SIAM J. SCI. COMPUT.
Vol. 22, No. 6, pp. 2113–2134

FAST BIT-REVERSALS ON UNIPROCESSORS AND SHARED-MEMORY MULTIPROCESSORS*

ZHAO ZHANG† AND XIAODONG ZHANG†

Abstract. In this paper, we examine different methods using techniques of blocking, buffering, and padding for efficient implementations of bit-reversals. We evaluate the merits and limits of each technique and its application and architecture-dependent conditions for developing cache-optimal methods. Besides testing the methods on different uniprocessors, we conducted both simulation and measurements on two commercial symmetric multiprocessors (SMP) to provide architectural insights into the methods and their implementations. We present two contributions in this paper: (1) Our integrated blocking methods, which match cache associativity and translation-lookaside buffer (TLB) cache size and which fully use the available registers, are cache-optimal and fast. (2) We show that our padding methods outperform other software-oriented methods, and we believe they are the fastest in terms of minimizing both CPU and memory access cycles. Since the padding methods are almost independent of hardware, they could be widely used on many uniprocessor workstations and multiprocessors.

Key words. cache optimizations, memory hierarchy, bit-reversals, shared-memory multiprocessors, parallel computing

AMS subject classifications. 68P05, 65Y20, 65Y05

PII. S1064827599359709

1. Introduction.

Many FFT algorithms require data reordering operations of *bit-reversal*. If the bit-reversal operations are not implemented properly, those FFT operations can slow down significantly. On the other hand, it is easy to improperly implement bit-reversals on uniprocessors and multiprocessors. This is because the performance of bit-reversals is highly sensitive to how caches and memory hierarchies are used in the implementations. In other words, a fast bit-reversal implementation must be cache effective. Several papers have well addressed the significance and effects of considering memory hierarchy to bit-reversals (e.g., [2], [11], and [15]). Besides the important usage for FFT, different versions of bit-reversal implementations can also be used as benchmark programs to evaluate the memory hierarchy of various computer systems.

With the rapid development of RISC and VLSI technology, the speed of processors has increased dramatically in the past decade. Processor clock rates have doubled every 1–2 years. Nevertheless, the memory speed has increased at a much slower pace. Therefore we have seen and will continue to see an increasing gap in speed between processor and memory, and this gap makes performance of application programs on both uniprocessor and multiprocessor systems rely more and more on effective usage of caches. Performance degradation of bit-reversals is mainly caused by cache conflict misses. Bit-reversals are often repeatedly used as fundamental subroutines for scientific programs, such as FFT. Thus, in order to gain the best performance, cache-

*Received by the editors September 17, 1999; accepted for publication (in revised form) November 20, 2000; published electronically April 12, 2001. This work is supported in part by the National Science Foundation under grants CCR-9400719 and CCR-9812187, by the Air Force Office of Scientific Research under grant AFOSR-95-1-0215, and by Sun Microsystems under grant EDUE-NAFO-980405. Preliminary results of this work were presented in the 1999 Supercomputing Conference, Portland, OR.

http://www.siam.org/journals/sisc/22-6/35970.html

†Department of Computer Science, College of William and Mary, Williamsburg, VA 23187-8795 (zzhang@cs.wm.edu, zhang@cs.wm.edu).

optimal methods and their implementations should be carefully and precisely done at the programming level. This type of performance programming for some special programs, such as bit-reversals, may significantly outperform an optimization from an automatic tool, such as a compiler.

A standard bit-reversal program is described as follows:

```
for i = 1, N
    Y[i'] = X[i]
```

The values of array X in their sequential positions i are copied to array Y in their bit-reversal positions, i' for $i = 1, \ldots, N$, where $N = 2^n$. The above program says that X is a bit-reversal reordering of Y. The indices of i and i' of X and Y are represented by a sequence of n binary digits. Positions i and its bit-reversal i' are defined in [11] as

$$i = \sum_{j=0}^{n-1} a_j 2^j \quad \text{and} \quad i' = \sum_{j=0}^{n-1} a_j 2^{n-1-j},$$

where a_j is either 0 or 1. For example, a 5-bit reversal of $i = 10010$ is $i' = 01001$.

The bit-reversal operations have following unique characteristics: First, in many implementations, each element in an array is used (read or written) only once for its copy operation. Thus, the reorderings have only spatial locality but no temporal locality for elements. Second, the loops follow certain sequences with high spatial locality. Bit-reversals are highly sensitive to problem sizes, cache sizes, and cache line sizes. Since the data array sizes are a power of 2, multiple elements stored in different memory locations could map to the same cache line, causing severe cache conflict misses and cache thrashing. The reason is simple. Most commercial computers use direct-mapped or n-way associative caches where the mapping functions of cache sizes are also related to powers of 2.

We use an identical unit, called an "element," to represent the sizes of data arrays, caches, and others such as buffers and blocking. One element may represent a 4-byte integer, a 4-byte floating point number, or an 8-byte double floating point number. Because the sizes of caches and cache lines are always a multiple of an element in practice, this identical unit for all sizes is practically meaningful for both architects and application programmers and makes the discussions straightforward. Here are the algorithmic and architectural parameters we will use to describe cache-optimal methods of bit-reversals.

- C: data cache size, which could be further defined as C_{L1} and C_{L2} for data cache sizes of L1 and L2, respectively.
- L: the size of a cache line, which could be further defined as L_{L1} and L_{L2} for cache lines of L1 and L2, respectively.
- K: cache associativity, which could be further defined as K_{L1} and K_{L2} for cache associativity of L1 and L2, respectively.
- K_{TLB}: translation-lookaside buffer (TLB) cache associativity. (A TLB cache is a small buffer that holds most recent memory page mappings. The concept will be discussed in detail later in the paper.)
- T_s: number of entries in the TLB cache.
- N: the data size for the bit-reversal vector of size $N = 2^n$, where n is the number bits used in the vector index.
- B_{cache}: blocking size of a $B \times B$ submatrix for cache.
- B_{TLB}: blocking size for TLB.
- P_s: a memory page size.

In this paper, we examine different methods using techniques of blocking, buffering, and padding for efficient implementations. We evaluate the merits and limits of each technique and its application and architecture-dependent conditions for developing cache-optimal methods. Although our methods are developed for out-of-place bit-reversals, they are also applicable to in-place bit-reversals where X and Y are the same array.

Symmetric multiprocessor (SMP) systems have become practical and cost-effective servers for scientific computing and other applications. Although parallel efficiency and communication latency reduction are major performance concerns, computations on an SMP share many common considerations with uniprocessors. The most important one is the effective usage of memory hierarchies. When the cache locality of each processor is effectively exploited, the memory accesses to the shared-memory will be reduced, and so will be the memory access contention. People have studied parallel data reordering algorithms on distributed-memory systems with special networks, such as hypercubes (see, e.g., [6] and [9]). In this study, we target parallel bit-reversals on SMPs and show the significant impact of the cache and TLB considerations for efficient method development and implementations. We also evaluate the performance impact of SMP interconnection networks.

Our algorithm designs and implementations are optimized by considering several nontraditional but practical and performance-effective factors, namely, the programming complexity, memory space requirement, instruction count, cross interference among the data arrays, and program portability. We will summarize the limits and merits of different bit-reversal methods based on these considerations after we have discussed the designs and presented the performance results, aiming at providing a guideline for performance programming and memory performance optimization for other scientific computing applications.

We present two contributions in this paper: (1) Our integrated blocking methods, which match cache associativity and TLB cache size and which fully use the available registers, are cache-optimal and fast. (2) We show that our padding methods outperform other software-oriented methods and believe they are the fastest in terms of minimizing both CPU and memory access cycles. Since the padding methods are almost independent of hardware, they could be widely used on many uniprocessor workstations and SMP multiprocessors.

The rest of the paper is organized as follows. We discuss the inherently blocking nature of bit-reverse operations and the effectiveness and limits of blocking techniques for solving the problems in section 2. In section 3, we evaluate a software buffering technique and our methods using existing hardware components for implementing the data reordering. Our new method integrating blocking and padding will be presented in section 4. We discuss blocking and padding techniques for TLB in section 5. The experimental measurements and analyses for evaluating different methods on uniprocessor workstations and SMP multiprocessors will be reported in sections 6 and 7. We summarize the work in section 8.

2. Blocking for bit-reversals. The blocked memory access patterns of bit-reversals can be easily viewed when we convert the one-dimensional vector to a two-dimensional equivalent array in Figure 1. All the reordering elements and elements in other groups will be allocated along the column in the two-dimensional equivalent array forming a block.

In this blocking method, the bit-reversal reordering is performed block by block, where the operations for each block are implemented similarly to the Evans method

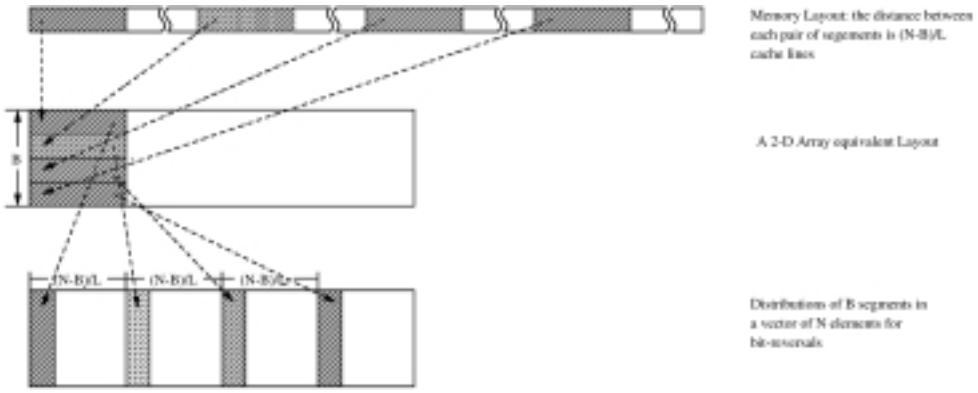

FIG. 1. *Memory layout of a blocked bit-reversals, where* $B = B_{cache}$.

[7]. (The Evans method is used to construct a hybrid method in [11].) The program in the appendix presents such an implementation along with padding technique. (The padding technique will be discussed in section 4.) The blocking algorithm we have used can be classified as a hybrid method.

In general, for a bit-reversal vector of $N = 2^n$ elements, the block size B_{cache} is a power of 2, denoted by $B_{cache} = 2^b$. Each of the B_{cache} elements in X has the address format of fg, where g is B_{cache} bits and f has $n - b$ bits. Each of the corresponding B_{cache} elements in Y has the address format of $g'f'$. Therefore, the distance between two nearest elements in the same group in Y is $2^{n-b} = N/B_{cache}$.

Choosing the cache line size as the minimum blocking size ($B_{cache} = L$), we can easily calculate the maximum N's for the bit-reversal vector based on different data cache sizes. For example, for a large cache of 2 MB, the blocking technique is effective up to an 18-bit-reversal reordering which represents 268,144 data elements, where each element is an 8-byte double type, and the cache line is 32 bytes. In practice, the data size of bit-reversals could easily be larger than $n = 20$ [11].

3. Blocking with buffers. As we have shown, the effectiveness of blocking is limited by the size of the data arrays. In theory, the smallest blocking size could be 2×2. A cache line in a modern processor usually holds more than 2 elements, i.e., is larger than 16 bytes. If we choose a 2×2 block, the data in a cache line will not be fully used before their replacement, causing more cache misses in the reorderings. The bit-reversal reordering demands large cache space to make blocking effective. In order to effectively use limited cache space, Gatlin and Carter [8] present an effective method using an additional buffer to first hold the conflict-missed elements of a block in one array temporarily and then copy the block to their reordered positions in the other array. In this section, we discuss implementations of blocking methods supported by both software and hardware buffers.

3.1. Blocking with a software buffer and its limits. Because this buffer is defined in a reordering program, we call it "software buffer." This buffer shares the allocation space with the data arrays X and Y in the cache.

There are two major limits in this approach. First, the buffer itself may interfere with arrays of X and Y, causing additional access conflicts. This interference is certain when the sizes of X and Y are larger than the size of the cache, C. Each cache block or set is mapped from arrays X and Y more than once. No matter where the buffer is

located in the cache, it will interfere with them. The larger the buffer size, the more interference will occur.

The second limit is the additional copy overhead time involved in moving data from the array X to the buffer and then in moving them to the target array in their reordered positions. This overhead exactly doubles the instruction cycles for data copying. The data copy through a buffer is a worthy investment if the number of cycles lost from cache misses is much higher than the additional CPU cycles for the data copy.

To overcome the two limits, we propose several alternatives to eliminate cache interference caused by the software and to reduce or eliminate the data copy time.

3.2. Cache structure dependent blocking. We will present several blocking methods which depend on the cache organization of the running machine. These methods can be implemented at the user programming level.

Blocking based on set associativity. The cache associativity, K, is an important factor to consider for blocking. If $K \geq L$, an $L \times L$ or a $K \times K$ blocking method for bit-reversals would effectively avoid conflict misses. Because the hit time is a less sensitive performance factor than the cache misses in the L2 cache, a higher associativity of the L2 cache is more effective than that of L1. If a cache line holds 4 double floating point elements ($L = 4$ elements of 32 bytes in Pentium processors), a 4×4 blocking method without any data buffer is able to fully use the cache associativity. The blocking method would gain more benefit from caches of associativity higher than 4, such as a design in [20].

What would we do if the associativity is not sufficiently high for the blocking, or $K < L$? One solution is to make a $K \times L$ rectangular blocking. Unfortunately bit-reversals require an $L \times L$ blocking.

Supplement with registers. We may also consider using the available registers to supplement a low associativity cache. The number of registers available to a user program is limited. Normally, a uniprocessor provides up to 16 registers to users. For example, for a 2-way associative cache, we need 8 registers to buffer 2 additional cache lines so that we could effectively make a 4×4 blocking as if we ran the program on a 4-way associative cache.

We develop a more efficient blocking method for bit-reversals, which requires only $(L-K) \times (L-K)$ registers. The operation sequence of this method is in three steps: (1) The $L - K$ cache lines of X are stored in K cache lines of Y and accessed by copying its $(L - K) \times K$ elements to Y in the reordered positions and copying the rest of $(L-K) \times (L-K)$ elements to a buffer consisting $(L-K) \times (L-K)$ registers. (2) The rest of K lines of X are brought to the cache set, and its $K \times K$ elements are copied to Y in the reordered positions. (3) Finally, the $(L - K) \times (L - K)$ elements in the register buffer and the rest of the $(L - K) \times K$ elements are copied to Y in their reordered positions. A cache set will be used more than twice if $K < L/2$.

Besides the advantage of no access conflicts between the register buffer and the arrays of X and Y, there is another advantage of using registers to buffer the data in a load/store processor. A data copy through the registers from X to Y is equivalent to the two-step process of load and store, and thus there will be no additional overhead. We will show our experimental performance in section 5.

Using registers as the buffer. If the cache is direct-mapped, we have to fully rely on a buffer for blocking. Here we discuss some ways to use registers to serve the buffer in order to eliminate the potential cache conflicts and eliminate extra data

copying by taking advantage of the load/store operations. The number of registers for a buffer of $L \times L$ elements is determined by the number of elements a cache line can hold. The length of a cache line of the L1 cache in some processors, such as Sun SPARC Micro I and II, is $L = 2$ of 16 bytes, which holds only two floating point elements. The blocking size could be as small as 2×2 using a buffer of 4 registers.

The cache line length of the L1 cache in many advanced workstations is 32 bytes, such as the Sun Ultra and Intel Pentium processors, each of which holds 4 double floating point elements. In this case, we need a buffer of $4 \times 4 = 16$ registers for a blocking. This would be difficult due to the limited number of available registers. We have two solutions for this. First, we use only the number of registers available to form a smaller buffer than it should be, which will not make each cache line fully used and will cause additional cache misses. Our experiments show that this blocking method of using a buffer of insufficient number of registers still achieves a reasonable performance improvement and outperforms the implementation using a software buffer.

The second method is to further reduce the size of the buffer, which reduces the required number of registers by using our $(L - K) \times (L - K)$ blocking method.

L1 cache versus L2 cache. The main objective of building two-level caches is to make the L1 cache small enough to catch up to the cycle time of the fast CPU and to make the L2 cache large enough to capture as many accesses as possible [12]. In practice, the data size of a bit-reversal is larger than the size of the L2 cache. L1 and L2 caches offer different sizes of the cache line, L, and the associativity, K. Both of the following alternatives are effective for blocking. (1) Taking advantage of a short cache line and fast hit time of the L1 cache, we could effectively use limited registers as the buffer and make a small $L \times L$ blocking effective. (2) Taking advantage of high associativity of the L2 cache, we could effectively use both associativity and supplemental registers as the buffer and make a large $L \times L$ blocking effective.

3.3. Victim-cache-aided blocking. Victim cache [13] is a small fully associative cache serving as the buffer containing only cache blocks due to conflict misses from L1 cache. This is an on-chip cache connected between L1 and the next level cache or memory. On a miss in L1, the victim cache is first checked before going to the next level. If the missed block is found there, the victim cache block and the L1 cache block are swapped and then the block is delivered to CPU from the L1 cache. Victim cache has been available in some commercial workstations, such as HP7200.

The minimum number of victim cache lines required for $L \times L$ blockings of transpose and bit-reversal reorderings is $L - K$. In the execution, $L \times L$ elements of each blocking are allocated in a set of K lines in L1 cache, and the rest of the elements are allocated in the $L - K$ lines of the victim cache. The victim cache is able to hold all the conflict misses in the reorderings by an $L \times L$ blocking. In addition, a conflict miss in the L1 cache that hits in the victim cache has only one additional cycle miss penalty. Thus, a simple $L \times L$ blocking method would be effective if such a victim cache is available.

However, the victim cache does not have a direct connection with the CPU. When a data hit happens in the victim cache, it has to be first swapped to the L1 cache and then delivered to CPU. This swapping operation is unnecessary for our reordering algorithms. Without counting the cold misses of bringing the elements in the first column for an $L \times L$ blocking and considering the LRU replacement policy, the entire blocking will have $L \times (L-1)$ conflict misses in the L1 cache, which are then found in the victim cache. This also means that each of such a blocking needs $L \times (L - 1)$ additional swapping cycles between the L1 cache and the victim

cache, which is independent of the associativity, K. In contrast with the blocking method based on the associativity supplemented by registers, the swapping cycles in the victim cache are additional overhead. Despite this, a victim-cache-aided blocking is more efficient than a blocking method with a software buffer because there are no cross interference conflicts between the victim buffer and arrays of X and Y.

4. Blocking with padding. Padding is a technique that modifies the data layout of a program so that the conflict misses are reduced or eliminated. The data layout modification can be done at run-time by system software [3, 19] or at compile-time by complier optimization [16]. Sharing the same objective of compiler optimization to change the base addresses of potentially conflicting cache blocks in the reorderings, we insert padding variables inside the data array. For example, the padding can be done as part of the last butterfly for the decimation in an FFT computation without additional cost, and the output is not padded.

However, we notice that this free padding opportunity may not be easily found, and the bit-reversal result may be padded in some cases. For example, the padding of a recursive implementation of the Cooley–Tukey FFT algorithm [5] is more complex than the padding in our implementations. The padding method produces padded results in a vector if the bit-reversals are done in an inplaced fashion. The accesses to the padded results need to go through a simple address converting process with additional CPU cycles. In addition, our methods target bit-reversals based on the data size of powers of 2. However, FFT algorithms are not limited to this data size. If the data size is not a power of 2, the padding method will be more complex to implement. Poor memory performance of bit-reversals has been reported even for nonpower of 2 data sizes (see, e.g., [2]).

Since the data arrays of bit-reversals form a vector whose size is power of 2, the padding is highly regular, inserting L elements or a cache line space starting at the vector positions of N/L, $2 \times N/L$, ..., and $(L-1) \times N/L$. Using L elements or a section data of a cache line to separate the vector at these L points can completely eliminate the cache conflicts caused by the address mapping based on powers of 2. Again during execution, the reordering data copies are directly conducted between the arrays X and Y without going through a data buffer. Another advantage is that the number of padding elements needed is only $L \times L$ or L cache lines and is independent of the data array size, N. Compared with the data size of bit-reversals, the number of padding elements is insignificant. Figure 2 shows how the data layout of a bit-reversal vector is modified by padding so that conflict misses are eliminated.

Compiler optimization targets a large range of application programs and automatically inserts padding variables in the programs for users. An optimal padding is application program dependent. For example, padding positions are different from different applications in order to effectively change base addresses of conflicting cache blocks [18]. Based on the unique nature of the data reordering, the optimal padding unit used by our methods for bit-reversals is a cache line with L elements. In contrast, a compiler optimization normally uses an element as the basic padding unit. How many padding units to use and where to pad in the data arrays are determined by some approximation models which may not precisely fit the unique memory access patterns of each case. In addition, applying the padding technique to bit-reversals embedded in applications would not increase complexity in the entire computation. For example, when a padded bit-reversal is performed in an FFT computation, it has little effect on the neighboring butterfly operations.

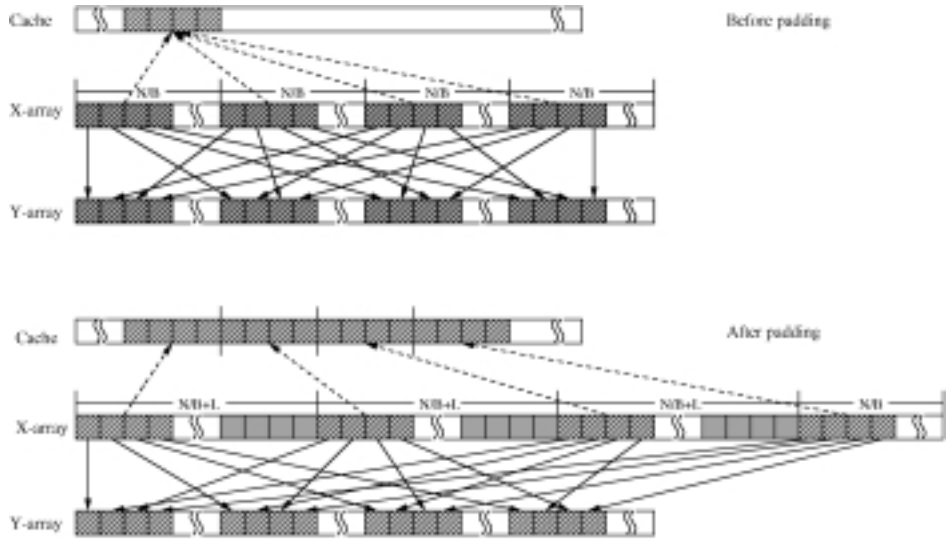

FIG. 2. *Data layout of a bit-reversal is modified by padding, where* $B = B_{cache} = L$.

5. Blocking and padding for TLB. The TLB is a special cache that stores the most recently used virtual-physical page translations for memory accesses. The TLB is a small and usually fully associative cache. Each entry points to a memory page of 4 KB to 64 KB. The page size is normally fixed at the level of operating systems and cannot be changed by user programs. A TLB cache miss will make the system retrieve the missing translation from the page table in memory and then select a TLB entry to replace. When the data to be accessed in our blocking method is larger than the amount of data of all the memory pages that the TLB can hold, we will have TLB thrashing. In this section, we will discuss and present blocking and padding methods for TLB cache optimizations.

5.1. Blocking for a fully associative TLB. Before giving a general model to show how the blocking size is affected by the TLB size, let's go through an example to show that a moderate N for bit-reversals would easily lead to TLB cache thrashing. The 64 pages in the TLB of the Sun UltraSparc-II processor hold $64 \times 1024 = 65536$ elements, which represents a 16-bit-reversal of $N = 2^{16}$. Since we have two vectors X and Y, the TLB can hold a 15-bit-reversal of $N = 2^{15}$ elements. This is also consistent with our experiments on this machine, where execution time per element was a constant until $n = 15$, but sharply increased at $n = 16$ bit-reversals caused by the TLB misses.

In our cache-optimal methods, we include an outer loop to form a blocking for TLB, whose size is denoted as B_{TLB}. The blocking size of B_{TLB} for bit-reversals when $N \geq T_s \times P_s$ is

$$B_{TLB} \leq T_s,$$

where P_s is the page size in elements, and T_s is the number of entries of the TLB. On the other hand, the B_{TLB} should be chosen as large as possible to make effective use of the page space. When $N < T_s \times P_s$, the data size of a bit-reversal will be less than the data size covered by the TLB. Thus there is no need for TLB optimizations.

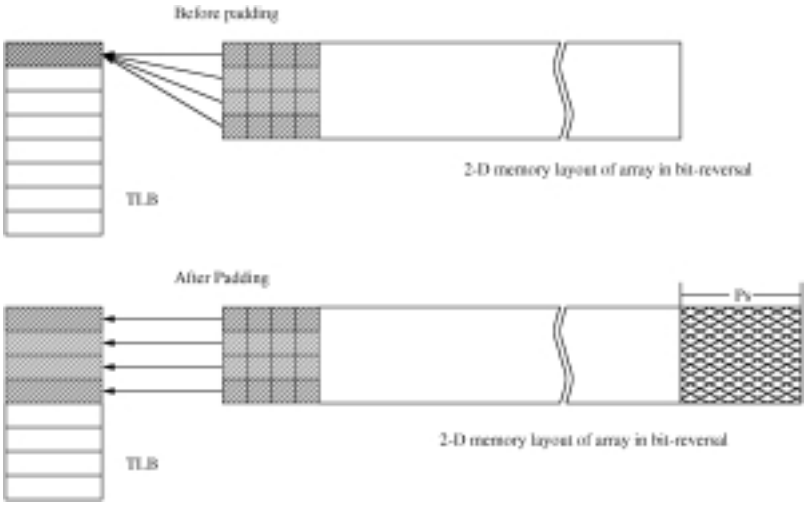

FIG. 3. *Padding for TLB: the data layout is modified by inserting a page space at multiple locations, where* $B_{TLB} = 4$, $K_{TLB} = 1$, $T_s = 8$.

5.2. Padding for a set-associative TLB. Some processors' TLBs are not fully associative, but set-associative. For example, the TLB in the Pentium-II 400 processor is 4-way associative ($K_{TLB} = 4$). A simple blocking based on the number of TLB entries is not cache-optimal, because multiple pages within a TLB-size-based blocking may map to the same TLB cache set and cause TLB cache conflict misses.

If the size N of a bit-reversal vector is a multiple of $T_s \times P_s$, where T_s is the number of TLB entries and P_s is the page size in elements, and if $K_{TLB} < B_{TLB}$, then TLB cache conflict misses will occur. This could easily happen in practice. For example, on the Pentium-II 400, N is equal to $128K$ elements (one element = 8 bytes) for a 17-bit-reversal, and this N is two times the value $T_s \times P_s$ of the machine, where $T_s = 64$, and $P_s = 1024$ elements.

In a way similar to the technique of padding for the data cache, we insert a page of elements or a page of space starting at the vector positions of N/L, $2 \times N/L, \ldots$ and $(L-1) \times N/L$ to eliminate the conflict of TLB cache misses. Figure 3 gives an example of the padding for TLB, where the TLB is a direct-mapped cache of 8 entries, blocking size is $B_{TLB} = 4$, and the number of elements of a row is a multiple of 8 page elements. Before padding, each of blocking row is mapped to the same cache line of the TLB. After padding, these rows are mapped to different cache lines of the TLB.

Combining padding for data cache and padding for TLB cache, we are inserting $L+P_s$ elements or a page plus a cache line space in L locations separated by a distance of N/L elements.

In practice, we selected more than N/L points to insert the padding variables to eliminate both data cache and TLB conflict misses. This approach could effectively merge two nested paddings (one for data cache and the other one for TLB) into a single one. An optimal number of inserting points can be easily determined experimentally based on the size of the TLB cache. The padding optimizations are all based on L2 cache in our experiments.

Partial index mapping addresses of bit-reversals are precalculated and stored in a small table as shown in the program in the appendix. This approach further improves

TABLE 1

Architectural parameters of the 5 workstations we have used for the experiments. All specifications on L1 cache refer to the L1 data cache, and all L2s are uniform. Each L2 cache block on UltraSPARC-IIi consists of 2 16-byte subblocks. The hit times of L1, L2 and the main memory are measured by lmbench [14], and their units are converted from nanosecond (ns) to their CPU cycles.

Workstations	SGI O2	Sun Ultra 5	Sun E-450	Pentium	XP1000
Processor type	R10000	UltraSparc-IIi	UltraSparc II	P-II 400	Alpha 21264
Clock rate (MHz)	150	270	300	400	500
L1 cache (KBytes)	32	16	16	16	64
L1 block size (Bytes)	32	32	32	32	64
L1 associativity	2	1	1	4	2
L1 hit time (cycles)	2	2	2	2	3
L2 cache (KBytes)	64	256	2048	256	4096
L2 block size (Bytes)	64	64	64	32	64
L2 associativity	2	2	2	4	1
L2 hit time (cycles)	13	14	10	21	15
TLB size (entries)	64	64	64	64	128
TLB associativity	64	64	64	4	128
Memory latency (cycles)	208	76	73	68	92

the performance because the table will be accessed in the cache during the computation, and the precalculation overhead is trivial. The time for the precalculation is included in the total execution time.

6. Experimental results and performance evaluation. We have implemented and tested all the bit-reversal methods discussed in the previous sections on an SGI O2 workstation, a Sun Ultra-5 workstation, a Sun SMP server E-450, a Pentium PC, and a Compaq XP1000 workstation. We will present and evaluate the performance of different methods on different machines.

6.1. Experimental environment and evaluation methodology. We used "lmbench" [14] to measure the latencies of memory hierarchies at different levels on each machine. The architectural parameters of the 5 machines are listed in Table 1.

We focus the performance evaluation on methods and implementations of bit-reversals in this paper. We compared all our methods with the method of blocking with a software buffer which was recently published in [8]. We denote this method as "bbuf-br"—blocking with buffer for bit-reversals. Two of our methods are experimentally compared: "breg-br"—blocking with associativity and registers for bit-reversals, and "bpad-br"—blocking with padding for bit-reversals. We have also applied blocking or padding technique for the TLB in these two methods based on the TLB associativity.

All the programs use a standard subroutine to calculate the bit-reversal value for a given address. The execution times were collected by "gettimeofday()," a standard Unix timing function. The resolution of this function is 1 μs on the machines being measured, which is significantly smaller than the execution times of any programs we have measured. A small bit-reversal table is precalculated, and we exclude this calculation time. The reported time unit is cycles 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 of the bit-reversal program. Besides the different methods of bit-reversals, we also measured the execution time of a program copying elements between X and Y. This program has the same number of data copying operations with a continuous memory access pattern. We use the execution time of this program to provide a base line reference for bit-reversal programs and show how close a bit-reversal execution is to its ideal time. We denote this reference program as "base." Each method is further divided into "float" data type using 4 bytes to represent an element, and "double" type using 8 bytes to represent an element. The data type divisions will show the performance impact of the cache line length.

For all experiments on different machines, the bit-reversal programs first call a routine to flush the cache to make sure that all the data are allocated only in the memory. All experiments were repeated multiple times.

6.2. Effects of TLB and virtual memory. Before measuring and comparing the performance of different bit-reversal methods, we experimentally evaluated the effects of TLB and virtual memory to confirm our assumptions and analyses.

Selection of TLB blocking size. The TLB blocking size is a sensitive performance parameter to be selected, which is determined by the size of the TLB if it is fully associative. We executed program "bpad-br" (blocking with padding for bit-reversals) with $n = 20$ on a single node of Sun E-450 by changing the blocking sizes for TLB from 8 to 128. The TLB of the E-450 is a fully associative cache with 64 entries. Figure 4 shows the measured cycles per element of the program of different blocking sizes on the node. Our experimental results are consistent with our analyses in the previous section. When the blocking size for TLB was 64, the execution time curve increased sharply. This is because arrays X and Y together demanded more than 64 pages and caused TLB thrashing.

Virtual memory versus physical memory addresses. All our analyses are based on cache mappings between memory pages in the virtual address space and cache blocks in the physical memory address space. This assumes that contiguous memory pages will be contiguously mapped to the cache. This assumption is guaranteed for the virtual-address caches [4]. However, all our experiments have been performed on machines with physical address L2 caches. Since the virtual-physical translations for L2 caches are handled by operating systems, our assumptions may sometimes be inaccurate. In order to show that many operating systems attempt to map contiguous virtual pages to cache blocks contiguously so that our virtual-address-based study is practically meaningful and effective, we conducted a simulation by using the SimOS [17] and measurements on different workstations to observe how an operating system makes translations from virtual memory addresses to their physical addresses.

The SimOS simulates a complete hardware of SGI machines and runs the IRIX 5.3 operating system in the simulation. We executed a blocking-only program of bit-reversals using the cache line L as the blocking size. The bit-reversal vector size was changed from $n = 15$ to $n = 22$. We measured the miss rates on array X. The cache size was set to 2 MB holding two double type arrays up to $n = 18$ in the virtual memory space. Figure 5 gives consistent results from the SimOS simulation: when $n > 18$, the miss rate on array X was sharply increased to 100% from 12.5%.

From this experiment, we have observed that virtual-physical translations from

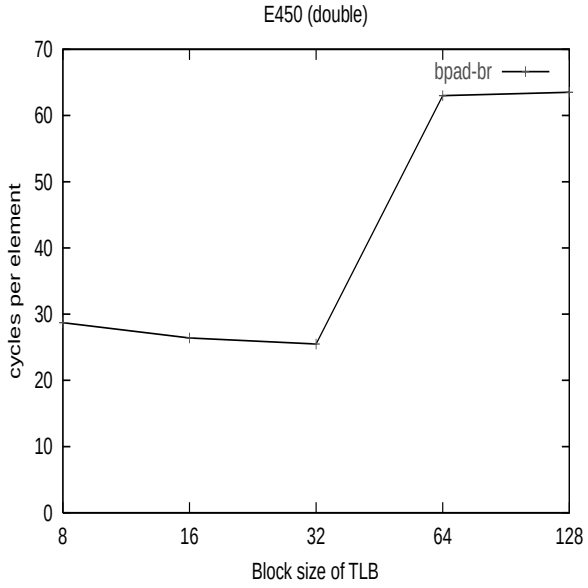

FIG. 4. *Changing the TLB blocking sizes on a single node of the Sun E-450: when the blocking size for TLB was larger than 32, the execution time curve was sharply increased.*

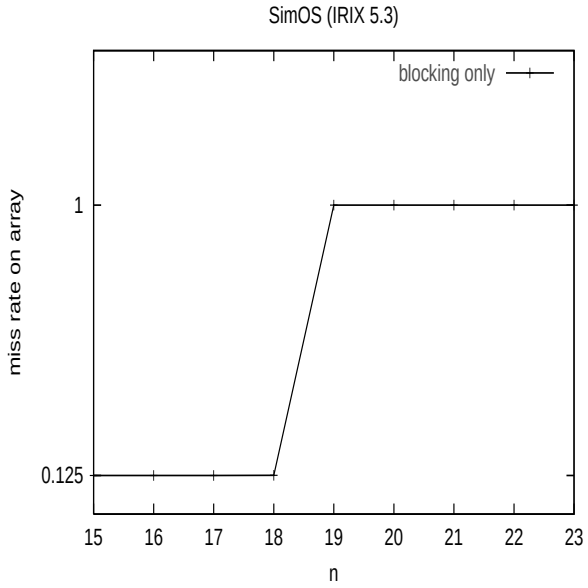

FIG. 5. *Using the SimOS to observe the miss rates by changing the size of the bit-reversal arrays of a blocking-only program: when* $n > 18$*, the miss rate was sharply increased to* 100%.

the IRIX 5.3 operating system are quite consistent with our assumption of "contiguous allocations."

We have also run the similar experiments on different targeted workstations with different operating systems, such as Linux and Solaris, to measure the changes of execution times when the data size is changed. Our measurements are also consistent

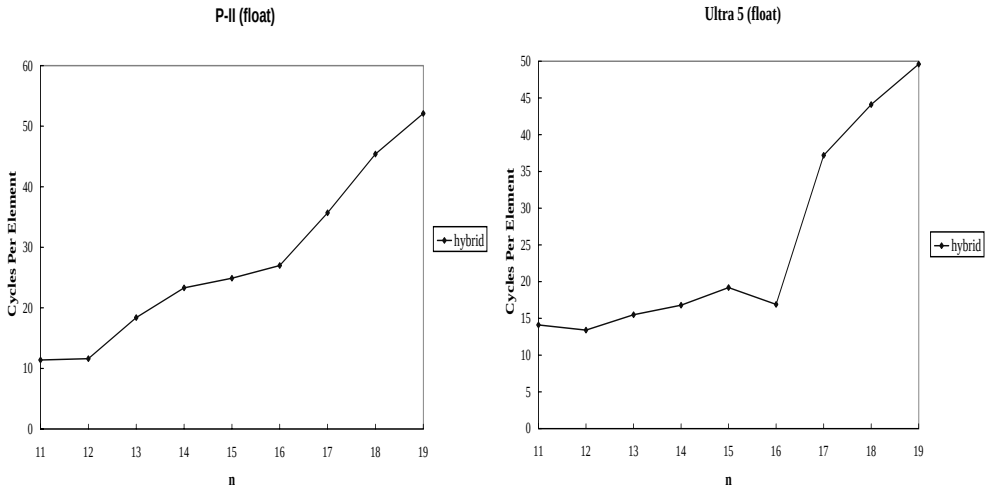

FIG. 6. *Execution times of the hybrid method on the Pentium-*II *(left figure) and on the Ultra-*5 *machine (right figure).*

to the SimOS results and indicate that the larger the data arrays to be used, the more likely an operating system will allocate the pages contiguously. Because our study targets large data sets, our analyses based on the virtual memory space is reasonably accurate. In addition, our methods assume that the operating system uses a uniform page size for page allocation, which is consistent with most commercial and commonly used operating systems.

6.3. Performance of the hybrid method for bit-reversals. In order to show the effectiveness of our cache optimizations, we first plot the measured execution times of the hybrid method[1] in "float" data types on the Pentium-II and the Ultra-5 machines in Figure 6. Although the hybrid method did reasonably well for $n \leq 16$ on Pentium-II and $n \leq 12$ on Ultra-5, the execution times significantly increased due to limited cache performance after the data size was further increased.

6.4. Performance comparisons on the SGI O2. The SGI O2 is a 1995 product using an R10000 processor of 150 MHz, 32 KB 2-way associative L1 cache, and 64 KB 2-way associative L2 cache. The cache line of L2 is 64 bytes. Since the associativity of L2 is low, and the cache line of L2 is relatively long, it is difficult to do blocking with associativity and available registers. We implemented only the blocking with padding method to compare with blocking with software buffer and the base reference.

We scaled bit-reversal methods from $n = 16$ to $n = 21$. Figure 7 shows the comparisons of CPE among the three programs of both "float" type and "double" type on the SGI O2 machine. The measurements show that the padding method slightly reduced the execution time compared with the method of blocking with software buffer. The time reduction was up to 6%. The reason for the small performance improvement comes from the extremely long memory latency (208 cycles) of the O2 machine. The reduction and saving of instruction cycles for data copies from padding became less significant because memory latencies caused by the required cold misses in both methods were dominant in execution.

[1]The program was written in Fortran by Alan Karp.

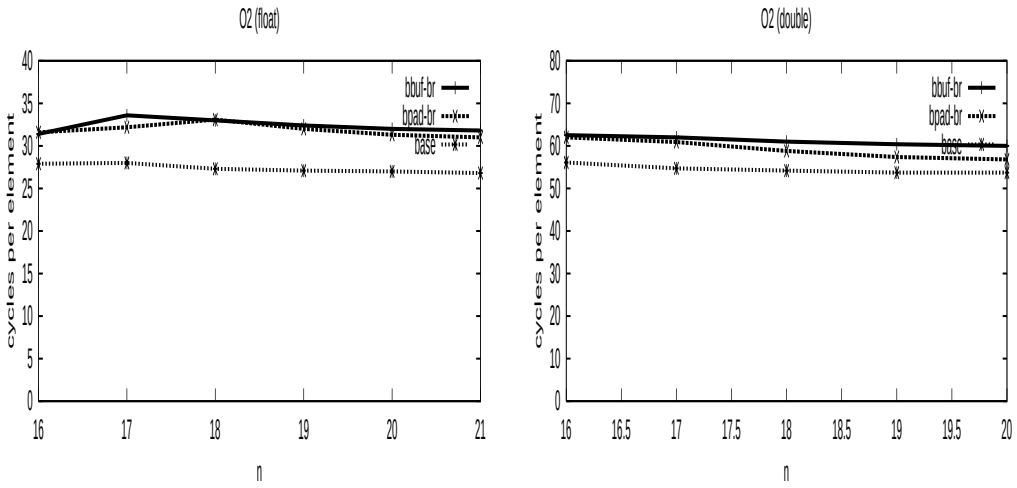

FIG. 7. *Execution comparisons on the SGI O2 workstation: "bbuf-br" represents the method of blocking with software buffer; "bpad-br" represents the method of blocking with padding; and "base" represents the ideal base line reference.*

6.5. Performance comparisons on the Sun Ultra-5. The Sun Ultra-5 is a 1998 product using an UltraSparc-IIi processor of 275 MHz, 16 KB direct-mapped L1 cache, and 256 KB 2-way associative L2 cache. The cache line of L1 is 32 bytes consisting of two 16-byte subblocks, and L2 is 64 bytes long. Similar to the SGI O2, the associativity of L2 on the Ultra-5 is low, and the cache line of L2 is relatively long, so it is difficult to do blocking with associativity and available registers. We implemented only the blocking with padding method to compare with blocking with software buffer and the base reference.

We scaled the bit-reversal methods from $n = 16$ to $n = 23$. Figure 8 shows the comparisons of cycles per element among the three programs of both "float" type and "double" type on the Ultra-5. The memory latency of the Ultra-5 (76 cycles) is significantly lower than that of the O2. We observed a more significant performance improvement from the method of blocking with padding over that of blocking with software buffer. For example, using "float" type, the padding program is 14% faster than that of blocking with buffer for $n = 20$ or larger. A L2 cache line of the Ultra-5 holds 16 "float" type elements ($L = 16$), and 8 "double" type elements ($L = 8$). The larger the L, the higher overhead the blocking with software buffer will have. This has been confirmed by our comparative experiments between the "float" and "double" types on the Ultra-5 shown in Figure 8.

6.6. Performance comparisons on the Sun E-450. The Sun E-450 is a 1998 4-processor SMP product. Each of the 4 nodes is an UltraSparc-2 processor of 300 MHz, 16 KB direct-mapped L1 cache, and 2 MB 2-way associative L2 cache. The cache line of L1 is 32 bytes consisting of two 16-byte subblocks, and L2 is 64 bytes long. Due to the limited associativity and a relatively long L2 cache line, we implemented only the blocking with padding method to compare with blocking with software buffer and the base reference.

We scaled the bit-reversal methods from $n = 16$ to $n = 25$. Figure 9 shows the comparisons of CPE among blocking with software buffer, blocking with padding, and the base program on a single node of E-450, each of which has both "float" type and

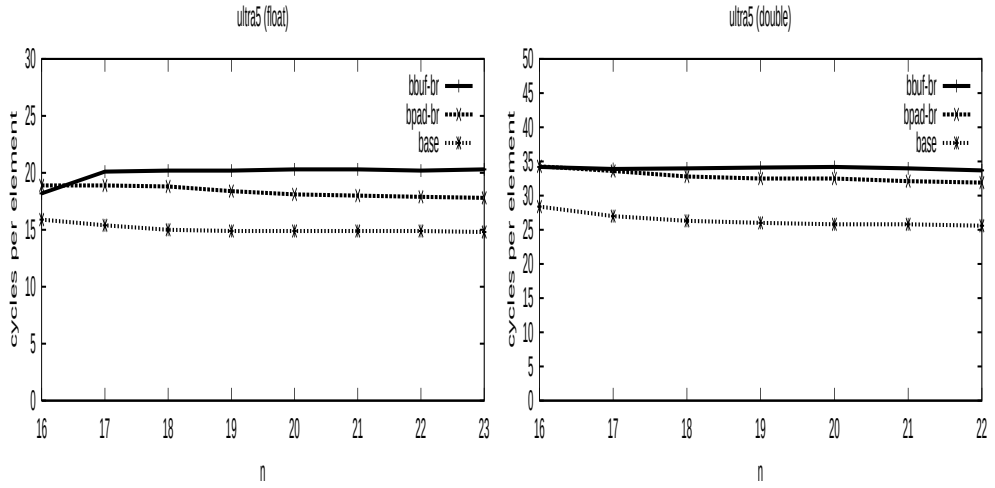

FIG. 8. *Execution comparisons on the Sun Ultra-5 workstation: "bbuf-br" represents the method of blocking with software buffer; "bpad-br" represents the method of blocking with padding; and "base" represents the ideal base line reference.*

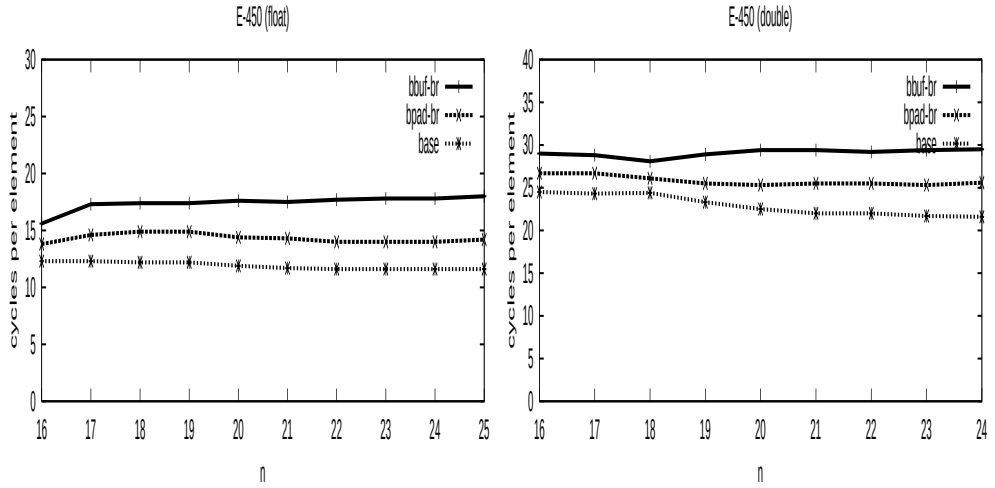

FIG. 9. *Execution comparisons on the Sun E-450 SMP: "bbuf-br" represents the method of blocking with software buffer; "bpad-br" represents the method of blocking with padding; and "base" represents the ideal base line reference.*

"double" type. The memory latency of the Ultra-5 (73 cycles) is slightly lower than that of Ultra-5. On this machine, we observed higher performance improvement from the method of blocking with padding over that of blocking with software buffer. For example, using "float" type, the padding program is 22% faster than that of blocking with buffer for $n = 20$ or larger. Our comparative experiments between the "float" and "double" types on E-450 in Figure 9 also confirms that the larger the L, the higher performance the padding method would achieve.

6.7. Performance comparisons on the Pentium-II 400. The Pentium PC we used is a 1998 product using a Pentium-II 400 processor of 400 MHz, 8 KB direct-mapped L1 cache, and 256 KB 4-way associative L2 cache. The cache lines of both

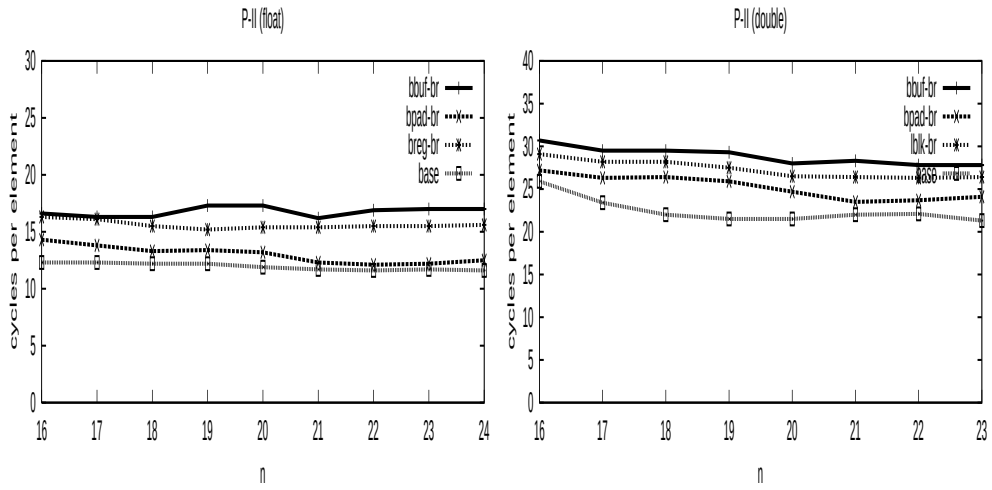

FIG. 10. *Execution comparisons on the Pentium*-II 4000 *PC: "bbuf-br" represents the method of blocking with software buffer; "bpad-br" represents the method of blocking with padding; "breg-br" represents the method of blocking with associativity and registers; and "base" represents the ideal base line reference.*

L1 and L2 are 32 bytes. Since the L2 associativity is high, we are able to implement the method of blocking with associativity and available registers, L2 cache line $L = 8$ elements for a "float" type, and we need $(L-K)(L-K) = 16$ registers to supplement the 4-way associative cache. An L2 cache line holds 4 "double" type elements ($L = 4$). Thus, we do not need any registers to supplement but simply make a 4×4 blocking. The TLB of the Pentium processor is a 4-way associative cache of 64 entries. We used our padding for the TLB technique to avoid TLB misses. We implemented the blocking with padding method and the blocking with associativity and registers to compare with blocking with software buffer and the base reference.

We scaled the bit-reversal methods from $n = 16$ to $n = 24$. Figure 10 shows the comparisons of cycles per element among the four programs. As we expected, the paddings for both cache and TLB were highly effective, and the padding program performed the best. For example, using "float" type, the padding program is about 40% faster than that of blocking with buffer for $n = 22$ or larger. We also show that the method using available registers to supplement associativity is effective. Although it is not as good as the padding program due to the increase of the instruction counts for additional data copies, it still achieved up to 12% execution reduction over the blocking with software buffer program. As we expected, the execution time of the method using the 4-way associative L2 cache without the supplement of registers to form a 4×4 blocking was delayed mainly by the longer L2 cache hit time. The performance of this method still outperformed the method of blocking with a software buffer.

6.8. Performance comparisons on the Compaq XP-1000. The Compaq XP-1000 is a 1999 product using an Alpha 21264 processor of 500 MHz, 64 KB 2-way associative L1 cache, and 4 MB 2-way associative L2 cache. The cache lines of both L1 and L2 are 64 bytes long. Similar to the SGI and Sun machines, the associativity of L2 on the XP 1000 is low, and the cache line of L2 is relatively long, so it is difficult to do blocking with associativity and available registers. We implemented only the

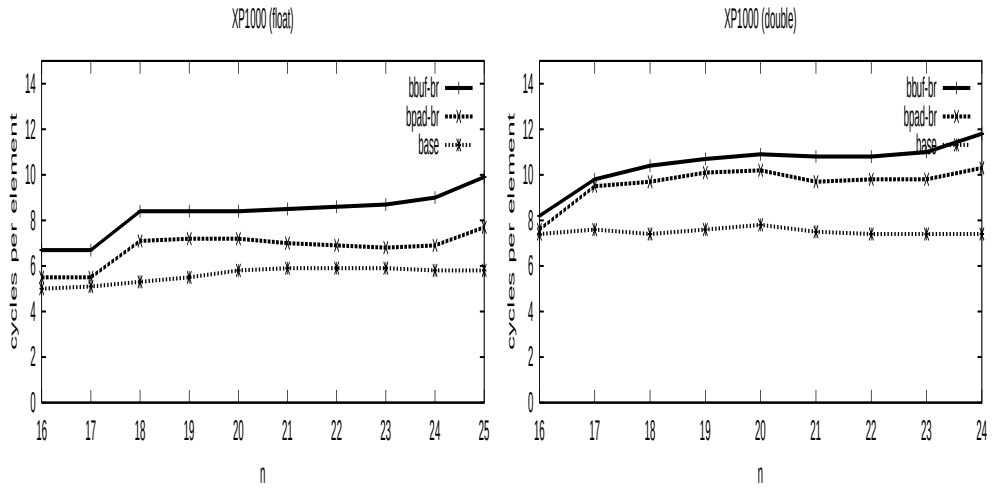

FIG. 11. *Execution comparisons on the Compaq XP-*1000 *workstation: "bbuf-br" represents the method of blocking with software buffer; "bpad-br" represents the method of blocking with padding; and "base" represents the ideal base line reference.*

blocking with padding method to compare with blocking with software buffer and the base reference.

We scaled the bit-reversal methods from $n = 16$ to $n = 25$. Figure 11 shows the comparisons of CPE among the three programs of both "float" type and "double" type on the XP-1000 machine. As we expected, we achieved better or comparable performance to the ones on the Sun machines. For example, using "float" type, for $n = 24$ or larger, the padding program is 30% faster than that of blocking with buffer, and 15% faster for "double" type.

7. Performance evaluation on SMP multiprocessors. We implemented the bit-reversal methods on two SMP multiprocessors: the Sun E-450 and the HP 9000 V2200. The parallel bit-reversal program on an SMP with M processors is described using POSIX thread primitives [10] as follows:

```
bit_reversal(id)
  my_start = id*(N/M);
  my_end = (id-1)*(N/M);
  for i = 1, N
    Y[i'] = X[i];
```

The bit-reversal operations are evenly distributed among M processors.

7.1. Performance comparisons on the Sun E-450. The Sun E-450 is a 1998 4-processor SMP product. Each of the 4 nodes is an UltraSparc-2 processor of 300 MHz, 16 KB direct-mapped L1 cache, and 2 MB 2-way associative L2 cache. The cache line of L1 is 32 bytes consisting of two 16-byte subblocks, and L2 cache line is 64 bytes. Due to the limited associativity and a relatively long L2 cache line, we implemented only the blocking with padding algorithm to compare with blocking with software buffer and the base reference.

We scaled the bit-reversal algorithms from $n = 16$ to $n = 24$. Figure 12 shows the comparisons of CPE among blocking with software buffer, blocking with padding, and the base program on the E-450 of 4 nodes, each of which has both "float" type and "double" type. On this machine, we observed some performance improvement

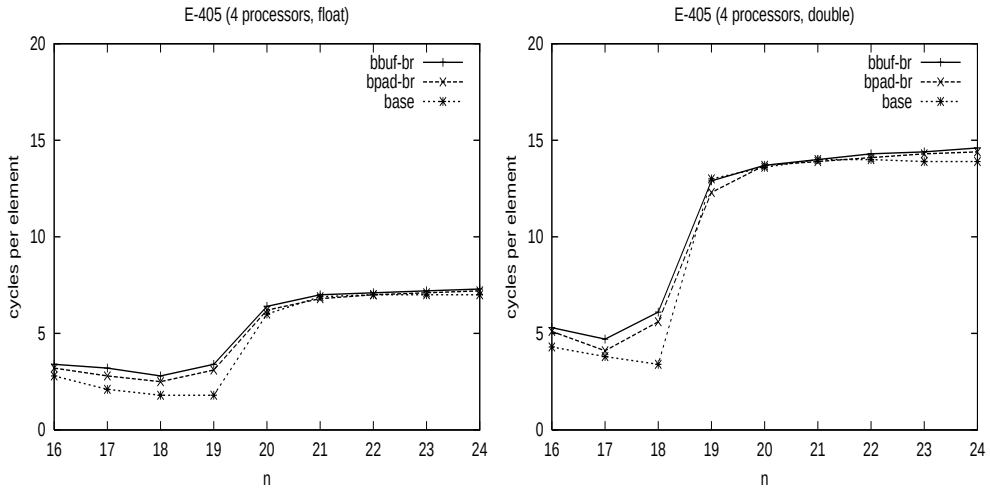

FIG. 12. *Execution comparisons on Sun E-450 SMP of 4 processors: "bbuf-br" represents the algorithm of blocking with software buffer; "bpad-br" represents the algorithm of blocking with padding; and "base" represents the ideal base line reference.*

when $n \leq 18$ from the algorithm of blocking with padding over that of blocking with software buffer.

However, when $n > 18$ of double type or $n > 19$ of float type, each processor has to process a data set larger than its cache capacity. Multiple processors simultaneously access the memory through a shared data link would cause the contention to degrade the performance. Since the data to be accessed from different processors are distributed in different locations, a crossbar interconnection network to link each processor to all the memory modules would significantly reduce the contention. The E-450 does have a 5×5 crossbar to connect 2 pairs of processors, 2 I/O ports, and the memory. The communications between the 4 processors the memory modules are connected through the single memory data link. Figure 13 shows the crossbar interconnections of the E-450 among the processors, the shared-memory modules, and the 2 I/O ports. The contention occurs in the memory data link when the multiple processors request memory accesses simultaneously.

We have observed severe performance degradation caused by the memory access contention. Figure 12 shows that this contention makes the execution time curves of the three programs jump sharply and merge together when $n > 18$ of double type and $n > 19$ of float type. In contrast, on a single processor of E-450, accesses to the memory through the memory bus have no contention so that the algorithms were scaled well.

7.2. Performance comparisons on the HP 9000 V2200. HP 9000 V2200 is a 1997 SMP product with up to 16 processors. We used 4 processors for performance comparisons. Each node is a HP PA-8200 processor of 200 MHz with a 2 MB direct-mapped L1 data cache. The cache line is 32 bytes. Due to limited associativity, we implemented only the blocking with padding algorithm to compare with blocking with software buffer and the base reference.

The HP SMP has a crossbar interconnection network, the HyperPlane crossbar, to connect up to 8 pairs of processors to 8 memory modules. Multiple pairs of processors can access different memory modules simultaneously. Each pair of the processors is

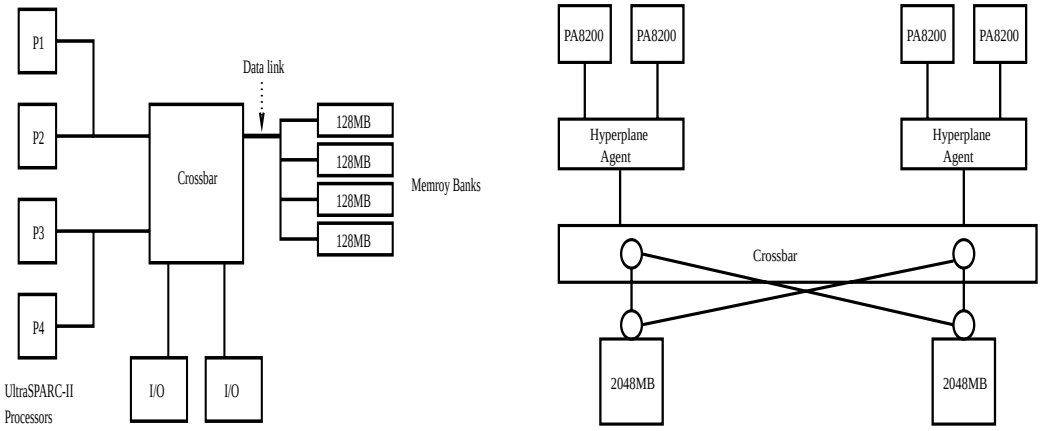

FIG. 13. *Architecture comparisons between Sun E-*450 *SMP (left) and HP* 9000 *V*2200 *SMP (right): the memory data link of the E-*450 *may become a bottleneck when simultaneous memory access requests from multiple processors; the HyperPlane crossbar connected between the memory modules and the processors on the HP* 9000 *V*2200 *can effectively reduce the contention.*

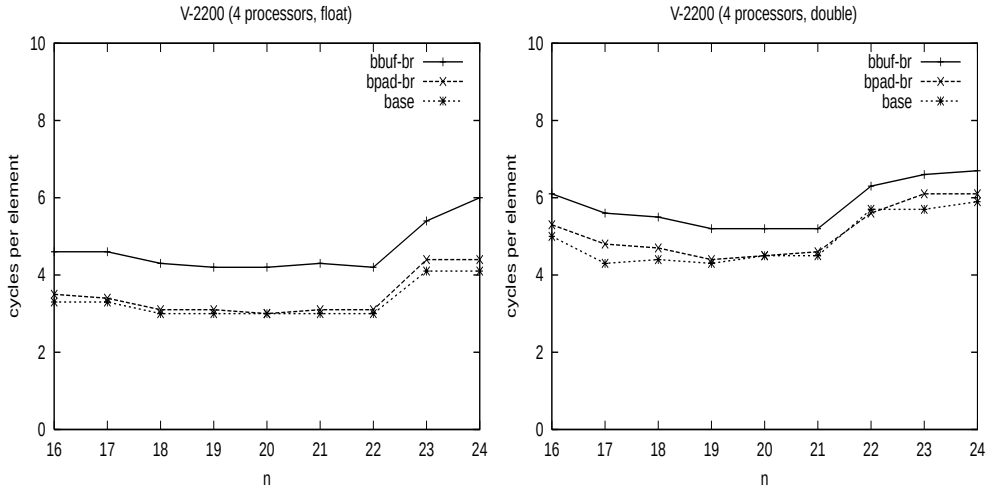

FIG. 14. *Execution comparisons on HP* 9000 *V*2200*: "bbuf-br" represents the algorithm of blocking with software buffer; "bpad-br" represents the algorithm of blocking with padding; and "base" represents the ideal base line reference.*

connected to the crossbar through an adaptor called HyperPlane Runway Agent. Figure 13 gives the interconnection structure of the HP 9000 V2200 of 4 processors.

In our experiments, the 4 processors are divided into 2 pairs which are connected to 2 memory modules by a 2×2 hyperplane crossbar. Each pair of processors may have contention to compete the adaptor, but the crossbar is able to allow simultaneous data accesses among the memory modules. The negative performance effect due to the data link contention observed on Sun E-450 was significantly reduced on the HP SMP, which shows the effectiveness of the crossbar. Figure 14 shows comparative execution time curves between the "float" and "double" types on E-450 in Figure 14. The execution times of the 3 programs are quite stable and independent of the size of n. Both the padding programs of the float type and of the double type outperformed

TABLE 2

Summary of the blocking methods and their impact on the three aspects of performance (cross interference, instruction count, and memory space) and on the program complexity. The performance of "blocking only" method is the base line for comparisons. Note: + means that the method quantitatively increases the factor and hurts the performance, and blank means it has no impact. The program complicity is subjective and compared with the "block only" method, with 1 *being a slightly more complex, and* 2 *a moderately more complex.*

Methods	Cross interference	Instruction count	Memory space	Program complexity	Comments
Blocking only				0	limited by data sizes.
Blocking with software buffer	+	+	+	1	system independent.
Blocking with register buffer				1	limited by the number of available registers.
Blocking with associativity and registers				2	works well on high associativity caches.
Blocking with padding			+	1	works well on all systems.
TLB blocking				0	a TLB size dependent outer loop, effective for fully associative TLBs.
TLB padding			+	1	paddings by using L pages, effective for set associative TLBs.

the blocking methods with buffer up to 40% and 18%, respectively. Their execution curves almost merge together with the base reference curve.

8. Conclusion. We have examined and developed cache-optimal methods for bit-reversal data reorderings. These methods have been tested on 5 representative uniprocessor workstations of 1995 to 1999 products to show their effectiveness. Different methods have their own merits and limits. The blocking-only method is limited by data sizes. Although the blocking-with-software-buffer method is architecture independent, it increases cross interference and instruction count and needs additional memory space. The blocking-with-a-register-buffer method is fast but is limited by the number of available registers. Blocking with associativity and with registers work well on high associativity caches. We have shown that the methods of blocking with padding, blocking for TLB, and padding for TLB can effectively exploit cache locality and are almost independent on hardware. Thus, they could be widely used on many uniprocessors workstations and SMP multiprocessors. We summarize different techniques and their merits and limits in Table 2, which gives a guideline for application users to choose a technique based on the size of the problem and the machines available.

The methods have also tested on two commercial SMP multiprocessors. By exploiting cache locality of each processor, we have effectively eliminated the conflict misses so that accesses to the shared memory and contention are minimized. However, another potential bottleneck on SMPs is the data access contention to the shared-memory. We show that crossbar interconnections between processors and memory modules play an important role to parallel bit-reversal data reorderings.

Appendix.

```
/* This is a padded bit-reversal program for cache optimization. */
void bit_reversal()
{
   int blk, blk_rev, i, i_rev, j, jump = PAD_LENGTH, k;
   int D = N >> 2*b, d = n - 2*b;
   DATA_TYPE *Xp[B];
   DATA_TYPE *Yp, f0, f1, f2, f3;

   for (i = 0; i < B; i ++)
      Xp[i] = &X[bitrev_tbl[i]*jump];

   for (blk = 0; blk < D; blk ++) {
      bitrev(blk, blk_rev, d);
      for (i = 0; i < B; i ++) {
         i_rev = bitrev_tbl[i];
         k = (blk << b) + i;
         Yp = &Y[(blk_rev<<b) + (i_rev<<(n-b))];
         for (j = 0; j < B; j += 4) {
            f0 = Xp[j][k];
            f1 = Xp[j+1][k];
            f2 = Xp[j+2][k];
            f3 = Xp[j+3][k];
            Yp[j] = f0;
            Yp[j+1] = f1;
            Yp[j+2] = f2;
            Yp[j+3] = f3;
         }
      }
   }
}
```

Acknowledgments. We feel fortunate to have had Alan Karp's expert views and comments on this work. We have also exchanged bit-reversal programs of different methods to compare the performance. The comments from the anonymous referees were helpful. We thank Kang Su Gatlin for his constructive suggestions on a preliminary version of this paper. Neal Wagner carefully read the manuscript and made useful comments.

REFERENCES

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp, *Compiler transformations for high performance computing*, ACM Computing Surveys, 26 (1994), pp. 345–420.

[2] D. H. Bailey, *FFTs in external or hierarchical memory*, J. Supercomputing, 4 (1990), pp. 23–35.

[3] B. Bershad, D. Lee, T. Romer, and B. Chen, *Avoiding conflict misses dynamically in large direct-mapped caches*, in Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, 1994, pp. 158–170.

[4] M. Cekleov and M. Dubois, *Virtual-address caches*, IEEE Micro, 17 (1997), pp. 64–71.

[5] J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comp., 19 (1965), pp. 297–301.

[6] A. Edelman, *Optimal matrix transpose and bit reversal on hypercube: All-to-all personalized communication*, J. Parallel Distrib. Comput., 11 (1991), pp. 328–331.
[7] D. M. W. Evans, *An improved digit-reversal permutation algorithm for the fast Fourier and Hartley transforms*, IEEE Trans. Acoust. Speech Signal Process., 35 (1987), pp. 1120–1125.
[8] K. S. Gatlin and L. Carter, *Memory hierarchy considerations for fast transpose and bit-reversals*, in Proceedings of Fifth International Symposium on High-Performance Computer Architecture, 1999, pp. 33–44.
[9] S. L. Johnsson and C.-T. Ho, *Algorithms for matrix transposition on Boolean N-cube configured ensemble architectures*, SIAM J. Matrix Anal. Appl., 9 (1988), pp. 419–454.
[10] IEEE, *POSIX P*1003.4*a: Threads Extension for Portable Operating Systems*, IEEE Press, Piscataway, NJ, 1994.
[11] A. H. Karp, *Bit reversal on uniprocessors*, SIAM Rev., 38 (1996), pp. 1–26.
[12] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan-Kaufmann, San Francisco, 1996.
[13] N. P. Jouppi, *Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers*, in Proceedings of 17th Annual International Symposium on Computer Architecture, 1990, pp. 364–373.
[14] L. McVoy and C. Staelin, *lmbench: Portable tools for performance analysis*, in Proceedings of the 1996 USENIX Technical Conference, 1996, pp. 279–295.
[15] P. N. Swarztrauber, *FFT algorithms for vector computers*, Parallel Comput., 1 (1984), pp. 45–63.
[16] C. Rivera and C.-W. Tseng, *Data transformations for eliminating conflict misses*, in Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation, 1998, pp. 38–49.
[17] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, *Using the SimOS machine simulator to study complex computer systems*, ACM Transactions on Modeling and Computer Simulation, 7 (1997), pp. 78–103.
[18] L. Xiao, X. Zhang, and S. A. Kubricht, *Improving memory performance of sorting algorithms*, 5 (2000), pp. 1–23.
[19] Y. Yan, X. Zhang, and Z. Zhang, *Cacheminer: A runtime approach to exploit cache locality on SMP*, IEEE Trans. Parallel Distrib. Systems, 11 (2000), pp. 357–374.
[20] C. Zhang, X. Zhang, and Y. Yan, *Two fast and high-associativity cache schemes*, IEEE Micro, 17 (1997), pp. 40–49.