

Efficient Sorting Using Registers and Caches

Lars Arge, Jeff Chase, Jeffrey S. Vitter, and Rajiv Wickremesinghe*

Department of Computer Science, Duke University, Durham, NC 27708, U.S.A.
<rajiv@cs.duke.edu>

Abstract. 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 theoretically superior under the models. R-MERGE is designed to minimize memory stall cycles rather than cache misses, considering features common to many system designs.

1 Introduction

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 cost. 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 managed to approximate the illusion of a large, fast memory. The levels of the memory hierarchy often have varying characteristics (eg: fully-associative vs. direct-mapped, random vs. sequential/blocked access).

Program performance is largely determined by its interaction with the memory system. It is therefore important to design algorithms with access behavior that is efficient on modern memory systems. Unfortunately, the RAM model and its alternatives are inadequate to guide algorithmic and implementation choices on modern machines. In recent years there has been a great deal of progress in devising new models that more accurately reflect memory system behavior. 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 [2]. The I/O model has yielded significant improvements for I/O-bound problems where the access gaps between levels of the hierarchy (e.g., memory and disk) are largest.

* This work supported in part by the National Science Foundation through ESS grant EIA-9870724 and RI grant EIA-9972879, and by the U.S. Army Research Office through MURI grant DAAH04-96-1-0013.

The difference in speed between upper levels of the memory hierarchy (CPU, caches, main memory) continues to widen as advances in VLSI technology widen the gap between processor and memory cycle time. It is not the number of steps or instructions, but the number of cache misses, and the duration of the resulting stalls that determine the execution time of a program. This contrasts with the models derived from the I/O model, which only count the number of cache misses (analogous to number of I/Os in the I/O model).

This makes it increasingly important for programs to make efficient use of caches and registers. It is therefore important to extend the benefits of the I/O model to incorporate the behavior of caches and registers. However, cache behavior is difficult to model adequately. Unlike main memory, caches are faster and tend to have low associativity. The former makes computationally expensive optimizations ineffective, and the latter increases the difficulty of modeling the memory. The complexity of memory systems, including cache hierarchies, prefetching, multiple- and delayed- issue processors, lockup-free caches, write buffers and TLBs (see Section 2) 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 affect algorithmic and implementation choices for a specific problem: sorting. Second, we present quantitative data from several memory-conscious sort implementations on Alpha 21x64 CPUs in order to quantify the effects of these choices. These results also apply in general to other architectures including Intel Pentium and HP PA-RISC (see appendix). 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 RISC processors. This sorting approach, R-MERGE, 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. In particular, R-MERGE uses a hybrid mergesort/quicksort approach that considers the duration of stalls, streaming behaviour, etc., rather than merely minimizing cache misses. R-MERGE also shows the importance of efficient register usage during the merge phase. We generalize from our experience to propose key principles for designing and implementing memory-conscious algorithms.

This paper is organized as follows. Section 2 briefly reviews memory system structure and extracts principles for memory-conscious algorithm design and implementation. Section 3 and Section 4 apply these principles to the problem of sorting, describe the memory-conscious sort implementations used in our experiments, and present experimental results from these programs. Section 5 sets our work in context with related work. We conclude in Section 6.

2 Models of Memory Systems

Modern memory systems typically include several levels of memory in a hierarchy. The processor is at level 0, and has a small number of *registers* that store individual data items for access in a single cycle. One contribution of this paper is to show the

benefits of incorporating explicit register usage into algorithm design. Register usage is usually managed by the compiler. Careful use of registers can reduce memory accesses and instructions executed by reducing the number of load/store instructions required.

Caches are small, fast, stores of “useful” data. In today’s systems, caches are managed by the memory system rather than the program. Several levels of cache are placed between main memory and the processor. Elements are brought into the cache in *blocks* of several words to make use of spatial locality. There are many hardware devices that help improve performance, but they also complicate analysis. A *stall* occurs when the processor is forced to wait for the memory system instead of executing other (independent) instructions. A *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 miss penalty. *Translation Look-aside Buffers* (TLBs) provide a small cache of address translations in a hardware lookup table. Accessing large numbers of data streams can incur performance penalties because of TLB thrashing [12].

The I/O model of Aggarwal and Vitter [2] is important because it bridges the largest gap in the memory hierarchy, that between main memory and disk. The model describes a system with a large, slow memory (disks) and a limited fast memory (main memory). The fast memory is fully-associative and data is transferred to and from it in blocks of fixed size (an *I/O operation*). 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 [11, 12, 16] have proposed cache models similar to the I/O model. The cache is analogous to the (fast) main memory in the I/O model. Main memory takes the place of disks, and is assumed to contain all the data. The following parameters are defined in the model:

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. A block transfer takes place when a cache miss occurs. The I/O model is not directly applicable to caches because of several important differences, described below.

- 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.
- Modern 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 cache misses.
- The cache (in our case) is direct-mapped, whereas the main memory is fully-associative. This restricts how we use the cache, since the cache location used is

determined by the location in memory where the data element is stored. There are no special instructions to control placement of data in the cache.

- The model only considers one cache; in practice there is a hierarchy of caches.
- Registers can be used to store the most frequently accessed data. They form what can be viewed as a very small, fast, controllable cache. Algorithmic and programming changes are usually required to exploit registers.
- The size of the TLB limits the number of concurrent data streams that can be handled efficiently [12].

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 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 number of misses). A more realistic measure of CPU time takes into account the memory stall time and the actual execution time [7]:

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

To consider the effects of these factors, we experiment with hybrid sorting algorithms. Our designs are guided by augmenting the cache model with three principles.

1. The algorithm should be instruction-conscious *and* cache-conscious: we need to balance the conflicting demands of reducing the number of instructions executed, and reducing the number of memory stalls.
2. Memory is not uniform; carefully consider the access pattern. Sequential accesses are faster than random accesses.
3. Effective use of registers reduces instruction counts as well as memory accesses.

3 Sorting

We consider the common problem of sorting N words (64-bit integers in our case) for values of N up to the maximum that will fit in main memory. This leads us to make several design choices.

1. We observe that quicksort does the least work per key as it has a tight inner loop, even though it is not cache-efficient. It also operates in place, allowing us to make full use of the cache. Therefore it is advantageous to sort with quicksort once the data is brought into the cache. Our mergesorts combine the advantage of quicksort with the cache efficiency of merging by forming cache-load-sized runs with quicksort. For example, when we compared two versions of our R-MERGE algorithm—one that does one pass of order $k = N/(BC)$, and one that does two passes of order $k = \sqrt{N/(BC)}$ but consequently has more efficient code—the two-pass version had double the misses in the merge phase, but executed 22% fewer instructions and

was 20% faster. The run formation was identical for both versions. This indicates that it is necessary to focus on instruction counts beyond a certain level of cache optimization.

2. 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.
3. The third key to good performance is the efficient use of registers. The merge heap is an obvious candidate, as it 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 also avoids cache interference misses, though this effect is small at small order [16], by eliminating memory accesses in the heap. However, the small number of registers limits the order (k) of the merge. The small order merge executes less instructions, but 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 is proportional to $N \times (\text{average cost of heap operation}) \times \log_k N$. The average cost of a heap operation averaged over the $\log k$ levels of the heap is less if the heap is smaller. The number of misses is $(N/B) \lceil \log_k(N/BC) \rceil$.

These factors lead us to use a hybrid mergesort/quicksort algorithm. We optimized the core merge routine at the source level to minimize the number of memory locations accessed, and especially to reduce the number of instructions executed. Runs shorter than cache size are used, and the input is padded so that all runs are of equal same size, and so that the full order of runs are merged. This appears to be the most efficient arrangement. We found merging to be at least 10% faster than the corresponding distribution sort.

Our hybrid algorithm is as follows. The merge order, k , is input into the program.

1. Choose number of passes, $\lceil \log_k(N/BC) \rceil$.
2. Form initial runs (which are approximately of length BC elements) using quicksort (in cache).
3. Merge sets of k runs together to form new runs.
4. Repeat merge step on new runs until all elements are in one run.

The merge itself is performed using techniques based on the selection tree algorithms from [8].

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. Note that the previous step created holes in the data at just the right places.

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()
      
```

We refer to this algorithm as Q-MERGE. One advantage of Q-MERGE is that with low merge orders it is possible to store the merge heap in registers, further improving performance. The next section shows that this approach—R-MERGE—is up to 36% faster than previous comparison-sorting algorithms.

It is essential to engineer the inner loop(s) so that the compiler can generate efficient code. Duplicating compiler optimizations (eg: loop unrolling, instruction reordering, eliminating temporary variables) is ineffective, and could actually hurt overall performance. This is partly because the compiler has more detailed knowledge of the underlying machine. Although merging is not done in-place, cache performance is not degraded because memory writes are queued in the write buffer, and do not incur additional penalties.

Data and control dependencies slow program execution by limiting the degree of parallelism (modern processors are superscalar—capable of executing multiple instructions in parallel). 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 is inherently limited by the necessity of performing $\Theta(n \log n)$ *sequential* comparisons. (Successful branch prediction and other techniques could reduce the actual number of sequential comparisons required).

4 Experiments

We performed most of our experiments on a Digital Personal Workstation 500au with a 500MHz Alpha 21164 CPU. We also validated our results on several other architectures (see appendix): (1) Alpha 21264 (500MHz) with a 4MB cache. (2) Alpha 21264A (467MHz) with a 2MB cache. (3) Intel Pentium III (500MHz) with a 512K cache. The 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). 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 ($C = 256K$), 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 [6].

We performed cache simulations and measured the actual number of executed instructions by instrumenting the executables with the ATOM [5] tool. We used the hardware counters of the Alpha 21164 and the `dcp.i` tool [3] to count memory stalls. This tool was also invaluable in optimizing the code as it allows instruction level tracking of

stalls. All the programs were written in C, and compiled with the vendor compiler with optimizations enabled.

We compare our programs with the original cache-conscious programs of LaMarca and Ladner [11] described below.

- TILED MERGESORT is a tiled version of an iterative binary mergesort. Cache-sized tiles are sorted using mergesort, and then merged using an iterative binary merge routine. Tiling is only effective over the initial $\Theta(\log(BC))$ passes.
- MULTI-MERGESORT uses iterative binary mergesort to form cache-sized runs and then a single $\Theta(N/BC)$ way merge to complete the sort.
- QUICKSORT is a memory-tuned version of the traditional optimized quicksort. It has excellent cache performance within each run, and is most efficient in instructions executed (among comparison-based sorts). However, quicksort makes $\Theta(\log N)$ passes, and thus incurs many cache misses. Memory-tuned quicksort sorts small subsets (using insertion sort) when they are first encountered (and are already in the cache). This increases the instruction count, but reduces the cache misses. To make our analysis more conservative, we improved this version by simplifying the code to rely more on compiler optimization (eg: *removing* programmer-unrolled loops), yielding better performance.
- DISTRIBUTION SORT uses a single large multi-way partition pass to create subsets which are likely to be cache-sized. Unlike quicksort, this 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 [12].

Our programs are as follows.

- R-DISTRIBUTION is a register-based version of distribution sort with a counting pass. It limits the order of the distribute in order to fit the pivots and counters in registers. This leads to an increased number of passes over the data.
- R-MERGE is one of two programs that implement the hybrid mergesort/quicksort algorithm described in Section 3: R-MERGE uses small merge order and direct addressing (discussed below), allowing the compiler to assign registers for key data items (the merge heap in particular).
- Q-MERGE also implements the hybrid algorithm. However, it uses an array to store the merge heap. We use this more traditional version to determine the effect of using registers.

Both the merge programs have heap routines custom-written for specific merge orders. These routines are generated automatically by another program. We also evaluated a traditional heap implementation.

It is possible to access a memory location by *direct addressing* or by *indirect addressing*. In the former, the address of the operand is specified in the instruction. In the latter, the address has to be computed, for example as an offset to a base; this is frequently used to access arrays. In a RISC processor like the Alpha 21164, indirect accesses to memory usually take two instructions: one to compute the address, and one to access the data. R-MERGE tries to be register efficient and thus uses direct addressing

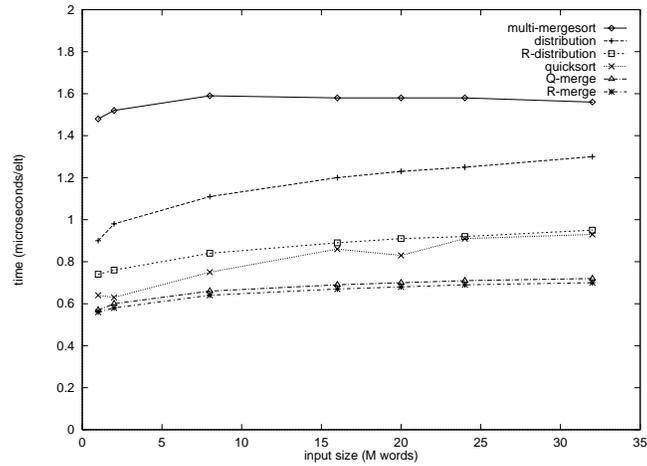


Fig. 1. Comparison with cache-conscious sort programs of LaMarca and Ladner [11]. Time per key (μs) vs. number of keys ($\times 2^{20}$).

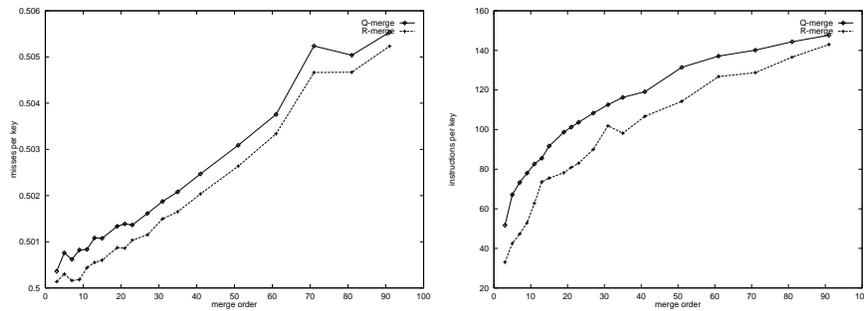


Fig. 2. Misses per key and instructions per key for R-MERGE and Q-MERGE (merge phase only). Misses are measured by simulation of a direct-mapped cache and do not reflect the entire memory system.

and a small merge order k so that the merge data structure can be accessed in registers. Q-MERGE, on the other hand, performs better with substantially larger merge order k .

Figure 1 shows performance results for R-MERGE and Q-MERGE compared with the fast implementations of [11]. We tested a range of input sizes requiring up to a gigabyte of main memory. The speedup for merge-based sorting is greater than 2; for distribution sort, 1.3. Comparing the fastest programs (quicksort and R-MERGE), R-MERGE obtains a speedup ranging from 1.1 to 1.36. Similar results were obtained on the 21264-based system.

In order to illustrate the tradeoffs between more efficient merging and a larger number of merge passes, we vary the merge order for R-MERGE and Q-MERGE. Figure 2 compares R-MERGE and Q-MERGE at varying merge order. Since these programs have identical run-formation stages, it is sufficient to examine the merge phase to compare them. R-MERGE is consistently better because it allocates at least part of the heap in

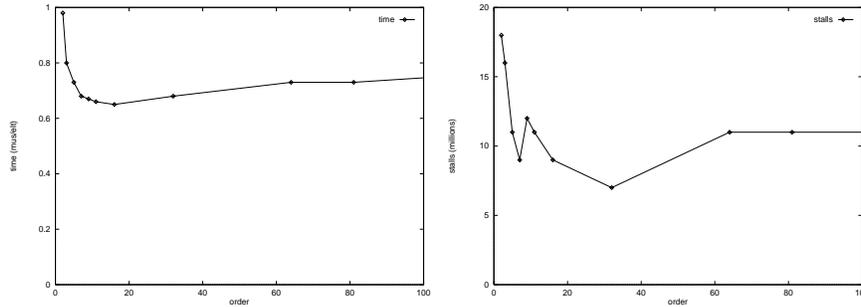


Fig. 3. Time per key (μs) and number of stalls (millions) sampled for R-MERGE for different merge orders. $N = 20 \times 2^{20}$ words.

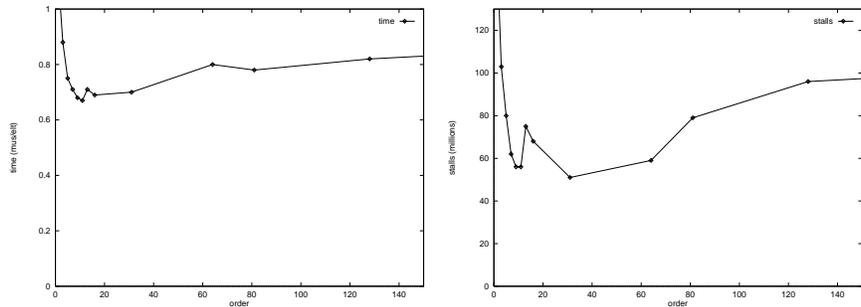


Fig. 4. Time per key (μs) and number of stalls (millions) sampled for R-MERGE for different merge orders. $N = 32 \times 2^{20}$ words.

registers. This reduces the number of instructions executed and memory locations referenced.

Figure 3 shows the time per key and number of stalls at varying merge orders k , when sorting 20×2^{20} words. The best performance is obtained at $k \approx 16$, even though the minimum stalls is obtained at $k \approx 32$ and the number of cache misses is minimized at $k \geq N/(BC) = 80$.

Figure 4 shows the time per key and number of stalls at varying merge orders k , when sorting 32×2^{20} words. The values at $k = 16$ and $k = 32$ are of particular interest. When $k \leq 16$, the merge heap structure can be stored in registers. The time taken is closely correlated with the number of stalls, indicating that the memory system is the bottleneck. When $k = 32$ the heap no longer fits, and the time increases even if the number of stalls decreases.

5 Related Work

LaMarca and Ladner [11] 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. We use these algorithms as a reference point for comparing the performance of our programs.

Rahman and Raman [12] 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 develop an MSB radix sort that makes use of integer operations to speed up the sorting, giving a speedup of 1.4 on uniform random data. In their subsequent paper [13], Rahman and Raman present a variant of LSB radix sort (PLSB radix sort). PLSB radix sort makes three passes over the 32-bit data, sorting 11-bit quantities at each pass. A pass consists of a presort and a global sort phase, both using counting sort. They report a speedup of roughly 2 over the sorting routines of Ladner and LaMarca [13]. 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 [16] present a model which attempts to combine cache misses and instruction count. They show that a cache-oblivious implementation of mergesort leads to inferior performance. This analysis includes the interference misses between streams of data being merged.

Sanders [14] 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 [15] 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 test in which we equalize 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.

Efficient register use has also been extensively studied in the architecture community [7, 4, 10]. Register tiling for matrix computations has been shown to give significant speedups. However, as discussed in section 3, sorting brings up different problems.

6 Conclusion

A key step toward developing better models is developing efficient programs for specific problems, and using them to quantify the performance effects of features not reflected in the models. We have introduced a simple cache model incorporating the principle that computation overlapped with memory access is free and then explored the effect of memory system features on sorting performance. We have formulated key principles of use in algorithm design and implementation: algorithms need to be both instruction-conscious and cache-conscious; memory access patterns can affect performance; using registers can reduce instruction counts and memory accesses.

We illustrate these principles with a new sorting implementation called R-MERGE that is up to 36% faster than previous memory-conscious comparison sorting algorithms on our experimental system.

Acknowledgements

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

References

1. A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. *Proceedings of the 19th ACM Symposium on Theory of Computation*, pages 305–314, 1987.
2. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
3. J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, M. Vandevoorde, C. Waldspurger, and B. Wehl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP)*, October 1997.
4. D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.
5. DEC. *Programmer's Guide*. Digital Unix Documentation Library. ATOM toolkit reference.
6. J. H. Edmondson, P. I. Rubinfeld, P. J. Bannon, B. J. Benschneider, D. Bernstein, R. W. Castolino, E. M. Cooper, D. E. Dever, D. R. Donchin, T. C. Fischer, A. K. Jain, S. Mehta, J. E. Meyer, R. P. Preston, V. Rajagopalan, C. Somanathan, S. A. Taylor, and G. M. Wolrich. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 7(1):119–135, 1995.
7. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2 edition, 1995.
8. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.
9. R. Ladner, J. Fix, and A. LaMarca. Cache performance analysis of traversals and random accesses. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999.
10. M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.
11. A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1997.
12. N. Rahman and R. Raman. Analysing cache effects in distribution sorting. In *3rd Workshop on Algorithm Engineering*, 1999.
13. N. Rahman and R. Raman. Adapting radix sort to the memory hierarchy. In *ALENEX, Workshop on Algorithm Engineering and Experimentation*, 2000.
14. P. Sanders. Accessing multiple sequences through set associative caches. *ICALP*, 1999.
15. P. Sanders. Fast priority queues for cached memory. In *ALENEX, Workshop on Algorithm Engineering and Experimentation*. Max-Planck-Institut für Informatik, 1999.
16. S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, 2000.

Appendix: Other Architectures

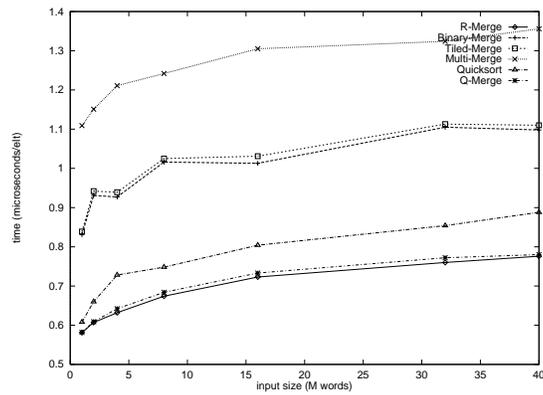


Fig. 5. Results on an Intel Pentium 500MHz with 512KB cache. Time per key (μs) vs. number of keys ($\times 2^{20}$). Element size is 32 bits for these experiments.

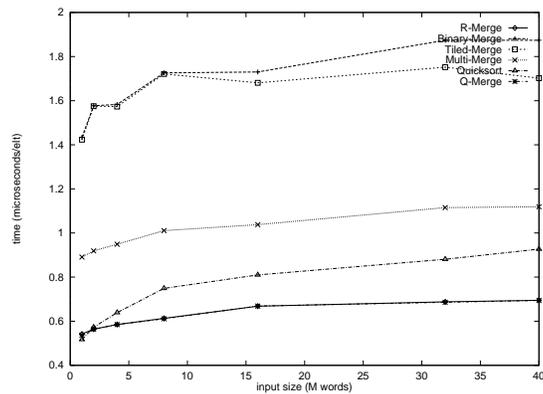


Fig. 6. Results on an Alpha 21164A 467MHz with 2MB cache. Time per key (μs) vs. number of keys ($\times 2^{20}$).