

# Improving Memory Performance of Sorting Algorithms

Li Xiao, Xiaodong Zhang, Stefan A. Kubricht  
Department of Computer Science  
College of William and Mary  
Williamsburg, VA 23187-8795

---

Memory hierarchy considerations during sorting algorithm design and implementation play an important role in significantly improving execution performance. Existing algorithms mainly attempt to reduce capacity misses on direct-mapped caches. To reduce other types of cache misses that occur in the more common set-associative caches and the TLB, we restructure the mergesort and quicksort algorithms further by integrating tiling, padding, and buffering techniques and by repartitioning the data set. Our study shows that substantial performance improvements can be obtained using our new methods.

General Terms: Caches, Memory Performance, Mergesort, Quicksort, TLB

---

## 1. INTRODUCTION

Sorting operations are fundamental and are often repeatedly used in many large-scale scientific and commercial applications. Because of this prominence, any effort to maximize the efficiency in these programs requires ensuring that the sorting algorithms used have been correctly selected and are precisely implemented. Restructuring standard efficient sorting algorithms (such as mergesort and quicksort) to exploit cache locality has proven to be an effective approach for improving performance on high-end systems. Since sorting algorithms are highly sensitive to both

---

This work is supported in part by the National Science Foundation under grants CCR-9400719, CCR-9812187, and EIA-9977030, by the Air Force Office of Scientific Research under grant AFOSR-95-1-0215, and by Sun Microsystems under grant EDUE-NAFO-980405.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

the memory hierarchy of the computer architecture and the types of data sets, care must be taken when choosing an algorithm to fully optimize the performance for a specific sorting operation. Existing restructured algorithms (e.g., [4]) mainly attempt to reduce capacity misses on direct-mapped caches. In this paper, we present several restructured mergesort and quicksort algorithms that exhibit substantial performance improvements by further increasing the locality of the memory references to reduce other types of cache misses, such as conflict misses and TLB misses. These new algorithms utilize both tiling and padding techniques, data set repartitioning, and knowledge of the processor hardware (such as cache and TLB associativity) to fully optimize the performance. Thus, in order to maximize efficiency, it is necessary to implement the cache-effective algorithms carefully and precisely at the algorithm design and programming levels.

Our efforts focus chiefly on restructuring mergesort and quicksort algorithms to more effectively utilize the cache. Our results and contributions are summarized below:

- By applying padding techniques we are able to significantly reduce cache conflict misses and TLB misses, which are not fully addressed in the algorithm designs of the tiled mergesort and the multi-mergesort [4]. For our two mergesort alternatives, the optimizations improve both cache and overall performance. Our experiments on different high-end workstations show that our algorithms achieve up to a 70% reduction in execution time compared with the base mergesort, and up to a 54% reduction versus the fastest of the tiled and multi-mergesort algorithms.
- By partitioning the data set based on data ranges, we are able to improve the cache locality of quicksort on unbalanced data sets. Our two quicksort alternatives significantly outperform the memory-tuned quicksort [4] and the flashsort [6] on unbalanced data sets.
- Cache-effective sorting algorithm design is dependent on the computer architecture as well as the type of data set. The algorithm design should include parameters such as the size and associativity of both the data cache and TLB, the ratio between the data set size and the cache size, and possibly other factors. Using our measurements and simulations, we show the importance of considering these factors by demonstrating how machines interact differently with the various algorithms.
- A major issue that must be considered when designing a sorting algorithm for practical use concerns the trade-offs resulting from increasing the instruction count in order to reduce cache misses and other high-latency memory operations. To address this, we give an execution timing model to quantitatively predict the performance of an algorithm. We also give analytical predictions of the number of cache misses for the sorting algorithms before and after the cache optimizations. We show that cycles lost from increasing the instruction count to maximize cache reuse can be a negligible price to pay when compared to the many cycles that would otherwise be lost from different types of cache misses.

## 2. ARCHITECTURAL/ALGORITHMIC PARAMETERS AND EVALUATION METHODOLOGY

In this section, we first list and describe the architectural-dependent parameters we used in designing the algorithms. We then introduce the performance evaluation methodology and present the data sets used in the experiments.

### 2.1 Architectural/algorithmic parameters

A data set consists of a number of elements. One element may be a 4-byte integer, an 8-byte integer, a 4-byte floating point number, or an 8-byte double floating point number. We use the same generic unit, an element, to specify the cache capacity. Because the size of caches and cache lines are always a multiple of an element in practice, a general unit is practically meaningful to both architects and application programmers. The algorithmic and architectural parameters we will use to describe cache-effective sorting algorithms are as follows:  $N$ : the size of the data set,  $C$ : the data cache size,  $L$ : the size of a cache line,  $K$ : the cache associativity,  $T_s$ : the number of entries in a TLB set,  $K_{TLB}$ : the TLB associativity, and  $P_s$ : the size of a memory page.

### 2.2 Performance evaluation methodology

Directly monitoring and measuring a program's cache behavior is an important task for providing insights and guidance for optimizing the memory performance of an algorithm. Since current systems are not able to directly report memory related performance statistics (such as the number of cache hits or misses) during program execution, users must use tools to gather these statistics. ATOM [10] is a system utility for DEC Alpha machines for instrumenting and analyzing program executables. The ATOM analysis tool accepts the results of an instrumented program and presents the cache performance statistics. Using the ATOM utility, users can directly monitor and measure the cache performance on DEC alpha machines. The analysis of sorting algorithms in [4] uses the ATOM tool. Due to its platform dependence, memory performance studies using ATOM are not feasible on other types of machines.

The need for studying a broad range of platforms necessitates an alternative approach. We conducted our performance evaluation in two steps: (1) completing algorithm analysis and measuring performance on different high-end workstations, and (2) utilizing execution-driven simulations to gather insight into the memory performance of the algorithms on these machines. Employing the first step, we are able to measure the algorithm performance on a wide variety of machines. From the second step we are able to gather a deeper understanding of how the cache behavior affects the execution performance.

For our simulation environment, we used the SimpleScalar tool set [1], a family of simulators for studying interactions between application programs and computer architectures. The simulation tools take an application program's binaries compiled for the SimpleScalar Instruction Set Architecture (a close derivative of the MIPS instruction set) and generate statistics during the execution of the program on the simulated architecture. The statistics generated include many detailed execution traces which are not available from measurements on a computer, such as the number of cache misses in the L1, L2 and TLB caches.

We ran the compared sorting algorithms on different simulated memory architectures with memory hierarchies similar to those on typical high-end workstations to observe the following performance factors:

- L1 or L2 cache misses per element*: to compare the number of data cache misses.
- TLB misses per element*: to compare the number of TLB misses.
- Instruction count per element*: to compare the algorithmic complexities.
- Reduction rate of total execution cycles*: to compare the percentage of cycles saved in comparison to the base mergesort or the memory-tuned quicksort.

### 2.3 Data sets

The algorithms are compared and evaluated experimentally and analytically. We tested the sorting algorithms on a variety of data sets consisting of 8-byte integer elements. The 9 data sets we used are enumerated below. (Probability Density Functions and Inverse Distribution Functions of some of the number generators used can be found in [7].)

- (1) *Random*: the data set is obtained by calling the random number generator `random()` from the C library, which returns integers in the range of 0 to  $2^{31} - 1$ .
- (2) *Equilikely*: function `Equilikely(a,b)` returns integers in the range `a` to `b`.
- (3) *Bernoulli*: function `Bernoulli(p)` returns integers 0 or 1.
- (4) *Geometric*: function `Geometric(p)` returns integers 0, 1, 2, ...
- (5) *Pascal*: function `Pascal(N,p)` returns integers 0, 1, 2, ...
- (6) *Binomial*: function `Binomial(N,p)` returns integers 0, 1, 2, ..., N.
- (7) *Poisson*: function `Poisson( $\mu$ )` returns integers 0, 1, 2, ...
- (8) *Zero*: the data set consists entirely of 0s.
- (9) *Unbalanced*: function returns integers in the range of 0 to  $2^{15} - 1$  for  $i = 0$  to  $\frac{127}{128}N - 1$ , by calling `rand()` from the C library, where  $i$  is the data element index and  $N$  is data set size; and returns integers  $MAX/100 + i$  for  $i = \frac{127}{128}N$  to  $N$ , where  $MAX = 2^{31} - 1$ .

## 3. CACHE-EFFECTIVE MERGESORT ALGORITHMS

In this section, we first briefly evaluate the two existing mergesort algorithms on their cache locality, as well as their merits and limits. We present two new mergesort alternatives to address these limits. The experimental performance evaluation done through measurements will be presented in section 5.

### 3.1 Tiled mergesort and multi-mergesort

LaMarca and Ladner [4] present two mergesort algorithms to effectively utilize the data cache. The first one is called *tiled mergesort*. The basic idea is to partition the data set into subarrays which are sorted individually. This is mainly done for two reasons: to avoid capacity misses and to fully use the data loaded in the cache before it must be replaced. The algorithm is divided into two phases. In the first phase, subarrays of length  $C/2$  (half the cache size) are sorted by the base mergesort algorithm to exploit temporal locality. The algorithm returns to the base mergesort

without considering cache locality in the second phase to complete the sorting of the entire data set.

The second mergesort, called *multi-mergesort*, addresses the limits of the tiled mergesort. In this algorithm, the first phase is the same as the first phase of the tiled mergesort. In the second phase, a multi-way merge method is used to merge all the sorted subarrays together in a single pass. A priority queue is used to hold the heads of the lists (the sorted subarrays from the first phase) to be merged. This algorithm exploits cache locality well when the number of subarrays in the second phase is less than  $C/2$  (half the cache size). However, the instruction count is significantly increased in this algorithm.

Our analysis of the two mergesort algorithms shows two areas for improvement. First, both algorithms significantly reduce capacity misses, but do not sufficiently reduce conflict misses. In mergesort, the basic idea is to merge two sorted subarrays to a destination array. In a cache with low associativity, mapping conflicts occur frequently among the elements in the three subarrays. Also, reducing TLB misses is not considered in the algorithm designs. Even when the data set is only moderately large, TLB misses may severely degrade execution performance, compounding on the effect of normal data cache misses. Our experiments show that the performance improvement of the multi-merge algorithm on several machines is modest—although it decreases the number of data cache misses, the heap structure significantly increases the number of TLB misses.

### 3.2 New mergesort alternatives

We present two new restructured mergesort alternatives for reducing conflict misses and TLB misses with a minimized instruction count increase: *tiled mergesort with padding* and *multi-mergesort with TLB padding*.

**3.2.1 Tiled mergesort with padding.** Padding is a technique that modifies the data layout of a program so that conflict misses are reduced or eliminated. The data layout modification can be done at run-time by system software [2; 12] or at compile-time by compiler optimization [8]. However, padding done at the algorithm design level using a full understanding of the data structures is expected to outperform optimizations using the two above system methods [13].

In the second phase of the tiled mergesort, pairs of sorted subarrays are sorted and merged into a destination array. One element from each of the two subarrays is selected at a time for a sorting comparison in sequence. These three data elements in the two different subarrays and the destination array can potentially be in conflicting cache blocks because they may be mapped to the same block in a direct-mapped cache and in a 2-way associative cache. This phenomenon occurs most often when the source array (containing the two subarrays) and the destination array are allocated contiguously in memory.

On a direct-mapped cache, the total number of conflict misses for the tiled mergesort in the worst case is approximately

$$\left(1 + \frac{1}{2C}\right)N \lceil \log_2 \frac{2N}{C} \rceil, \quad (1)$$

where  $\log_2 \frac{2N}{C}$  is the number of passes in the second phase of the sorting and  $1 + \frac{1}{2C}$  represents 1 conflict miss per comparison and  $\frac{1}{2C}$  conflict misses for every time an

element is placed into the destination array following a comparison, respectively.

In order to change the base addresses of these potentially conflicting cache blocks, we insert  $L$  elements (or a spacing the size of a cache line) to separate every section of  $C$  elements in the data set in the second phase of the tiled mergesort. These padding elements can significantly reduce the cache conflicts in the second phase of the mergesort. The memory used by the padding elements is trivial when compared to the size of the data set. The increase in the instruction count (resulting from having to move each element in a subarray to its new position for the padding) is also minor. We call this method as *tiled mergesort with padding*.

On a direct-mapped cache, the total number of conflict misses for the tiled mergesort with padding is at most

$$\frac{3}{4}N \lceil \log_2 \frac{2N}{C} \rceil, \quad (2)$$

where  $\log_2 \frac{2N}{C}$  is the number of passes in the second phase of the sorting and  $\frac{3}{4}$  represents the number of conflict misses per element. After the padding is added, the one conflict miss per comparison is reduced to  $\frac{3}{4}$ , and the  $\frac{1}{2C}$  conflict misses from the placement in (1) are eliminated. Comparing the two approximations in (1) and (2), we see that the tiled mergesort with padding reduces the conflict misses of the tiled mergesort by about 25%. (Our experimental results on the Sun Ultra 5, a workstation with a direct mapped cache, show that execution times of the tiled mergesort were reduced 23% to 68% by the tiled mergesort with padding. These execution time reductions mainly come from the abatement of conflict misses.)

Figure 1 shows an example of how the data layout of two subarrays in the second phase of tiled mergesort is modified by padding to reduce conflict misses. In this example, a direct-mapped cache holds 4 elements. In the figure, identical lines represent a pair comparison and the corresponding action to store the selected element in the destination array. The letter “m” in the figure represents a cache miss. Without padding, there are 8 conflict misses when merging the two sorted subarrays into the destination array; there are only 4 after padding is added.

Figure 2 shows the L1 misses (see the left figure) and the L2 misses (see the right figure) of the base mergesort, the tiled mergesort, and the tiled mergesort with padding on a simulated machine with the cache architecture of a Sun Ultra 5 using SimpleScalar. On this machine, the L1 cache is direct-mapped and contains 16 KBytes, and the L2 cache is 2-way associative holding 256 KBytes. The experiments show that the padding reduces the L1 cache misses by about 23% compared with the base mergesort and the tiled mergesort. These misses are conflict misses that cannot be reduced through tiling. The L2 cache miss reduction by the tiled mergesort with padding is almost the same as that by the tiled mergesort, which shows that the padding is not very effective in reducing L2 conflict misses on this machine. This is because the 2-way associativity in the L2 cache significantly reduces the percentage of conflict misses, in comparison to the direct-mapped L1 cache.

Capacity misses in the second phase of the tiled mergesort are unavoidable without a complex data structure, because the size of the working set (two subarrays and a destination array) is normally larger than the cache size. As we have shown, conflict misses can be reduced by padding in this phase. However, the padding may not completely eliminate all conflict misses due to the randomness of the order in

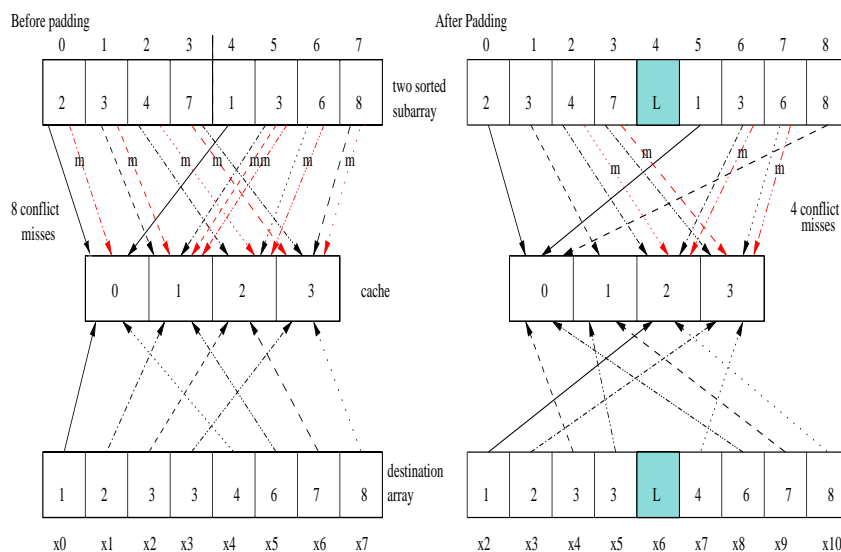


Fig. 1. Data layout of subarrays is modified by padding to reduce the conflict misses.

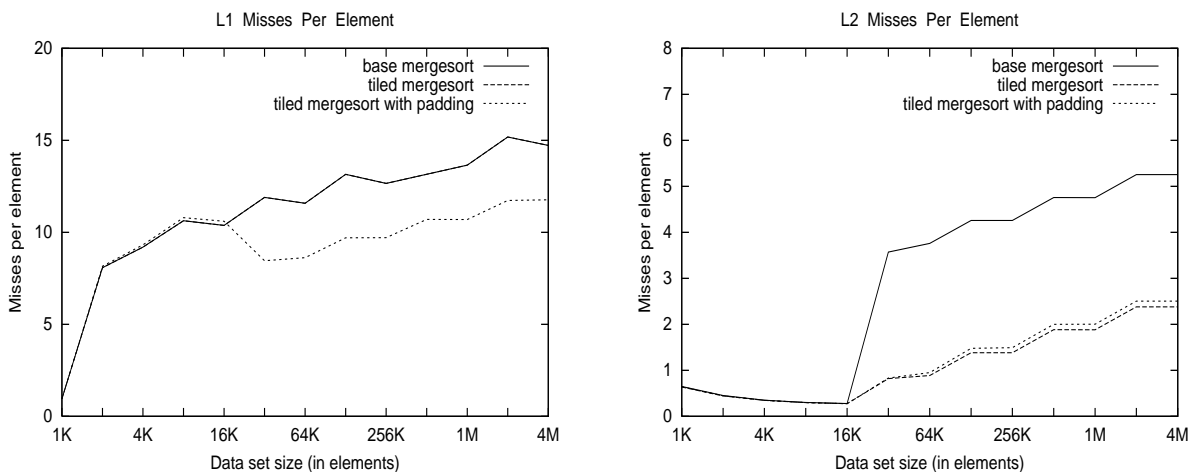


Fig. 2. Simulation comparisons of the L1 cache misses (left figure) and L2 misses (right figure) of the mergesort algorithms on the *Random* data set on the simulated Sun Ultra 5. The L1 cache miss curves (left figure) of the base mergesort and the tiled-mergesort are overlapped.

the data sets. Nevertheless, our experimental results presented in section 5 and the appendix using the 9 different data sets consistently show the effectiveness of the tiled mergesort with padding on the Sun Ultra 5.

**3.2.2 Multi-mergesort with TLB padding.** In the second phase of the multi-mergesort algorithm, the multiple subarrays are completely sorted in a single pass. Multiple

subarrays are used only once to complete the sorting of the entire data set to effectively use the cache. This single pass makes use of a heap structure to hold the head elements of the multiple subarrays. (Because of this structure, we will often refer to these subarrays as lists.) However, since the heads come from all the lists being multi-merged, the working set is much larger than that of the base mergesort (where only three subarrays are involved at a time). This large working set causes TLB misses which degrade performance. (We will explain the TLB structure in the following paragraph). Our experiments indicate that the multi-mergesort does significantly decrease the number of data cache misses. However, it also increases the TLB misses, which offsets the performance gain. Although a rise in the instruction count leads to additional CPU cycles in the multi-mergesort, this has a minimal effect. The performance of the algorithm is degraded mainly from the high number of TLB misses—memory accesses are far more expensive than CPU cycles.

The TLB (Translation-Lookaside Buffer) is a special cache that stores the most recently used virtual-physical page translations for memory accesses. The TLB is generally a small fully associative or set-associative cache. Each entry points to a memory page of 4 to 64Kbytes, depending on the architecture. A TLB cache miss forces the system to retrieve the missing translation from the page table in the memory, and then to replace an existing TLB entry with this translation. The TLB can hold a limited amount of data for sorting. When the data to be accessed spans more memory pages mapping to the same TLB set than the TLB can hold, TLB misses will occur. For example, the TLB cache of the Sun UltraSparc-III processor holds 64 fully associative entries ( $T_s = 64$ ), each of which points to a page of 8 KBytes ( $P_s = 1024$  8-byte elements). The 64 pages in the TLB of Sun UltraSparc-III processor hold  $64 \times 1024 = 65536$  elements, which represents a moderately-sized data set for sorting. Furthermore, in practice we often have more than one data array being operated on at a time. Some processors' TLBs are not fully associative, but set-associative. For example, the TLBs in the Pentium II and Pentium III processors are 4-way associative ( $K_{TLB} = 4$ ).

In the second phase of the multi-mergesort, we insert  $P_s$  elements (or a page space) to separate every sorted subarray in the data set in order to reduce or eliminate the TLB cache conflict misses. The padding changes the base addresses of these lists in page units to avoid potential TLB conflict misses.

Figure 3 exemplifies how padding for the TLB works; in this case the TLB is a direct-mapped cache of 8 entries, and the number of elements in each list is a multiple of 8 page elements. Before padding, each of the lists in the data set is mapped to the same TLB entry. After padding, these lists are mapped to different TLB entries.

When the multi-mergesort is performed on a large data set, when the size of each list is a multiple of  $T_s$ , the number of TLB misses per element is close to 1. After the TLB padding, the average number of TLB misses per element for the multi-mergesort algorithm becomes approximately

$$\frac{A}{A + K_{TLB}}, \quad (3)$$

where  $A = \frac{C}{T_s}$  is the number of average misses for each TLB set entry. The above

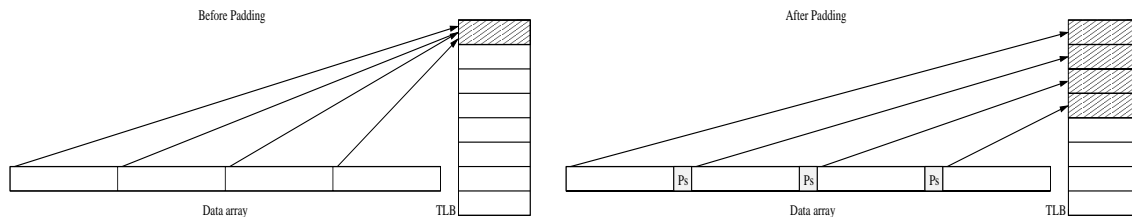


Fig. 3. Padding for TLB: the data layout is modified by inserting a page space at multiple locations, where  $K_{TLB} = 1$ , and  $T_s = 8$ .

approximation is further derived to

$$\frac{C}{C + K_{TLB} \times T_s}. \quad (4)$$

Figure 4 shows the number of L2 misses and TLB misses for the 5 mergesort algorithms on the Pentium II memory architecture as simulated using SimpleScalar, where the L1 cache is 4-way set associative with 16 KBytes, the L2 cache is 4-way set associative with 256 KBytes, and the TLB is a 4-way set associative cache having 64 entries. The simulation shows that the multi-mergesort and the multi-mergesort with TLB padding have the lowest L2 cache misses among the different algorithms (see the left figure in Figure 4). The multi-mergesort also had the highest number of TLB misses, but these misses are considerably reduced by the TLB padding. (see the right figure in Figure 4).

Here is an example verifying the approximation in (4) of TLB misses of the multi-mergesort. Substituting the parameters of Pentium II to the approximation,  $C = 256$ ,  $K_{TLB} = 4$ , and  $T_s = 64$ , we get 0.5 TLB misses per element for the multi-mergesort with TLB padding, which is very close to our experimental result, 0.47 (in the right figure of Figure 4). We will show in section 5 that the multi-mergesort with TLB padding significantly reduces the TLB misses and improves overall execution performance.

### 3.3 Trade-offs relating to an instruction count increase and the performance gain

Figure 5 shows the instruction counts of the 5 mergesort algorithms and the percentage of total cycles saved by the 4 improved mergesort algorithms compared to the base mergesort on the simulated Pentium II. The simulation shows that the multi-mergesort had the highest instruction count, while the tiled mergesort had the lowest instruction count. Taking advantage of the low number of L2 cache misses in the multi-mergesort and by reducing the TLB misses through padding, the multi-mergesort with TLB padding saved cycles by about 40% on large data sets compared to the base mergesort even though it has a relatively high instruction count. The tiled-mergesort with padding did not improve performance on the Pentium II. This is because this machine has a 4-way set associative cache where conflict misses are not major concerns.

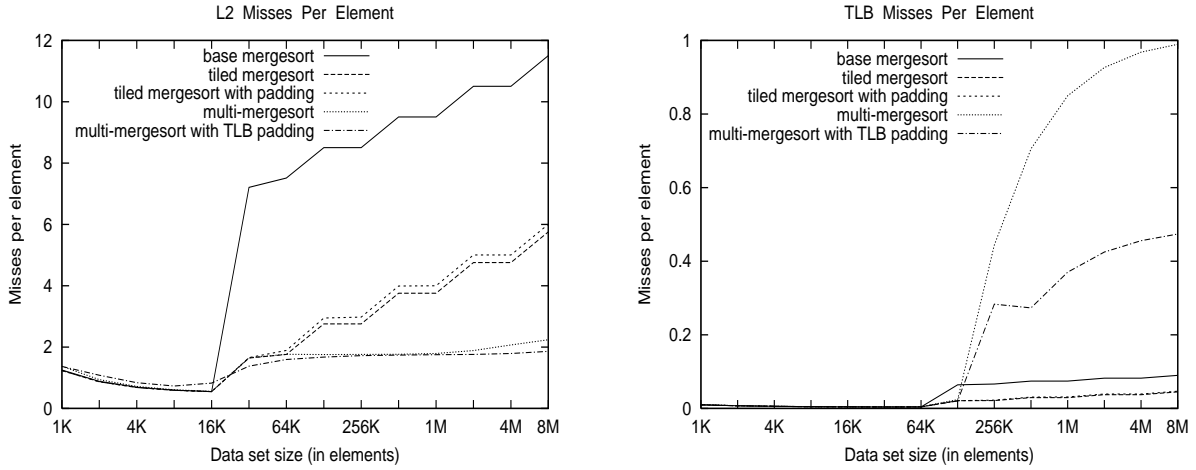


Fig. 4. Simulation comparisons of the L2 cache misses (left figure) and TLB misses (right figure) of the mergesort algorithms on the *Random* data set on the simulated Pentium II.

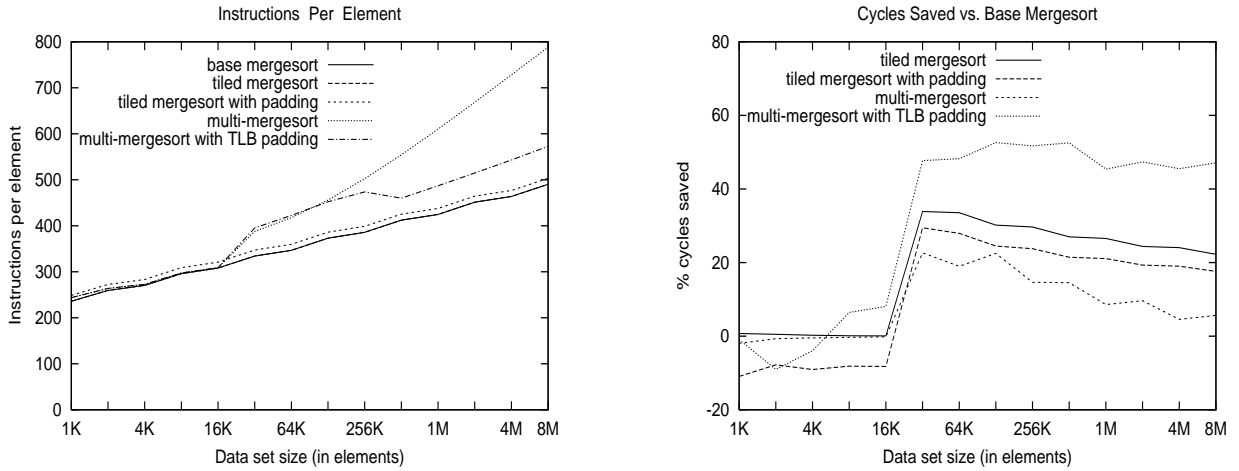


Fig. 5. Simulation comparisons of the instruction counts (left figure) and saved cycles in percentage (right figure) of the mergesort algorithms on the *Random* data set on the simulated Pentium II. The instruction count curves (left figure) of the base mergesort and the tiled mergesort are overlapped.

#### 4. CACHE-EFFECTIVE QUICKSORT

We first briefly assess the merits and limits of the two existing quicksort algorithms, especially considering their cache locality. We present two new quicksort alternatives for improving memory performance further. Experimental results will be reported in the next section.

#### 4.1 Memory-tuned quicksort and multi-quicksort

LaMarca and Ladner in the same paper [4] present two quicksort algorithms for cache optimization. The first one is called *memory-tuned quicksort*, which is a modification of the base quicksort [9]. Instead of saving small subarrays to sort in the end, the memory-tuned quicksort sorts these subarrays when they are first encountered in order to reuse the data elements in the cache.

The second algorithm is called *multi-quicksort*. This algorithm applies a single pass to divide the full data set into multiple subarrays, with the hope that each subarray will be smaller than the cache capacity.

The performance gain of these two algorithms from experiments reported in [4] is modest. We implemented the two algorithms on simulated machines and on various high-end workstations and obtained consistent performance. We also found that the performance of quicksort and its cache-optimized alternatives are very sensitive to the types of the data set being used. These algorithms were not efficient on unbalanced data sets.

#### 4.2 New quicksort alternatives

In practice, the quicksort algorithms exploit cache locality well on balanced data. A challenge is to make the quicksort perform well on unbalanced data sets. We present two cache-optimized quicksort alternatives that work well on both balanced and unbalanced data sets.

**4.2.1 *Flash Quicksort*.** Flashsort [6] is extremely fast for sorting balanced data sets. The maximum and minimum values are first identified in the data set to identify the data range. The data range is then evenly divided into classes to form subarrays. The algorithm consists of three steps: “classification” to determine the size of each class, “permutation” to move each element into its class by using a single temporary variable to hold the replaced element, and “straight insertion” to sort elements in each class by using Sedgewick’s insertion sort [9]. This algorithm works very well on balanced data sets because the sizes of the subarrays after the first two steps are similar and are small enough to fit in the cache. This makes the Flashsort highly effective ( $O(N)$ ). However, when the data set is not balanced, the sizes of the generated subarrays are disproportionate, causing ineffective usage of the cache, and making the flashsort as slow as the insertion sort ( $O(N^2)$ ) in the worst case.

In comparison with the pivoting process of the quicksort, the classification step of the flashsort is more likely to generate balanced subarrays, which favors better cache utilization. On the other hand, the quicksort outperforms the insertion sort on unbalanced subarrays. Through combining the advantages of the flashsort and the quicksort, we present a new quicksort alternative called *flash quicksort*, where the first two steps are the same as the ones in the flashsort, and the last step uses the quicksort to sort the elements in each class.

**4.2.2 *Inplaced Flash Quicksort*.** To further improve overall performance, we employ another cache optimization to improve temporal locality in the flash quicksort. This alternative is called *inplaced flash quicksort*. In this algorithm, the first and third steps are the same as in the flash quicksort. In the second step, an additional

array is used as a buffer to hold the permuted elements. In the original flashsort, a single temporary variable is used to hold the replaced element. A cache line normally holds more than one element. The data structure of the single variable minimizes the chance of data reuse. Using the additional array, we attempt to reuse elements in a cache line before their replacement, and to reduce the instruction count for copying data elements. Although this approach increases the required memory space, it improves both cache and overall performance.

### 4.3 Simulation results

Figure 6 shows the instruction counts (left figure) and the L1 misses (right figure) of the memory-tuned quicksort, the flashsort, the flash quicksort, and the inplace flash quicksort, on the *Unbalanced* data set on the simulated Pentium III memory architecture which has a higher memory latency and a larger L2 cache (512 KBytes) than the Pentium II. The instruction count curve of the flashsort was too high to be presented in the left figure of Figure 6. The same figure shows that the instruction count of the memory-tuned quicksort also increases rapidly as the data set size grows. In contrast, the instruction counts of the flash quicksort and the inplace flash quicksort changes little as the data set size increases. The simulation also shows that the number of L1 misses increases much more rapidly as the size of the data set grows in the memory-tuned quicksort and the flashsort than in the flashsort and inplace flashsort algorithms. The simulation results are consistent with our algorithm analysis, and show the effectiveness of our new quicksort alternatives on unbalanced data sets.

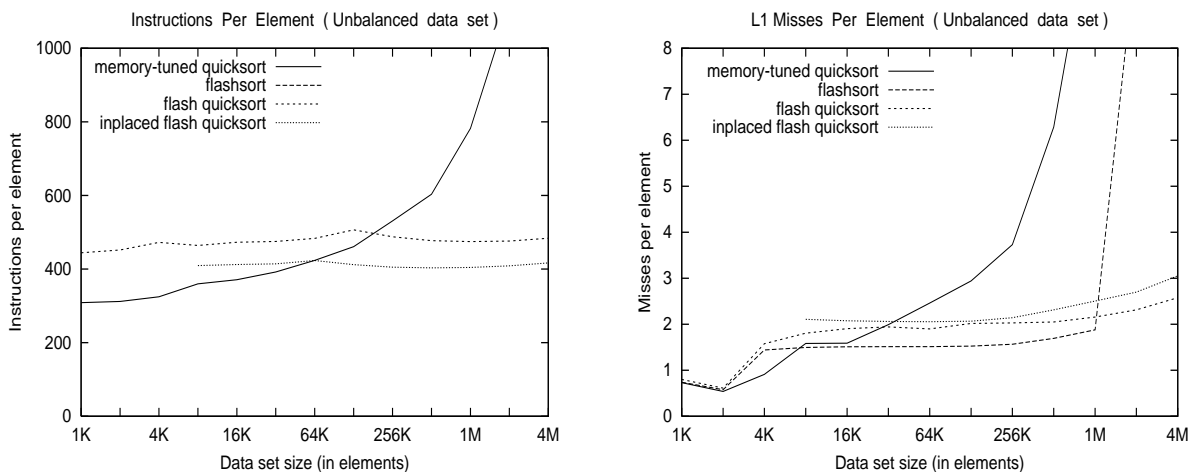


Fig. 6. Simulation comparisons of the instruction counts (left figure) and the L1 misses (right figure) of the quicksort algorithms on the *Unbalanced* data set on the simulated Pentium III. (The instruction count curve of the flashsort was too high to be presented in the left figure).

Workstations	SGI O2	Sun Ultra 5	Pentium	Pentium
Processor type	R10000	UltraSparc-III	Pentium II 400	Pentium III Xeon 500
clock rate (MHz)	150	270	400	500
L1 cache (KBytes)	32	16	16	16
L1 block size (Bytes)	32	32	32	32
L1 associativity	2	1	4	4
L1 hit time (cycles)	2	2	2	3
L2 cache (KBytes)	64	256	256	512
L2 associativity	2	2	4	4
L2 hit time (cycles)	13	14	21	24
TLB size (entries)	64	64	64	64
TLB associativity	64	64	4	4
Memory latency (cycles)	208	76	68	67

Table 1. Architectural parameters of the 4 machines we have used for the experiments.

## 5. MEASUREMENT RESULTS AND PERFORMANCE EVALUATION

We have implemented and tested all the sorting algorithms discussed in the previous sections on all the data sets described in section 2 on a SGI O2 workstation, a Sun Ultra-5 workstation, a Pentium II PC, and a Pentium III PC. The data sizes we used for experiments are limited by the memory size available on the experimental machines since we focus on cache-effective methods. We used “lmbench” [5] to measure the latencies of the memory hierarchy at its different levels on each machine. The architectural parameters of the 4 machines are listed in Table 5, where all the L1 cache specifications refer to the L1 data cache; all the L2 caches are uniform. The hit times of the L1 and L2 caches and the main memory measured by lmbench have been converted to the corresponding number of CPU cycles.

We compared all our algorithms with the algorithms in [4] and [6]. The execution times were collected by “gettimeofday()”, a standard Unix timing function. The reported time unit is cycle per element (*CPE*):

$$CPE = \frac{\text{execution time} \times \text{clock rate}}{N},$$

where *execution time* is the measured time in seconds, *clock rate* is the CPU speed (in cycles/second) of the machine where the program is run, and *N* is the number of elements in the data set.

Each execution time reported in this paper represents the mean of 20 runs. The variances range from 0.096 to 23.72 cycles<sup>2</sup> (corresponding to standard deviations ranging from 0.31 to 4.87 cycles). As a result, the coefficients of variation calculated by the ratio of the standard deviation to the mean is in a range of 0.00035 to 0.01.

The performance results on all the data sets are fairly consistent. Because of this, we only present the performance results of the mergesort algorithms using the *Random* data set on the 4 machines (plus performance results of the other data sets on the Ultra 5 to show the effectiveness of the tiled mergesort with padding). We present the performance results of the quicksort algorithms using the *Random* and the *Unbalanced* data sets on the 4 machines.

### 5.1 Mergesort performance comparisons

We compared 5 mergesort algorithms: the base mergesort, the tiled mergesort, the multi-mergesort, the tiled mergesort with padding, and the multi-mergesort with TLB padding. Proportional to each machine's memory capacity, we scaled the mergesort algorithms from  $N=1K$  up to  $N=16M$  elements. All our algorithms demonstrated their effectiveness as the data set size grew. Figure 7 shows comparisons of cycles spent per element for the 5 algorithms on the SGI O2 and the Sun Ultra 5. The multi-mergesort with TLB padding performed the best on the O2, with execution times reduced 55% compared to the base sort, 35% compared to the tiled mergesort, and 31% compared to the multi-mergesort on 2M elements. On the other hand, the tiled mergesort with padding performed the best on the Ultra 5, reducing execution times 45% compared to the multi-mergesort, 26% to the base mergesort, and 23% to the tiled mergesort on 4M elements. The multi-mergesort with TLB padding on Ultra 5 also did well, with a 35% improvement over the multi-mergesort, 13% over the base mergesort, and 9% over the tiled mergesort on 4M elements. The reason for the incredible performance improvements on the O2 result from its long memory latency (208 cycles). This makes the cache miss reduction techniques very effective in improving the overall performance of the sorting algorithms. The L2 cache size of the SGI is relatively small (64 KBytes), and the TLB is frequently used for memory accesses. Thus, the TLB padding is very effective. In addition, both L1 and L2 caches are 2-way set associative, where the padding is not as effective as on a direct-mapped cache. In contrast, the Ultra 5's L1 cache is direct-mapped and the L2 cache is 4 times larger than that of the O2. On this platform the data cache padding is more effective than the TLB padding.

In order to show the effectiveness of the tiled-mergesort with padding on a cache system with a low associativity, the performance curves of the 5 mergesort algorithms from the Sun Ultra 5 on the other 8 data sets are provided in the Appendix. Our experiments show that the tiled-mergesort with padding consistently and significantly outperforms the other mergesort algorithms on the Ultra 5. For example, the tiled mergesort with padding achieved 70%, 68%, and 54% reductions in execution time on the *Zero* data set compared with the base mergesort, the tiled mergesort, and the multi-mergesort, respectively. Using other data sets, the tiled mergesort with padding achieved 24% to 53% reductions in execution time compared with the base mergesort, 23% to 52% reductions compared with the tiled mergesort, and 23% to 44% reductions compared with the multi-mergesort.

Figure 8 shows the comparisons of cycles per element between the 5 mergesort algorithms on the Pentium II 400 and the Pentium III 500. The measurements on both machines show that the multi-mergesort with TLB padding performed the best, reducing execution times 41% compared with the multi-mergesort, 40% compared with the base mergesort, and 26% compared with the tiled sort on 16M elements. The L1 and L2 caches of both machines are 4-way set associative so the issue of data cache conflict misses is not a concern (as we discussed in section 3.1). Since TLB misses are the main source of inefficiency in the multi-mergesort algorithm, the padding for the TLB is very effective in improving the performance.

In summary, the tiled mergesort with padding is highly effective in reducing conflict misses on machines with direct-mapped caches, and the multi-mergesort

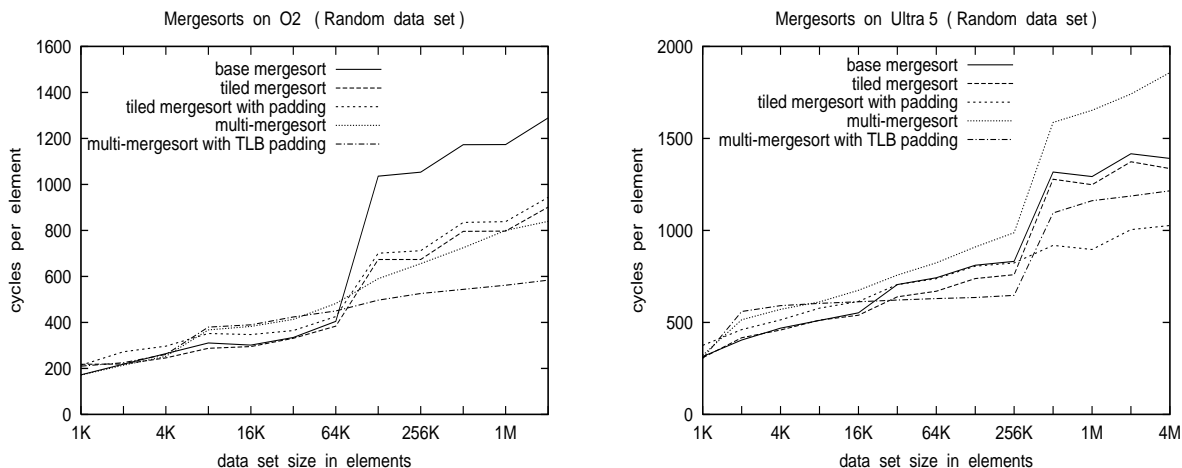


Fig. 7. Execution comparisons of the mergesort algorithms on SGI O2 and on Sun Ultra 5.

with TLB padding performs very well on all types of architectures.

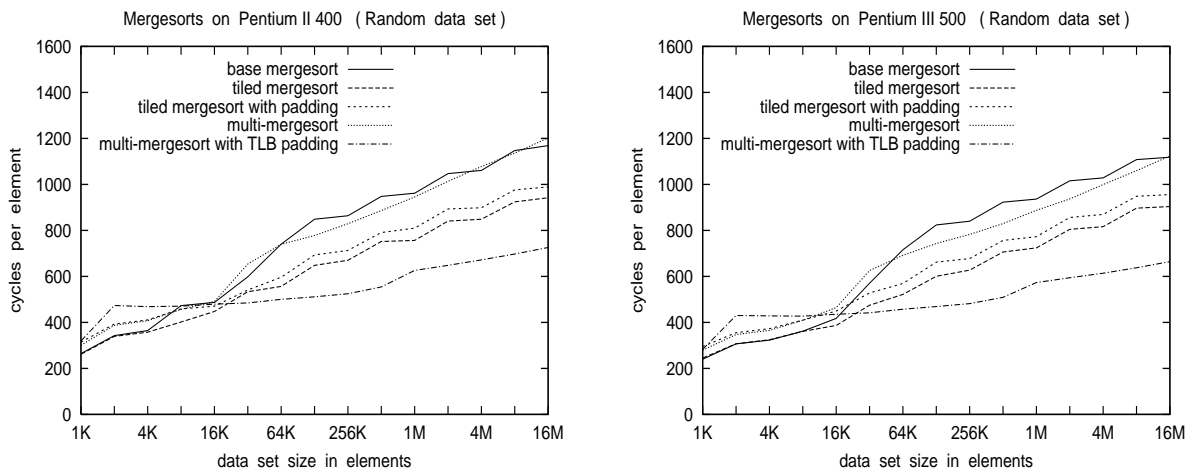


Fig. 8. Execution comparisons of the mergesort algorithms on Pentium II and on Pentium III.

### 5.2 Quicksort performance comparisons

We present the results of quicksort algorithms on the 4 machines using the *Random* data set and the *Unbalanced* data set. The 4 quicksort algorithms are: the memory-tuned quicksort, the flashsort, the flash quicksort, and the inplace flash quicksort.

Figure 9 shows the comparisons of cycles per element between the 4 quicksort algorithms on the *Random* data set (left figure) and the *Unbalanced* data set (right

figure) on the SGI O2 machine. The performance results of the 4 quicksort algorithms using the *Random* data set are comparable, with the memory-tuned algorithm slightly outperforming the others. The performance results using the *Unbalanced* data set are much different. As we expected, the number of cycles spent to sort each element is relatively stable for the flash quicksort and the inplace flash quicksort as the size of the data set increases, while the performance of memory-tuned quicksort and the flashsort greatly diminishes. The timing curves of the flashsort are even too high to be shown in the right figure in Figure 9.

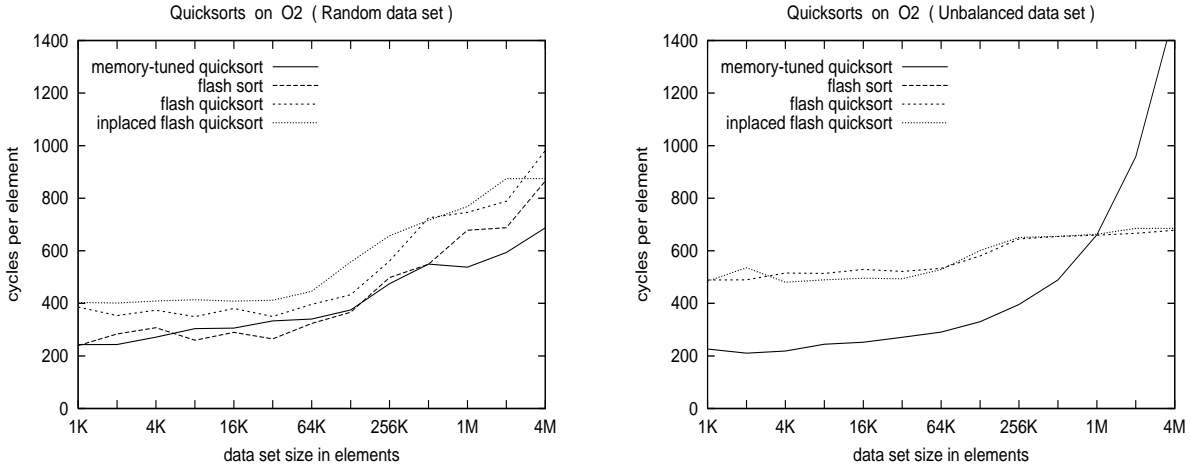


Fig. 9. Execution comparisons of the quicksort algorithms on the *Random* data set (left figure) and on the *Unbalanced* data set (right figure) on the SGI O2. (The timing curve of the flashsort is too high to be presented in the right figure).

Figure 10 shows the comparisons of cycles per element among the 4 quicksort algorithms on the *Random* data set (left figure) and the *Unbalanced* data set (right figure) on the Sun Ultra 5 machine. On the Ultra 5, all 4 algorithms showed little difference in their execution times on the *Random* data set. On the other hand, the flash and inplace flash quicksorts exhibited much better performance on the *Unbalanced* data set. For example, when the data set increased to 128K elements, the execution time of the flashsort is more than 10 times higher than that of the other three algorithms (the curve is too high to be plotted in the figure). When the data set is increased to 4M elements, the execution time of the memory-tuned quicksort is more than 3 times higher than the flash quicksort and the inplace flash quicksort, and the execution time of the flashsort is more than 100 times higher than that of the others.

Figure 11 and Figure 12 show the comparisons of cycles per element between the 4 quicksort algorithms on the *Random* data set (left figure) and the *Unbalanced* data set (right figure) on the Pentium II and the Pentium III machine respectively. On both Pentiums using the *Random* data set the flashsort, the flash quicksort, and the inplace flashsort displayed similar execution performance and reduced execution times around 20% compared to the memory-tuned quicksort on large data sets.

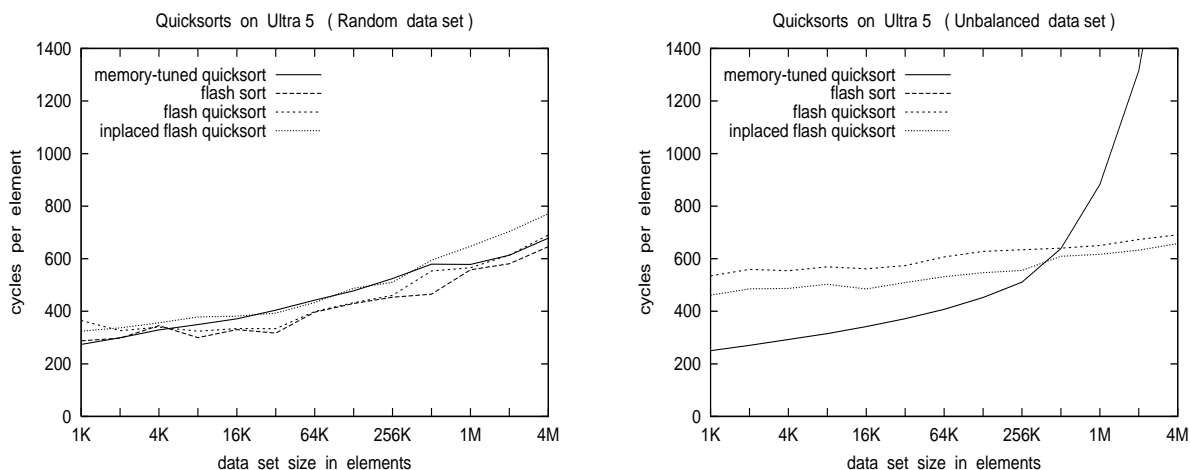


Fig. 10. Execution comparisons of the quicksort algorithms on the *Random* data set (left figure) and on the *Unbalanced* data set (right figure) on the Ultra 5. (The timing curve of the flashsort is too high to be presented in the right figure).

Again, the flash quicksort and inplace flash quicksort significantly outperformed the memory-tuned quicksort algorithm on the *Unbalanced* data sets on the two Pentium machines.

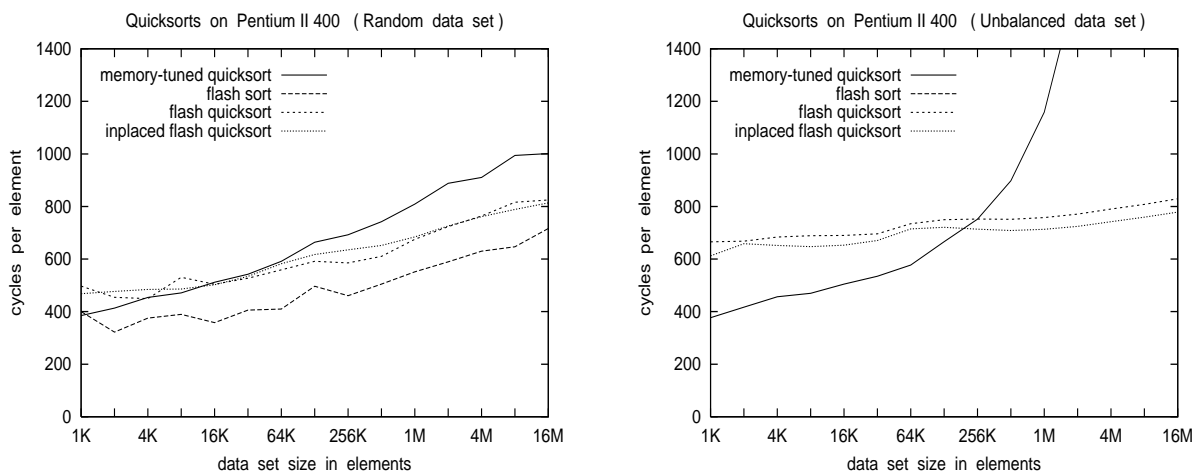


Fig. 11. Execution comparisons of the quicksort algorithms on the *Random* data set (left figure) and on the *Unbalanced* data set on the Pentium II. (The timing curve of the flashsort is too high to be presented in the right figure).

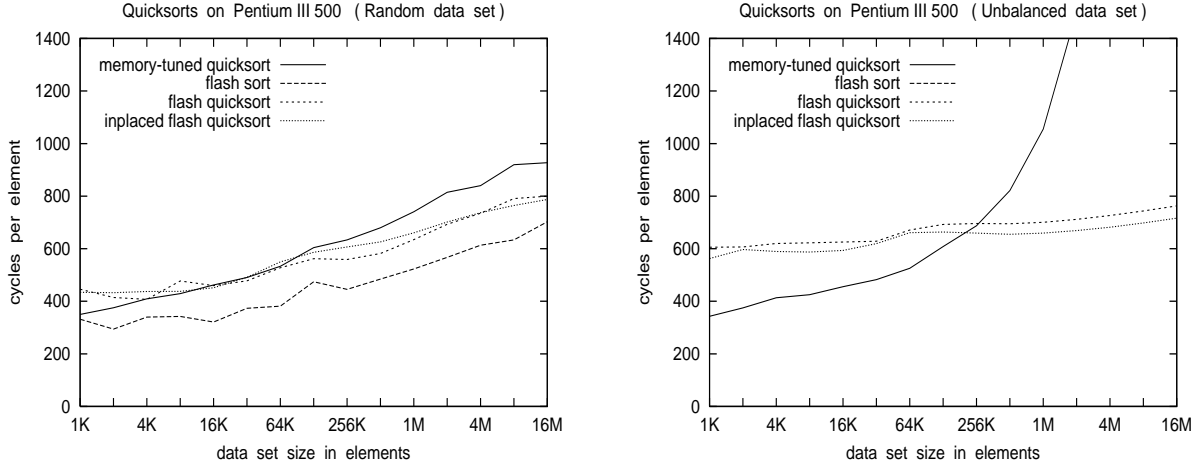


Fig. 12. Execution comparisons of the quicksort algorithms on the *Random* data set (left figure) and on the *Unbalanced* data set on the Pentium III. (The timing curve of the flashsort is too high to be presented in the right figure).

## 6. A PREDICTION MODEL OF PERFORMANCE TRADE-OFFS

The essential issue that must be considered when designing an algorithm that has an efficient memory access pattern is the trade-off between the optimization achievement—the reduction of cache misses, and the optimization effort—the increment in the instruction count. The optimization objective is to improve overall performance—to reduce the execution time of a base algorithm. This trade-off and the objective can be quantitatively predicted through an execution timing model. The execution time of an algorithm on a computer system based on Amdahl’s Law [3] is expressed as

$$T = CPU \text{ clock cycles} + \text{memory stall cycles} = IC \times CPI + CA \times MR \times MP, \quad (5)$$

where  $IC$  is the instruction count of the algorithm,  $CPI$  is the number of CPU cycles per instruction for the algorithm,  $CA$  is the number of cache accesses of during the algorithm’s execution,  $MR$  is the cache miss rate of the algorithm, and  $MP$  is the miss penalty in cycles of the system. The execution time for a base algorithm,  $T_{base}$ , is expressed as

$$T_{base} = IC_{base} \times CPI + CA_{base} \times MR_{base} \times MP, \quad (6)$$

and the execution time for an optimized algorithm,  $T_{opt}$ , is expressed as

$$T_{opt} = IC_{opt} \times CPI + CA_{opt} \times MR_{opt} \times MP, \quad (7)$$

where  $IC_{base}$  and  $IC_{opt}$  are the instruction counts for the base algorithm and the optimized algorithm,  $CA_{base}$  and  $CA_{opt}$  are the numbers of cache accesses of the base algorithm and the optimized algorithm, and  $MR_{base}$  and  $MR_{opt}$  are the cache miss rates of the base algorithm and the optimized algorithm, respectively.

In some optimized algorithms such as the tiled mergesort and the tiled mergesort with padding, the total number of cache accesses may be nearly the same as for

the base algorithm. For this type of algorithms, we combine equations (6) and (7) with  $CA_{base} = CA_{opt} = CA$  to predict the execution time reduction rate of an optimized algorithm as follows:

$$R = \frac{T_{base} - T_{opt}}{T_{base}} = \frac{\Delta MR \times CA \times MP - \Delta IC \times CPI}{IC_{base} \times CPI + CA_{base} \times MR_{base} \times MP}, \quad (8)$$

where  $\Delta MR = MR_{base} - MR_{opt}$  represents the miss rate reduction, and  $\Delta IC = IC_{opt} - IC_{base}$  represents the instruction count increment. In order to obtain a positive reduction in execution time, the following must hold true:

$$\Delta MR \times CA \times MP > \Delta IC \times CPI.$$

This model describes the quantitative trade-off between the instruction count increase and the miss rate reduction, and gives the condition for an optimized algorithm to improve the performance of a base algorithm:

$$\frac{\Delta IC}{\Delta MR} < \frac{CA \times MP}{CPI}. \quad (9)$$

For multi-phase optimized algorithms which have different cache access patterns in each phase, such as the multi-mergesort and the multi-mergesort with TLB padding, we combine equations (6) and (7) with  $CA_{base} \neq CA_{opt}$  to obtain the condition for an optimized algorithm to improve the performance of a base algorithm:

$$\frac{\Delta IC}{\Delta(MR \times CA)} < \frac{MP}{CPI}, \quad (10)$$

where  $\Delta(MR \times CA) = MR_{base} \times CA_{base} - MR_{opt} \times CA_{opt}$ .

There are architecture related and algorithm related parameters in this prediction model. The architecture related parameters are  $CPI$  and  $MP$  which are machine dependent and can be easily obtained. The algorithm related parameters are  $IC$ ,  $CA$ , and  $MR$ , which can be either predicted from algorithm analysis or obtained from running the program on a simulated architecture, such as SimpleScalar. The algorithm related parameters can also be predicted by running the algorithms on relatively small data sets that are larger than the cache capacity on a target machine.

Using the prediction model and the parameters from the SimpleScalar simulation, we are able to predict the execution time rate of reduction for the optimized algorithms. Our study shows that the predicted results using the model are close to the measurement results, with a 6.8% error rate.

## 7. CONCLUSION

We have examined and developed cache-effective algorithms for both mergesort and quicksort. These algorithms have been tested on 4 representative processors dating from 1995 to 1999 to show their effectiveness. We also use simulations to provide additional evaluation of performance. We have shown that the memory architecture plays the largest role in affecting the performance of the various mergesort algorithms, while the type of data set used affects the quicksort algorithms the most.

Our techniques of padding, partitioning, and buffering can also be used for other algorithms for optimizations directed at the cache. Whenever a program regularly

accesses to a large data set that cannot be entirely stored in the cache, the danger of conflict misses exists, particularly when the algorithm partitions the data sets in sizes that are a power of 2. Padding is effective for this type of program to eliminate or reduce conflict misses. Examples include matrix accesses/manipulations and data reordering and swapping between data sets. When a program sequentially and repeatedly scans a large data set that cannot be stored in the cache in its entirety, the program will suffer capacity cache misses. Partitioning the data set based on the cache size to localize the memory used by a stage in execution is effective for this type of program. Tiling for the mergesort is one example where this is used; other tasks where this optimization approach can be used include data accesses by loops and data manipulations of a large data file in a sequential order. The buffering technique is effective to reduce or eliminate conflict misses by using an additional buffer to temporarily hold data elements for later reuse that would otherwise be swapped out of the cache. Examples where this can be employed include programs manipulating data in an inplaced fashion and programs where data accesses easily cause conflict cache misses.

The only machine dependent architecture parameters for implementing the 4 methods we presented in this paper are the cache size ( $C$ ), the cache line size ( $L$ ), cache associativity ( $K$ ), the number of entries in the TLB cache, and a memory page size ( $P_s$ ). These parameters are becoming increasingly known to users. They can be defined as variables in the programs, making migration from one platform to another easy for a user. In this way, the programs are easily portable—all that is required is the knowledge of the 4 required parameters.

There are several ways to provide sorting algorithms with architecture-dependent parameters. One approach leaves the work to an informed user who is familiar with the machine architecture; this user could simply input the required parameters into the programs. A second possibility: users could conduct some brief executions using a runtime library to obtain estimated architectural parameters for the program optimizations. The overhead caused by this approach is normally acceptable [12]. ATLAS [11] uses a tool to first automatically determine architectural parameters by extensive tests on the target machine. The program is then recompiled with these parameters included. A third possibility would be to utilize the ATLAS approach to support our sorting program optimizations, easing the burden on an end-user.

#### ACKNOWLEDGMENTS

Many students in the Advanced Computer Architecture class offered in the Spring 1999 participated discussions of cache-effective sorting algorithms and their implementations. Particularly, Arun S. Mangalam made an initial suggestion to combine the quicksort and the flashsort. We also appreciate Alma Riska, Zhao Zhang, and Zhichun Zhu for their comments and their help with the simulations. Finally, the comments from the anonymous referees were constructive and helpful to us for improving the quality and readability of the paper.

### 8. APPENDIX: MERGE-SORT PERFORMANCE COMPARISONS ON ULTRA 5 USING 8 DIFFERENT DATA SETS

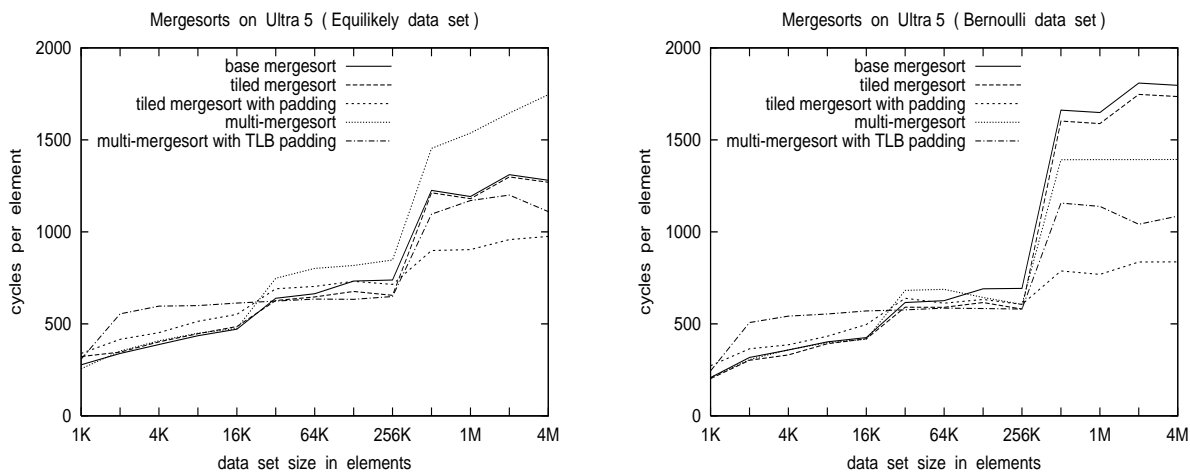


Fig. 13. Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Equilikely data set (left figure) and the Bernoulli data set (right set).

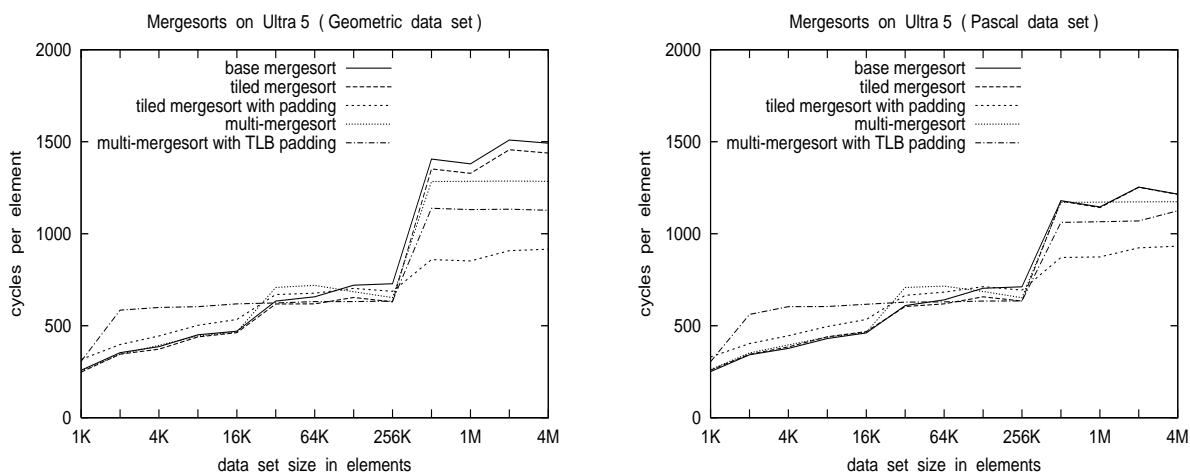


Fig. 14. Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Geometric data set (left figure) and the Pascal data set (right set).

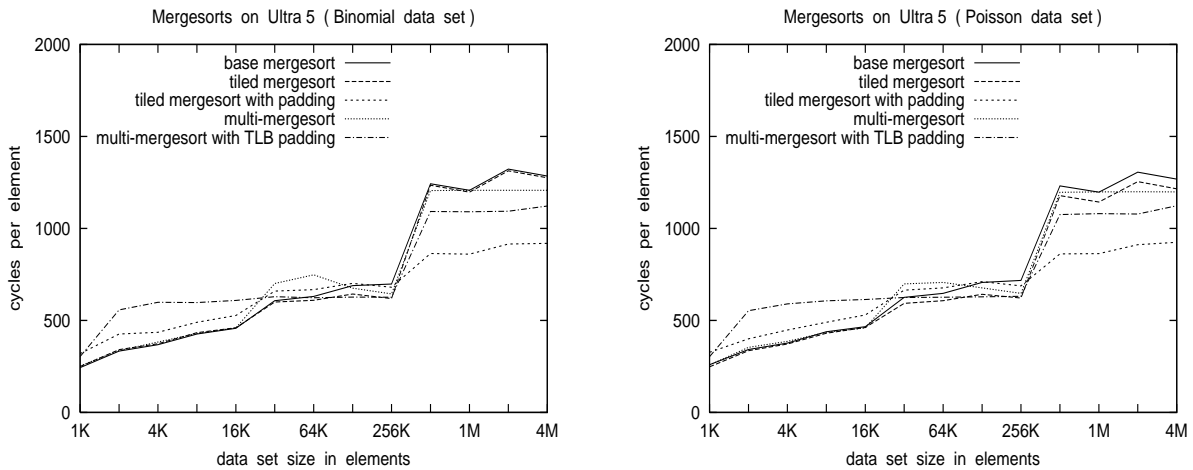


Fig. 15. Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Binomial data set (left figure) and the Poisson data set (right set).

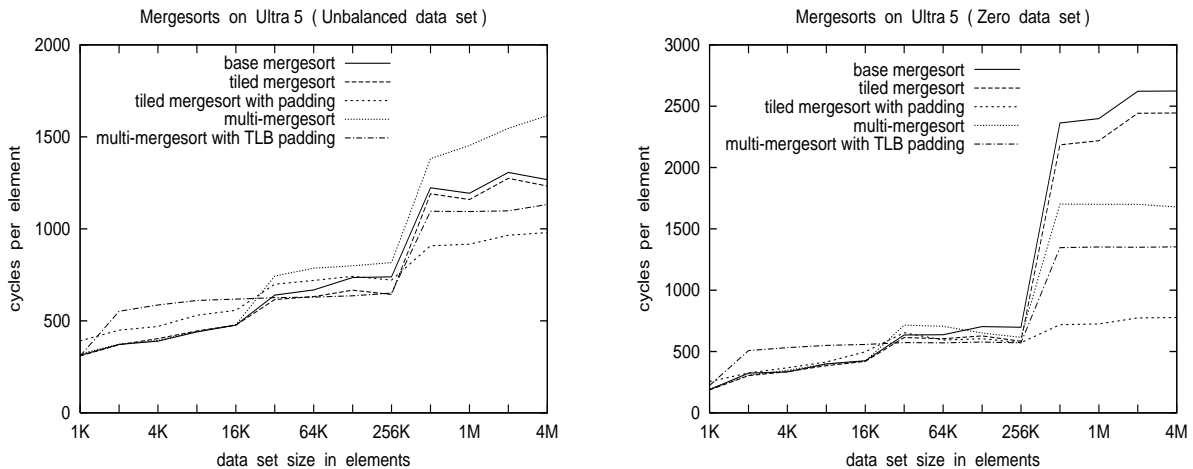


Fig. 16. Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Unbalanced data set (left figure) and the Zero data set (right set).

## REFERENCES

- [1] D. Burger and T. M. Austin, *The SimpleScalar Tool Set, Version 2.0*, Technical Report 1342, Department of Computer Sciences, University of Wisconsin, Madison, June 1997.
- [2] B. Bershad, D. Lee, T. Romer and B. Chen, "Avoiding conflict misses dynamically in large direct-mapped caches", *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, October, 1994.
- [3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, Inc., 1996.
- [4] A. LaMarca and R. E. Ladner, "The influence of caches on the performance of sorting",

- Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1997, pp. 370-379.
- [5] L. McVoy and C. Staelin, "lmbench: portable tools for performance analysis", *Proceedings of the 1996 USENIX Technical Conference*, San Diego, California, 1996, pp. 279-295.
  - [6] K.-D. Neubert, "The Flashsort1 algorithm", *Dr. Dobbs's Journal*, February 1998, pp. 123-125.
  - [7] S. Park and L. Leemis, *Discrete-Event Simulation: A First Course*, Lecture Notes, College of William & Mary, Revised Version, January 1999. Preprint of a Prentice-Hall book, August, 1999.
  - [8] C. Rivera and C.-W. Tseng, "Data transformations for eliminating conflict misses", *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation*, July 1998.
  - [9] R. Sedgewick, "Implementing quicksort programs", *Communications of the ACM*, Vol. 21, No. 10, 1978, pp. 847-857.
  - [10] A. Srivastava and A. Eustace, "ATOM: a system for building customized program analysis tool", *Proceedings of the 1994 ACM Symposium on Programming Languages Design and Implementation*, pp. 196-205.
  - [11] R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software", *Proceedings of Supercomputing'98*, November 1998.
  - [12] Y. Yan, X. Zhang and Z. Zhang, "Cacheminer: a runtime approach to exploit cache locality on SMP", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 11, No. 4, 2000, pp. 357-374.
  - [13] Z. Zhang and X. Zhang, "Cache-optimal methods for bit-reversals", *Proceedings of Supercomputing'99*, November, 1999.