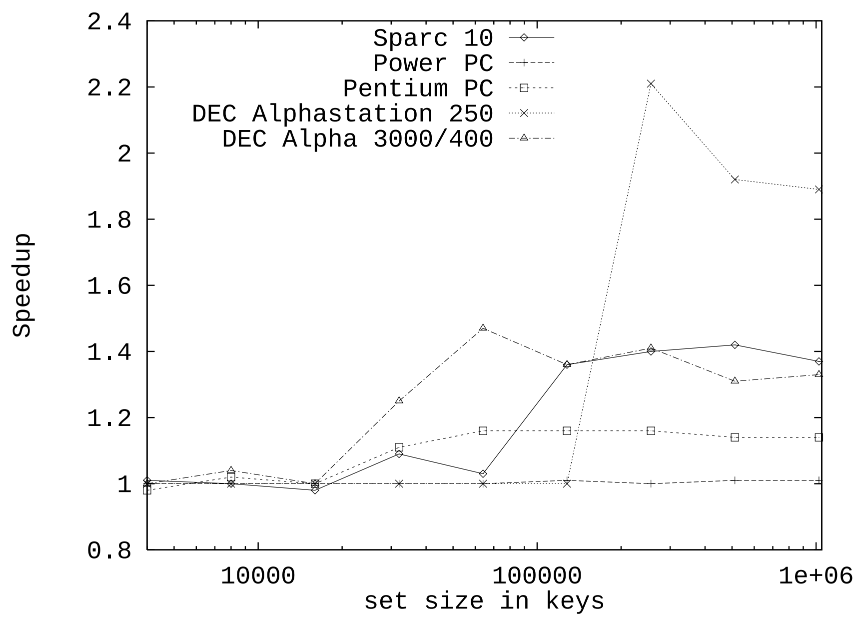

Figure 11: Speedup of tiled mergesort over base mergesort on five architectures.