Bit Reversal on Uniprocessors

Alan H. Karp
HPL-93-89
October 1993

Bit reversal, hierarchical memory, Fast Fourier Transform

Many versions of the Fast Fourier Transform require a reordering of either the input or the output data that corresponds to reversing the order of the bits in the array index. There has been a surprisingly large number of papers on this subject in the recent literature.

This paper collects 30 methods for bit reversing an array. Each method was recoded into a uniform style in Fortran and its performance measured on several different machines, each with a different memory system. This paper includes a description of how the memories of the machines operate to motivate two new algorithms that perform substantially better than the others.

Index terms: Fast Fourier Transform, bit reversal, cache memory

1 Introduction

There is a wide variety of Fast Fourier Transform (FFT) algorithms. Some of them require extra working storage; some can be done in place. Some algorithms use vectors of varying lengths; some use long vectors throughout. Some algorithms take the input in natural order and produce output in natural order; some either require scrambled input or produce scrambled output[32]. Since they all do the exact same number of arithmetic operations, memory access is the deciding factor in choosing one over another.

There are conflicting goals when deciding which algorithms to use. On a machine with a limited memory, an in-place algorithm is best. On a vector machine, algorithms with long vectors perform best. Unfortunately, many of these algorithms require a so-called *bit reversal* reordering of the data. If the bit reversal is not done properly, it can take a substantial fraction of the total time to do the FFT. In fact, it is common wisdom that bit reversal reordering is too slow to be used on a machine with a hierarchical memory[31].

Figure 1 illustrates the problem. It shows the time to do the bit reversal of an array of the indicated length in machine cycles per element. Two methods are shown, a simple scatter operation[20] and one of the first published methods[7]. Also shown is the time it takes to do one FFT butterfly using an algorithm tuned to the IBM 3090 vector processor used. It is clear that the bit reversal step will take a substantial fraction of the total time for large arrays. One conclusion drawn from these measurements is that the performance will be poor if the algorithm is not designed with the memory structure in mind.

People have looked at measurements like this and concluded that autosort methods are best. The idea, which is a good one, is to do the data movement required by the bit reversal while storing the results of each butterfly. Any bit reversal algorithm that requires $log_2 N$ passes over the data can be combined with the $log_2 N$ butterfly steps to make an autosort method. However, the problem is not so much with the *amount* of data movement as it is with the `pattern` of data movement.

An autosort method is effectively limited to using a multipass bit reversal which limits the data access patterns available for the FFT. Such methods don't perform optimally on vector processors because the vector length changes on each butterfly. Bailey[2] solves this problem by switching between Stockham variant I and Stockham variant II when the vectors get too short. The downside is that data movement equivalent to a matrix transpose is needed when switching algorithms. This transpose step is at least as expensive as the best bit reversal described in this paper.

The main point of this paper is to show people that bit reversals are not to be feared. There are quite a few bit reversal algorithms that require only a single pass over the data. In many cases, a single pass bit reversal is quite fast leaving more freedom in selecting an FFT algorithm. My own measurements, as yet unpublished, on a Cray Y-MP and an IBM 3090

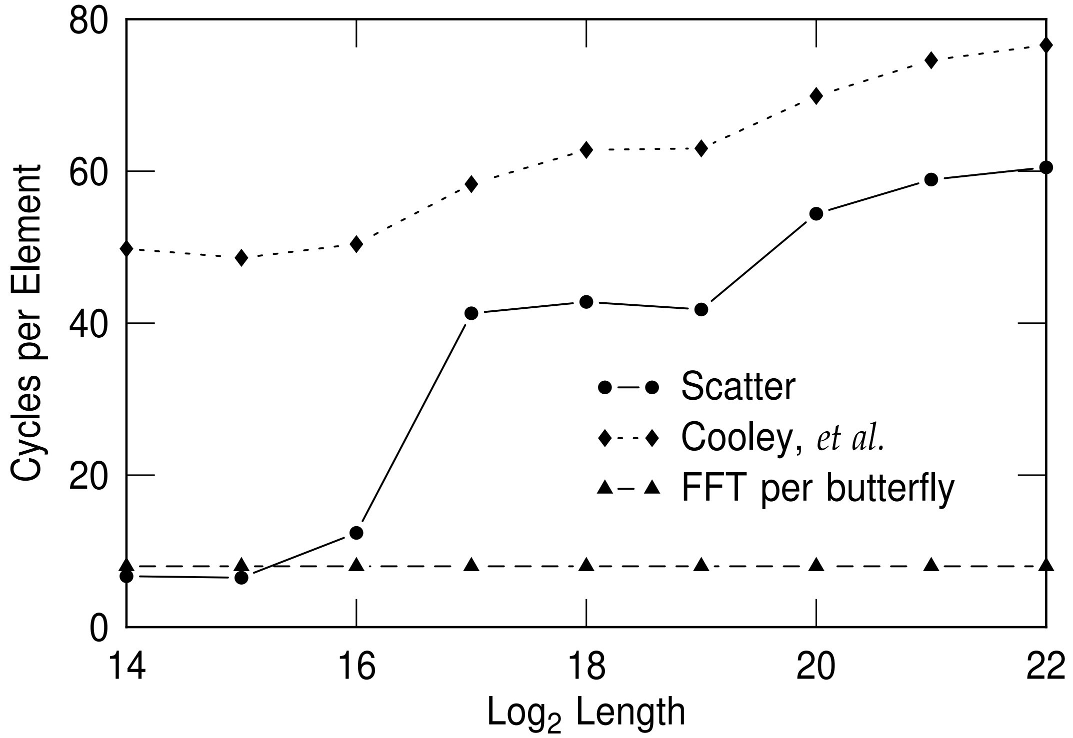

Figure 1: The time to do a bit reversal in units of machine cycles per element versus the size of the array using two different approaches on an IBM 3090 vector processor. The first jump in time occurs when the array becomes too large to fit into the cache; the second jump, when the data exceeds the TLB size (see text). Also shown is the number of cycles per element for one butterfly of a high-performance FFT algorithm.

vector processor show that the Pease algorithm with a separate bit reversal step is every bit as fast as the library routines with their less favorable data access patterns.

Another reason for this paper is the recent interest in bit reversals. There have been nearly 20 publications in the last several years on this subject. This study is the first comprehensive review of the literature in the last 15-20 years. (Rösel's[25] survey is more limited in scope than this one.)

While there is a good deal of interest in bit reversals on parallel processors[9, 14, 16, 17], this paper considers only those written for uniprocessors. I have collected 30 bit reversal algorithms from the literature and from a request made over na-net. Each has been coded into a uniform style of Fortran, tuned, and run on a number of machines – one node of an Intel iPSC/860, an IBM RS/6000 workstation, an HP-9000 workstation, and in vector and scalar modes on both a Cray Y-MP and an IBM 3090J/VF.

Section 2 gives an summary of some ways of viewing the bit reversal. In order to help explain why some of the methods perform poorly, Section 3 gives a brief description of the operation

Table 1: Index Vector by Recursion

0	0							
1	0	1						
2	0	2	1	3				
3	0	4	2	6	1	5	3	7

of interleaved memory and hierarchical memory systems. Section 4 describes the machines used. Next, Section 5 describes the algorithms. A new algorithm for interleaved memory machines is given in Section 5.3 and a new algorithm which is designed for hierarchical memories is presented in Section 5.6. The performance of all these algorithms measured on all the machines is given next in Section 6. The paper finishes with an interesting historical note, a plot of the speed of the bit reversal versus year of publication.

2 Properties of Bit Reversal

A bit reversal reordering is a simple operation.[1] For an array of length $N = 2^n$, exchange two elements $x(k)$ and $x(\tilde{k})$ for

$$k = \sum_{j=0}^{n-1} a_j 2^j \qquad \text{and} \qquad \tilde{k}_n = \sum_{j=0}^{n-1} a_j 2^{n-1-j}, \tag{1}$$

where the a_j are either 0 or 1. From this equation we see that we need to know the value of the index and how many bits are involved. For example, a 4-bit reversal of $1 = 0001$ is $1000 = 8$ while a 5-bit reversal is $10000 = 16$.

We can view the bit reversal as a permutation[20]. Let $\mathbf{x}$ be the vector of the $x(k)$, and $\tilde{\mathbf{x}}$ be the vector of the $x(\tilde{k})$. Then, we can write $\tilde{\mathbf{x}} = P\mathbf{x}$ where $P = P^{-1}$. In other words, the bit reversal of a bit reversed array is the array in natural order.

In practice, the permutation is written as an index vector with components $\tilde{k}$. The index vector can be computed quite efficiently by Horner's rule,

$$\tilde{k}_n = a_{n-1} + 2(a_{n-2} + 2(\cdots + 2a_0)\cdots).$$

Our index vector computation produces $\tilde{k}_n$ for all $0 \leq k_j < N$. We can save a lot of arithmetic if we use the fact that the high order bits of the binary form of k_j are 0 when k_j is small. For example, we know that the last $n-2$ bits of $\tilde{k}_j$ are zero for $k_j < 4$. This approach leads to the recursive algorithm illustrated in Table 1[36]. At each step we multiply the current list by 2 and concatenate 1 plus the list.

[1]While the methods described here can be modified to work for mixed radix problems, we limit our discussion to radix 2 problems only.

The primary reason we are interested in bit reversals is that many FFT algorithms leave the result bit reversed or require input in bit reversed order. This means that we can use the data access patterns of these FFT algorithms to affect the bit reversal. For example[20],

$$\tilde{\mathbf{x}} = \prod_{k=0}^{n-2} Q(2^k, N)\mathbf{x}. \tag{2}$$

Here $Q(m, N)$ is a generalized shuffle that performs a perfect shuffle taking m elements at a time.

The bit reversal reordering is also closely related to matrix transposition applied repeatedly on matrices of different shape[33]. One scheme is to reshape $\mathbf{x}$ into an $N/2 \times 2$ matrix, transpose it, and copy the elements in row major order. This new vector is reshaped into an $N/4 \times 4$ matrix. This procedure is repeated $n-1$ times and is exactly identical to the generalized perfect shuffle just described.

These permutation and multiple pass methods touch all the data. In fact, only some of the elements of $x(k)$ need to be moved depending on the value of k. Since we are exchanging elements, we need only consider $k < \tilde{k}$. The next thing to notice is that we need not look at all the indices. For example, it is obvious that no exchange is necessary for $k = 0$ and $k = N - 1$ since the former is all 0 bits and the latter all 1 bits.

Rodriguez[24] showed that the actual upper bound is the number whose bits are all 1 except for a 0 near the middle. If n is odd, this number has 2 more 1 bits to the right of the 0 than to the left; if n is even, there is one extra 1 bit to the right of the zero. The actual value for the index of the last exchange is then $N - 1 - m$, where $m = \sqrt{N}$ for n even and $m = \sqrt{2N}$ for n odd.

We don't even have to examine all the indices to see which ones should be swapped[35]. Consider for the moment n even. We can write $k = k_1\sqrt{N} + k_2$ so that $\tilde{k} = \tilde{k_1} + \tilde{k_2}\sqrt{N}$. If we think of a matrix with $\sqrt{N}$ rows and columns and entries consisting of pairs $(k_1, \tilde{k_2})$, it is clear we need only exchange elements corresponding to entries below the diagonal. Those on the diagonal are their own bit reversal, and those above are the same as those below. If n is odd, we generate separate matrices for the middle bit 0 and for the middle bit 1.

Another approach is to view the bit reversal process as one of finding the set of pairs k and $\tilde{k}$ that require an exchange. Rutkowska[27] describes a way to find this set recursively. Divide the set of all indices $0 \leq k < N$ into 4 sets. Elements in set A_n have n bits and both a leading and trailing 0 in their binary representation; B_n, a leading 0 and a trailing 1; C_n, a leading 1 and a trailing 0; D_n, both leading and trailing 1. Clearly, if $k \in A_n, \tilde{k} \in A_n$; if $k \in D_n, \tilde{k} \in D_n$; if $k \in B_n, \tilde{k} \in C_n$. These sets are simply related to the sets involving the $n-2$ middle bits of k. We build up to the desired set starting with (1,2) and (0,3) for n even and (0,1) for n odd.

3 Memory Architectures

As processor speed has increased, computer architects have been forced to use a number of tricks to keep the cost of their systems down. One thing they have done is to use memory chips that are considerably slower than those used in the processor. Since many instructions refer to data stored in memory, slow memories can only be used effectively if there is some way to improve their apparent performance.

One commonly used approach that is adopted by all the machines studied in this paper is to use a small amount of high-speed memory to hold the most frequently used data. These *registers* may hold as few as 4 words on some machines and as many as a few thousand on a vector processor. Since this small amount is not enough to hide enough of the memory delay, system designers use two other approaches – interleaved and hierarchical memories. Each of these are described in the following sections.

3.1 Interleaved Memory

If it takes several processor cycles to deliver one word from memory, then taking consecutive requests from independent memories can deliver the data fast enough. One way to implement this approach is to have an *interleaved* memory.

An interleaved memory is divided into *banks*, each holding part of the memory. Consecutive words are stored in consecutive banks in wrap mode. In other words, data is dealt to the banks as are cards in a bridge game. When a word in memory is accessed, the associated memory bank starts to deliver the word. The number of cycles needed until this bank can start to process another request is called the *busy time*.

The number of banks is usually a small multiple of the ratio of the processor to memory speeds. For example, the Cray X-MP/1 uses memory chips that are a quarter the speed of the processor chips and has 16 banks[15]. Other systems use more; the Fujitsu VP-200 uses 256 memory banks. Some machines use more complicated structures. The Cray 2 has a two level hierarchy, 4 sectors with 4 memory banks each.

A request to another bank that is not busy can start immediately. Thus, if there are enough memory banks, stepping through consecutive words in memory, stride 1 access, results in data being delivered to the processor fast enough to keep it busy. The performance from using other strides will depend on the number of banks and the relative speeds of the processor and memory. Fortunately, the relationship is easy to understand.

Let the busy time be m cycles, the number of memory banks be b, and the stride be s. If the stride is not commensurate with the number of memory banks, data will be delivered as fast as at stride 1, typically 1 cycle per element. If the stride is a multiple of b, then the average time to deliver a word to the processor is m. If the stride is such that $b/s < m$ is a small integer, the average time will be ms/b. This last case is easy to understand if we think of the memory as b/s independent memories each delivering a word every m machine cycles.

It is relatively easy to figure out which memory bank contains a specific address if the number of banks is a power of two, simply look at a few bits in the address. Hence, most machines use a power of two memory banks. However, power of two strides are quite common in practice, and such strides cause problems for these machines. Burroughs attempted to address this problem by producing a machine with 17 memory banks. Special algorithms were used to find which bank held a given word[38].

3.2 Hierarchical Memory

Hierarchical memories are complicated because they use many tricks to shield the user from delays caused by the slower circuits lower in the hierarchy[30]. While these techniques speed up the average memory access time, they make understanding the system more difficult.

At the top of the memory hierarchy are the registers. The time to move data between two registers sets the basic unit of time, usually a machine cycle. Since this is the case on all the machines considered here, the unit of time for all measurements will be machine cycles.

Next in the hierarchy is the cache memory, sometimes called high speed buffer memory. The cache is designed to deliver data to the registers as fast as it is requested, typically one word per cycle. Such high performance requires that an expensive technology be used so cache memories are typically small, 8 KBytes to 256 KBytes.

Below the cache is the main memory. Main memory is typically several cycles from the cache. This means that a memory reference to a word that is not in the cache will cause a delay. However, it is possible to pipeline data transfers from main memory to the cache which reduces the average access time per word. The amount of data transferred on each such *cache miss* is called a *cache line*. Cache lines typically contain between one and 32 words.

Below the main memory is the backing storage, either slow memory or disk. A reference to a word not in the main memory can take anywhere from hundreds to thousands of cycles to get the data into the cache. The average delay is reduced by moving relatively large blocks, typically 4 KBytes, at a time time. The data transferred on each such *page fault* is called a *page*. Main memory is divided into *frames*, each of which can hold a page of data.

The cache is the most important part of the hierarchy for this discussion since all the cases measured fit in main memory. System designers have a great deal of freedom in designing their cache system. They can choose the amount of data brought into the cache as a block, the cache line size. They can also choose the algorithm by which they replace a cache line when the cache gets filled up. In addition, they often speed up the processor by limiting the number of places in the cache that can hold a specific word.

Choosing the cache line size involves a trade-off between two, conflicting goals. We want a line as short as possible to avoid bring data into the cache that we will never use. On the other hand, we want a cache line to be as long as possible to amortize the cost of going to memory. The machines in this study have cache lines ranging from 32 to 128 bytes.

In an ideal world, we would be able to put a cache line into any available slot in the cache. Unfortunately, each memory reference would involve searching the entire cache for the cache line of interest. Such a search would take so long that memory references to cached data would take many machine cycles. The process can be sped up by limiting the number of places that can hold a specific address.

We can think of the cache as containing rows and columns of the places the data can sit. *Set associativity* is implemented by using a subset of the bits in the address to select the row, called the *class*. We can now find our entry on a memory reference by using these address bits to tell us which row to search. Typically machines are *direct mapped* in which there is only one slot that can hold a specific address (one column), or n-way set associative (n columns), where n is in the range of 2 to 8. There are also *fully associative* caches that allow any cache line to go into any available slot.

When we need to bring a new cache line into a full cache, we need to decide which cache line to replace. The best algorithm is one that replaces the line that will be used farthest in the future. Since we don't have full knowledge of future memory references, we need an approximation. If we have a direct mapped cache, there is no decision to be made; we put our new line into the only slot that can hold it. In a set associative or fully associative cache, we have a choice of algorithms. Most designers choose to replace the least recently used cache line, LRU replacement, an algorithm that needs a good deal of hardware to implement. Surprisingly, simply replacing a random line works quite well.

These choices impact performance. Consider the following program run on a machine with a 4-way, set associative, 32 KB cache with a 16 word line that uses LRU replacement.

```
dimension a(2**13,5)
do i = 1, 100
   do j = 1, 5
      a(i,j) = 0.0
   enddo
enddo
end
```

Each of the first 5 accesses should take a relatively long time since each results in a cache miss. However, we would expect the next 15×5 access to take only 1 cycle each since the

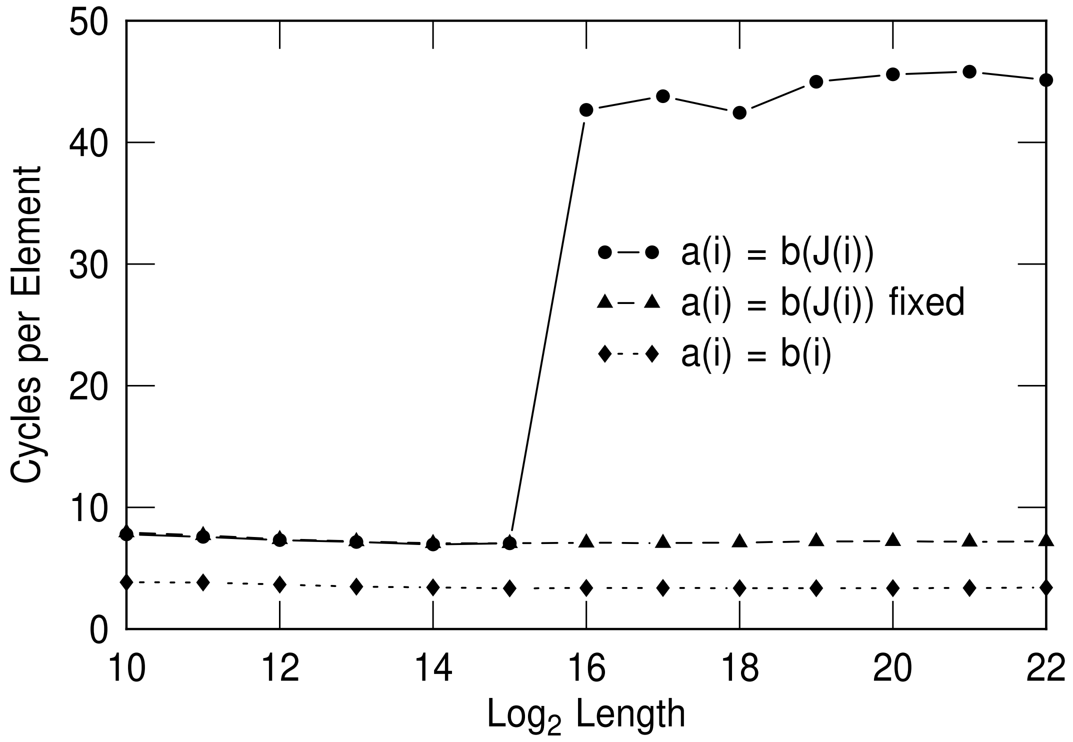

Figure 2: The time to copy an array at stride 1 using two methods. The dotted line is for `a(i) = b(i)` while the solid line is for `a(i) = b(J(i))` with `J(i) = i`. The improved version runs with the start of `b` offset 1024 words from the end of `a`.

data should be in cache. What we find is that every access is slow.

In our example, we are accessing the data with a stride equal to the size of the cache. Thus, each reference falls into the same row. The first 4 references find free slots in the row, but the 5th flushes one of them. Since we are replacing the least recently used line, the 5th reference flushes the cache line that will be used on the 6th reference. That one flushes the line needed for the 7th, and so on.

Other power of two strides will also be bad. A stride of half the cache size gives me only 2 rows; one quarter the cache size, 4 rows; *etc.* Surprisingly, other strides can also cause problems[12]. A 4-way set associative, 32 KByte cache with a 128 byte line size will perform poorly when the stride is 103!

We can fix our example by changing the stride by a small amount. If the array in our simple program were dimensioned `a(2**13+16,5)`, each reference would be to a different row in the cache, and our performance would improve markedly.

Another problem occurs for even larger strides. Figure 2 compares the performance of two ways of copying data at stride 1 on the IBM 3090. The dotted line uses `a(i) = b(i)`; the

Table 2: Parameters for Machines Studied

Machine	Cycle (ns)	Cache(KB)	Line(B)	Assoc.	Replac.	TLB (KB)
RS/6000-520	50.0	32	64	4	LRU	512
Intel i860	25.0	32	32	2	Random	128
HP-730	15.2	256	32	1	–	380
IBM 3090J	14.5	256	128	4	LRU	1024
Cray Y-MP	6.0	–	–	–	–	–

solid line, `a(i) = b(J(i))`, with `J(i) = i`. The intent is to measure how much more time a gather operation takes than a simple assignment. The time in cycles per element is constant until the array gets up to 2^{16} words at which point it increases by 36 cycles. The problem is not related to the cache which can hold only 2^{13} words.

To understand this behavior we have to look at the virtual memory. Although this problem fits into the 128 MByte memory on the machine used, the virtual memory system still must do *address translation*, a process that relates the virtual address of the page to the real address of the frame holding the data. The system keeps a table in memory to do this translation.

Doing address translation out of a table kept in main memory is slow. The process is speeded up by using a special hardware device, the translation look-aside buffer (TLB). The TLB on the 3090 is 2-way set associative with 128 entries, *i.e.*, the TLB can be pictured as having 64 rows and 2 columns. A reference to an address in the TLB is resolved in one machine cycle; a TLB miss costs 36 cycles.

Any stride larger than the 64 rows (256 KBytes or 64 KWords) will cause all TLB references to fall into the same class. Since the loop using `a(i) = b(i)` has only two memory references, and the TLB is 2-way set associative, the code runs at full speed. The other loop has three arrays falling in the same class; thus every memory reference results in a TLB miss and an increase of 36 cycles per element. As with the cache, this problem can be avoided by adding an increment to the size of the arrays.

There are many more subtleties of hierarchical memories. Some systems prefetch data; others move blocks of pages at a time; yet others have multiple level caches. Fortunately, none of these are important for the algorithms described here.

4 The Machines Measured

In this Section, I will describe the memory subsystems of the machines measured. Their characteristics are summarized in Table 2. Each has some unique features that make its performance different from the others.

4.1 Cray Y-MP

The Cray Y-MP is the only machine in this study that does not have a memory hierarchy beyond the registers. All addresses refer to the physical addresses in memory since there is no virtual memory support. The Y-MP can do 2 loads and a store every machine cycle giving it the highest memory bandwidth of any of the machines studied. The Cray can also chain operations together, such as load, multiply, add, store, to reduce the time to complete a calculation.

The Y-MP uses a 2-stage network to connect each of its 8 processors to each of the 256 memory banks. The first stage is an 8-way switch and the second, a 4-way switch. Hence, the memory looks like a 32-way interleave to each processor. Since the bank busy time is 5 cycles, strides of 2 and 4 suffer no penalty. The time to complete a load doubles with every doubling of the stride from 4 to 16. Stride 32 loads take 5 cycles each.

It takes about 17 cycles to get a single word from memory to the registers, either a scalar or a vector element. Subsequent words on a vector load or store are done at a rate of one cycle per element barring bank conflicts. Hence, stepping through an array takes $12 + 63 = 75$ cycles for each vector register, an average of 1.2 cycles per element. If the compiler can generate two independent loads, the rate doubles.

4.2 IBM 3090

The IBM 3090J is has a hierarchical memory with virtual memory support that uses a 2-way set associative TLB and a 4-way set associative cache. Unlike some other supercomputers, vector memory operations use the cache. The 3090 uses the longest cache line of any of the machines studied. The 3090 is the only machine studied that has arithmetic operations where one of the operands comes directly from the memory; all others require that the data be moved to the registers before it is used.

The memory system is built to deliver entire cache lines. When a cache miss occurs from either a read or a write, the memory delivers an entire cache line starting with the requested double word (64 bits). After a start-up of approximately 16 cycles, one double word is put into the cache every cycle; this word is available for use immediately. Since only one memory operation may be in progress at any time, the next load or store will have to wait for the entire 32 cycles it takes to transfer the cache line.

Once the data is in the cache, data is delivered to the registers in the cycle in which they are requested. Thus, stepping through an array of 32-bit numbers takes $16 + 31 = 47$ cycles for every 32 numbers in the cache line, just less than 1.5 cycles per element.

Since the 3090 has a 4-way cache, we never run into a problem of set associativity as we never reference more than 4 arrays in any of the loops. However, the TLB is only 2-way set associative. We have seen from Figure 2 that a gather copy can take over 45 cycles per element if the three arrays fall into the same associativity set. To avoid this problem, the measurements were made with arrays offset from each other by 1024 words.

4.3 IBM RS/6000-520

The memory subsystem of the RS/6000 model 520 is similar to that of the 3090. Its cache is smaller, 32 KBytes, and its cache lines are shorter, 64-bytes[3]. It also has a smaller TLB.

When a reference is made to a word not in cache, a line is transferred from main memory starting with the quadword (16 bytes) containing the requested byte. The first word is delivered to the register after a delay of 8 cycles; the rest of the quadwords arrive at one cycle intervals. This data can be moved from the cache to the registers while the rest of the cache line is being moved from memory. The memory subsystem is busy for 12 cycles on every cache miss. However, up to two memory requests to independent memory banks can be processed simultaneously.

Once the data is in the cache, data is delivered to the registers in one cycle. Thus, sequentially stepping through a single precision array takes $8+15 = 23$ cycles for every 16 words accessed, an average of 1.5 cycles per element. As with the 3090, we avoid the set associativity problem in the TLB by offsetting the arrays from each other by 1024 words. A TLB miss costs about 35 cycles.

4.4 HP/9000-730

The HP/9000 is the only machine studied with a direct mapped cache which means that there is only one place in the cache for each piece of data. To compensate, this machine has the largest cache of the machines studied, 256 KB, and a cycle time shorter than the other RISC processors. We can avoid problems of having a cache line from one array replace a line from another array by separating the arrays by the length of a cache line, 8 words, from each other.

When a reference is made to a word not in cache, a line is transferred starting with the word requested. Subsequent double words are delivered on successive cycles. It takes about 30 cycles to get the first word from the memory to the cache. Unlike the other machines studied, there is a one cycle delay between loading a word from the cache to a register and when that word can be used. Fortunately, this *delay slot* can usually be filled with other work. Hence, stepping through a single precision array takes $30 + 7 = 37$ cycles for every 8 words, almost 5 cycles per element.

The TLB is fully associative with LRU replacement and contains 96 entries, making this the only machine that has a cache or TLB not a power of two. A TLB miss takes about 30 cycles.

4.5 Intel i860

The Intel i860 processor is the basic compute node of the Touchstone Gamma machine. One node was used for the measurements reported here. Unlike the other machines studied, the i860 will not load a cache line on a write. If the word is in the cache, it is stored; if not, the data is sent directly to the memory. This means that stores never suffer a miss penalty.

When a load misses in the cache, the word referenced is delivered to the cache first and forwarded to the register. Each remaining double word (64 bits) is delivered in subsequent cycles[19]. It takes about 40 cycles to get the first word into the cache. Hence, loading a vector at stride 1 takes about 6 cycles per element. The i860 has a 2-way set associative cache that chooses which of the two lines to replace using a pseudo-random number generator. This replacement rule makes the i860 less susceptible to problems related to associativity, but can result in higher miss rates in some circumstances.

The TLB is 4-way set associative with 64 entries. Hence, our tests should show no effects of conflict in the TLB.

5 The Algorithms

Even though bit reversal is an inherently simple thing to do, there is a surprisingly large number of ways to do it. Even more surprising is how much work is still being done; over half of the algorithms reported here were published or developed in the last 5 years. Each of the sections that follows describes algorithms that use similar techniques to perform the reversal.

5.1 Examine the Index

These algorithms step through the array of indices and decide whether the corresponding data element should be exchanged with another. One disadvantage shared by all of these approaches is that they are inherently sequential and do not vectorize. Also, there is no locality in the data reference pattern so these approaches do poorly on machines with hierarchical memories. On the other hand, they are efficient in their data movement, touching each element to be moved once and not touching elements that remain in place.

The prototype of this class[7] works with 2 indices. One increases while the other is modified until it contains the bit reversal of the other. At this point, the elements are exchanged if they have not already been exchanged in an earlier iteration of the loop.

Buneman[6] made 3 improvements. First, he noted that the last 2 elements will never be exchanged with each other, so the loop could be shortened. Secondly, he replaced the divisions by 2 that Cooley used with an array of powers of 2. This latter change had no effect on the results reported here because I replaced the divisions with shifts. Finally, he moved the exchange of elements to the end of the loop which results in one fewer index computation. Buneman's code runs slower than does Cooley's because the way Buneman builds the bit reversed index takes one more step than Cooley's. Had I been clever enough to optimize Buneman's routine to use Cooley's version, its performance would have been identical to Cooley's.

Rodriguez[24] does Cooley's algorithm using the actual value of the index of the last exchanged element as the upper bound on the loop. He showed that the loop was made shorter by about $\sqrt{N}$. Unfortunately, the savings is small for arrays of the length considered in this study.

Rutkowska[26] improved on Rodriguez's idea by using some simple identities of an integer and its bit reversal to do 4 swaps of elements for each index computation. A similar modification of Cooley's algorithm results in similar performance improvement. Yong[37] used a similar trick to improve on Cooley's algorithm by exchanging pairs of elements. This code runs a bit faster than Rutkowska's because it does less integer work in the inner loop.

Although we normally think of arithmetic operations as if they took all the time, Duhamel[8] noted that the test in the inner loops of many bit reversals takes a substantial amount of time. He used the close relation between bit reversal and matrix transpose to completely eliminate the test of whether the two elements had been exchanged previously.

Brigham[5] took a conceptually simpler approach that, unfortunately, performs much worse. For each index in the loop, he explicitly computes the bit reversed value in $log_2 N$ steps. He then tests the values to see if the data elements should be exchanged. Not only does this approach use a lot of integer operations, it makes the computer time increase as $N log_2 N$ instead of as N with all the other algorithms in this section.

5.2 Build an Index Vector

Since we frequently do many FFTs of the same length, recomputing the bit reversal indices can be avoided. The methods reported in this Section compute the bit reversal indices once with the intent of using them many times. There are many ways to generate this list. I timed one that uses an optimized version of Horner's rule for evaluating polynomials[36] as described in Section 2. Once we have the index vector `J(i)`, we can do the bit reversal using a gather, `a(i) = b(J(i))`, or a scatter, `a(J(i)) = b(i)`.

Middleditch[21] builds an index vector 1/8 the length of the array. He determines which segment of the elements to work on from the short index vector, then does three bits worth of reversal explicitly. This approach saves storage, since a smaller index vector is needed, but it does not vectorize.

One of Van Loan's algorithms from an early draft of his book[33] computes the index vector inside the loop that exchanges the elements using the algorithm described in this section.

Odd/even pairs of elements are exchanged on each pass, and the inner loop does vectorize.

The problem with using these methods is that the memory references are not localized. Hence, most of the loads on a gather or stores on a scatter do not hit in the cache. On most machines, the performance of these two is the same. On the Intel i860, however, the scatter operation runs considerably faster because it does not bring in a cache line when a store operation misses.

5.3 New Gather/Scatter Method

A different problem occurs on the Cray. The bit reversal index vector contains all possible power of 2 strides which leads to lots of bank conflicts. All is not lost, though. We can modify the data access patterns by scrambling the index vector[1]. Of course, since the index vector is scrambled, we will have to use both a scatter and a gather to do the bit reversal. We will gain by avoiding memory bank conflicts, but we lose because gathers and scatters take almost twice as long as straight loads and stores.

Recall that our gather method uses `a(i) = b(J(i))`. The strategy used is to replace `J(i)` with `J(mod(k*i,n))` for an array of length `n`. We can use any value for `k` that will break up the bad reference patterns. The choice used for the runs reported in this paper is `k = n/4 - 1`. This choice guarantees that the largest power of two difference on any vector access is 4, a value that does not degrade performance on the Cray Y-MP.

Simply using `n/4-1` will cause problems on machines with 32-bit integers since `k*i` can overflow even for arrays of modest size. We are safe if we use the smaller of `n/4-1` and `2**31/n - 1`. One final nicety is to prevent the program from crashing if it is fed a very small or very large value of `n`. The code used to build the gather vector is

```
k = max(n/4 - 1,3)
k = min(k,2**31/n-1)
do i = 0, n - 1
   G(i) = mod(k*i,n)
enddo
```

Since we have scrambled the data on gather, we must reorder the data we store. Fortunately, this part is easy; we simply use the standard index vector for bit reversal for the scatter. The bit reversal is then done with both a gather and a scatter, `a(J(G(i))) = b(G(i))`. While this scheme runs a bit slower in scalar mode due to the extra memory traffic, it reduces the time in vector mode by about 1/3.

5.4 Make log_2N Passes

The bit reversal step is needed because many of the FFT algorithms bit reverse the data in log_2N passes. This observation means that we can also do just a bit reversal in log_2N passes as done by Singleton[28] whose algorithm was designed to minimize the amount of words transferred between memory and backing storage. His code uses the perfect shuffle on successively smaller segments on each pass. The loops are arranged to keep a block of data in main memory as long as possible. Not surprisingly, this approach works well on hierarchical memory machines.

Swarztrauber[31] presented two algorithms that are similar to Singleton's but are coded to look like matrix transposes. Routine `ctsort` is an inverse perfect shuffle; successive elements are stored in separate halves of the output array. Routine `ptsort` implements a perfect shuffle similar to Singleton's but the inner loop is coded in such a way that the data is always accessed at large stride. No special blocking is done in either implementation.

Van Loan[33] has implemented the same algorithms as Swarztrauber using explicit computation of the indices as opposed to transposing elements in an array with 3 indices. There is a saving in not needing a subroutine call, but on many machines explicit address computation is not as efficient as array indexing.

A bit reversal can be done by a sequence of shuffles as shown by Korn and Lambiotte[20]. First do a perfect shuffle of the two halves of the array. Next, do a perfect shuffle taking pairs of elements. Repeat this process, doubling the size of the groups shuffled for log_2N steps. Each step is implementable using the `merge` operation on the STAR 100 they used.

Polge's `usbin` algorithm[23] does multiple passes over the data but does not need any auxiliary storage. The price paid is that some elements may be moved more than once. At step k elements are moved when the bits of k and $log_2N - k$ differ. Since blocks of elements share these bits, the blocks can be exchanged as units making the algorithm vectorizable. While Polge describes higher radix versions of this algorithm, his published code is for radix 2 exchanges only.

Rutkowska[27] makes explicit the recursive nature of these multistep bit reversals. The recursions are based on looking at the elements with odd and even indices separately and splitting these two sets into a top and bottom half. The bit reversal of an element in one set puts it into another or the same set. For example, the bit reversal of an even index in the lower half of the array results in another element in the same set. Rutkowska presents two implementations that apply this scheme recursively and use only a few shift and add operations. The first recursively calls itself until it has identified a pair of elements to exchange, a version that is done in place. Since some machines do not execute recursive algorithms efficiently, she also presents an implementation that uses a work array but handles the recursion implicitly. This latter version vectorizes.

5.5 Use a Short Index Vector

It is possible to do a bit reversal with an index vector that is much shorter than we might think. The methods described here are based on the observation that the bit reversal of an integer $k = k_2 + k_1\sqrt{N}$ is $\tilde{k} = \tilde{k_2}\sqrt{N} + \tilde{k_1}$ when N is even. When k is odd, we do the same thing for the two halves with differing middle bits. This approach is used in Polge's `usone` algorithm[23].

Evans[10] independently discovered Polge's approach and produced a slightly simpler implementation. Evans later improved the algorithm very slightly by reducing the number of multiplications needed.[11].

Evans' papers generated a flurry of improvements. Walker[35] observed that the array indices and their bit reversals can be written as a matrix. Given the idempotency of the bit reversal operation, it is easy to show that one need only consider the sub-diagonal part of this matrix. The observation allowed Walker to reduce the size of the index vector by a factor of 2 when $log_2 N$ is odd and reduce the run time marginally. Biswas[4] presents an algorithm virtually identical to that of Walker, even including a matrix showing the symmetry of the transformation.

Vesely[34] studied the properties of mixed radix bit reversals and concluded that the best performance comes from using 3 index vectors of length about $\sqrt[3]{N}$. The fully symmetric version used here is similar to Evans' algorithm for odd n except that the middle section is wider than one bit and an additional reversal stage is needed to process these bits.

Khan[18] formulates the bit reversal in a manner similar to Rutkowska[27], splitting the indices into top and bottom half of the even and odd values. He breaks up the bit reversal into groups of elements that can be processed together, and an index vector of order $\sqrt{N}$ in size.

The algorithm Heller[13] uses was designed for the CM-2, so some of the details are a bit obscure. However, the structure is similar to that of Khan in the way groups of elements are handled. Heller starts with a base version that computes a bit reversal index vector of length 32. Then groups of 1024 elements are bit reversed in groups of 32. Heller tried a number of variants – explicit unrolling of the inner loop, reordering the statements to improve locality, using temporaries to keep data in the registers longer. The measurements show that the simplest code produced the best results on all machines.

5.6 New Hybrid Method

It is clear from Figure 1 and the results in Section 6 that the standard approaches to bit reversal reordering don't perform well for arrays larger than the cache. With only one exception[28], none of the methods previously published take explicit account of the structure of the various parts of the memory hierarchy. In this section, I describe a two step approach

```
                        In     Out of
                        Cache  Cache
Do k = 0 to N-1 by L
   Load J(k)            D+B    D+B
   Load b(k)            D+B    D+B
   Store a(J(k))        D+B+1  D+B+1
   Do j = 1 to L-1
      Load J(j+k)       1      1
      Load b(j+k)       1      1
      Store a(J(j+k))   2      D+B+1
   End Do
End Do
```

Table 3: Cycles for a bit reversal that fits entirely in the cache and one that does not.

that runs close to the in-cache rate. In the first step, the data is loaded with fixed stride and stored at stride one. The second step does a bit reversal that works on data in the cache.

In what follows I will use the following definitions. The cache can hold somewhat more than C data elements, and the cache line size is L data elements. I will assume it takes 1 machine cycle to load an element from the cache to a register. If the data is being gathered, one additional machine cycle is needed per element. Thus, it takes $2L$ cycles to gather L elements from the cache. If the data is not in the cache, I assume it takes B cycles for the data to reach the cache and D additional cycles before the memory can start the next request, a total of $D + B$ cycles. If a gather or scatter is used, each load of an element takes 1 additional cycle.

The number of cycles to complete the bit reversal is shown in Table 3. The first iteration is done separately to account for the difference in access time between the first reference to data on a cache line and subsequent references. The average time to do the bit reversal if all the data fits in the cache can be seen to be $4 + 3(B + D - 1)/L$ cycles per element. For the RS/6000-520 with $B = 8$, $D = 4$, and $L = 16$ this simple model predicts 6.1 cycles per element compared to the 7.2 cycles measured.

We should expect the model to underestimate the time because all it includes are memory accesses; other work that must be done is ignored. The limits of this simple model can also be seen by comparing the time for a gather and that for a scatter. According to the model, the performance should be the same. Instead, they differ because the CPU must wait for the data on a load but not on a store. Note that the Intel i860 does not bring data into the cache on a store. Hence, the scatter operation is considerably faster than the gather when the data does not fit into the cache.

Table 4: Radix-4 Digit Reversal by Recursive Transposes

1	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_a	d_b	d_c	d_d	d_e	d_f
2	d_0	d_1	d_2	d_3												
	d_4	d_5	d_6	d_7												
	d_8	d_9	d_a	d_b												
	d_c	d_d	d_e	d_f												
3	d_0	d_4	d_8	d_c												
	d_1	d_5	d_9	d_d												
	d_2	d_6	d_a	d_e												
	d_3	d_7	d_b	d_f												
4	d_0	d_4	d_8	d_c												
	d_2	d_6	d_a	d_e												
	d_1	d_5	d_9	d_d												
	d_3	d_7	d_b	d_f												
5	d_0	d_4	d_8	d_c	d_2	d_6	d_a	d_e	d_1	d_5	d_9	d_d	d_3	d_7	d_b	d_f

We can now see why a straightforward scatter operation does not perform well on a machine with a cache. Consider a large bit reversal of length $N \gg C$. The first few elements of `J` are $0, N/2, N/4, N/2 + N/4$. Thus, when storing `a(J(0))`, data elements `a(0)` through `a(L-1)` are brought into the cache; `a(J(1))` brings elements `a(N/2)` through `a(N/2+L-1)`, *etc.* Since each element of `a` is used only once, the average access time per element is $D + B$ cycles. The indirect addressing used costs an additional cycle per element.

The elements of `b` and `J` must also be brought into the cache before being stored, but they are accessed contiguously. The sum of these times gives the total for the bit reversal. The best performance from the simple gather operation will be about $D+B+3+2(B+D-1)/L$ cycles per element as shown in Table 3. For the RS/6000-520 the model predicts 16.4 cycles per element, compared to the 24 cycles measured.

In order to improve the cache performance of the bit reversal reordering, we must make better use of the data when it is brought into the cache. The approach used here is to do one or more large-radix bit reversals followed by a series of radix-2 bit reversals. For example, Table 4 shows how to do a radix-4 digit reversal of 16 elements.

Row 1 is the data in natural order; row 2, the data written as a 4×4 array; and row 3, the transpose of row 2. Row 4 represents a step needed when the radix is not prime; the indices of the rows must be bit reversed. (In the code, steps 3 and 4 are combined.) Finally, row 5 represents the radix-4 bit reversal reordering of the data. If $C = 4$, we can finish the radix-2 bit reversal by bit reversing each segment in turn.

The indices in row 5 can be looked at as 4 sequences, the elements of each sequence being

```
Do k = 0 to N-1 by C
   Load b(k)                D+B
   Store a(k)               D+B
   Do j = 1 to L-1
      Load b(L*j+k)         D+B
      Store a(j+k)          1
   End Do
   Do j = 1 to C-1 by L
      Load b(j+k)           1
      Store a(j+k)          D+B
      Do i = 1 to L-1
         Load b(L*i+j+k)    1
         Store a(i+j+k)     1
      End Do
   End Do
End Do
```

Table 5: Cycles for large radix bit reversal.

parts of the original array accessed at stride 4. The scheme proposed here uses the data patterns present in the indices of the large radix bit reversal. These segments are exactly the same as the groups described by Khan[18].

I will now describe the algorithm for a sequence of length $N = LC$. Since the cache line is L elements long, I will use a radix-L bit reversal.

First, load $Z = C/L$ data elements at stride L starting at element 0. (If my vector registers can hold Z elements, this sectioning loop is done automatically.) These elements get stored into another array at stride 1 starting at element 0. Since each number used results in L numbers being transferred into the cache, I will have loaded C elements into the cache.

Next, I load another Z elements from the cache to the register at stride L starting at element 1. This data gets stored in the other array at stride 1 starting at element $ZL/2$, where $L/2$ is obtained by reversing the bits in 1. The next set of Z elements go into the output array starting at element $ZL/4$, $L/4$ coming from the bit reversal of 2. This process is repeated in chunks of Z elements until the entire array has been processed, *i.e.*, a total of L times.

At the end, I will have produced a radix-L bit reversal reordering of the original data. Now, I can take each of the L segments of the output array and do a radix-2 bit reversal on the $N/L = C$ elements using any method that performs well when the data fits into the cache.

Even though I make two passes over the data instead of one for the conventional algorithm, the new approach is much faster. Cycle counting explains the improvement. The first load takes $D + B$ cycles per element since each number is on a different cache line. The next $C - 1$ loads take only 1 cycle per element since the data has already been put into the cache. Each store is done at stride 1 and, therefore, takes $1 + (B + D - 1)/L$ cycles per element. The average time for the radix-L part is only a little more than $2 + (B + D - 1)/L$ cycles per element as shown in Table 5. The predicted value of 2.7 cycles per element is a bit smaller than the measured value of 4 on the RS/6000-520.

The length C radix-2 bit reversals are also efficient since all the data fits in the cache. All measurements were made using the algorithm that performs best on data that fits in the cache. On the RS/6000, HP-9000, and Intel i860 I used `walker`[35]; on the 3090 scalar run, `unsone`[23]; on the 3090 vector run, `gather`.

If N/L is too large to fit in the cache, we must repeat the process as many times as necessary to allow an efficient reordering. Each fixed stride pass over the data will take an additional $T_f = 2 + (B + D - 1)/L$ cycles per element. However, since we are decreasing the length of the sequence by a constant factor, the time grows only as $\log_L N$. In fact, we will rarely have to worry about when to use another method. The total time needed by this hybrid method is

$$T_h = 4 + \frac{3(B + D - 1)}{L} + T_f \log_L \frac{N}{C}.$$

Equating this time to the time for a simple scatter and solving for the break-even point N_* gives

$$\log_L \frac{N_*}{C} = \frac{(D + B - 1)(1 - 1/L)}{2 + (B + D - 1)/L}.$$

For the RS/6000-520, $C = 4096$, $L = 16$, $D = 4$, and $B = 8$, $\log_2 N_* = 27$; for the HP-9000 with $C = 262144$, $L = 8$, $D = 8$, and $B = 30$, $\log_2 N_* = 27$. The situation is even better for the 3090; with $C = 16384$, $L = 32$, $D = 16$, and $B = 16$, $\log_2 N_* = 48$, an impractically large value.

The full algorithm is presented in Table 6. Here `N` is the length of the array; `C`, somewhat less than the number of elements that fit in the cache; `L`, the size of the cache line. In addition, we take `m = min(L,nr/C)` instead of simply `L` to ensure that we always do length `C` bit reverses in the next phase. `J(j)` is the m-bit bit reversal of `j`.

In routine `trans` the shapes of the arrays are changed on each entry. Thus, on the first pass we deal with one array of length `N`; on the second pass, `L` arrays of length `N/L`; *etc.* Note, too, that the order of the loops is important. One might think that because `i,k` appears on both sides of the assignment that these two loops could be coalesced. Doing so would spoil the optimal use of the cache. The additional dimension that does not appear in Swarztrauber's[31] code accounts for the performance difference.

The performance of the algorithm can be improved slightly by violating one of the constraints,

```
nr = N
Do While ( nr > C )
  m = min(L,nr/C)
  trans(a,b,m,C,nr,N)
  nr = nr/L
  !  Interchange meaning of a and b
End Do
Do i = 0 to N-1 by C
  !  Bit reverse b(i:i+C)
End Do
End
Routine trans(a,b,L,C,nr,N)
Dimension a(C/L,nr/C,L,N/nr)
Dimension b(L,C/L,nr/C,N/nr)
Do m = 1, N/nr
  Do k = 1, nr/C
    Do j = 1, L
      Do i = 1, C/L
        a(i,k,j,m) = b(J(j),i,k,m)
      End Do
    End Do
  End Do
End Do
End
```

Table 6: Two stage bit reversal reordering.

namely that we load the array with a stride equal to the cache line size. We can reduce the number of passes made over the data by increasing the stride.

Figure 3 shows that we can increase the stride much more than might be expected, up to 256 on a machine with 16 word cache lines, before the performance degrades dramatically. To understand this phenomenon, we must look at the load and store operations separately.

Consider using a stride of $S = pL$. The loads proceed as before; each load brings in a full cache line all of which gets used. However, we will be forced to store fewer than L consecutive words, L/p words instead of L.

The optimum stride can now be predicted. If a larger stride means we can do the fixed stride work in one pass instead of two, we save time if the large stride version takes less than half

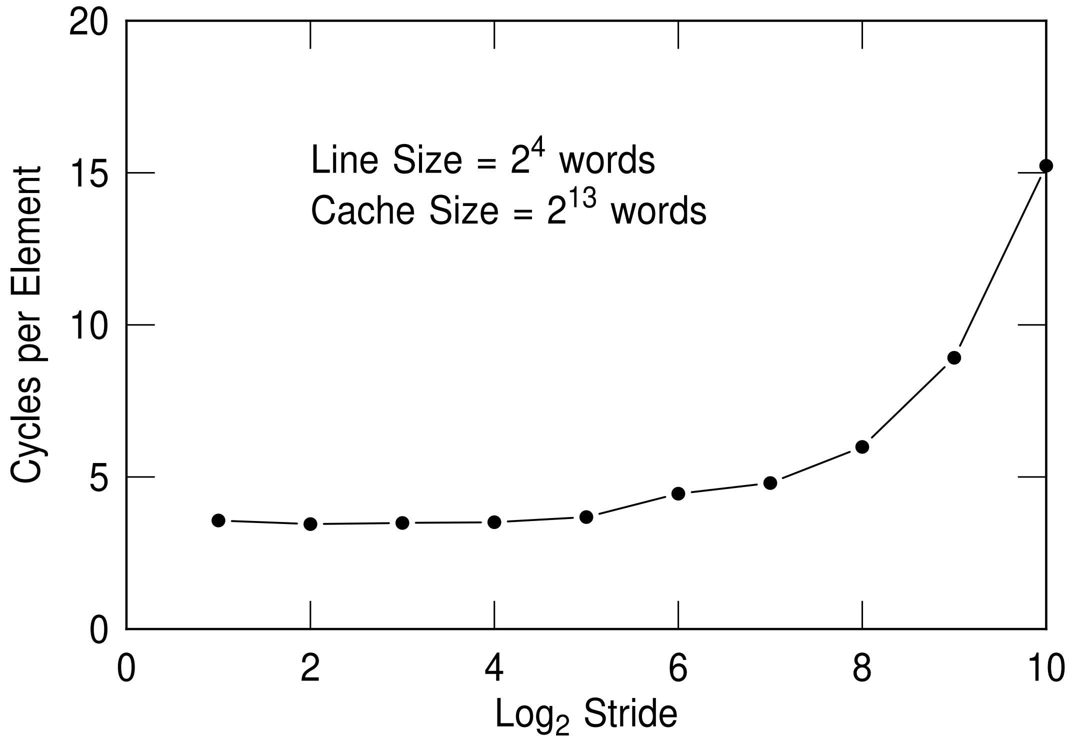

Figure 3: The time to copy an array at the indicated stride using the segmented copy of the new bit reversal method. Note that the time per element does not increase dramatically until the stride exceeds 256 words.

the time of the stride L version. From our discussion in Section 3.2 we reach this point on the RS/6000 when we store an eighth of a line, a stride of 256, just the value measured. A similar argument holds for the HP-9000 which has higher memory latency and a shorter cache line. On this machine, the cut-off stride is also 256. In this case, the fixed stride part is so slow that we can afford to have the gather step run partly outside the cache if it saves an extra pass over the data.

This approach can be extended to machines with more memory levels, either second level caches, electronic backing storage, or even to distributed memory parallel machines. All we need do is hierarchically nest the bit reversal with successively smaller values for C and L. For example, we might do a radix-1024 bit reversal to optimize the use of the second level cache. Next, we would call a bit reversal routine to work on segments that fit entirely in the second level cache. Since these segments would not fit in the first level cache, we could do a radix-32 bit reversal on them. Finally, we could use a gather on the subsegments that fit into the first level cache.

Table 7: Baseline Measurements for $\log_2 N = 20$ ($\log_2 N = 17$ on i860)

	Machine						
Case	3090J Scalar	3090J Vector	RS/6000 Model 520	HP-730	Intel i860	Y-MP Scalar	Y-MP Vector
Empty loop	5.0	5.0	2.0	2.0	2.0	11.0	0.3
Straight copy	9.1	3.2	3.4	14.0	6.1	24.2	1.4
Stride copy	10.2	3.2	3.4	38.3	27.0	23.4	2.4
Transpose copy	10.2	3.2	3.4	37.8	27.0	6.4	1.9
Gather copy	16.2	6.1	7.2	17.3	11.8	33.2	2.0
Scatter copy	17.6	5.4	6.2	17.1	12.0	26.3	1.8
Index vector	16.0	5.6	7.8	72.7	12.8	32.1	2.8

6 Measurements

The reports in the literature were coded in different languages and were run on a variety of machines over more than 20 years. In order to be fair, I coded each of them into more or less modern Fortran 77. Thus, any differences due to coding style and quality of compilers has been removed.

Secondly, I made a reasonable attempt to tune the codes. For example, using shift operations instead of multiplications and divisions by powers of 2 sped up some codes more than 3 times. Reordering loops to improve the locality of reference sped up others by an even larger factor. However, if a code was presented with certain computations in the inner loop, those were left there. I also did not unroll loops, a procedure which is known to improve performance on some machines. I did check the results for correctness for every case run.

Different approaches were taken to time the runs on different machines. Since I could not get the 3090 stand-alone, I measured CPU times which fluctuated somewhat due to the heavy load on the machine. (We shared the machine with chemists.) In addition, to get higher clock resolution, I used an experimental version of the timer. When the system was heavily loaded, this timer sometimes forgot to tick. To account for this problem, I took the largest CPU time of several runs.

On the RS/6000 Model 520 and HP-9000 Model 730 I ran when I was the only user. Unfortunately, Unix dæmons kept rearing their ugly heads so an occasional measurement was clearly spurious. These were the short runs; there is no way I can be sure that the longer runs were not affected. In all cases I report the shortest measured elapsed time of several runs.

The iPSC/860 returns the elapsed time on the single node I used. Even here, there were fluctuations from one run to the next, albeit small ones. Again, I report the shortest elapsed time of several runs. The times on the Cray were very repeatable.

6.1 Base Measurements

First, I made some measurements to find out the base performance of the machines. Table 7 summarizes the results.

The empty loop is just that, a loop with no executable statements. Only the Cray compiler completely eliminated this loop. Straight copy refers to `a(i) = b(i)`. Stride copy and transpose copy both move the data at fixed stride. Stride copy treats the arrays as one dimensional and computes the offsets; transpose copy, as two dimensional and lets the compiler do the address arithmetic.

Gather copy is `a(i) = b(J(i))`, and scatter copy is `a(J(i)) = b(i)`, both with `J(i) = i`. Once the TLB problem described in Section 3.2 was fixed, the time per element for these loops was independent of the number of elements.

Since many users of FFTs compute many transforms of the same length, I did not include the computation of the index vector in the timings presented in the next section. The index vector `J(i)` was computed using the method described in Section 2. This approach performs poorly on the HP-9000 because there are a lot of conflicts in the direct mapped cache. A different algorithm should be used on this machine.

6.2 Measurements

The measurements made are summarized in Tables 9–13. For each machine, I measured the largest bit reversal that would fit entirely in the cache up to the largest that would not cause excessive paging. On the Cray the largest run was limited by the 10 MWord allowed for interactive use.

The table entries are divided into groups separated by horizontal lines. The first group of algorithms are those that step through the index values and exchange the appropriate elements. None of these methods vectorize so this group is denoted by a double horizontal line in Table 10. Group 2 methods use an index vector of order N. Multiple passes over the data are used by the methods in Group 3, while Group 4 consists of methods that use a small seed table. The new method described in Section 5.6 is presented last.

Double vertical lines are used in the tables of machines with hierarchical memories to delineate the various levels in the memory hierarchy. The left section is for arrays that fit in cache; the middle section, those that fit in the translation look-aside buffer; the right section for larger arrays. In general, the methods do worse on larger arrays. Those that use less memory do better longer while those that need extra memory degrade sooner.

Some comments on the tables are in order. The Cray times are so stable and vary so little with problem size that only the largest four cases are shown. The code `perm1` uses recursive calls. Due to a bug in the RS/6000 Fortran compiler, this code did not run correctly. Also,

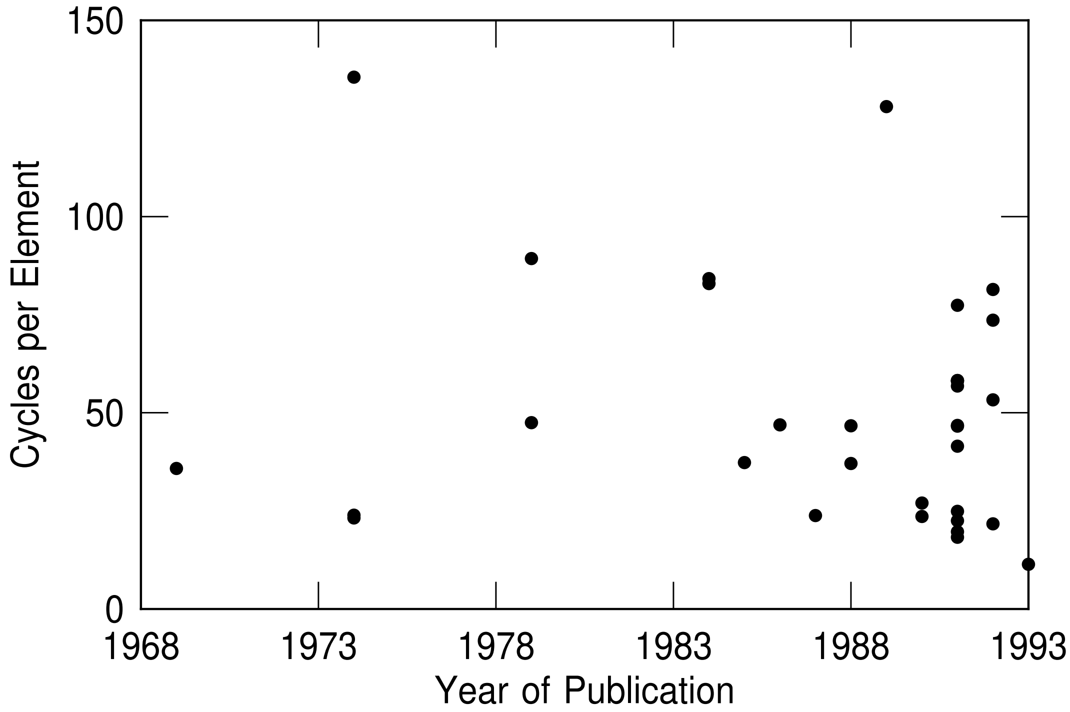

Figure 4: Time to do a bit reversal of an array of length 2^{20} words on an IBM RS/6000 Model 520 versus the year the algorithm was published.

even following the recommended procedures for recursive calls on the Intel i860, the code would not run. I did not use the C versions of this method on these machines because the performance is poor on the machines where it did run, and the differences between the C and Fortran compilers would have been an issue.

7 Conclusions

The idea that the memory structure is an important factor in determining the performance of the bit reversal algorithm is not new[28]. However, this fact seems to have been forgotten in the 25 years since its first publication; few of the methods described in this paper consider the memory structure of the machines. Some authors even publish performance numbers without actually moving the data[4, 18]! Further evidence of the problem is illustrated in Figure 4 which shows the time in cycles per element versus year of publication. I have chosen to show the performance on the RS/6000 for an array with $log_2 N = 20$.

I am being somewhat unfair since many of the algorithms were written for PCs and I am running them on a high performance, scientific workstation. On the other hand, the average number of citations to previous work in these papers is disappointingly small. Of course, bit reversals are not the most important thing in the world so the lack of rigor in the literature search may well be justified.

The two new methods presented show the importance of considering the memory structure. The improvement is particularly dramatic for the machines having long cache lines.

While this paper has concentrated on radix-2 bit reversals, most of the algorithms used have mixed-radix analogs, some of which have been reported in the literature[22, 29]. However, it is difficult to predict their performance because their memory access patterns are quite different from those of the radix-2 methods. Perhaps a follow-up study is needed.

Acknowledgements: I would like to thank Rad Olson for helping me understand the operation of the cache and TLB on the IBM machines. Eric Barszcz helped me understand the Intel i860. David Bailey taught me the `gatscat` algorithm. I also thank the people who sent me algorithms or allowed me to use their work in advance of publication – Alan Edelman, Steve Heller, Fuad Khan, and John Middleditch. Numerous others sent comments and pointers to the literature. Thanks to you all. I would also like to thank David Gustavson, Chuck Dickens, and Charles Boeheim of SLAC for helping me get machine time to verify the performance numbers for the 3090 in scalar mode.

Table 8: Bit Reversals on Cray Y-MP in cycles per element

	Scalar				Vector			
Version	17	18	19	20	17	18	19	20
cooley[7]	80.1	80.1	80.2	80.2	78.7	78.7	78.7	78.8
r-cooley[26]	30.6	30.7	30.6	30.8	30.9	30.9	30.8	30.9
yong[37]	28.1	28.1	28.1	28.2	28.2	28.2	28.1	28.2
buneman[6]	112.4	112.4	112.4	112.5	112.5	112.3	112.4	112.5
rodrig[24]	87.2	87.1	87.2	87.4	87.1	87.1	87.4	87.5
r-rodr[26]	37.4	37.4	37.4	37.5	37.3	37.3	37.4	37.5
brigham[5]	405.2	448.2	491.2	534.2	400.6	440.7	480.7	520.8
duhamel[8]	37.0	37.1	37.1	37.1	37.1	37.1	37.1	37.2
middled[21]	73.7	73.7	73.7	73.7	72.3	72.2	72.2	72.4
cvl143[33]	32.5	32.6	32.4	32.4	8.2	8.3	8.2	8.2
gather	29.3	29.4	29.3	29.3	6.6	6.7	6.7	6.7
scatter[20]	26.3	26.4	26.4	26.3	6.0	6.0	6.0	6.1
gatscat[1]	34.4	34.5	34.4	34.4	4.8	4.8	4.8	4.8
singl[28]	72.5	80.3	80.1	87.7	10.8	11.7	11.8	12.3
ctsort[31]	173.6	179.1	184.6	190.0	27.5	28.9	30.7	31.8
psort[31]	207.9	213.2	219.3	225.7	103.4	104.9	106.6	109.0
cvl154[33]	343.5	364.2	383.9	404.7	42.3	44.3	46.8	48.3
cvl153[33]	344.4	365.3	385.2	405.7	39.1	41.0	43.1	44.6
kornlam[20]	205.3	211.5	216.7	222.6	53.4	58.2	63.3	69.6
unsbin[23]	65.2	72.7	79.8	87.4	50.5	51.0	51.6	52.3
perm1[27]	124.5	123.9	124.3	124.4	124.7	123.8	124.3	124.2
perm2[27]	38.5	38.6	38.7	38.7	7.0	7.0	6.8	6.6
unsone[23]	17.8	17.9	17.7	17.7	7.3	7.3	6.6	6.4
evans[10]	17.7	17.8	17.7	17.7	8.2	8.2	8.1	8.1
walker[35]	16.3	17.9	16.2	17.7	7.9	8.1	8.1	8.2
biswas[4]	20.5	20.8	20.5	20.7	20.5	20.7	20.5	20.7
vesely[34]	56.8	55.8	55.4	55.6	15.4	13.0	13.0	13.1
khan[18]	15.0	16.9	15.1	16.8	13.1	8.1	13.0	8.1
heller[13]	28.0	27.7	27.7	27.9	7.2	7.1	7.0	7.3
hybrid	24.7	24.3	24.2	24.0	8.8	8.7	8.6	8.6

Table 9: Bit Reversals on IBM 3090 Scalar in cycles per element

Version	15	16	17	18	19	20	21	22
cooley[7]	48.9	50.6	58.4	61.9	62.8	69.9	74.6	76.6
r-cooley[26]	22.4	22.4	27.2	29.0	29.5	32.6	35.9	36.6
yong[37]	18.9	20.3	28.9	29.3	30.9	39.0	41.9	42.5
buneman[6]	60.7	61.2	71.1	75.2	75.4	82.1	86.4	88.7
rodrig[24]	47.1	47.6	57.2	61.5	61.9	68.7	74.1	75.7
r-rodr[26]	22.8	23.0	27.5	29.5	30.3	33.3	36.1	37.2
brigham[5]	238.0	254.5	274.1	290.4	304.0	323.8	342.1	357.0
duhamel[8]	24.5	24.6	38.9	38.6	39.3	53.9	54.3	55.9
middled[21]	30.4	31.1	41.2	43.3	44.4	53.5	58.5	62.0
cvl143[33]	29.8	40.5	55.0	54.7	55.5	78.5	83.8	86.2
gather	15.4	22.0	40.5	42.1	41.8	59.9	64.0	67.2
scatter[20]	16.3	24.2	44.8	46.6	46.6	67.6	71.5	74.2
gatscat[1]	36.3	86.1	106.0	112.0	161.8	199.6	211.8	196.6
singl[28]	24.6	31.3	34.4	38.0	38.1	41.3	41.3	44.2
ctsort[31]	124.4	136.4	140.0	144.4	146.9	151.2	154.6	155.0
psort[31]	130.7	153.3	161.2	173.0	180.8	189.3	197.6	204.0
cvl154[33]	115.5	137.4	145.6	155.7	164.1	171.9	180.2	188.7
cvl153[33]	114.8	136.9	146.0	155.4	163.9	171.5	179.8	187.1
kornlam[20]	119.0	137.8	147.0	155.0	166.0	176.3	185.6	192.8
unsbin[23]	31.9	36.7	40.6	44.0	44.0	47.2	48.1	50.2
perm1[27]	10.9	103.9	108.3	108.1	108.0	115.0	115.3	114.6
perm2[27]	21.5	24.0	29.0	37.3	43.9	51.1	57.0	60.7
unsone[23]	10.0	10.2	22.8	23.2	23.3	33.7	35.3	37.4
evans[10]	9.9	10.3	20.4	24.6	24.9	32.4	37.3	38.8
walker[35]	11.7	11.4	22.8	23.1	26.4	34.2	42.6	41.2
biswas[4]	12.3	14.1	26.0	28.4	28.5	26.8	42.9	42.6
vesely[34]	29.9	80.6	82.1	86.6	95.2	102.1	108.4	121.8
khan[18]	12.8	14.2	23.3	24.2	25.1	34.4	36.9	37.8
heller[13]	18.7	20.8	43.3	44.3	47.3	63.6	63.3	64.6
hybrid	10.0	10.4	20.6	22.3	22.2	22.0	22.5	22.4

Table 10: Bit Reversals on 3090 Vector in cycles per element

Version	15	16	17	18	19	20	21	22
cvl143[33]	30.2	39.4	54.8	55.8	56.5	78.9	84.3	86.8
gather	6.5	12.4	41.3	42.8	41.8	54.4	58.9	60.5
scatter[20]	12.4	22.7	46.7	47.8	46.5	69.4	74.1	75.5
gatscat[1]	39.2	94.7	122.2	128.5	168.1	210.0	213.7	202.1
singl[28]	12.7	14.1	18.8	20.3	20.0	21.8	21.7	22.3
ctsort[31]	42.8	113.2	115.5	122.9	124.5	132.2	136.0	136.1
psort[31]	98.4	124.8	132.9	143.6	151.1	159.3	166.7	172.6
cvl154[33]	53.1	69.7	73.0	77.2	80.2	84.4	87.7	88.7
cvl153[33]	59.7	77.2	80.4	85.5	88.3	92.8	96.7	97.9
kornlam[20]	101.7	113.5	122.9	132.3	143.6	153.7	162.9	170.7
unsbin[23]	48.0	52.4	54.7	56.2	56.3	58.1	58.9	59.2
perm2[27]	17.2	19.3	30.6	46.8	54.9	69.9	83.2	89.5
unsone[23]	9.9	9.6	42.3	44.5	43.6	59.4	61.4	62.7
evans[10]	7.5	10.0	22.2	42.2	42.8	56.3	60.5	60.6
walker[35]	9.8	10.3	36.2	42.1	53.1	59.9	63.6	65.7
vesely[34]	20.5	74.8	86.9	84.9	106.6	111.0	116.1	138.0
heller[13]	19.7	32.5	52.7	51.9	54.1	54.1	51.2	64.8
hybrid	6.8	9.6	10.5	10.5	10.4	10.5	11.0	10.7

Table 11: Bit Reversals on RS/600-520 in cycles per element

Version	12	13	14	15	16	17	18	19	20	21
cooley[7]	15.9	16.6	20.0	21.1	21.8	22.1	30.9	34.7	35.8	36.1
r-cooley[26]	6.9	6.9	9.4	10.7	11.0	11.1	14.6	17.5	18.3	18.6
yong[37]	6.4	6.3	10.8	11.3	11.6	11.8	20.1	22.4	22.5	22.6
buneman[6]	27.0	26.5	29.9	31.1	31.7	32.0	41.4	45.7	46.9	47.2
rodrig[24]	16.5	17.0	20.9	21.6	22.1	22.5	31.6	35.9	37.1	37.5
r-rodr[26]	8.0	8.1	10.4	12.0	12.4	12.5	16.0	18.9	19.7	20.0
brigham[5]	59.9	63.8	71.9	77.1	81.9	86.0	99.7	107.7	135.5	117.5
duhamel[8]	7.5	7.5	12.5	12.7	12.6	12.9	27.0	27.1	27.0	27.0
singl[28]	10.8	11.4	17.6	16.9	18.3	18.7	20.1	20.1	21.7	21.6
ctsort[31]	30.0	53.0	53.8	59.1	62.3	66.9	74.1	78.5	82.9	89.5
psort[31]	38.9	56.3	58.9	61.9	66.5	70.2	76.4	80.2	84.2	88.2
cvl154[33]	29.5	49.3	52.4	56.2	60.7	65.3	73.2	77.2	81.4	88.0
cvl153[33]	28.3	45.9	48.6	51.2	54.7	59.2	66.1	69.9	73.6	77.7
kornlam[20]	35.2	60.0	59.2	62.3	66.5	77.9	81.4	85.4	89.3	93.7
unsbin[23]	12.5	12.3	18.2	18.2	19.9	20.0	21.6	21.6	23.2	23.2
perm1[27]										
perm2[27]	10.6	12.6	17.8	23.3	27.2	28.7	32.2	38.1	41.5	45.3
middled[21]	16.8	16.9	20.3	21.7	22.9	22.7	32.1	36.1	37.3	37.7
cvl143[33]	12.1	16.7	22.4	22.2	23.6	25.1	52.6	52.9	53.3	53.9
gather	7.8	10.5	17.4	17.4	17.9	18.8	46.2	46.5	46.7	46.6
scatter[20]	7.2	12.3	24.2	23.7	23.8	24.9	47.2	47.5	47.5	47.5
gatscat[1]	21.7	36.4	36.9	36.6	39.8	93.3	128.1	127.8	128.0	129.6
unsone[23]	4.6	4.6	12.7	12.3	12.4	12.6	23.9	23.9	23.9	23.9
evans[10]	4.6	4.8	9.2	11.0	11.2	13.7	15.3	17.2	23.8	24.0
biswas[4]	5.5	4.5	10.5	10.4	10.6	10.7	24.7	19.5	24.9	23.7
walker[35]	4.4	4.2	8.8	11.1	11.1	11.4	15.3	17.1	23.6	24.0
vesely[34]	12.0	33.1	36.6	35.9	48.5	47.7	50.3	72.7	77.4	78.1
khan[18]	5.5	5.1	9.0	10.0	10.8	11.2	15.7	15.8	21.7	21.0
heller[13]	6.4	10.7	23.7	23.5	21.0	24.2	46.5	46.8	46.8	46.5
hybrid	4.4	4.2	8.1	8.3	8.3	8.8	9.4	10.0	11.4	12.6

Table 12: Bit Reversals on HP-730 in cycles per element

Version	12	13	14	15	16	17	18	19	20	21
cooley[7]	19.4	19.4	19.3	19.5	19.6	30.4	46.7	53.7	55.9	57.4
r-cooley[26]	11.2	11.2	11.3	11.3	11.3	33.7	40.9	45.2	47.8	47.4
yong[37]	10.5	10.6	10.6	10.6	10.7	28.8	40.9	44.1	45.5	48.3
buneman[6]	23.3	23.8	23.3	23.4	23.5	33.9	50.4	57.1	59.4	61.0
rodrig[24]	19.6	19.6	19.6	19.8	19.9	31.7	48.1	57.0	58.5	58.8
r-rodr[26]	11.5	11.5	11.5	11.5	11.8	33.8	41.9	45.9	48.2	48.6
brigham[5]	113.8	122.1	130.8	141.3	147.1	167.8	193.2	211.8	219.0	233.9
duhamel[8]	11.5	11.4	11.3	11.5	11.4	35.2	44.9	45.1	46.2	46.3
middled[21]	21.8	21.8	21.7	21.8	21.9	33.7	49.8	57.1	60.0	62.5
cvl143[33]	31.2	36.9	31.4	31.3	37.0	81.7	97.3	102.8	102.6	104.5
gather	17.4	17.4	17.4	18.2	20.1	65.2	80.3	85.2	85.8	87.6
scatter[20]	17.6	18.9	17.5	17.6	20.7	70.4	83.4	85.9	87.4	89.8
gatscat[1]	48.4	48.5	48.7	49.4	130.4	176.0	199.0	230.5	233.6	255.0
singl[28]	19.1	19.0	21.3	21.3	25.3	55.4	74.2	90.2	94.7	94.9
ctsort[31]	81.6	86.0	90.7	96.3	221.4	282.4	325.8	369.3	422.7	463.2
psort[31]	69.6	73.0	76.8	81.3	219.9	281.8	319.8	360.9	416.2	432.2
cvl154[33]	77.6	84.0	90.0	98.4	223.9	286.2	329.4	380.3	429.9	467.8
cvl153[33]	78.1	84.4	90.2	98.2	232.6	285.7	317.8	366.3	405.9	434.3
kornlam[20]	65.9	69.1	73.0	77.7	143.5	567.5	633.5	707.2	782.6	848.4
unsbin[23]	21.6	22.0	23.7	23.8	26.2	57.2	75.8	91.1	96.0	97.1
perm1[27]	20.5	21.0	20.5	20.3	20.3	40.7	41.0	41.7	42.3	41.9
perm2[27]	28.6	28.6	29.0	29.0	29.3	44.6	54.0	63.9	70.8	92.1
unsone[23]	8.7	8.6	8.5	8.6	8.4	31.2	42.5	42.7	43.6	45.2
evans[10]	9.6	9.5	9.5	9.6	9.5	17.8	31.5	35.6	47.1	48.1
walker[35]	8.7	8.5	8.6	8.6	8.6	18.7	32.6	34.3	45.4	47.3
biswas[4]	9.5	8.7	9.4	8.7	9.4	26.5	43.4	44.8	45.7	45.7
vesely[34]	22.1	21.6	22.0	21.3	133.2	139.8	140.0	141.2	149.3	151.8
khan[18]	8.7	8.4	8.5	8.4	8.6	22.4	30.9	32.7	41.3	41.7
heller[13]	14.3	14.2	14.2	14.3	14.8	44.1	44.0	43.9	44.3	44.5
hybrid	8.7	8.5	8.6	8.7	8.6	37.3	37.5	37.4	37.7	38.1

Table 13: Bit Reversals on Intel i860 in cycles per element

Version	8	9	10	11	12	13	14	15	16	17
cooley[7]	27.3	26.3	25.6	26.1	29.0	32.5	33.5	33.9	34.1	42.7
r-cooley[26]	14.9	13.4	12.5	13.3	16.2	18.8	19.3	19.4	19.2	23.7
yong[37]	14.1	12.5	12.1	12.9	16.8	18.8	19.9	19.9	20.0	28.1
buneman[6]	33.8	32.9	32.5	32.9	36.1	39.1	40.5	40.9	41.2	50.0
rodrig[24]	27.1	25.7	25.7	25.8	29.6	32.8	33.8	34.3	34.6	43.2
r-rodr[26]	14.9	13.6	13.1	13.8	17.0	19.7	20.1	20.3	20.4	24.7
brigham[5]	83.9	89.5	95.8	103.1	114.3	123.9	132.1	139.6	147.1	163.1
duhamel[8]	16.9	15.2	14.0	14.2	19.8	20.8	20.7	20.7	20.8	28.1
middled[21]	26.4	24.9	24.3	24.7	27.7	30.4	31.4	31.7	32.5	41.8
cvl143[33]	32.4	29.3	28.3	31.9	38.6	38.1	38.0	38.0	42.6	57.2
gather	18.6	17.3	17.3	19.9	28.5	29.7	29.6	29.7	31.4	45.2
scatter[20]	19.0	20.6	19.8	17.6	16.2	15.4	15.0	15.1	17.4	31.4
gatscat[1]	27.8	25.6	27.6	44.8	46.3	46.3	46.2	53.7	83.0	103.7
singl[28]	35.0	31.5	32.6	33.9	47.3	47.8	51.9	51.7	54.9	54.9
ctsort[31]	57.1	56.6	65.5	92.4	108.3	120.8	132.8	145.1	157.5	169.9
psort[31]	80.6	80.7	85.0	110.6	116.1	120.3	126.5	133.0	140.1	146.6
cvl154[33]	53.9	54.9	63.7	89.0	99.9	112.1	124.1	136.4	148.9	161.6
cvl153[33]	53.1	54.5	60.2	81.1	86.3	91.5	98.8	106.2	113.9	121.5
kornlam[20]	80.8	80.1	84.6	128.4	136.4	149.2	161.8	176.1	190.0	203.0
unsbin[23]	28.8	24.0	25.1	24.5	36.8	38.8	42.5	42.1	45.5	45.7
perm1[27]										
perm2[27]	25.6	22.1	21.6	23.7	30.6	36.6	38.9	40.0	41.2	45.0
unsone[23]	13.3	11.5	9.6	9.8	16.7	17.8	17.7	17.7	18.5	32.5
evans[10]	11.4	10.0	9.1	10.1	14.8	18.1	19.2	19.9	20.1	22.4
walker[35]	10.8	9.3	9.0	9.4	14.5	18.4	19.5	19.7	19.7	21.1
biswas[4]	12.5	9.8	9.3	9.1	16.5	17.9	17.9	17.6	18.5	28.6
vesely[34]	29.4	23.9	22.0	27.2	39.0	42.1	42.9	44.1	47.2	48.1
khan[18]	12.2	10.3	8.7	9.8	13.8	16.3	16.4	17.2	16.9	19.2
heller[13]	11.9	10.0	14.4	14.7	14.9	14.3	14.0	14.5	18.1	36.2
hybrid	12.2	10.3	8.7	9.8	13.8	16.3	17.8	19.3	22.8	30.5

8 References

[1] David Bailey. Private communication, 1989.

[2] David H. Bailey. FFTs in External or Hierarchical Memory. *Journal of Supercomputing*, 4(1):23–35, March 1990.

[3] Ron Bell. IBM RISC System/6000 NIC Tuning Guide for Fortran and C. Technical Report GG24-3611-01, IBM International Technical Support Center, Poughkeepsie, NY, July 1991.

[4] Amitava Biswas. Bit Reversal in FFT from Matrix Viewpoint. *IEEE Trans. Signal Proc.*, 39(6):1415–1418, June 1991.

[5] E. Oran Brigham. *The Fast Fourier Transform.* Prentice–Hall, Inc., Englewood Cliffs, N. J., 1974.

[6] Oscar Buneman. Conversion of FFT's to Fast Hartley Transforms. *SIAM J. Sci. Stat. Comput.*, 7(2):624–638, April 1986.

[7] James W. Cooley, Peter A. W. Lewis, and Peter D. Welch. The Fast Fourier Transform and its Applications. *IEEE Trans. Education*, 12(1):27–34, March 1969.

[8] Pierre Duhamel. A Connection between Bit Reversal and Matrix Tranposition: Hardware and Software Consequences. *IEEE Trans. Acoustics, Speech, Signal Proc.*, 38(11):1893–1896, November 1990.

[9] Alan Edelman. Optimal Matrix Transposition and Bit Reversal on Hypercubes: All–to–All Personalized Communication. *J. Parallel Distributed Computing*, 11:328–331, 1991.

[10] David M. W. Evans. An Improved Digit–Reversal Permutation Algorithm for the Fast Fourier and Hartley Transforms. *IEEE Trans. Acoustics, Speech, Sig. Proc.*, ASSP-35(8):1120–1125, August 1987.

[11] David M. W. Evans. A Second Improved Digit–Reversal Permutation Algorithm for Fast Fourier Transforms. *IEEE Trans. Acoustics, Speech, Sig. Proc.*, 37(8):1288–1291, August 1989.

[12] Jeanne Ferrante, Vivek Sarkar, and Wendy Thrash. On Estimating and Enhancing Cache Effectiveness. *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, August 1991. To appear in Springer Verlag's Lecture Notes in Computer Science series.

[13] Steven Heller and Alan Edelman. Private communication, 1991.

[14] Ching-Tien Ho and M. T. Raghunath. Efficient Communication Primitives on Hypercubes. Technical Report RJ 7932, IBM Almaden Research Center, 650 Harry Rd., San Jose, CA 95120, 1991.

[15] Roger W. Hockney and C. R. Jesshope. *Parallel Computers 2*. Adam Hilger, Bristol, England, 1988.

[16] S. Lennart Johnson and Ching-Tien Ho. Algorithms for Matrix Transposition on Boolean N-cube Configured Ensemble Architectures. *SIAM J. Matrix Anal. Appl.*, 9(3):419–454, July 1988.

[17] S. Lennart Johnson and Ching-Tien Ho. Optimal Communication Channel Utilizatio for Matrix Transposition and Related Permuations on Boolean Cubes. Technical Report TMC-187, Thinking Machines Corp., Cambridge, MA, 1991.

[18] Fuad Imran Khan. Another Fast Algorithm for Bit Reversal. Preprint, 1992.

[19] Leslie Kohn and Neal Margulis. The i860™ 64-Bit Supercomputing Microprocessor. In *Proceedings of Supercomputing '89*, pages 450–456, Los Alamitos, CA, November 1989. IEEE Computer Society Press.

[20] David G. Korn and Jules J. Lambiotte, Jr. Computing the Fast Fourier Transform on a Vector Computer. *Math. Comp.*, 33(147):977–992, July 1979.

[21] John Middleditch. Private communication, 1992.

[22] Robert J. Polge and B. K. Bhagavan. Efficient Fast Fourier Transform Programs for Arbitrary Factors with One Step Loop Unscrambling. *IEEE Trans. Computers*, pages 534–539, May 1976.

[23] Robert J. Polge, B. K. Bhagavan, and James M. Carswell. Fast Computational Algorithms for Bit Reversal. *IEEE Trans. Computers*, C–23(1):1–9, January 1974.

[24] Jeffery J. Rodriguez. An Improved Bit-Reversal Algorithm for the Fast Fourier Transform. In *IEEE Int. Conf. on Acoustics, Speech, Sig. Proc.*, pages 1407–1410, 1988.

[25] Petr Rösel. Timing of Some Bit Reversal Algorithms. *Signal Processing*, 18:425–433, 1989.

[26] Urzula Rutkowska. A Simple Algorithm for the Bit-Reversal Permutation. *Signal Processing*, 23:313–317, 1991.

[27] Urzula Rutkowska. Fast Recursive Unscrambling Algorithms. *J. New Gener. Comput. Syst.*, 4(1):29–40, 1991.

[28] Richard C. Singleton. A Method for Computing the Fast Fourier Transform with Auxiliary Memory and Limited High-Speed Storage. *IEEE Trans. Audio Electroacoust.*, AU-15:91–97, June 1967.

[29] Richard C. Singleton. An Algorithm for Computing the Mixed Radix Fast Fourier Transform. *IEEE Trans. Audio Electroacoust.*, AU-17:93–103, June 1969.

[30] Alan Jay Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.

[31] Paul N. Swarztrauber. FFT Algorithms for Vector Computers. *Parallel Computing*, 1:45–63, 1984.

[32] Paul N. Swarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5:197–210, 1987.

[33] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, Philadelphia, PA, 1992.

[34] Vítězslav Veselý. Fast Cell-Structured Algorithm for Digit Reversal of Arbitrary Length. *SIAM J. Sci. Stat. Comput.*, 12(2):298–310, March 1991.

[35] James S. Walker. A New Bit Reversal Algorithm. *IEEE Trans. Acoustics, Speech, Sig. Proc.*, 38(8):1472–1483, August 1990.

[36] H. H. Wang. On Vectorizing the Fast Fourier Transform. *Bit*, 20:233–243, 1980.

[37] Angelo A. Yong. A Better FFT Bit-Reversal Algorithm without Tables. *IEEE Trans. Signal Processing*, 39(10):2365–2367, October 1991.

[38] Hyunsoo Yoon, K. Y. Lee, and A. Bahiri. On the Modulo N Translators for the Prime Memory System. *J. Parallel Distrib. Comput.*, 8(1):72–76, January 1990.