ACM Journal on Experimental Algorithmics, Vol. 5, 2000

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. In order to further exploit cache locality to reduce other types of cache misses, we present several restructured mergesort and quicksort algorithms and their implementations by fully using existing processor hardware facilities, such as cache associativity and TLB, by integrating tiling and padding techniques, and by properly partitioning the data set. Our study shows that substantial performance improvement can be obtained by using our new methods.

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

1. INTRODUCTION

Sorting operations are fundamental in many large scale scientific and commercial applications. Sorting algorithms are highly sensitive to the memory hierarchy of the computer architecture on which the algorithms are executed, as well as sensitive to the types of data sets. Restructuring standard and algorithmically efficient sorting algorithms (such as mergesort and quicksort) to exploit cache locality is an effective approach for improving performance on high-end systems. Such ex-

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.

isting restructured algorithms (e.g., [4]) mainly attempt to reduce capacity misses on direct-mapped caches. In this paper, we report substantial performance improvement obtained by further exploiting memory locality to reduce other types of cache misses, such as conflict misses and TLB misses. We present several restructured mergesort and quicksort algorithms and their implementations by fully using existing processor hardware facilities (such as cache associativity and TLB), by integrating tiling and padding techniques, and by properly partitioning the data set for cache optimizations. Sorting as a fundamental subroutine, is often repeatedly used for many application programs. Thus, in order to gain the best performance, cache-effective algorithms and their implementations should be done carefully and precisely at the algorithm design and programming level.

We focus on restructuring mergesort and quicksort algorithms for cache optimizations. Our results and contributions are summarized as follows:

—Applying padding techniques, we are able to effectively reduce cache conflict misses and TLB misses, which are not fully considered in the algorithm design of the tiled mergesort and the multi-mergesort [4]. For our two mergesort alternatives, optimizations improve both cache and overall performance. Our experiments on different high-end workstations show that some algorithms achieve up to 70% execution time reductions compared with the base mergesort, and up to 54% reductions versus the fastest of the tiled and multi-mergesort algorithms.

—Partitioning the data set based on data ranges, we are able to exploit 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 both architecture and data set dependent. The algorithm design should include parameters such as the data cache size and its associativity, TLB size and its associativity, the ratio between the data set size and the cache size, as well as others. Our measurements and simulations demonstrate the interactions between the algorithms and the machines.

—The essential issue to be considered in sorting algorithm design is the trade-off between the reduction of cache misses and the increase in instruction count. We give an execution timing model to quantitatively predict the trade-offs. We also give analytical predictions of the number of cache misses for the sorting algorithms before and after the cache optimizations. We show that an increase in instruction count due to an effective cache optimization can be much cheaper than cycles lost from different types of cache misses.

2. ARCHITECTURE/DATA PARAMETERS AND THE SIMULATION TOOL

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 unit, element, to specify the cache capacity. Because the sizes of caches and cache lines are always a multiple of an element in practice, this identical unit is practically meaningful to both architects and application programmers, and makes the discussions straight-forward. Here are the algorithmic and architectural parameters we will use to describe cache-effective sorting algorithms: N: the size of the data set, C: data cache size, L: the size of a cache line, K:

cache associativity, T_s: number of set entries in the TLB cache, K_{TLB}: TLB cache associativity, and P_s: a memory page size.

Besides algorithm analysis and performance measurements on different high-end workstations, we have also conducted simulations to provide performance insights. The SimpleScalar tool set [1] is 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 concerning the program in relation with the simulated architecture. The statistics generated include many detailed execution traces which are not available from measurements on a computer, such as cache misses on L1, L2 and TLB.

We run sorting algorithms on different simulated architectures with memory hierarchies similar to that of high-end workstations to observe the following performance factors:

—*L1 or L2 cache misses per element:* to compare the data cache misses.

—*TLB misses per element:* to compare the TLB misses.

—*Instruction count per element:* to compare the algorithmic complexities.

—*Reduction rate of total execution cycles:* to compare the cycles saved in percentage against the base mergesort or the memory-tuned quicksort.

The algorithms are compared and evaluated experimentally and analytically. We tested the sorting algorithms on a variety of data sets, each of which uses 8-byte integer elements. Here are the 9 data sets we have used (Some probability density functions of number generators are described in [7].):

(1) *Random:* the data set is obtained by calling random number generator `random()` in the C library, which returns integers in the range 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(`μ`)` returns integers 0, 1, 2, ...

(8) *Zero:* All 0s in the data set.

(9) Unbalanced: the 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 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 overview the two existing mergesort algorithms for their cache locality, as well as their merits and limits. We present two new mergesort alternatives to address these limits. The experimental performance evaluation by measurements will be presented in section 5.

3.1 Tiled mergesort and multi-mergesort

LaMarca and Ladner [4] present two mergesort algorithms to effectively use caches. The first one is called *tiled mergesort*. The basic idea is to partition the data set into subarrays to sort individually mainly for two purposes: to avoid the capacity misses, and to fully use the data loaded in the cache before its replacement. 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 to be merged. This algorithm exploits cache locality well when the number of subarrays in the second phase is less than $C/2$. However, the instruction count is significantly increased in this algorithm.

Conducting experiments and analysis of the two mergesort algorithms, we show that the sorting performance can be further improved for two reasons. First, both algorithms significantly reduce capacity misses, but do not sufficiently reduce conflict misses. In mergesort, a basic operation is to merge two sorted subarrays to a destination array. In a cache with low associativity, conflict mapping occurs frequently among the elements in the three subarrays. Second, reducing TLB misses is not considered in the algorithms. Even when the data set is moderately large, the TLB misses may severely degrade execution performance in addition to 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 data cache misses, the heap structure significantly increases the TLB misses.

3.2 New mergesort alternatives

With the aim of reducing conflict misses and TLB misses while minimizing the instruction count increase, we present two new alternatives to further restructure the mergesort for cache locality: *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; 10] or at compile-time by compiler optimization [8]. Padding at the algorithm level with a full understanding of data structures is expected to significantly outperform optimization from the above system methods [11].

In the second phase of the tiled mergesort, pairs of sorted subarrays are sorted and merged into a destination array. One element at a time from each of the two subarrays is selected for a sorting comparison in sequence. These data elements in the two different subarrays and the destination array are potentially 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.

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

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

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 per element placement into the destination array after the comparison, respectively.

In order to change the base addresses of these potentially conflicting cache blocks, we insert L elements (or a cache line space) 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. Compared with the data size, the number of padding elements is insignificant. In addition, the instruction count increment (resulting from moving each element in a subarray to its new position after the padding) is also trivial. 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

$$\frac{3}{4}N\lceil \log_2 \frac{2N}{C} \rceil, \tag{2}$$

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 above 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 show the execution times of the tiled mergesort on the Sun Ultra 5, a workstation with a direct-mapped cache, were reduced 23% to 68% by the tiled mergesort with padding. The execution time reductions mainly come from the reductions 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 so that conflict misses are reduced. In this example, a direct-mapped cache holds 4 elements. In the figure, the same type of lines represent a pair comparison and the 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 (left figure) and the L2 (right figure) misses of the base mergesort, the tiled mergesort, and the tiled mergesort with padding on a simulated Sun Ultra 5 machine by the SimpleScalar. On this machine, L1 is a direct-mapped cache of 16 KBytes, and L2 is a 2-way associative cache of 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 which cannot be reduced by the tiling. The L2 cache miss reduction by the tiled mergesort with padding is almost the same as that by the tiled mergesort, which means that the padding is not very effective in reducing conflict misses in L2 on this machine. This is because the conflict misses are significantly reduced in L2 by the 2-way associative cache.

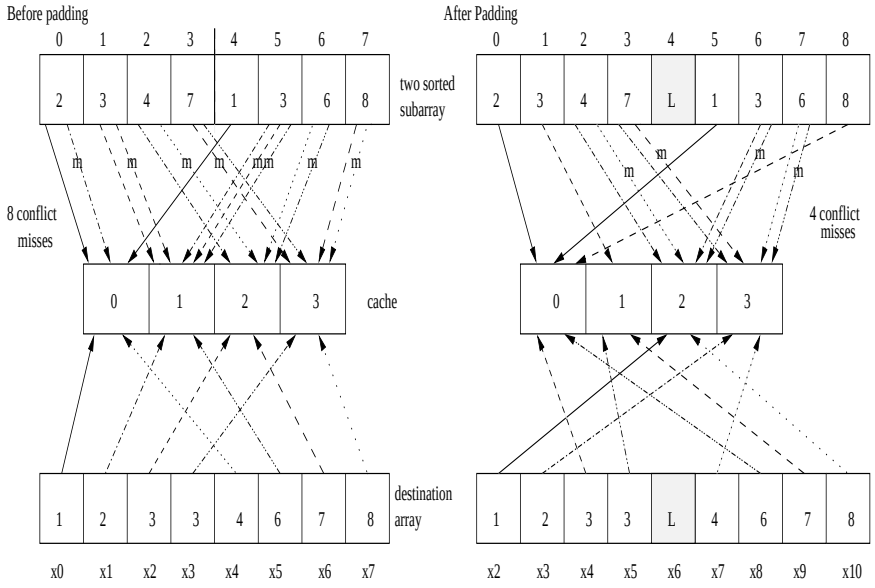

Fig. 1. Data layout of subarrays is modified by padding to reduce the conflict misses.

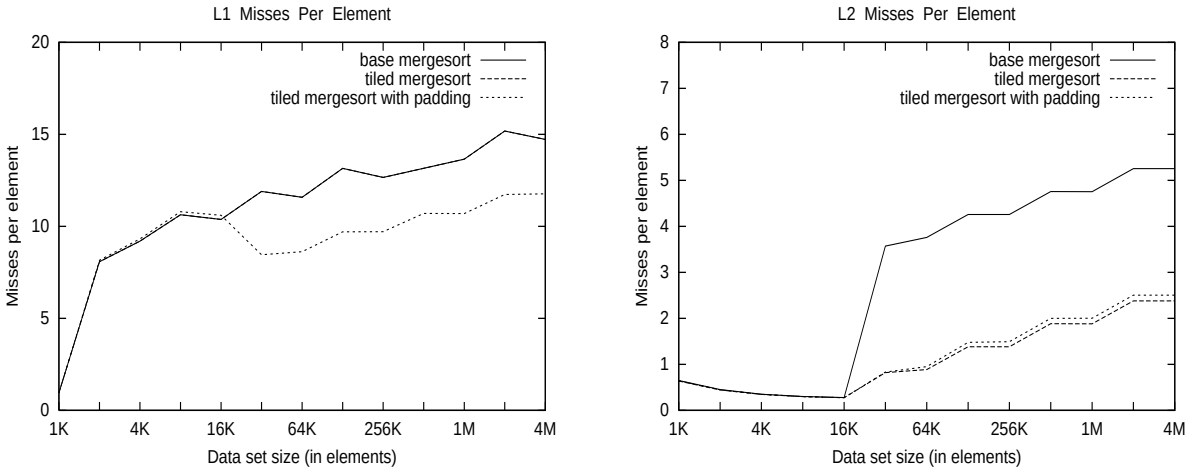

Fig. 2. Simulation comparisons of the L1 cache misses (left figure) and L2 misses (right figure) of the mergesort algorithms on the *Random* data set on the simulated Sun Ultra 5. The L1 cache miss curves (left figure) of the base mergesort and the tiled-mergesort are overlapped.

The 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, the potential conflict misses could be reduced by padding in this phase. However, the padding may not completely eliminate the conflict misses due to the randomness of the order in the data sets. Despite of this, our experimental results presented

in section 5 and the appendix using the 9 different data sets consistently show the effectiveness of the padding on the Sun Ultra 5.

3.2.2 *Multi-mergesort with TLB padding.* In the second phase of the multi-mergesort algorithm, 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 to hold the heads of the multiple lists. However, since the heads come from all the lists being multi-merged, the practical 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 following this paragraph). Our experiments indicate that the multi-mergesort significantly decreases 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, the performance degradation of the algorithm comes mainly from the high number of TLB misses since memory accesses are much 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 4K to 64KBytes. A TLB cache miss forces the system to retrieve the missing translation from the page table in the memory, and then to select a TLB entry to replace. When the data to be accessed is larger than the amount of data that all the memory pages in the TLB can hold, TLB misses 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. In practice, we have more than one data array being operated on at a time. Thus, the TLB can hold a limited amount of data in sorting.

Some processors' TLBs are not fully associative, but set-associative. For example, the TLB in the Pentium II and Pentium III processors is 4-way associative ($K_{TLB} = 4$). A simple blocking based on the number of TLB entries does not work well because multiple pages within a TLB space range may map to the same TLB set entry and cause TLB cache conflict misses.

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 gives an example of the padding for TLB, where the TLB is a direct-mapped cache of 8 entries, and the number of elements of 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 operates on a large data set, if 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 TLB miss per element of the multi-mergesort algorithm

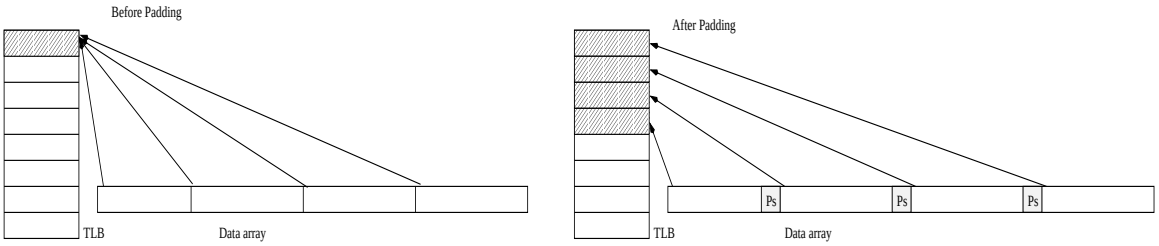

Fig. 3. Padding for TLB: the data layout is modified by inserting a page space at multiple locations, where $K_{TLB} = 1$, and $T_s = 8$.

becomes approximately

$$\frac{A}{A + K_{TLB}}, \tag{3}$$

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

$$\frac{C}{C + K_{TLB} \times T_s}. \tag{4}$$

Figure 4 shows the L2 misses and TLB misses of the 5 mergesort algorithms on the simulated Pentium II by the SimpleScalar, where L1 is a 4-way set associative cache of 16 KBytes, L2 is a 4-way associative cache of 256 KBytes, and TLB is a 4-way set associative cache of 64 entries. The simulation shows that the multi-mergesort and the multi-mergesort with TLB padding had the lowest L2 cache misses (see the left figure in Figure 4). The multi-mergesort had the highest TLB misses. These misses are significantly 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 between instruction count increase and performance gain

Figure 5 shows the instruction counts and the total cycles saved in percentage of the 5 mergesort algorithms compared with 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 counts. Taking advantage of low L2 misses of the multi-mergesort and significantly reducing the TLB misses by 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. We also show that the tiled-mergesort with padding did not gain performance improvement on the Pentium II. This is because this machine has a 4-way set associative cache where conflict misses are not major concerns.

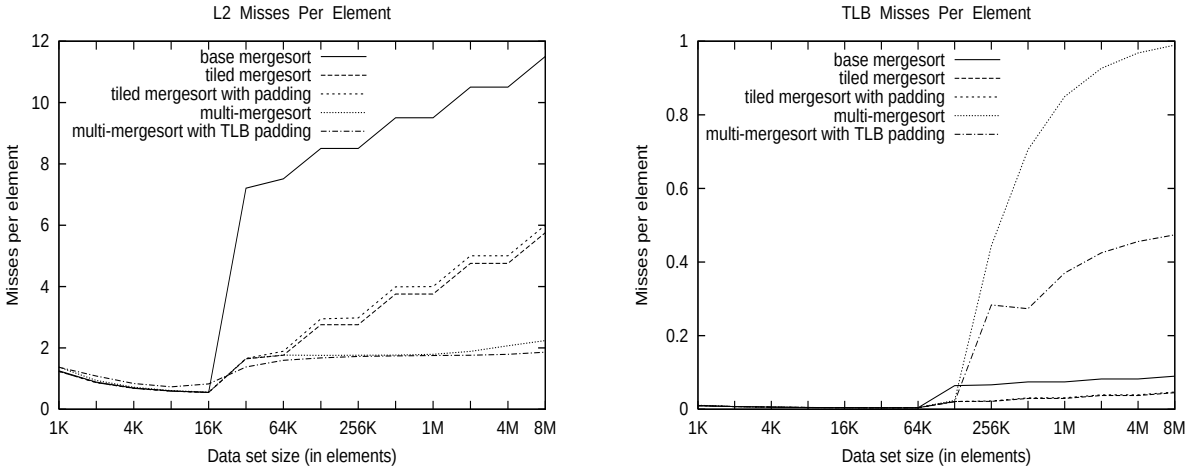

Fig. 4. Simulation comparisons of the L2 cache misses (left figure) and TLB misses (right figure) of the mergesort algorithms on the *Random* data set on the simulated Pentium II.

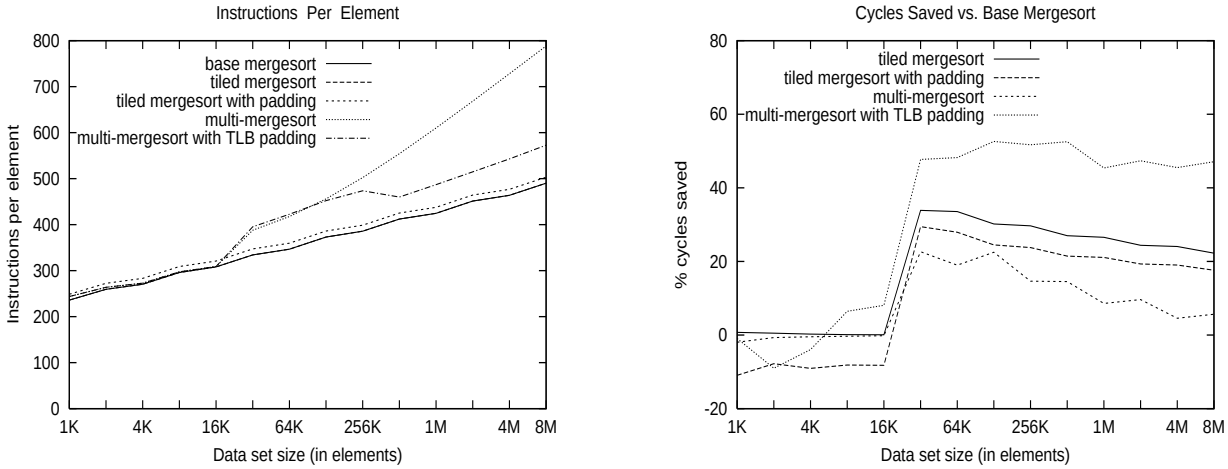

Fig. 5. Simulation comparisons of the instruction counts (left figure) and saved cycles in percentage (right figure) of the mergesort algorithms on the *Random* data set on the simulated Pentium II.

4. CACHE-EFFECTIVE QUICKSORT

We again first briefly evaluate the two existing quicksort algorithms concerning their merits and limits, including their cache locality. We present two new quicksort alternatives for further memory performance improvement. 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, each of which is hoped to 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 quicksort and its alternatives for cache optimizations are highly sensitive to the types of data sets. These algorithms did not work well 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 quicksort alternatives for cache optimizations which 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]. The reason this algorithm works well on balanced data sets is because the numbers of elements stored in the subarrays after the first two steps are quite similar and are sufficiently small to fit the cache capacity. This makes the Flashsort highly effective ($O(N)$). However, when the data set is not balanced, unbalanced amounts of elements among the subarrays are generated, causing ineffective cache usage, and making the flashsort as slow as the insertion sort ($O(N^2)$) in the worst case.

Compared with the pivoting process of the quicksort, the classification step of the flashsort is more likely to generate balanced subarrays, which is in favor of a cache optimization. On the other hand, the quicksort outperforms the insertion sort on unbalanced subarrays. Taking the advantages of both the flashsort and the quicksort, we present this 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 elements in each class.

4.2.2 *Inplaced Flash Quicksort.* We employ another cache optimization to improve temporal locality in the flash quicksort, hoping to further improve overall performance. This alternative is called *inplaced flash quicksort*. In this algorithm, the first and third steps are the same as the ones in the flash quicksort. In the second step, an additional array is used 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 reusage. 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 inplaced flash quicksort, on the *Unbalanced* data set on the simulated Pentium III which has a faster processor (500 MHz) 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 began to increase rapidly as the data set size grew. In contrast, the instruction counts of the flash quicksort and the inplaced flash quicksort had little change as the data set size increased. The simulation also shows that the L1 misses of the memory-tuned quicksort and the flashsort increased much more rapidly than that of the flashsort and the inplaced flashsort algorithms. The simulation results are consistent to our algorithm analysis, and show the effectiveness of our new quicksort alternatives on unbalanced data sets.

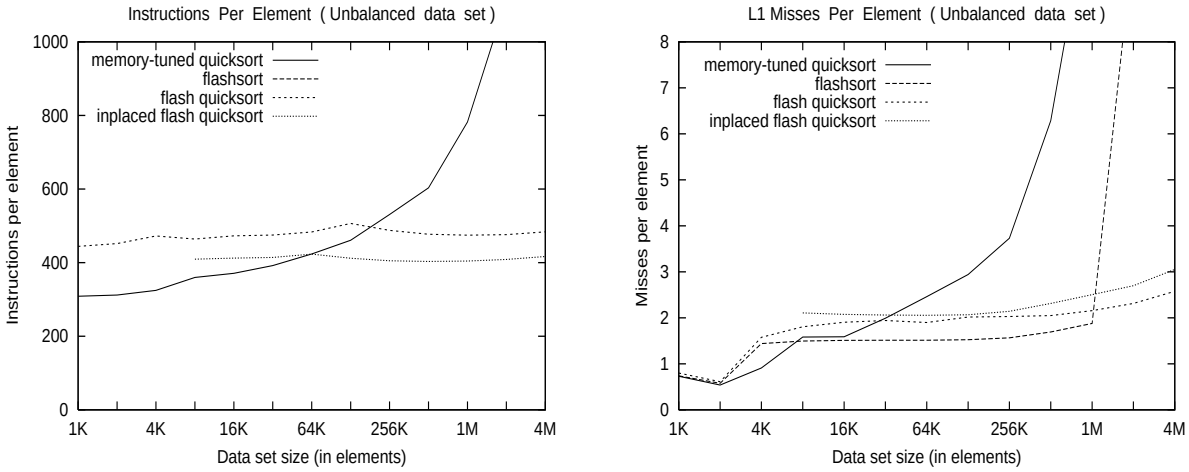

Fig. 6. Simulation comparisons of the instruction counts (left figure) and the L1 misses (right figure) of the quicksort algorithms on the *Unbalanced* data set on the simulated Pentium III. (The instruction count curve of the flashsort was too high to be presented in the left figure).

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

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.

used for experiments are limited by the memory size because 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 specifications on L1 cache refer to the L1 data cache, and all L2s are uniform. The hit times of L1, L2 and the main memory are measured by lmbench [5], and their units are converted from nanoseconds (ns) to their 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):

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

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

The performance results on all the data sets are quite consistent in our analysis. Since the performance of the sorting algorithms using different data sets on different machines is consistent in principle, 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), and 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 showed their effectiveness on large data sets. Figure 7 shows the comparisons of cycles per element among the 5 algorithms on the SGI O2 and the Sun Ultra 5.

The measurements on the O2 show that the multi-mergesort with TLB padding performed the best, with execution times reduced 55% compared with the base sort, 35% compared with the tiled mergesort, and 31% compared with 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 with the multi-mergesort, 26% compared with the base mergesort, and 23% compared with 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 super performance improvement on the O2 comes from its long memory latency (208 cycles). This makes the cache miss reduction techniques highly 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 associative, where the data cache padding is not as effective as the padding on a direct-mapped cache. In contrast, the Ultra 5's L1 cache is direct-mapped, and L2 is 4 times larger than that of the O2. Thus, the data cache padding is more effective than the TLB padding.

In order to further show the effectiveness of the tiled-mergesort with padding on a low-associativity cache system, such as the Sun Ultra 5, we plot the performance curves of the 5 mergesort algorithms using other 8 data sets on the Ultra 5 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% execution time reductions on the *Zero* data set compared with the base mergesort, the tiled mergesort, and the multi-mergesort, respectively. Using other data sets, we also show that the tiled mergesort with padding achieved 24% to 53% execution time reductions 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 among 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, thus, the issue of data cache conflict misses is not a concern (as we discussed in section 3.1). Since the TLB misses degraded performance the most in the multi-mergesort algorithm, the padding for TLB becomes very effective in improving the performance.

In summary, the tiled mergesort with padding on machines with direct-mapped caches is highly effective in reducing conflict misses, while the multi-mergesort with TLB padding performs very well on all the machines.

5.2 Quicksort performance comparisons

We used the *Random* data set and the *Unbalanced* data set to test the quicksort algorithms on the 4 machines. The 4 quicksort algorithms are: the memory-tuned quicksort, the flashsort, the flash quicksort, and the inplaced flash quicksort.

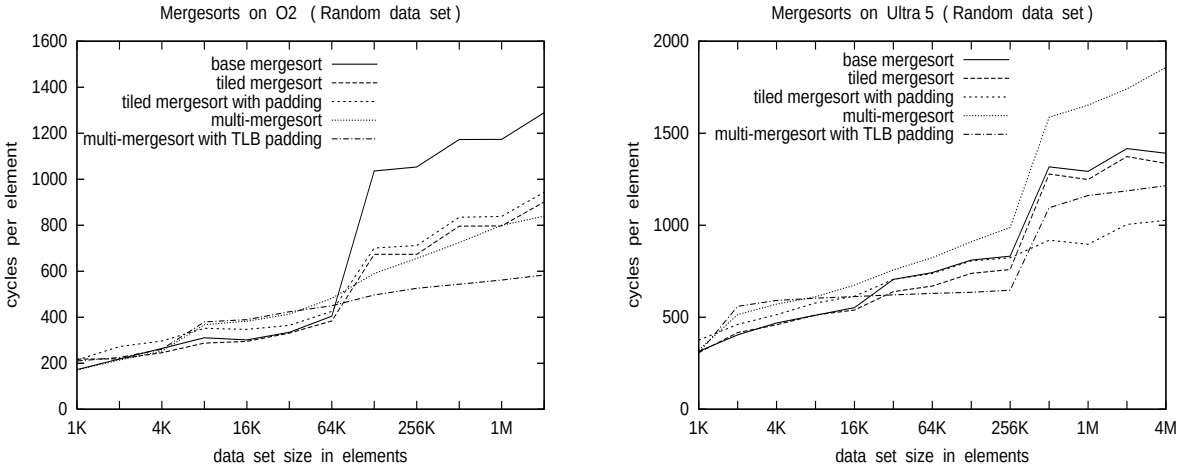

Fig. 7. Execution comparisons of the mergesort algorithms on SGI O2 and on Sun Ultra 5.

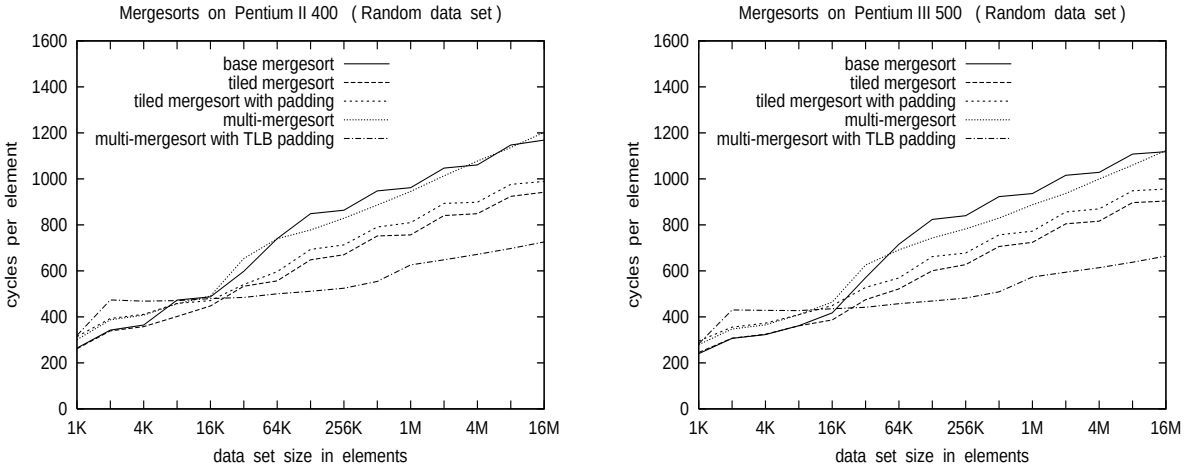

Fig. 8. Execution comparisons of the mergesort algorithms on Pentium II and on Pentium III.

Figure 9 shows the comparisons of cycles per element among the 4 mergesort 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 mergesort algorithms using the *Random* data set are comparable, where the memory-tuned algorithm slightly outperformed the others. In contrast, the performance results using the *Unbalanced* data set are significantly different. As we expected, the execution times of the flash quicksort and the inplaced flash quicksort are stable, but the memory-tuned quicksort and the flashsort performed much worse as data set sizes increased. The timing curves of the flashsort are even too high to be presented in the right figure in Figure 9.

Figure 10 shows the comparisons of cycles per element among the 4 mergesort

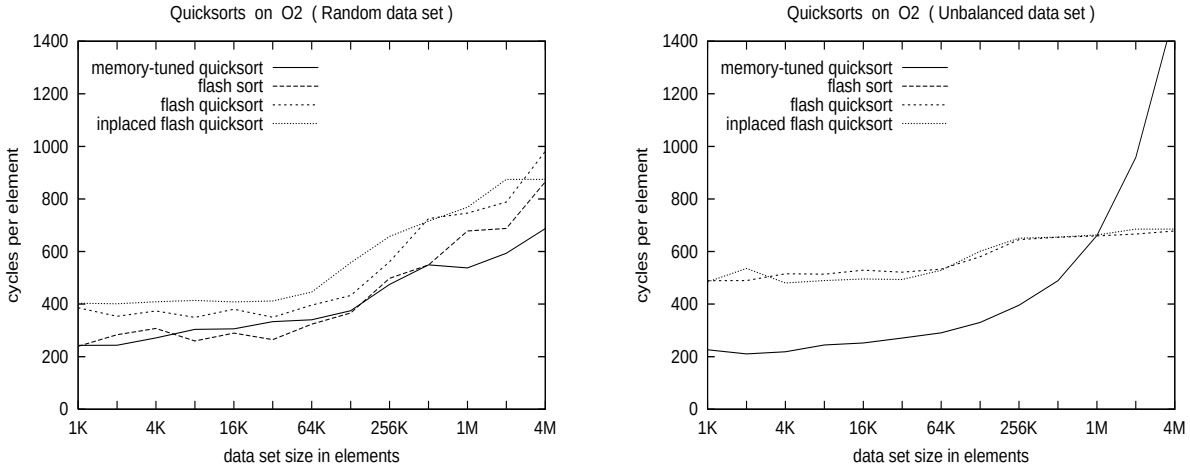

Fig. 9. Execution comparisons of the quicksort algorithms on the *Random* data set (left figure) and on the *Unbalanced* data set (right figure) on the SGI O2. (The timing curve of the flashsort is too high to be presented in the right figure).

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. The flash quicksort and the inplaced flash quicksort show their strong effectiveness 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 increased to 4M elements, the execution time of the memory-tuned quicksort is more than 3 times higher than the flash quicksort and the inplaced flash quicksort, and the execution time of the flashsort was more than 100 times higher than that of the others.

Figure 11 and Figure 12 show the comparisons of cycles per element among the 4 mergesort 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. The measurements on both Pentiums on the *Random* data set showed that the flashsort, the flash quicksort, and the inplaced flashsort had similar execution performance and reduced execution times around 20% compared with the memory-tuned quicksort. Again, the flash quicksort and inplaced flash quicksort significantly outperformed the memory-turned quicksort algorithm on the *Unbalanced* data sets on the two Pentium machines.

6. A PREDICTION MODEL OF PERFORMANCE TRADE-OFFS

The essential issue to be considered in sorting algorithms design and other algorithms design for memory optimization is the trade-off between the optimization achievement—the reduction of cache misses, and the optimization effort—the increment of 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.

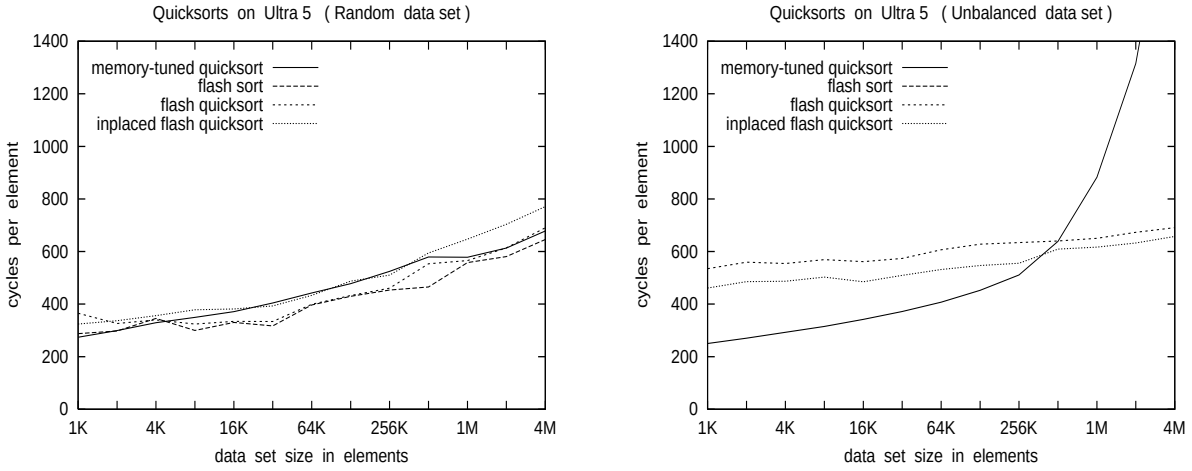

Fig. 10. Execution comparisons of the quicksort algorithms on the *Random* data set (left figure) and on the *Unbalanced* data set (right figure) on the Ultra 5. (The timing curve of the flashsort is too high to be presented in the right figure).

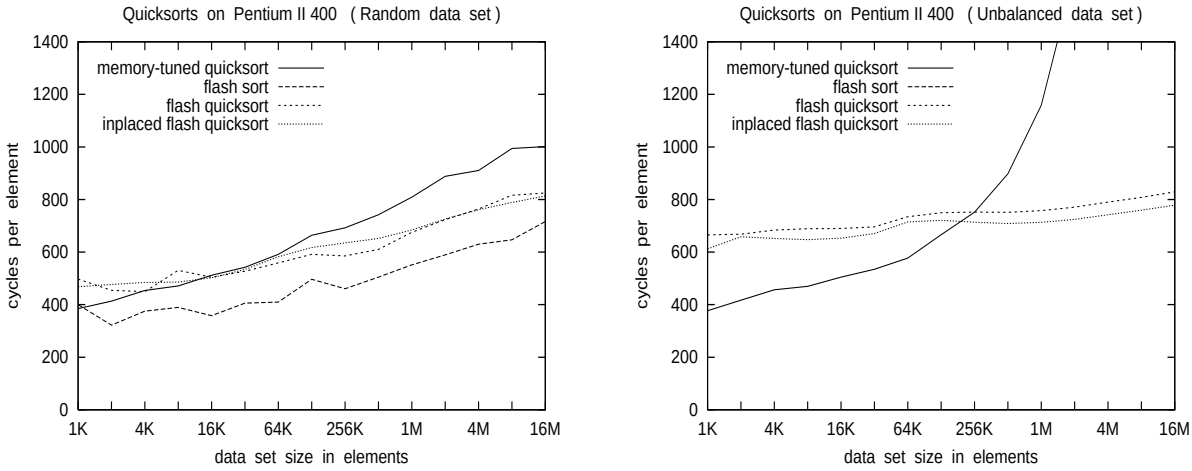

Fig. 11. Execution comparisons of the quicksort algorithms on the *Random* data set (left figure) and on the *Unbalanced* data set on the Pentium II. (The timing curve of the flashsort is too high to be presented in the right figure).

The execution time of an algorithm on a computer system based on Amdahl's Law [3] is expressed as

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

where IC is the instruction count of the algorithm, CPI is the number of cycles per instruction of the CPU for the algorithm, CA is the number of cache accesses of the algorithm in the execution, MR is the cache miss rate of the algorithm in

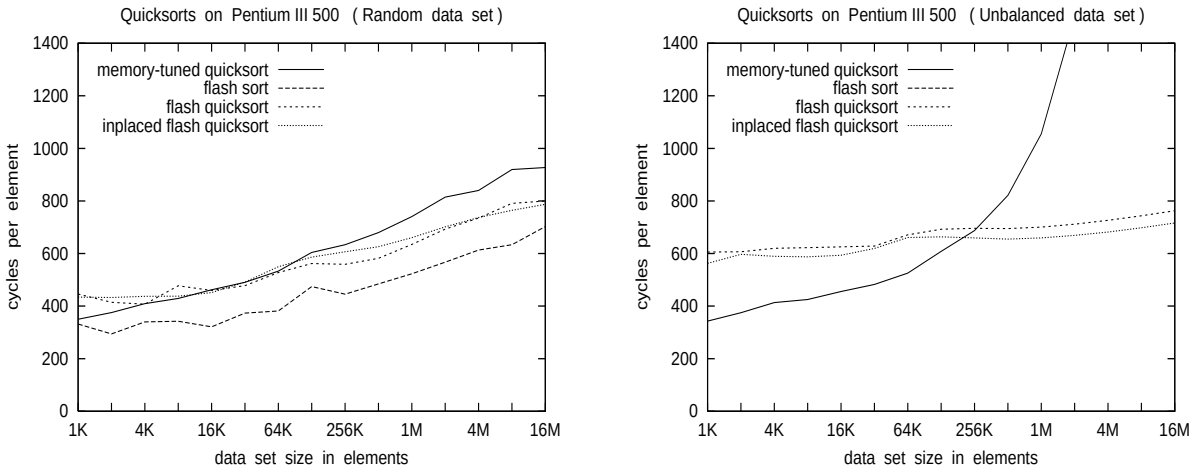

Fig. 12. Execution comparisons of the quicksort algorithms on the *Random* data set (left figure) and on the *Unbalanced* data set on the Pentium III. (The timing curve of the flashsort is too high to be presented in the right figure).

the execution, and MP is the miss penalty in cycles of the system. The execution time for a base algorithm, T_{base}, is expressed as

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

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

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

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 numbers of cache accesses are kept almost the same as that of 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:

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

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 execution time reduction rate, we must have

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

This model describes the quantitative trade-off between the instruction count increase and the miss rate reduction, and gives the condition for an optimized algo-

rithm to improve the performance of a base algorithm as follows:

$$\frac{\Delta IC}{\Delta MR} < \frac{CA \times MP}{CPI}. \tag{9}$$

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 $CA_{base} \neq CA_{opt}$, and obtain the condition for an optimized algorithm to improve the performance of a base algorithm as follows:

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

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 which oversize 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 reduction rate of 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 of products dating from 1995 to 1999 to show their effectiveness. The simulations provide more insightful performance evaluation. We show that mergesort algorithms are more architecture dependent, while the quicksort algorithms are more data set dependent. Our techniques of padding and partitioning can also be used for other algorithms for cache optimizations.

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 more and more commonly known to users. These parameters can also be defined as variables in the programs, which will be adaptively changed by users from machine to machine. Therefore, the programs are easily portable among different workstations.

ACKNOWLEDGMENTS

Many students in the Advanced Computer Architecture class offerred 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 on the work and helps on simulations.

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

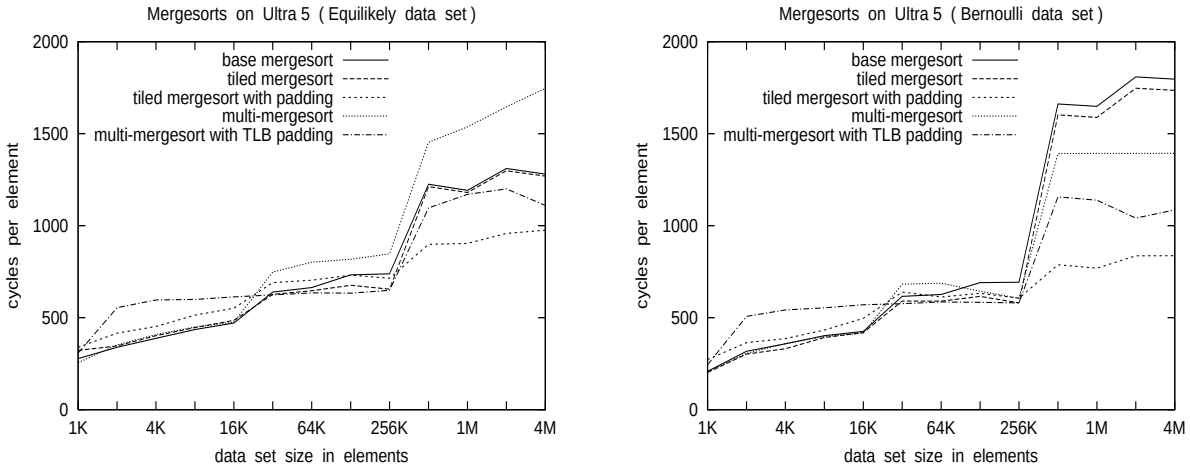

Fig. 13. Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Equilikely data set (left figure) and the Bernoulli data set (right set).

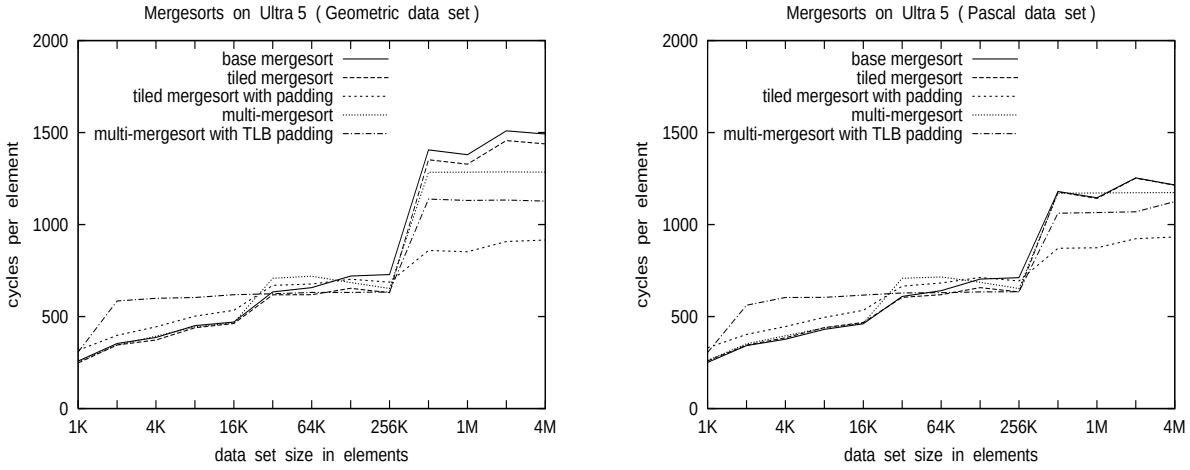

Fig. 14. Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Geometric data set (left figure) and the Pascal data set (right set).

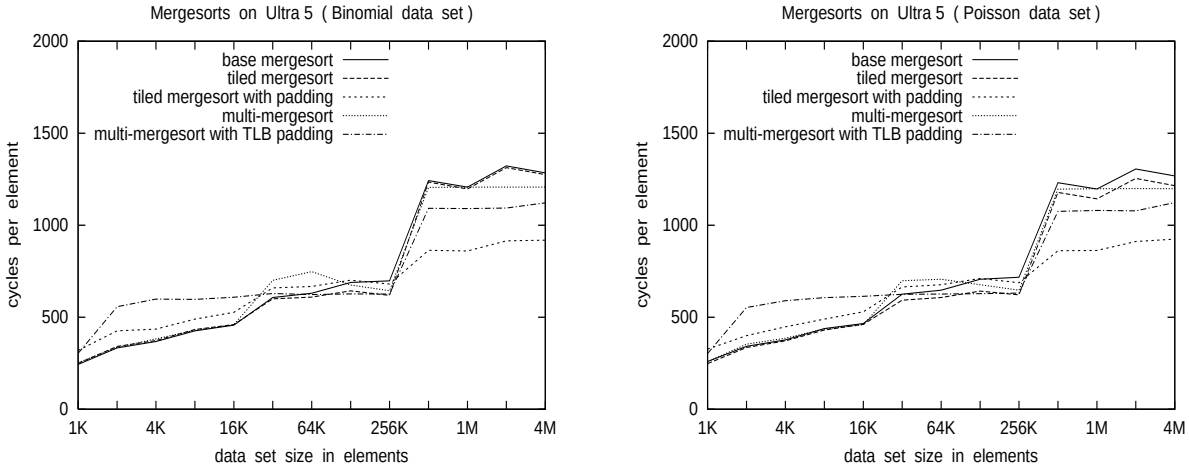

Fig. 15. Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Binomial data set (left figure) and the Poisson data set (right set).

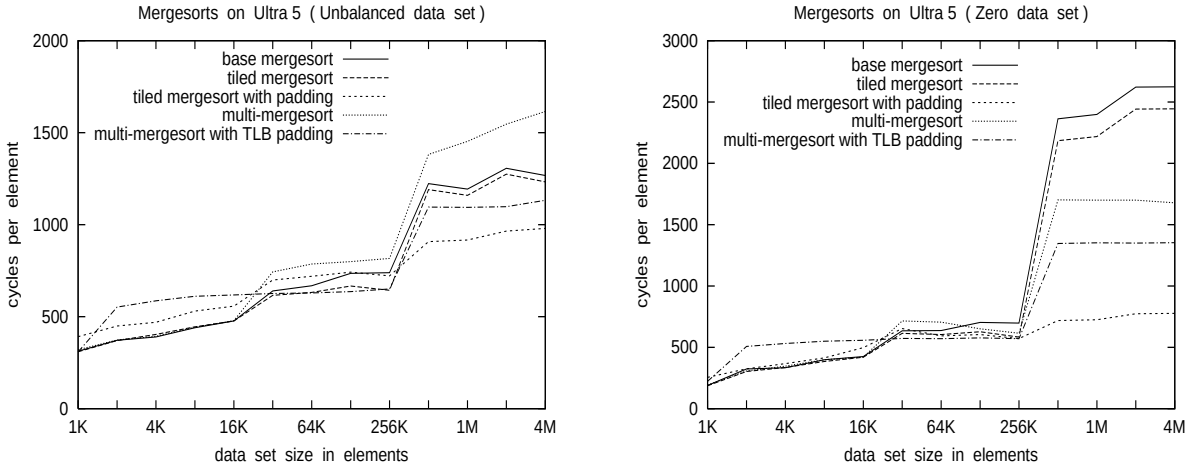

Fig. 16. Execution comparisons of the mergesort algorithms on Sun Ultra 5 using the Unbalanced data set (left figure) and the Zero data set (right set).

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. Dobb'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] 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.

[11] Z. Zhang and X. Zhang, "Cache-optimal methods for bit-reversals", *Proceedings of Supercomputing'99*, November, 1999.