The Influence of Caches on the Performance of Sorting[*]

TR # UW-CSE-96-10-01
Anthony LaMarca[†] & 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 heapsort, mergesort and quicksort 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 sorting algorithm, its relatively poor cache performance results in worse overall performance than the efficient comparison based sorting algorithms.

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, 7]. 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 which have been designed to minimize instruction count may not achieve the performance of algorithms which 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: heapsort [25], mergesort[1], quicksort [12], and radix sort[2]. Heapsort, mergesort, and quicksort 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 which enable us to predict the memory performance of these algorithms in terms of cache misses.

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 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 heap-

[*]Appears in the Proceedings of the *Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, January 1997.*

[†]LaMarca was supported by an AT&T fellowship. He is currently at Xerox PARC, 3333 Coyote Hill Road, Palo Alto CA 94304

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

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

sort 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.

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 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 it will be used 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]. Our design techniques will attempt to improve both the temporal and spatial locality of the 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 [23]. Atom is a toolkit developed by DEC for instrumenting program executables on Alpha workstations. Dynamic instruction counts are obtained by inserting an increment to an instruction 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 are less oblivious requiring more sophisticated techniques and approximations to accomplish the analysis. For example, in traditional heapsort the memory access pattern is very non-oblivious. We describe some of these techniques in Section 10.

4 Heapsort.

The heapsort algorithm first builds a heap containing all of the keys and then removes them all from the heap in sorted order [25]. With n keys, building the heap takes $O(n \log n)$ steps, and removing them in sorted order takes $O(n \log n)$ steps. In 1965 Floyd proposed an improved technique for building a heap with better average case performance and a worst case of $O(n)$ steps [8]. As a base heapsort algorithm, we follow the recommendations of algorithm textbooks and use a binary heap constructed using Floyd's method. In addition, we employ standard optimizations to reduce instruction count. The literature contains a number of optimizations that reduce the number of comparisons performed for both adds and removes [5, 2, 10], but in practice these do not improve performance and we do not include them in the base heapsort.

4.1 Memory Optimized Heapsort. 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 [25] for building a binary heap incurs fewer cache misses than Floyd's method. In addition we have shown [17, 16] that two other optimizations reduce the number of cache misses incurred by the remove-min operation. 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 are the size of a cache block. If d is relatively small, say 4 or 8, then 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]. The algorithm dynamically chooses between the repeated-adds method and Floyd's method for building a heap. If the heap is larger than the cache and repeated-adds can offer a reduction in cache misses, it is chosen over Floyd's method. We call this algorithm *memory-tuned heapsort.*

4.2 Performance of Heapsort. We now compare the performance of base heapsort and memory-tuned heapsort. Since the DEC Alphastation 250 has a 32 byte block size and our keys are 64 bits, there are 4 keys per block. As a consequence we choose $d = 4$ in our memory-tuned heapsort. The top row of figure 1 shows the performance of the variants of heapsort. The instruction count graph shows that memory-tuned heapsort executes fewer instructions than base heapsort ($d = 2$). The cache miss graphs shows that for large data sets the number of cache misses incurred by memory-tuned heapsort is less than half the misses incurred by base heapsort. Finally, the execution time shows that memory-tuned heapsort outperforms base heapsort for all sizes of data sets. Memory-tuned heapsort initially outperforms base heapsort due to lower instruction cost, and as the set size is increased the gap widens.

5 Mergesort.

For our base mergesort algorithm, we chose an iterative mergesort [15] to which a number of traditional optimizations can be applied. 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 small subarrays with a fast in-line sorting method, and loop unrolling. Our base mergesort algorithm has very low instruction count, executing fewer than half as many instructions as the base heapsort.

5.1 Memory Optimized Mergesort. In our study we employ two memory optimizations with 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 [26]. Tiled mergesort has two phases. To improve temporal locality, in the first phase subarrays of length $BC/2$ are sorted using mergesort, where B is the number of keys per cache block and C is the capacity of the cache in blocks. The second phase returns to the base mergesort to complete the sorting of the entire array.

Multi-mergesort adds an additional optimization to reduce the number of cache misses in the second phase of tiled mergesort. Multi-mergesort uses a multiway 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. 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.

5.2 Performance of Mergesort. The second row of figure 1 shows the performance of our mergesort variants. The instruction count graph shows that base mergesort and tiled mergesort essentially execute the same number of instructions while multi-mergesort has a much higher instruction count when the input set exceeds the cache capacity. The cache miss graph shows that both the base mergesort and the tiled mergesort make a sudden leap in cache misses when the size of the input array exceeds half the cache capacity. The multi-mergesort incurs slightly more than 1 miss per key regardless of the input size. The execution time graph shows that tiled mergesort and multi-mergesort perform much better than base mergesort and for the largest data sets multi-mergesort performs the best.

6 Quicksort.

Quicksort is an in-place divide-and-conquer sorting algorithm considered by many to be the fastest comparison-based sorting algorithm when the set of keys fits in memory [12]. Among the optimizations that Sedgewick suggests is one that sorts small subsets using a faster sorting method [21]. 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.

6.1 Memory Optimized Quicksort. Again, 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, using insertion sort, each small subarray when it is first encountered. While saving them all until the end makes sense from an instruction cost 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 $(1-x)^k$ [6, 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.

6.2 Performance of Quicksort. The third row of Figure 1 shows the performance of our quicksort variants. The instruction count graph shows that the number of instructions executed by the base quicksort is the least followed closely by memory-tuned quicksort. For large data sets multi-quicksort executes up to 20% more instructions than the memory-tuned quicksort. 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. The execution time graph shows that the execution times of the three quicksort algorithms are quite similar. However, this graph also 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.

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 [22]. 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" 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 [9] 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. The obvious memory optimization is to choose the radix which minimizes cache misses. As it happens, for the particular implementation used in our study, a radix of 16 bits minimizes both cache misses and instruction count on an Alphastation 250. In Section 10 we take a closer look at the cache misses incurred by radix sort.

7.1 Performance of Radix Sort. 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. The last row of figure 1 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. 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.

8 Performance Comparison.

To compare performance across sorting algorithms, Figure 2 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, while multi-mergesort has the least. For the largest data set radix sort has approximately 3 times as many cache misses as multi-mergesort or 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 and multi-mergesort ran 23% faster than radix sort yet performed almost four times as many instructions.

9 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 3 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 4 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 [11]. 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 [24]. 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.

10 Analysis of Cache Misses.

Cache misses cannot be analyzed precisely due to a number of factors such as context switching 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.

10.1 Analysis of Quicksort and Mergesort. The number of cache misses for our mergesort and quicksort variants are fairly easy to approximate because the memory reference patterns of these algorithms are fairly oblivious. As an example, for multi-mergesort let C be the cache capacity in cache blocks and let B be the number of keys per cache block. We assume that the number of keys, n, is such that $\lceil n/(BC/2) \rceil = k$ for some integer $k > 1$. Because the auxiliary data structures needed for merging and other parts of the program are small we assume they remain in the cache most of the time and have little effect on cache performance. In the first phase of multi-mergesort there are exactly two cache misses for every B keys, one miss in the source array and one in the destination array. In the second phase there are again two cache misses for every B keys, one in the source array of the k-way merge and one in the destination. In addition, there may be an additional miss per every B keys because the keys may need to be copied. This extra copy happens if $\log_2(BC/2)$ is even. Thus, the total number of misses per key is either 4 or 5 misses for every B keys, depending on the parity of $\log_2(BC/2)$. In the case of our cache performance study, $B = 4$ and $C = 2^{16}$. Thus, we calculate 4 misses for every 4 keys which is 1 miss per key. This corresponds almost exactly to the number of misses per key reported in the cache miss graph of multi-mergesort in figure 1. Naturally, if k were very large then we cannot assume that the k-way merging data structure has little effect on memory performance. This case would be much more difficult to analyze, but is probably not relevant in practice because this case would only arise if n were at least the order of $(BC)^2$. The analysis of the other versions of mergesort and of the versions of quicksort can be done in a similar way.

10.2 Analysis of Radix Sort. The approximate cache miss analysis of radix sort is more complicated. Let n be the number of keys, b be the number of bits per key, r be the radix, B be the number of keys per block, A be the number of counts per block, and C the capacity of the cache in blocks. Assume that the size of the two count arrays is less than the cache capacity, $2^{r+1} \leq AC$, and that the number of keys is larger than the cache capacity, $n > BC$. Although it is not necessarily true, we assume that in the first iteration every block in one of the count arrays is accessed at least once every BC visits to the input array, in the last iteration every block in one of the count arrays is accessed at least once every $BC/2$ visits to the source array, and in each of the other iterations every block in the two count arrays is accessed at least once every $BC/2$ visits of the source array. In the first iteration of radix sort n keys are visited in the input array. This results in n/B cache misses. In addition every BC visits flushes the count array used to accumulate counts. Hence, there are an additional $(n/BC)(2^r/A)$ cache misses. In the last iteration, n keys are visited in each of the source and destination arrays resulting in $2n/B$ cache misses. Also in the last iteration, one of the two count arrays is flushed on average every $BC/2$ accesses yielding an additional $(n/(BC/2))(2^r/A)$ cache misses. In the remaining iterations n keys are visited in each of the source array and destination array. This results in $2(\lceil b/r \rceil - 1)(n/B)$ misses. In all these iterations, on average, every $BC/2$ visits flushes both the count arrays. This leads to an additional $(n/(BC/2))(\lceil b/r \rceil - 1)(2^{r+1}/A)$ misses. In all, the total number of misses per key is approximated by

$$\frac{2\lceil \frac{b}{r} \rceil + 1}{B} + \frac{2^r(4\lceil \frac{b}{r} \rceil - 1)}{ABC}.$$

Figure 5 compares this approximation with the actual number of cache misses incurred per key by radix sort for $n = 4,096,000$, $b = 64$, $A = 8$, $B = 4$, and $C = 2^{16}$. Both the analytic approximation and the simulation agree that the radix that minimizes cache misses is 16.

Our approximation loses accuracy as r grows. There are at least two sources of the inaccuracy. The first is our assumption that each block in a count array is accessed at least once every M steps for some M. In reality, the probability that a block will not be accessed at least once in M steps is $(1 - 1/S)^M$ where S is the number blocks in a count array. In our example shown in the last row of figure 1, $S = 2^{16}/8 = 2^{13}$ and $M = 2^{16} \cdot 4 = 2^{18}$ so that the probability that a block will not be accessed is $(1 - 1/2^{13})^{2^{18}} \approx e^{-32}$. Hence, this is not a significant source of inaccuracy. A second and significant source of inaccuracy is the unmodeled interference among accesses to the destination array and count arrays. In reality, the destination array in all the iterations but the first has 2^r pointers into it. Each pointer traverses left-to-right from its original position to the original position of the next pointer at a rate proportional to the original distance between them. If r is large enough then these pointer accesses can interfere significantly with each other and with accesses to the count arrays causing "conflict misses." We believe that this is the major source of inaccuracy in our analysis.

10.3 Analysis of Heapsort. In a previous paper [17] we introduced *collective analysis* which enabled us to approximate the cache performance of heaps. In that paper we analyzed the cache performance of d-heaps in the hold model which models the priority queue in a discrete event simulator. That analysis serves as the basis for an approximate analysis of our heapsort variants. In collective analysis of an algorithm we divide the cache into a set of regions R and the algorithm's memory accesses into a set of independent stochastic processes P, where a process is intended to approximate the memory behavior of an algorithm or part of an algorithm. Areas of the virtual address space accessed in a uniform way should be represented with a single process. Collectively, all of the processes represent the accesses to the entire virtual address space and hence represent the algorithm's overall memory behavior. Collective analysis assumes that the references to memory satisfy the *independent reference assumption* [4]. In this model each access is independent of all previous accesses, that is, the system is memoryless.

If λ_{ij} is the access intensity of process j in cache region i then by extending a result of Rao [20] it can be shown [17] that the hit intensity of the algorithm is

$$\eta = \sum_{i \in R} \frac{1}{\lambda_i} \sum_{j \in P} \lambda_{ij}^2$$

and the miss intensity is $\lambda - \eta$ where $\lambda_i = \sum_{j \in P} \lambda_{ij}$ and $\lambda = \sum_{i \in R} \lambda_i$.

Memory accesses in the d-heap can be approximately modeled using collective analysis [17]. In particular, the cache can be divided into regions corresponding to levels of the d-heap that are smaller than the cache. The memory accesses can be divided into processes that correspond to accesses to the levels that are smaller than the cache and to parts of a level that are equal to the cache size.

Assume there are n keys, C cache blocks, and each block holds B keys. In the building a heap phase of heapsort there are at least n/B misses incurred in traversing the array. In addition, there are more misses incurred during the insertions into the d-heap. Because we use William's repeated-adds algorithm [25] we pessimistically assume that all adds percolate to the root and that the most recent leaf-to-root path is in the cache. In this case, the probability that the next leaf is in the cache is $1 - 1/B$, the parent of that leaf is in the cache is $1 - 1/(dB)$, the grandparent of that leaf is in the cache is $1 - 1/(d^2B)$, and so on. We divide the removes 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. In figure 6 we measured the results of heapsort using the traditional binary heap (base heapsort) and the 4-heap (memory-tuned heapsort) on a DEC Alphastation 250 with a 2 MB cache. Exactly four 64-bit keys fit in a cache block. We are somewhat surprised how well this analysis matches the measured results considering the assumptions we made in the approximation.

11 Conclusions.

This paper explores 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. 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 5 and 6 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.

Acknowledgement.

We thank Jim Fix and Mike Salisbury for carefully reading preliminary versions of this paper and for making many useful suggestions for improvements.

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] E. Coffman and P. Denning. *Operating Systems Theory*. Prentice–Hall, Englewood Cliffs, NJ, 1973.

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

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

[7] 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.

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

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

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

[11] J. Hennesey and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufman Publishers, Inc., San Mateo, CA, 1990.

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

[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–Wesely, 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, Feb 1996. UW department of computer science technical report # 96-02-03, to appear in *Journal of Experimental Algorithmics*.

[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] G. Rao. Performance analysis of cache memories. *Journal of the ACM*, 25(3):378–395, 1978.

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

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

[23] 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.

[24] 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.

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

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

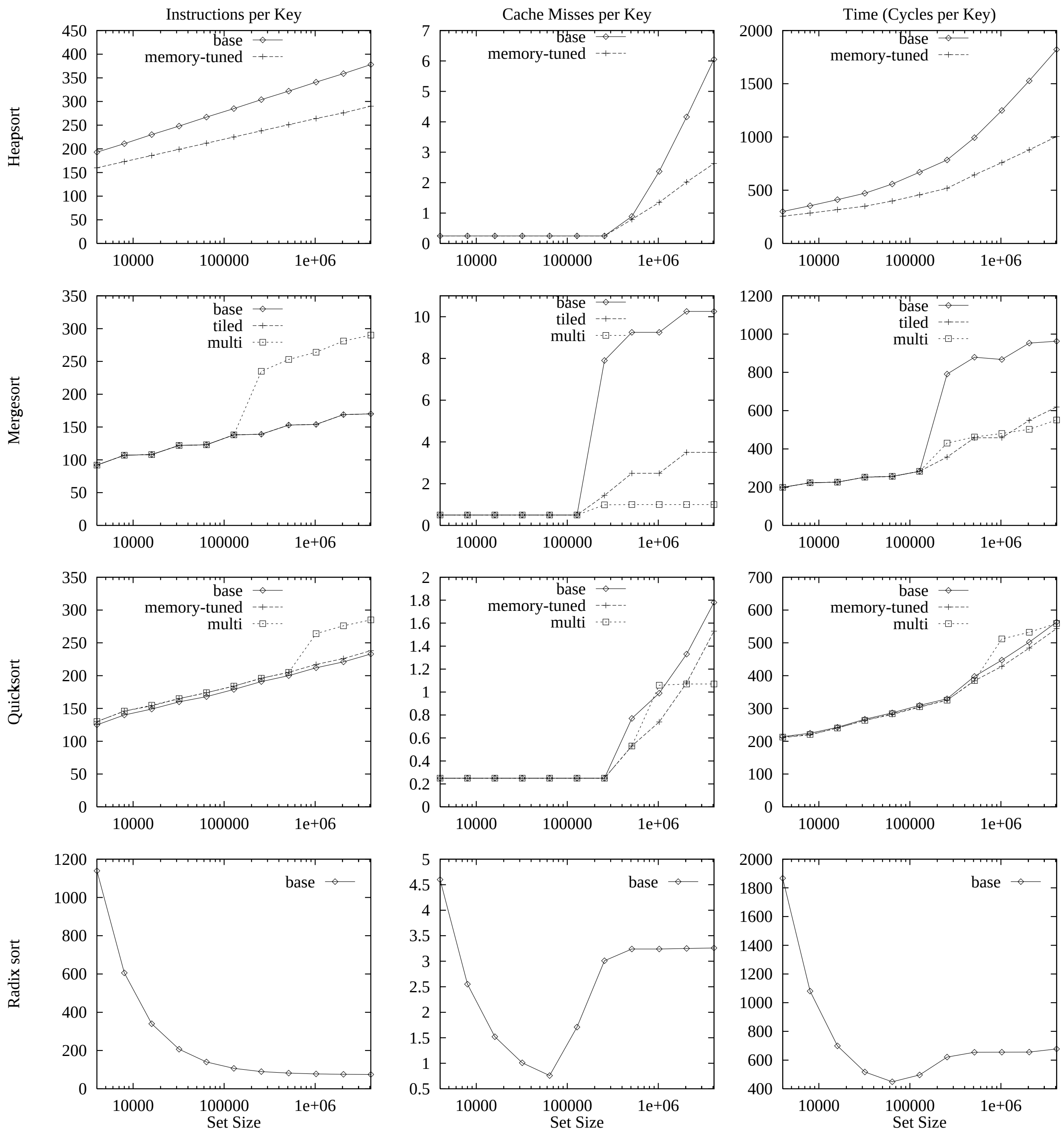

Figure 1: Performance of heapsort, mergesort, quicksort and radix sort on sets between 4,000 and 4,096,000 keys. The first column of graphs shows instruction counts per key, the second column shows cache misses per key and the third column shows execution times per key. Executions were run on a DEC Alphastation 250 and simulated cache capacity is 2 megabytes with a 32 byte block size.

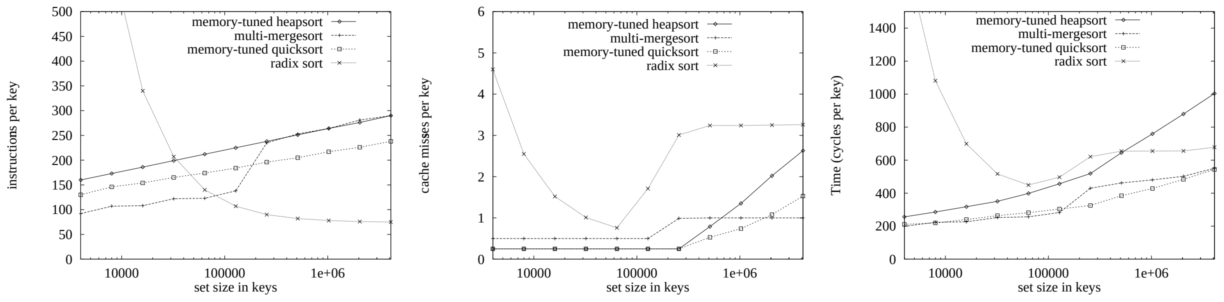

Figure 2: Instruction count, cache misses and execution time per key for the best heapsort, mergesort, quicksort and radix sort on a DEC Alphastation 250. Simulated cache capacity is 2 megabytes, block size is 32 bytes.

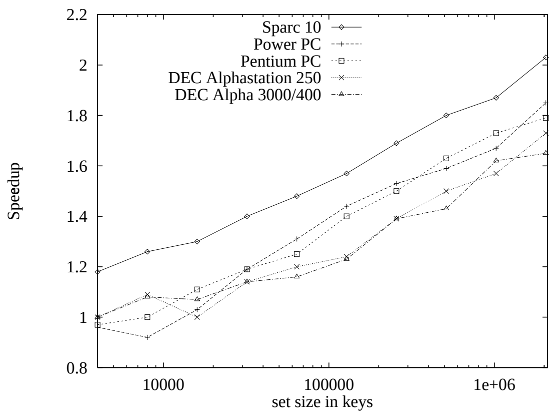

Figure 3: Speedup of memory-tuned heapsort over base heapsort on five architectures.

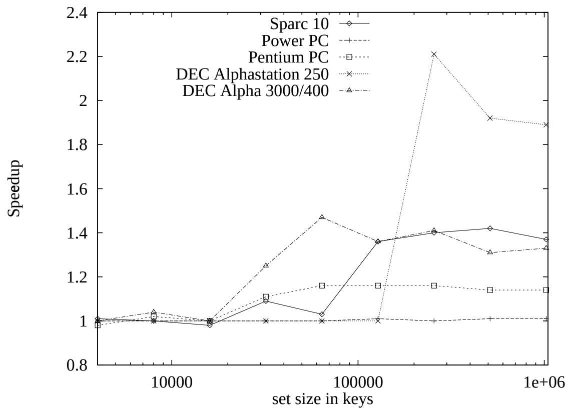

Figure 4: Speedup of tiled mergesort over base mergesort on five architectures.

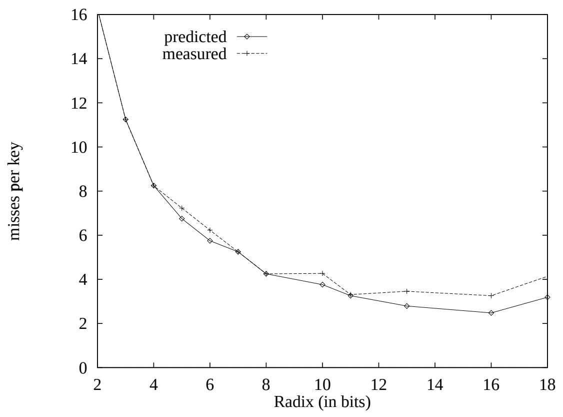

Figure 5: Cache misses incurred by radix sort, measured versus predicted.

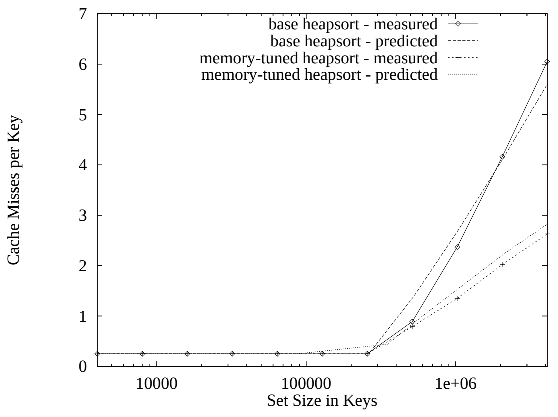

Figure 6: Cache misses incurred by heapsort, measured versus predicted.