Efficient Sorting Using Registers and Caches

Rajiv Wickremesinghe <rajiv@cs.duke.edu>
Lars Arge, Jeff Chase, Jeffrey Scott Vitter
Duke University

Modern computer systems have increasingly complex memory systems. Common machine models for algorithm analysis do not reflect many of the features of these systems, e.g., large register sets, lockup-free caches, cache hierarchies, associativity, cache line fetching, and streaming behavior. Inadequate models lead to poor algorithmic choices and an incomplete understanding of algorithm behavior on real machines.

A key step toward developing better models is to quantify the performance effects of features not reflected in the models. This paper explores the effect of memory system features on sorting performance. We introduce a new cache-conscious sorting algorithm, R-MERGE, which achieves better performance in practice over algorithms that are superior in the theoretical models. R-MERGE is designed to minimize memory stall cycles rather than cache misses by considering features common to many system designs.

1. INTRODUCTION

Program performance on today's computer systems is largely determined by interactions with the memory system. While new compiler techniques can help to improve a program's memory system behavior, algorithmic choices play a key role in determining memory access patterns for data-intensive computation. Despite recent progress, development of software that can perform well on on modern machines is hampered by a lack of adequate models to guide "memory-friendly" algorithm design.

Algorithm designers and programmers have traditionally used the *random access machine* (RAM) model to guide expectations of algorithm performance. The RAM model assumes a single processor with an infinite memory having constant access

This work was supported in part by the National Science Foundation through ESS grant EIA–9870724, RI grant EIA–9972879, and CAREER grant EIA–9984099, and by the U.S. Army Research Office through MURI grant DAAH04–96–1–0013.
An earlier version of this paper appeared in the *Proceedings of the Workshop on Algorithm Engineering* [Arge et al. 2000]
Address: Department of Computer Science, Duke University, Box 90129, Durham, NC 27708, U.S.A.

cost. Unfortunately, the RAM model and its variants are increasingly inadequate to guide algorithmic and implementation choices. It is well-understood that the RAM model does not reflect the complexity of modern machines, which have a hierarchy of successively slower and larger memories. Higher levels of the memory hierarchy are managed as caches to approximate the illusion of a large, fast memory.

There has been a great deal of recent progress in devising new models that more accurately reflect memory system behavior [LaMarca and Ladner 1997; Ladner et al. 1999; Rahman and Raman 1999; Rahman and Raman 2000; Sen and Chatterjee 2000; Sanders 1999a]. These models are largely inspired by the theory of *I/O Efficient Algorithms* (also known as *External Memory Algorithms*), based on the I/O model of Aggarwal and Vitter [1988]. The I/O model measures the complexity of an algorithm not by the number of processing steps or instructions, but by the number of I/O operations required. The I/O model yields significant improvements for I/O-bound problems, which spend most of their time transferring data between levels of the hierarchy (memory and disk) with the greatest performance difference.

One approach to cache-conscious algorithm design applies the I/O model to higher levels of the memory hierarchy, in order to optimize data-intensive computations in internal memory that do not require I/O [LaMarca and Ladner 1997; Rahman and Raman 1999]. Direct adaptation of the I/O model to caches leads to algorithms designed to minimize the number of *cache misses*, i.e., accesses to main memory.

As the gap between processor and memory cycle times continues to widen, cache-conscious or memory-friendly algorithm design is increasingly important for program performance. At the same time, increasingly complex memory system architectures arise to mask the memory access time gap, making it even more difficult to devise simple models that are predictive of algorithm performance on real machines. The complexity of memory systems (see Section 2.1), including cache hierarchies, prefetching, multiple- and delayed- issue processors, lockup-free caches, write buffers and TLBs make it difficult to analyze different approaches theoretically. Thus empirical data is needed to guide the development of new models.

This paper makes three contributions. First, we outline key factors affecting memory system performance, and explore how they impact algorithmic and implementation choices for a specific problem: sorting. Second, we present quantitative data from several memory-friendly sort implementations on Alpha 21x64 CPUs in order to quantify the effects of these choices. These results also apply qualitatively to other architectures including Intel Pentium and HP PA-RISC. Finally, we show how a broader view of memory system behavior leads to a sorting implementation that is more efficient than previous approaches on the Alpha CPU, which is representative of modern processors. This sorting approach, R-MERGE, is a hybrid mergesort/quicksort algorithm that benefits from optimizations that would not be considered under a simple model of cache behavior, showing that a straightforward extension of the I/O model is inadequate for memory systems.

We generalize from our experience to propose principles for designing and implementing memory-friendly algorithms. In particular, R-MERGE considers the *duration* of the processing delays (stalls) resulting from cache misses, rather than merely minimizing the number of cache misses; these delays vary significantly due to streaming behavior and other factors. R-MERGE also shows the potential of efficient register usage in algorithm design.

This paper is organized as follows. Section 2 briefly reviews memory system structure, discusses the problems associated with modeling the memory system, and introduces sorting as an illustrative application. Section 3 discusses principles for memory-friendly algorithm design and implementation, and describes our sort implementations. Section 4 presents experimental results. Section 5 sets our approach in context with related work. We conclude in Section 6.

2. BACKGROUND

2.1 Modeling Memory Systems

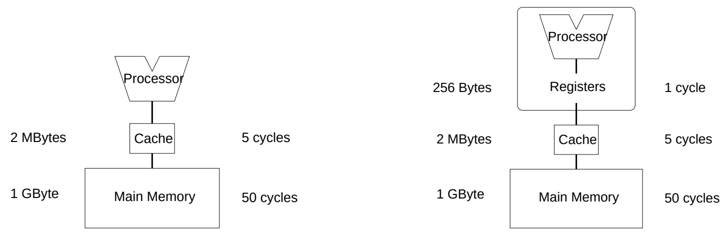
Modern memory systems have been undergoing a constant process of increasing sophistication in an effort to bridge the speed differential between faster processors and main memory. Current systems typically include one or more levels of *cache* memory arranged in a hierarchy. Although caches are much faster than main memory, they can only store a small amount of useful data. Furthermore, they are automatically managed by the memory system; the program can only indirectly control what is stored in the caches. Processors also have access to a limited number of *registers* that store individual data items for access in a single CPU cycle; data stored in registers is immediately available for computation.

If the memory system is unable to provide the data required by the processor in a timely manner, *stalls* will reduce the overall performance of the program by increasing the time spent idling, waiting for data. Much work has been done in the realm of computer architecture in an effort to reduce the delays caused by stalls. Unfortunately, these innovations also complicate the analysis of programs. For example, the *write buffer* reduces the number of write-stalls the processor suffers; it allows the processor to continue immediately after the data is written to the buffer, avoiding the cache-miss penalty. Because main memory is virtually addressed, a *translation lookaside buffer* (TLB) provides a small cache of address translations or hardware lookup table mapping virtual addresses to physical memory.

The above and other hardware optimizations are motivated by, and exploit, the locality of references made by programs. Accessing data in sequence (good spatial locality) makes the good use of these resources.

A simple memory system consisting of a single cache and main memory (as illustrated in Figure 1(a)) is similar to the system described by the I/O model of Aggarwal and Vitter [1988]. This model describes a system with a large, slow memory (disks) and a limited fast memory (main memory). The fast memory is *fully associative*; any item of data may be placed at any location in the memory. An *I/O operation* transfers a fixed size *block* of data to or from memory. Block transfers amortize the cost of a data transfer because programs tend to access data close to already accessed data; I/O efficient programs, in particular, have good *locality.* Computation is performed on data that is in fast memory. Computation time is usually not taken into account because the I/O time dominates the total time. Algorithms are analyzed in this model in terms of number of I/O operations.

Others have proposed cache models [LaMarca and Ladner 1997; Rahman and Raman 1999; Sen and Chatterjee 2000] based on the I/O model. The cache is analogous to the (fast) main memory in the I/O model. Elements are brought into



(a) Simple memory system

(b) Memory system showing registers

Fig. 1. Diagram of a memory system showing main memory, cache, and registers. Representative sizes and access times for the components are indicated.

the cache in blocks of several words to make use of spatial locality. Main memory takes the place of disks, and is assumed to contain all the data. The basic cache model defines the following parameters:

N	$=$	number of elements in the problem instance
B	$=$	number of elements in a cache block
C	$=$	number of blocks in the cache

The capacity of the cache is thus $M = BC$ elements. When a cache miss occurs because data was not found in the cache, a block is transfered from memory to the cache. The I/O model is not directly applicable to caches because of several important differences:

—Computation and main memory access cost can be ignored in the I/O model because the I/O cost dominates. The access times of caches and main memory are much closer, so the time spent processing becomes significant.

—Modern processors can have multiple outstanding memory operations, and may execute instructions out of order. This means that a cache miss does not increase the execution time provided other instructions are able to execute while the miss is handled. Memory accesses overlapped with computation do not contribute toward total execution time.

—Registers may store the most frequently accessed data. Under proper control, they act as a small, fast cache. Figure 1(b) depicts a memory system with registers.

—The size of the TLB limits the number of different sections of main memory that can be accessed together. Algorithms that access too many streams can cause *TLB thrashing* [Rahman and Raman 1999].

—Caches have low associativity, so that data from a particular memory location can be cached in only one (or a few), cache locations. This restricts how programs use the cache, since the cache location is determined by the data element's address. There are no special instructions to control placement of data in the cache. In comparison, the I/O model assumes a fully-associative main memory.

—Memory systems are optimized for sequential accesses. An algorithm that performs sequential (streaming) accesses has an advantage over another that accesses the blocks randomly, even if both have the same total number of accesses or misses. Although I/O systems also optimize sequential access, the I/O model does not need to take this into account because the block size is typically sufficiently large to hide access delays. In the case of the cache, however, the block size is hardware-dependent (and small).

In addition to these differences between I/O and cache models, the basic cache model considers only one cache; in practice there is a hierarchy of caches.

Although there are many shortcomings of the basic cache model, it has proved useful in guiding the design of memory-friendly algorithms. In the following section, we describe the sorting problem, and the implications of using the cache model in analyzing sorting algorithms.

2.2 Sorting

We consider the common problem of ordering a collection of values. In particular, we consider sorting N words (64-bit or 32-bit integers depending on machine architecture) for values of N up to the maximum that fit in main memory while allowing space for the result. The complexity is $\Omega(N \log N)$ comparisons (in the RAM model) and $\Omega((N/B)\log(N/B)/\log C)$ cache misses (in the cache model) [Knuth 1998; Aggarwal and Vitter 1988; Sen and Chatterjee 2000]. There has been much recent work on cache-conscious sorting algorithms using the basic cache model [LaMarca and Ladner 1997; Rahman and Raman 1999; Ranade et al. 2000]. The work of LaMarca and Ladner [1997] is a common reference point for comparison. They explored several algorithms designed to minimize the number of cache misses, as guided by the cache model.

—Mergesort is an iterative version of the traditional binary mergesort. Mergesort initially forms $\Theta(N)$ sorted runs of constant size, and then makes $\Theta(\log N)$ merge-passes over the data forming successively larger runs (the runsize at each pass doubles until all keys are in a single run). Each pass touches all the data, resulting in many cache misses.

—Tiled mergesort is a tiled version of the iterative binary mergesort. Cache loads (or tiles) are sorted using mergesort, and then merged using an iterative binary merge routine. Tiling increases locality, and improves the cache performance of mergesort, but is effective only over the initial $\Theta(\log(BC))$ passes, when mergesort is operating in-cache.

—Multi-mergesort uses the iterative binary mergesort to form cache-sized runs (as in tiled mergesort), and a single merge or order $\Theta(N/BC)$ to complete the sort. The merge of large order reduces cache misses as compared to tiled mergesort by reducing the number of passes over the data. We confirm the prediction of LaMarca and Ladner [1997] that multi-mergesort is faster than tiled mergesort for large input sizes. However, the results in Section 4 show that this is not always the case.

—Quicksort is a memory-tuned version of the traditional quicksort. It partitions the data around a pivot to form two sets; partitioning is repeated until the sets formed are constant-sized and can be trivially sorted. QUICKSORT has excellent cache performance within each run, and is most efficient in instructions executed (among comparison-based sorts) because of its efficient inner loops. However, QUICKSORT makes $\Theta(\log N)$ passes, and thus incurs many cache misses. Memory-tuned QUICKSORT uses INSERTION SORT to sort small subsets while they are in the cache, instead of partitioning further. This increases the instruction count, but reduces the cache misses.

—DISTRIBUTION SORT (multi-QUICKSORT) uses a single large multi-way partition pass to create subsets that are likely to be cache-sized. Unlike QUICKSORT, it is not in-place, and also has a higher instruction overhead. However, cache performance is much improved. This is similar to the FLASHSORT variant of Rahman and Raman [1999].

The design of these algorithms was guided by the basic cache model, and thus minimizes the total number of cache misses. We find that the cache model often leads to sub-optimal algorithm design choices, and the following section discusses how we improved the performance of our algorithms.

3. IMPROVED APPROACH

Using cache misses as a metric can give misleading results because only the time spent *waiting* for the memory system (memory stalls) contributes to the total running time. The number of cache misses alone does not determine the stall time of a program. A program that spends many cycles computing on each datum may have very few memory stalls, whereas a program that just touches data may be dominated by stalls. We are interested in the total running time of the program which is dependent on *both* instructions executed and duration of memory system *stalls*–not the number of misses. A more realistic measure of time takes into account the memory stall time and the actual execution time [Hennessy and Patterson 1995]:

$$\text{time} = (\text{CPU execution cycles} + \text{memory stall cycles}) \times \text{clock cycle time}$$

Memory stalls are hard to model because of the many factors affecting memory system performance, as discussed in Section 2.1. We experiment with hybrid sorting algorithms to consider the role of these factors in a simple abstract model. Our designs are guided by three principles:

(1) The algorithm should be instruction-conscious *and* cache-conscious: we need to balance the sometimes-conflicting demands of reducing instructions executed, and reducing memory stalls.

(2) Memory is not uniform; in particular, sequential accesses are less costly than random accesses.

(3) Effective use of registers can reduce instruction counts as well as memory accesses.

This leads us to several design choices for sorting:

—Once data is in the cache, use the most instruction-efficient method to sort it. Although QUICKSORT is not cache-efficient, it does the least work per key. QUICKSORT also operates in place, making full use of the cache; no space is wasted in storing the result. This means that it is advantageous to sort cache-sized loads using QUICKSORT rather than a MERGESORT variant.

—The streaming behavior of memory systems favours algorithms that access the data sequentially, e.g., merging and distribution (multi-way partitioning). Merging has an overall computational advantage over distribution because layout is not dependent on the ordering of the input data. This eliminates the extra pass needed by distribution methods to produce sequential output, and more than offsets the extra work required to maintain a dynamic merge heap, compared to a fixed search tree.

—The third key to good performance is limiting the size of data structures. For merge sorting, this constrains the merge order. Although this is contrary to the basic cache model, it has several advantages that increase overall performance: (a) it is possible to effectively use registers to store small structures; (b) interference or conflict misses due to low associativity in the caches are reduced; and (c) TLB faults are reduced.

Our decision to bound the merge order (k) enables the efficient use of registers. In merge-based sorts, the merge heap is accessed multiple times for every key, and is relatively small. Storing the heap in registers reduces the number of instructions needed to access it, and eliminates the need to load the heap elements from cache memory. It also avoids cache interference misses (between the heap and data streams), although this effect is small at small merge order. Limited merge order also reduces cache interference misses between data streams [Sen and Chatterjee 2000]. It has also been shown that TLB thrashing can significantly degrade performance [Rahman and Raman 1999]. Limiting the merge order concentrates memory accesses within the small set of pages whose mappings can be stored in the TLB.

The small order merge executes fewer instructions, but it also causes more cache misses. Because the processor is not stalling while accessing memory, the increase in misses does not lead to an increase in memory stalls, and so the overall time is decreased. The number of instructions executed in merging is proportional to $N\times$ (average cost of heap operation) $\times \log_k N$. If the cost of a heap operation is $\log k$, we obtain the familiar $N \log N$ bound. However, we find that the average cost of a heap operation averaged over the $\log k$ levels of the heap is less if the heap is smaller because of the factors mentioned above. This favors smaller merge orders.

To investigate the impact of these factors, we developed the HYBRID-MERGE sorting algorithm outlined in Figure 2.

Applying our design principles we created several sorting implementations:

—Q-MERGE implements HYBRID-MERGE using an array to store the merge heap. We use this version to contrast the effect of using registers.

—R-MERGE also implements HYBRID-MERGE but uses a small merge order allowing the compiler to assign registers for key data items (the merge heap in particular).

—R-DISTRIBUTION is a limited-order version of DISTRIBUTION with an additional counting pass that determines the length of each resulting stream in advance. This greatly simplifies the distribution by eliminating bounds checks and the requirement for variable-length result streams. R-DISTRIBUTION fits the key data structures—the pivots that define the substreams, and the counters that indicate the length of each substream—in registers by limiting the distribute order.

Hybrid merge sort (merge order = k):

(1) Determine number of merge passes, $p = \lceil \log_k(N/BC) \rceil$.
(2) Form initial runs of length approximately BC elements using in-cache QUICKSORT.
(3) Merge sets of k runs together to form new runs.
(4) Repeat merge step on new runs until all elements are merged into a single run.

The merge itself is based on the *selection tree* algorithms from Knuth [1998].

(1) Take the first element from each of the k runs and make a heap. Each element of the heap consists of the value, `val`, and pointer to the next element of the run, `nextptr`.
(2) Insert a sentinel (with key value $+\infty$) after the last element of each run. The sentinel eliminates end of run checks in the inner loop. Fortunately, the previous step removes items from the runs at just the right places to insert the sentinels, so no rearrangement is required.
(3) Repeat N times: output the min; obtain the next item in same run and replace the min; heapify. We halve the number of calls to heapify by not maintaining the heap property between the 'extract min' and insert.

```
*output++ ← min.val
min.val ← *min.nextptr++
heapify()
```

Fig. 2. Outline of HYBRID-MERGE sorting algorithm

We also evaluated a traditional heap implementation. This version, probably similar to the one used by Ranade, Kothari, and Udupa [2000], was not competitive because of the additional index computations. We focus our efforts on merge-sorting because we find it to be at least 10% faster than distribution.

3.1 Getting the right program

The speed differences between the processor, registers and cache are usually no more than 5-10 cycles. Although this translates into a proportional performance gain, we found that it is very easy to inadvertently obscure the effect of a better algorithm. Our implementations attempt to be as efficient as possible while not resorting to non-portable optimizations.

We optimized the core merge routine at the source level. The heap operations constitute a major part of the inner loops of our programs. A key design decision was using a custom merge routine for each merge order; a program generates C source for these routines from a specification.

Although most modern processors are capable of executing multiple instructions in parallel, data and control dependencies slow program execution by limiting the degree of parallelism. We structure the code to reduce dependencies as much as possible (for example, by avoiding extraneous checking within the inner loop). However, comparison based sorting inherently must perform $\Theta(N \log N)$ sequential comparisons. (Successful branch prediction and other speculative techniques could perform some comparisons in parallel).

4. EXPERIMENTS

This section presents empirical results to validate the ideas presented in the previous sections. To focus on memory system and CPU performance, all experiments sort datasets that fit in the physical memory of the machine, with no I/O. Dataset sizes range from 2 million up to 40 million keys.

Most experiments ran on a Digital Personal Workstation 500au with a 500MHz Alpha 21164 CPU with a 2MB cache.[1] We obtained qualitatively similar results on several other architectures including: (1) Alpha 21264 (500MHz) with a 4MB cache, (2) Alpha 21214A (467MHz) with a 2MB cache, (3) Intel Pentium III (500MHz) with a 512K cache, (3) HP PA-8200 (240MHz) with a 2MB cache.

We performed cache simulations and measured the actual number of executed instructions by instrumenting the executables with ATOM [Srivastava and Eustace 1994]. The hardware counters of the Alpha 21164 and `dcpi` [Anderson et al. 1997] provided information on memory stalls; `dcpi` also proved invaluable in better understanding the cause of stalls. All the programs were written in the `C` programming language, and compiled with the vendor's compiler with optimizations enabled.

The following sections examine different aspects of the algorithms' performance. Section 4.1 compares with previous algorithms. Sections 4.2 and 4.3 examine the run-formation and merge phases separately, and Section 4.4 considers the complete mergesort.

4.1 Comparison with previous algorithms

We compare our programs (R-DISTRIBUTION, Q-MERGE, R-MERGE) described in Section 3 with the original cache-conscious algorithm implementations (MULTI-MERGESORT, DISTRIBUTION, QUICKSORT) of LaMarca and Ladner [1997] described in Section 2.2. Because their fastest program was QUICKSORT, we made several minor improvements to make our comparisons more conservative. In particular, we simplified their code to rely more on compiler optimization (e.g. *removing* programmer-unrolled loops), yielding better performance.

Figure 3 shows comparative sorting performance in microseconds per key. We tested a range of input sizes requiring up to a gigabyte of main memory. The speedup of R-MERGE and Q-MERGE over MULTI-MERGESORT is greater than 2. The speedup of R-DISTRIBUTION over DISTRIBUTION is 1.3. Comparing R-MERGE with QUICKSORT, the fastest sort from [LaMarca and Ladner 1997], our speedup ranges from 1.10 to 1.36. The results also confirm the supposition of LaMarca and Ladner [1997] that although MULTI-MERGESORT was relatively slow for small input sizes (less than 2×2^{20} keys) it is better than TILED MERGESORT for large inputs.

Experiments on an Alpha 21264 yield similar results. QUICKSORT on a PA-RISC does relatively better; it is only a little slower than R-MERGE and Q-MERGE. Figure 4 shows that similar speedups can be obtained on a Pentium.

[1]The Alpha 21164 is a four-way superscalar processor with two integer pipelines. The memory subsystem consists of a two-level on-chip data cache and an off-chip L3 cache. It has 40 registers (31 usable) with a word size of 8 bytes. The memory subsystem does not block on cache misses, and has a 6-entry $\times$ 32-byte block write buffer that can merge at the peak store issue rate. The L1 cache is 8K direct-mapped, write-through. The L2 cache is 96KB, 3-way associative, write-back. The L3 cache (which is the only one we directly consider in this paper) is 2MB ($BC = 256$K), direct-mapped, write-back, with 64-byte blocks ($B = 8$). The access time for the off-chip cache is four CPU cycles. Machine words are 64-bits (8-bytes) wide. The TLB has 64 entries [Edmondson et al. 1995].

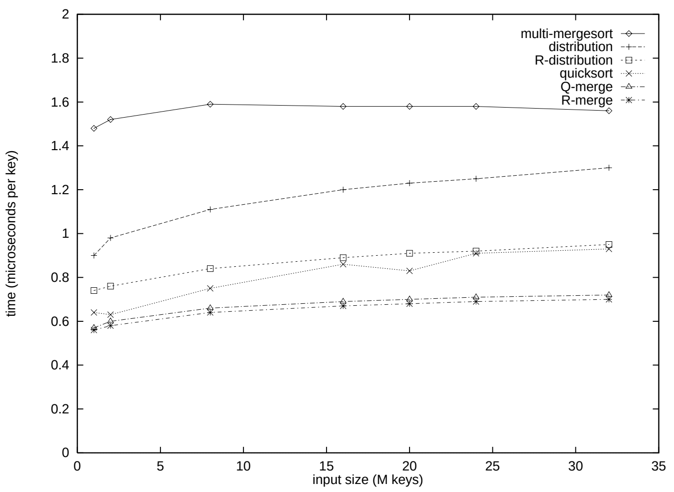


Fig. 3. Comparison with cache-conscious sort programs of LaMarca and Ladner [1997]. Time per key (microseconds/key) vs. number of keys ($\times 2^{20}$ keys). These results were obtained on an Alpha 21164; the Alpha 21264 produces a similar graph. Our MERGE programs were run with a merge order of 7, and use two merge passes.

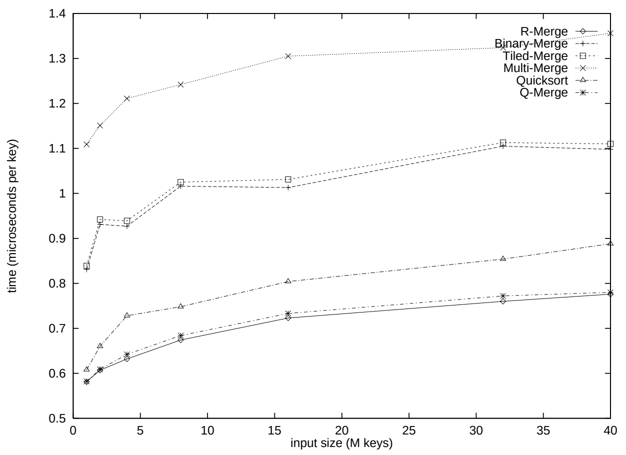
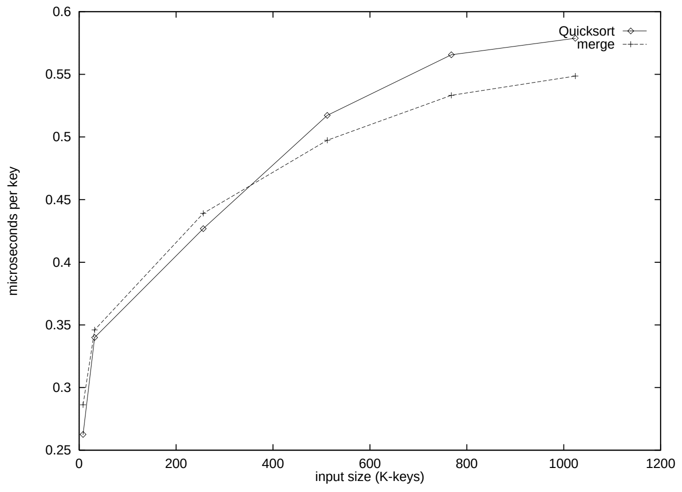
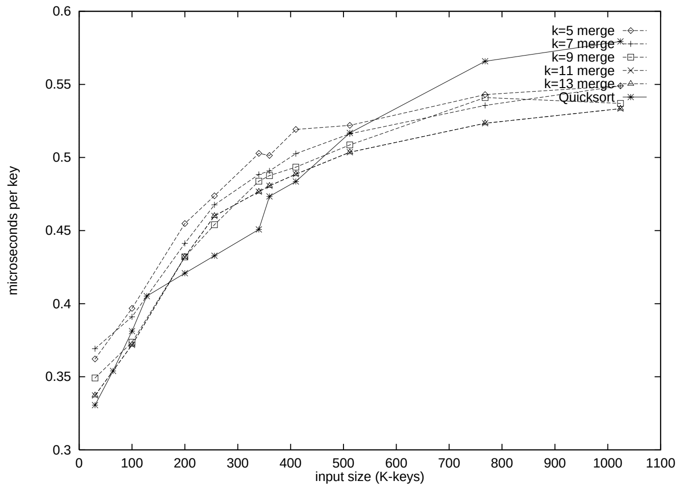


Fig. 4. Results on an Intel Pentium 500MHz with 512KB cache. Time per key (μs) vs. number of keys ($\times 2^{20}$). Element size is 32 bits for these experiments.



(a) Comparison between QUICKSORT and merging for run formation.



(b) Effect of additional passes of R-MERGE with merge order k.

Fig. 5. Total time per key (microseconds) for the run-formation phase, as a function of run size. Note: Experimental overhead (and thus the margin of error) is more significant at lower run sizes.

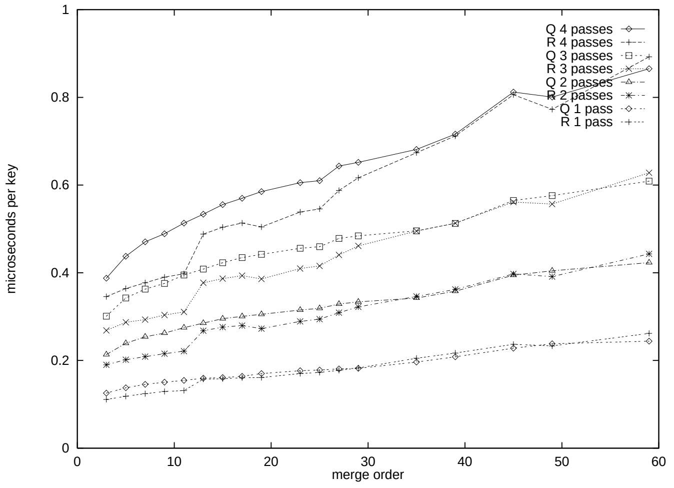


Fig. 6. Total time per key (microseconds) for the merging phase of R-Merge and Q-Merge, varying the merge order.

4.2 Run formation

Merging uses the cache more efficiently than Quicksort, but it executes more instructions for a given problem size. Thus Quicksort is a better choice for sorting runs that fit in cache.

Figure 5(a) compares a merge-based approach, internal merge, to Quicksort, illustrating this tradeoff. The cache size in this experiment is 2^{18} keys (= 256 K keys). The routines used by LaMarca and Ladner [1997] are slower than Quicksort and the merge-based approach by at least a factor of 2, so we do not include them here. Quicksort is faster for runs smaller than the cache, while merge-sorting is faster for runs larger than the cache. However, this tradeoff is affected by the implementation details and parameterization of the algorithms: Figure 5(b) shows that the break-even point between Quicksort and the merge order of internal merge. An additional merge pass on a cache-sized run (resulting in more, smaller quicksort runs) reduces the total time in some situations.

Other experiments have shown the benefit of using Quicksort to form runs much smaller than the cache size. The above experiments indicate that Quicksort performs well for sorting cacheloads, although it may not be the clear winner in all situations.

4.3 The merge phase

The key difference between R-Merge and Q-Merge is the use of registers in the merge phase. If the merge order is sufficiently small, we expect that R-Merge will merge faster using registers to store the heap consisting of the head of each stream being merged. This is significant because the merge phase dominates the total run time; run formation cost is linear with problem size.

Figure 6 shows the total time per key for the merge phase of R-Merge and Q-Merge, varying the merge order, and making one to four merge passes. As expected, using registers for merging is faster at low merge orders. As the merge order increases, the gap narrows as R-Merge runs out of registers, and finally both programs converge. The merge cost scales with the number of passes, as does the speedup gained by using registers at small merge order. This confirms our supposition that using registers can improve the performance of merging algorithms in the context of a higher level algorithm that uses modest merge orders.

4.4 Putting it all together

Sorting illustrates limitations of the simple cache model. For a given sort input size (N), there is a tradeoff between the merge order (k), the number of passes (p), and the run size.

$$\text{run size} \times k^p \approx N$$

A smaller run size yields lower cost per key during run formation (Figure 5), but requires more merge passes and/or a higher merge order to complete the sort. Lower merge order allows more efficient merging using registers (Figure 6), but requires larger runs and/or additional passes to complete the sort. Fewer merge passes reduces the number of cache misses, but requires larger runs and/or higher merge order. All three factors are related; it is not possible to change one in isolation. This is illustrative of a common problem: it is hard or impossible to isolate one factor; each parameter constrains the others.

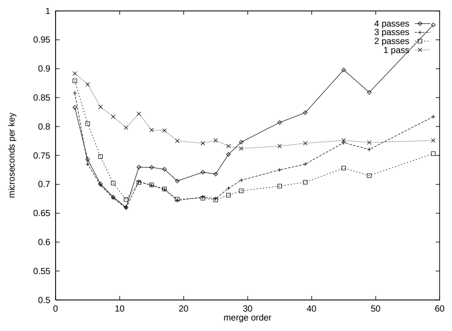
Figures 7 and 8 compare different merge orders and varying numbers of merge passes for R-Merge and Q-Merge. Figure 7 shows the result when sorting 32×2^{20} keys (128 cache loads on the test machine). Figure 8 shows the result when sorting 8×2^{20} keys (32 cache loads on the test machine).

Figures 7(a) and 8(a) show that for R-Merge with moderate merge order, the 2-pass and 3-pass versions are most effective, while the 4-pass version does poorly at all but the smallest merge orders. Large merge order and more passes result in smaller runs which waste merge effort, and under-utilize the cache. We also see that at very small merge order (less than 10), the run sizes are too large for cache-efficient run formation. Additional merge passes can reduce the run size, but waste processor cycles during each merge pass. That is, some idle instructions wait for data to arrive; these instructions could be used to increase the merge order at little cost. This confirms that there is a tradeoff between the parameters of the sort, although it is hard to determine exact form because of the coarse granularity of the parameter space.

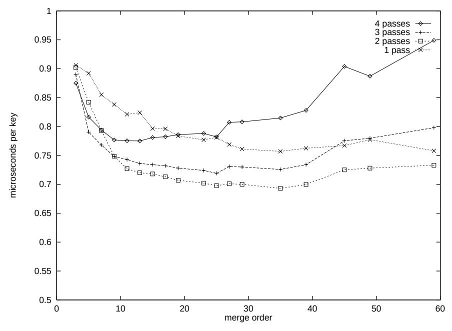
The figures also show that the number of cache misses sometimes does not matter. For example, in Figure 7(a), for merge order between 13 and 23, the 2-pass and 3-pass versions have similar performance, though the latter makes an extra pass through the data, and subsequently has $\Theta(N/B)$ extra cache misses.

Figures 7(b) and 8(b) show that Q-Merge has more regular behaviour than Q-Merge (Figures 7(a) and 8(a)). It is significant that the most effective configuration of Q-Merge uses a relatively large merge order (approximately 35), and only one pass. This is because the merge is not as efficient as in R-Merge, so the benefit of smaller runs is outweighed by increased merge cost.

A comparison between Figures 7 and 8 shows that a reduction in input size shifts to the left the tradeoff points between the different merge passes. This is consistent; as the input size is reduced, the merge order and number of merge passes required

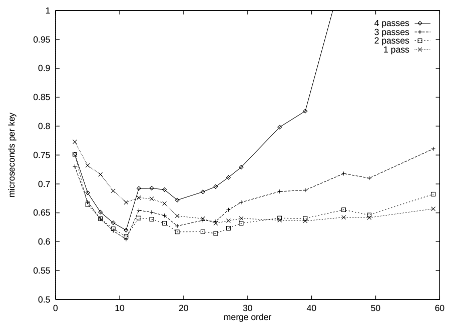


(a) R-Merge

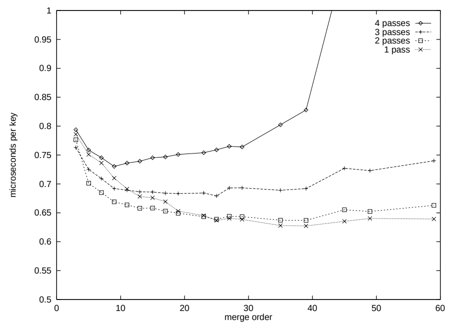


(b) Q-Merge

Fig. 7. Time per key (microseconds) while varying merge order. Time is for 1–4 merge passes. Sort size is 32×2^{20} keys.



(a) R-Merge



(b) Q-Merge

Fig. 8. Time per key (microseconds) while varying merge order. Time is for 1–4 merge passes. Sort size is 8×2^{20} keys.

to obtain the same run sizes is reduced.

We have seen that R-MERGE offers a significant speedup over Q-MERGE at a given merge order. However, this benefit is limited because R-MERGE requires smaller merge orders, which in turn could require extra passes. Q-MERGE reduces this advantage by using a significantly higher merge order. If we were being guided by the basic cache model, we would always pick the 1-pass merge with larger merge order because it has fewer cache misses, as seen with Q-MERGE. However, R-MERGE, which makes multiple passes and uses a smaller merge order is faster in practice. This shows that the basic cache model can mislead algorithm designers in some situations.

5. RELATED WORK

LaMarca and Ladner [1997] analyze the number of cache misses incurred by several sorting algorithms. They compare versions optimized for instructions executed with cache-conscious versions, and show that performance can be improved by reducing cache misses even with increased instruction count. Their work shows that the basic cache mode is often more predictive of real performance than the RAM model on modern machines. We use these algorithms (see Section 2.2) as a reference point for comparing the performance of our programs.

Rahman and Raman [1999] consider the slightly different problem of sorting single precision floating point numbers. They analyze the cache behaviour of Flashsort and develop a cache-efficient version obtaining a speedup of 1.1 over quicksort when sorting 32M floats. They also present an MSB radix sort that uses integer operations to speed up the sorting, giving a speedup of 1.4 on uniform random data. Rahman and Raman [2000] present a variant of LSB radix sort (PLSB radix sort). PLSB radix sort makes three passes over the 32-bit data, sorting on 11-bit fields at each pass. A pass consists of a presort and a global sort phase, both using counting sort. This yields a speedup of roughly 2 over the sorting routines of Rahman and Raman [2000]. However, the number of required passes for PLSB increases when working with 64-bit words rather than 32-bit words. We expect R-MERGE to be competitive with PLSB when sorting long words.

Sen and Chatterjee [2000] present a model that combines cache misses and instruction count. They show that a cache-oblivious implementation of mergesort leads to inferior performance. The analysis includes interference misses between streams of data being merged, and shows that small merge order is desirable.

Sanders [1999a] analyzes the cache misses when accessing multiple data streams sequentially. Any access pattern to $k = \Theta(M/B^{1+1/a})$ sequential data streams can be efficiently supported on an a-way set associative cache with capacity M and line size B. The bound is tight up to lower order terms. In addition, any number of additional accesses to a working set of size $k \leq M/a$ does not change this bound. In our experimental setup, we have $M = 256 \times 2^{10}$, $B = 8$ and $a = 1$, suggesting that values of k up to several thousand may be used without significant penalty.

Sanders [1999b] presents a cache-efficient heap implementation called *sequence heap*. Since the sequence heap operates on $(key, value)$ pairs, it is not possible to directly compare a heap sort based on a heap with our integer sorting program. However, in a simple experiment which equalizes the data transfer by setting the key and value to half word size, our new approaches are more than three times faster than Sanders' heapsort.

General models have been proposed for deep hierarchical memories, including the Hierarchical Memory Model [Aggarwal et al. 1987], Hierarchical Memory with Block Transfer [Aggarwal et al. 1987] and Uniform Memory Hierarchy Model [Alpern et al. 1994]. Frigo, Leiserson, Prokop, and Ramachandran [1999] present an algorithm design strategy that does not require explicit knowledge of the parameters describing the memory hierarchy. It is interesting to note that recursive algorithms are typically more memory friendy than their iterative versions unless the iterative version is cache-conscious.

Efficient register use has also been extensively studied in the architecture community [Hennessy and Patterson 1995; Callahan et al. 1990; Lam et al. 1991]. Register tiling for matrix computations has been shown to give significant speedups. However, as discussed in section 3, sorting raises different problems.

Ranade, Kothari, and Udupa [2000] also present a register-efficient mergesort which stores the head-element and position of each stream being merged in registers. However, their version does not explicitly use a heap; instead, the state of the merge is encoded in the program state itself. This results in an exponentially longer program, but eliminates register–register moves. However, we believe our approach is more efficient because: (1) they incur *many* more instruction-cache misses (which are relatively expensive); and (2) register–register moves are relatively cheap on a superscalar machine because they can frequently be scheduled in parallel with other instructions.

6. CONCLUSION

The performance of algorithms is determined by the accuracy of the model in which they are designed and analyzed. A key step toward developing better models is developing efficient programs for specific problems, and using them to observe the performance effects of features not reflected in the models. This paper illustrates key principles through an investigation of sorting algorithms.

Studying algorithms with access behavior that is efficient on modern memory systems shows that overlapping computation with memory accesses as much as possible leads to better sorting performance. This can be achieved by observing that: algorithms need to be both instruction-conscious and cache-conscious; memory access patterns affect performance; using registers and small data structures can improve performance. We embody these principles in a new sorting implementation called R-MERGE that is up to 36% faster than previous cache-conscious comparison sorting algorithms on our expermental system. We hope that application of these principles will lead to performance gains for other problems, and ultimately to a better understanding and representation of real machines in algorithm design.

ACKNOWLEDGMENTS

Thanks to Andrew Gallatin, Rakesh Barve, Alvin Lebeck, Laura Toma, and Song-Bac Toh for their invaluable contributions. Thanks also to the many reviewers, who provided comments on the paper.

REFERENCES

Aggarwal, A., Alpern, B., Chandra, A. K., and Snir, M. 1987. A model for hierarchical memory. In *Proceedings of the 19th ACM Symposium on Theory of Computation* (New York, NY, 1987), pp. 305–314.

Aggarwal, A., Chandra, A., and Snir, M. 1987. Hierarchical memory with block transfer. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science* (Los Angeles, CA, 1987), pp. 204–216.

Aggarwal, A. and Vitter, J. S. 1988. The Input/Output complexity of sorting and related problems. *Communications of the ACM 31*, 9, 1116–1127.

Alpern, B., Carter, L., Feig, E., and Selker, T. 1994. The uniform memory hierarchy model of computation. *Algorithmica 12*, 2-3, 72–109.

Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S. A., Sites, R. L., Vandervoorde, M. T., Waldspurger, C. A., and Weihl, W. E. 1997. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, Volume 31,5 of *Operating Systems Review* (New York, Oct. 5–8 1997), pp. 1–14. ACM Press.

Arge, L., Chase, J., Vitter, J. S., and Wickremesinghe, R. 2000. Efficient sorting using registers and caches. In *WAE, Workshop on Algorithm Engineering*, Lecture Notes in Computer Science (Sept. 2000). Springer.

Callahan, D., Carr, S., and Kennedy, K. 1990. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation* (White Plains, NY, June 1990).

Edmondson, J. H., Rubinfeld, P. I., Bannon, P. J., Benschneider, B. J., Bernstein, D., Castelino, R. W., Cooper, E. M., Dever, D. E., Donchin, D. R., Fischer, T. C., Jain, A. K., Mehta, S., Meyer, J. E., Preston, R. P., Rajagopalan, V., Somanathan, C., Taylor, S. A., and Wolrich, G. M. 1995. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal 7*, 1, 119–135.

Frigo, Leiserson, Prokop, and Ramachandran. 1999. Cache-oblivious algorithms. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)* (1999).

Hennessy, J. L. and Patterson, D. A. 1995. *Computer Architecture: A Quantitative Approach* (second ed.). Morgan Kaufmann.

Knuth, D. E. 1998. *Sorting and Searching* (second ed.), Volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA.

Ladner, R., Fix, J., and LaMarca, A. 1999. Cache performance analysis of traversals and random accesses. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms* (1999).

Lam, M. S., Rothberg, E. E., and Wolf, M. E. 1991. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1991).

LaMarca, A. and Ladner, R. E. 1997. The influence of caches on the performance of sorting. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (January 1997), pp. 370–379.

Rahman, N. and Raman, R. 1999. Analysing cache effects in distribution sorting. In *3rd Workshop on Algorithm Engineering* (July 1999).

Rahman, N. and Raman, R. 2000. Adapting radix sort to the memory hierarchy. In *ALENEX, Workshop on Algorithm Engineering and Experimentation* (2000).

Ranade, A., Kothari, S., and Udupa, R. 2000. Register efficient mergesorting. In *International Conference on High Performance Computing* (2000).

Sanders, P. 1999a. Accessing multiple sequences through set associative caches. In J. Wiedermann, P. van Emde Boas, and M. Nielsen Eds., *Automata, Languages and Programming, 26th International Colloquium, ICALP'99, Prague, Czech Republic, July 11-15, 1999, Proceedings*, Volume 1644 of *Lecture Notes in Computer Science* (1999). Springer.

Sanders, P. 1999b. Fast priority queues for cached memory. In *ALENEX, Workshop on Algorithm Engineering and Expermentation*, Lecture Notes in Computer Science (1999).

Sen, S. and Chatterjee, S. 2000. Towards a theory of cache-efficient algorithms. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms* (2000).

Srivastava, A. and Eustace, A. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation* (June 1994), pp. 196–205.

Vitter, J. S. and Shriver, E. A. M. 1994. Algorithms for parallel memory I: Two-level memories. *Algorithmica 12*, 2–3, 110–147.