Towards an Optimal Bit-Reversal Permutation Program

Larry Carter and Kang Su Gatlin
{carter, kgatlin}@cs.ucsd.edu
Department of Computer Science and Engineering, UCSD
9500 Gilman Drive, La Jolla, CA, 92093-0114

Abstract

The speed of many computations is limited not by the number of arithmetic operations but by the time it takes to move and rearrange data in the increasingly complicated memory hierarchies of modern computers. Array transpose and the bit-reversal permutation – trivial operations on a RAM – present non-trivial problems when designing highly-tuned scientific library functions, particular for the Fast Fourier Transform. We prove a precise bound for RoCol, a simple pebble-type game that is relevant to implementing these permutations. We use RoCol to give lower bounds on the amount of memory traffic in a computer with four-levels of memory (registers, cache, TLB, and memory), taking into account such "messy" features as block moves and set-associative caches. The insights from this analysis lead to a bit-reversal algorithm whose performance is close to the theoretical minimum. Experiments show it performs significantly better than every program in a comprehensive study of 30 published algorithms.

1. Background and related work

Given binary strings `a` and `b`, let `ab` denote their concatenation and `r(a)` denote the *reversal* of `a`. Thus, for instance, `r(01101) = 10110`, and `r(ab) = r(b)r(a)`.

Arrays will be indexed by binary strings. The pseudocode statement "`for i = 0 to N-1`" means that `i` iterates through all binary strings of length lg(`N`), where lg represents log base two.

Consider the following three programs, where `N` is a power of 2, `N` = `N1`$\times$`N2`, and `A` and `B` are arrays of length `N`:

```
Copy(A,B):
   for i = 0 to N-1
      B[i] = A[i]

Transpose(A,B):
   for i = 0 to N1-1
      for j = 0 to N2-1
         B[j,i] = A[i,j]

BitReverse(A,B):
   for i = 0 to N-1
      B[r(i)] = A[i]
```

In the Random Access Machine (RAM) model of computation [AHU74], all three programs have the same complexity, Θ(`N`). If we only count the cost of Loads and Stores of array elements (i.e., we assume that all addressing and looping computations are free) then each of these permutations has complexity exactly 2`N`.

Yet in practice, in the minds of people who write high-performance programs, these three permutations have very different costs. `Copy` is very fast. `Transpose` is likely to be slow because of the computer's memory hierarchy (e.g. the data cache), but with a little bit of work, it can be rewritten to be about as fast as `Copy`. But `BitReverse` has so many performance problems, due to architectural features such as cache and TLB associativity, that it is best avoided if at all possible.

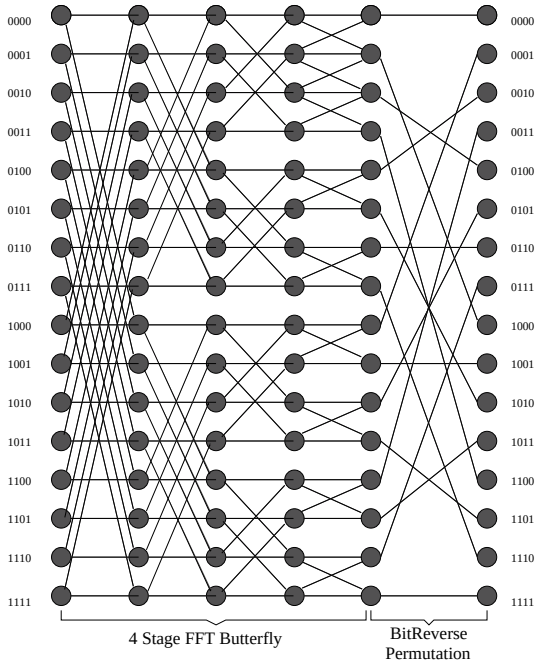

Figure 1. The "standard" picture of a FFT network, followed by the `BitReverse` permutation.

Our interest in very accurate analyses of `Transpose` and `BitReverse` comes from our interest in high-performance library programs for the Fast Fourier Transform (FFT). Figure 1 shows a directed acyclic graph (DAG) representation of a 16-point FFT, assuming all edges go from left to right. The right-most stage of the DAG is a `BitReverse`. This stage is necessary in practice so that repeated applications of the FFT can swap between the "time domain" and "frequency domain".

It is difficult to program a FFT that gets anywhere near the advertised "peak MFLOP/sec speed" of a modern workstation. For example, without any tuning, the 3-D FFT of the NPB 1.0 FT benchmark program runs on our desktop computer (a 60 MHz SPARCstation 20) at about 3.7 MFLOP/sec. It turns out that `fftz1`, the subroutine that actually does the "real work" of multiplying and adding floating point numbers, runs at a slightly more respectable 17 MFLOP/sec, but 78% of the runtime is "wasted" transposing arrays and copying data into contiguous memory locations so that `fftz1` can run without memory hierarchy problems.

This example highlights that a major difficulty with programming FFT's efficiently is choreographing the data movement. Because of the tremendous importance of FFT's,[1] many papers and books (e.g. [V-L92]) deal extensively with this question.

In practice, most FFT implementations avoid bit reversals, using "autosort" methods instead, which weave the bit reversal into the rest of the computation. One of the more popular algorithms is the "Four Step" FFT [GS66], advocated by Bailey [B90] particularly for computers with hierarchical memories. The algorithm performs a one-dimensional FFT by storing the data in a 2-D array in column major order, performing FFT's on the rows of the array, transposing the array (simultaneously multiplying the elements by appropriate "twiddle factors"), and finally performing FFT's on the rows of the transposed array.

Under a Uniform Memory Hierarchy model of computation [ACFS94], which models the hierarchical nature of computer memory, a recursively-implemented Four Step FFT is $\Theta(N \lg(N) \lg\lg(N))$. Alpern *et al* also develop a communication-efficient FFT with complexity $\Theta(N \lg(N))$. The elimination of the $\lg\lg(N)$ term is due to the replacement of the recursive transposes of the Four Step FFT by a single BITREVERSE. This result has provoked our interest in developing a highly-tuned implementation of BITREVERSE.

In a later section, we will describe the features of modern computers that affect the performance of these programs. Here, we discuss some theory whose goal is to provide more accurate modeling of the cost of computing permutations.

It is natural to consider a two-level memory model, in which data must be moved from a large, slow memory to a small, fast memory for processing. Permutations involve no re-use of data (each element is used only once) so models (such as the Red-Blue pebble game [HK81]) that ignore the spatial structure of memory don't provide any insight.

However, a two-level model becomes relevant when there is an added restriction that only contiguous *blocks* of data can be moved between the two levels. Floyd [F72] shows that if the small memory can hold only two blocks, each of size B elements, and $B \leq \min(N_1, N_2)$, then transposing a $N_1 \times N_2$ array requires exactly $2(N/B)\lg(B)$ block moves between the two level, where $N = N_1 N_2$. Aggarwal and Vitter [AV88] extend this result to show that if the small memory can hold $K \geq 2$ blocks, then transpose requires $\Theta((N/B)\lg(\min(KB, N_1, N_2, N/B))/\lg(K))$ block moves. In practice, when the two levels being modeled are disk and memory, or memory and cache, then K will be at least B. In this case, the above reduces to what has been known in practice since the earliest computers; that "tiling" an array into subarrays of size $B \times B$, and processing one tile at a time, allows transpose

[1] It has been estimated [JJ97] that in 1990, 40% of all CPU cycles executed by Cray Research supercomputers were devoted to FFT's.

to be computed with each data element making only one trip into the smaller memory. For some other architectural scenarios (narrow TLB's or cache associativity problems), K is smaller that B. In this case, the the result can be interpreted as saying that at most $O(\log_K(B))$ passes through the data are needed.

Our interest is in designing programs that perform optimally in practice, and proving their optimality. Doing this requires even more realistic models. A two-level analysis assumes that once a block of data is brought into the smaller memory, any permutation can be performed on the data at zero cost. This model is appropriate when the input and output arrays are stored on disk storage, since the cost of moving a block between disk and memory dominates all other costs. However, when the arrays are in main memory, the cost of moving data from memory to cache, for instance, has the same order of magnitude as moving data from cache to registers. Modeling this accurately enough to determine the constants requires a hierarchical model of memory.

In the Block Transfer (BT) model of hierarchical memory [ACS87], copying a block of consecutive locations takes one unit of time per element, after an initial access time that is a function of the source and target locations. The cited paper provides results for a variety of smooth access time cost functions. Unfortunately, there is no obvious way to translate these asymptotic analyses to specific results for the step-wise functions that occur in practice.

A particularly provocative result in [AC88] is that when block transfers in a certain BT model[2] are controlled by a virtual memory system, then any transpose program that moves each element of the source array directly to its final destination has cost $\Omega(n^{5/4})$. Nevertheless, there is a recursive algorithm that costs only $\Theta(nlog^2n)$. Although the assumptions correspond to unrealistically large costs, this result gives theoretical support to the idea that "unnecessary" data movement can result in a faster algorithm.

The Memory Hierarch model [ACF90, ACFS94] represents the memory of a single-processor computer as a sequence of progressively smaller modules, where the ith module can hold k_i blocks of size b_i elements. Transferring a block between a module and the next larger module requires time t_i. It is shown, assuming $k_i \geq b_i$ for each module and that $t_i = b_i$ (i.e., each bus is unit-bandwidth), transposing a "nicely-aligned" array requires only $(2+\epsilon)N$ cycles (where ϵ depends on the exact model, but is very small). Unfortunately, the assumption that $k_i \geq b_i$ doesn't hold for the problematic architectural scenarios. Further, many computers cannot overlap communication as required by the model.

[2]The specific model here assumes the initial access time of a block at address x is x.

Savage [S95] presents a multilevel pebble game and briefly suggests an extension that can model block moves, but his results don't apply to the issues we address in this paper.

The rest of the paper is organized as follows. First (to appeal to the theoretically-inclined reader) we introduce the game of RoCol and prove an exact bound. Section 3 then presents the unavoidable architectural details that affect the speed of real permutation programs. Section 4 applies RoCol to our permutation problems, making the following arguments for large `BitReverse` problem instances and *very* large `Transpose` instances:

- There is a trade-off between how often data is moved into cache and how often it is moved into registers. It is necessary either to move most elements into registers *twice* or into cache *multiple times*. For typical computers, it's better to optimize for cache.
- Given that each element is brought into cache only once, each page must be brought into TLB nearly $b_2/\sqrt{2b_1k_1}$ times.

The final section presents an optimized `BitReverse` program, and shows it is better than any other known method. This last task is made easier since a comprehensive study [K96] shows that Alan Karp's "Hybrid" bit reversal is superior to the 29 other algorithms he found in a thorough literature search. Our program beats Hybrid significantly.

2. The RoCol^{TM} pebble game

EQUIPMENT:
Two buckets, labeled A and B.
N pebbles (initially in A.)
An (infinitely large) "Go" board.
An integer K.

OBJECT OF THE GAME:
To move all the pebbles from A to B in as few moves as possible.

RULES:

1. Initially, all the pebbles are in A.
2. At most K pebbles can be on the Go board at any time.
3. There are two types of moves:
   - ROW move: Choose a row of the Go board. Place as many stones from A as desired (subject to the limit of rule 2) on that row, in any positions (but only one pebble per position).

- COLUMN move: Choose a column of the Go board, and move as many pebbles as desired from that column to bucket B.

4. The game is over when all the pebbles are in bucket B. The score is the number of moves. The goal is to use as few moves as possible.

STRATEGY:

A poor strategy would be to repeatedly make one ROW move to place K pebbles on one row of the board, and then to make K COLUMN moves to pick them up one at a time. This strategy would require $\lceil N(1+1/K) \rceil$ moves.

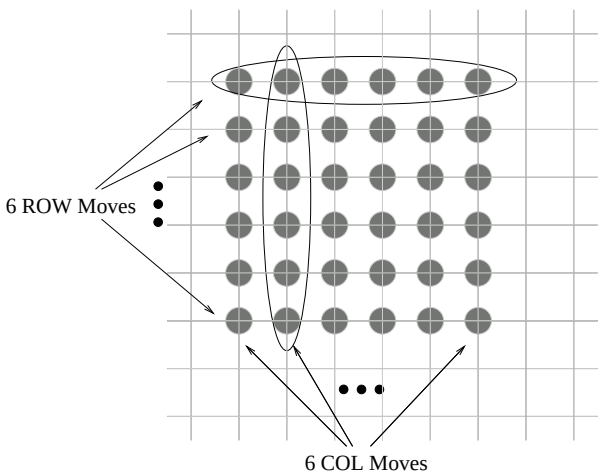

Figure 2. The "square" strategy for RoCol with $K = 36$. Six ROW moves create a square, then six COLUMN moves remove the 36 pebbles. It is optimal when $N = K$

A much better strategy is illustrated in figure 2 for the game with $K = 36$. For simplicity we assume $K = H^2$. First H ROW moves are made to create an $H \times H$ square of pebbles, and then H COLUMN moves empty the board into bucket B. This strategy has an average "bandwidth" from A to B of $\sqrt{K}/2$ pebbles per move, that is, assuming N is a multiple of K, the score will be $2N/\sqrt{K}$.

However, there is a still better strategy, as illustrated in figure 2 with $K = 36$. Assume K is a triangular number, i.e. $K = H(H+1)/2$. An initial H ROW moves are made to create a right triangle with legs of length H. Thereafter, we alternate one COLUMN move placing H pebbles in bucket B with one ROW move of H pebbles, restoring the triangle to its full size. The asymptotic "bandwidth" of this strategy is $H/2 = (\sqrt{1+8K}-1)/4$ pebbles per move, which is nearly $\sqrt{2}$ better than the previous strategy.

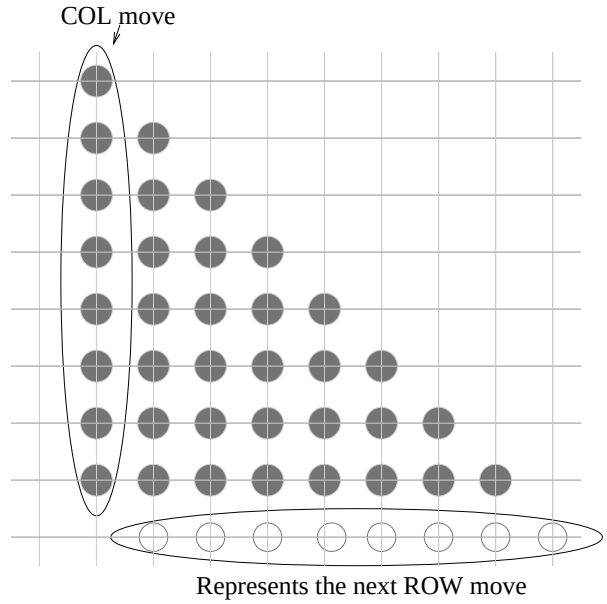

Figure 3. The optimal "triangle" strategy for RoCol with $K = 36$. Moves alternate removing a column of 8 pebbles and adding a row, restoring the triangle.

We will now prove that the triangle strategy is optimal within an additive constant.

Define $T(h) = h(h+1)/2$ on the positive real numbers, and let T^{-1} to be the inverse of T (i.e. $T^{-1}(k) = (\sqrt{1+8k}-1)/2$.)

Given a "board position" G of pebbles on the Go board, let r_j denote the number of pebbles on the j-th row and c_j the number on the j-th column. Define the potential P of position G as:

$$P(G) = \sum_j T(c_j) - \sum_j T(r_j)$$

P can be thought of as a measure of the "vertical-ness" of the arrangement of pebbles — a lot of pebbles in a small number of columns will result in a large P. A large P suggests there are a number of good COLUMN moves available.

Let G_i denote the board position after i moves have been made. Note that in a game of m moves, G_0 and G_m are the empty board position.

We first give a bound on how much the potential of a position can change in one move, then use that to bound the "bandwidth" of an entire game.

Lemma 1: $P(G_i) - P(G_{i-1}) \leq K - T(k_i)$, where k_i is the number of pebbles involved in move i.

Proof:

Case 1: Move i is a ROW move.

For a given j, the term $T(c_j)$ in the first summation in the definition of $P(G_i)$ can be visualized as the number of edges (including one self-edge) that can be drawn between pebbles in the j-th column. Thus, adding at most one pebble per column will increase this first summation by at most K (the maximum number of pebbles that can be on the board), since there will be one new edge per pebble in each column that gets a new stone.[3] The second summation increases by at least $T(k_i)$, even more if pebbles were placed on a row that already had some pebbles.

Case 2: Move i is a COLUMN move.

The proof is similar. (If you transpose the board and run time backwards, a COLUMN move becomes a ROW move.)

$\diamondsuit$

Lemma 2: In a game of m moves, $\sum_{i=1}^{m} T(k_i) \leq mK$.

Proof: Rewrite Lemma 1 as $T(k_i) \leq K + P(G_{i-1}) - P(G_i)$. Summing over all i from 1 to m and telescoping the interior terms yields $\sum_{i=1}^{m} T(k_i) \leq mK + P(G_0) - P(G_n) = mK$.

$\diamondsuit$

Lemma 3: $\sum_{i=1}^{m} k_i \leq mT^{-1}(K)$

Proof: Let $S = \sum_{i=1}^{m} T(k_i)$. Suppose we allow the k_i's to be arbitrary non-negative reals (rather than restricting them to the integers). It follows from the fact that the second derivative of T is positive that $\sum k_i$ is maximized, subject to the constraint that $\sum T(k_i) = S$, when the k_i's are all the same value, call it k_o.[4]

We can calculate k_o by observing that $S = \sum_{i=1}^{m} T(k_i) = mT(k_o)$, so $k_o = T^{-1}(S/m)$. Thus, the maximum value of $\sum_{i=1}^{m} k_i$ is $mT^{-1}(S/m)$.

It follows that for *any* set of k_i's that satisfy $\sum T(k_i) = S$, in particular for the game of interest, $\sum k_i \leq mT^{-1}(S/m)$. We also know from Lemma 2 that $S \leq mK$, and since T is an increasing function, it follows that $\sum_{i=1}^{m} k_i \leq mT^{-1}(K)$.

$\diamondsuit$

The above lemma leads to:

Theorem 1: A game of RoCol that allows at most K pebbles on the board at a time requires at least $2N/H$ moves to transfer N pebbles from A to B, where $H = T^{-1}(K)$ (i.e. $K = H(H+1)/2$).

[3] An alternate algebraic proof follows simply from $T(c_j) - T(c_j - 1) = c_j$.

[4] Proof: suppose to the contrary that $k_1 \neq k_2$. Let $k_1' = k_2' = (k_1 + k_2)/2$, and $k_i' = k_i$ for $i > 2$. We then have $\sum k_i' = \sum k_i$ but $\sum T(k_i') < \sum T(k_i)$. We could now increase k_1' to satisfy the constraint $\sum T(k_i) = S$, resulting in a larger value of the objective function $\sum k_i$.

Proof: A complete game of m moves involves moving all N pebbles out of bucket A and moving N pebbles into B, so the total number of pebble-moves, $\sum_{i=1}^{m} k_i$, is $2N$. The theorem follows immediately from Lemma 3.

$\diamondsuit$

Theorem 2: If $H = T^{-1}(K)$ is an integer, then a game of RoCol that allows at most K pebbles on the board at a time can move N pebbles from A to B in at most $2N/H + 2(H-1)$ moves.

Proof: The triangle strategy described earlier requires $H-1$ ROW initial moves to set up the triangle and $H-1$ to COLUMN moves to clear it at the end. The remaining moves proceed at a "bandwidth" of H pebbles per move. (Even if N is not a multiple of H, enough pebbles are moved during the initialization and final moves to handle the remainder.)

$\diamondsuit$

The reader is undoubtedly wondering what RoCol has to do with TRANSPOSE and BITREVERSE. The short answer is, the buckets model a large memory module, the ROW and COLUMN moves correspond to moving blocks of data into a smaller module, and the K pebbles represent the total memory available in a still smaller module. Thus, RoCol models the relationship among *three* levels of the memory hierarchy. But before this can be made clearer, we must describe the relevant architectural features.

3. A primer of architecture

A simplified description of the architectural features of modern PC's and workstations that most affect the performance of these programs, along with some sizes suitable for "back-of-the-envelop" intuition, follows:

- The computer's memory is partitioned into blocks of contiguous storage called *pages*. Typically, a page holds 1024 (4-Byte) array elements. Before the processor can access data in a page, the page must "be moved into the TLB".[5] The TLB can hold only a limited number of pages, say 64. Moving a page into the TLB might take 15 or 50 cycles, depending on the computer.

- Depending on assorted details[6] there may be restrictions on which pages can be in the TLB simultaneously. This so called "associativity problem" can be particularly acute when the addresses of the pages differ by multiples of a large power of two, as often happens with `BitReverse` and `Transpose`.

[5] This phrase describe something that is a bit more complicated. You don't want to know the details.

[6] They include the associativity of the TLB and whether the TLB mapping is randomized. You don't want to know.

- Pages in TLB are partitioned into blocks of contiguous storage called *cachelines*. Typically, a cacheline holds between 8 and 32 array elements (depending on the machine). Before the processor can can access data in a cacheline, it must be moved into cache. The cache can hold only a limited number of cachelines, say 64 or 1024 (depending on the machine). Moving a cacheline into cache takes perhaps 20 cycles.

- Depending on even more details[7] there are limitations on which cachelines can be in cache simultaneously. The cache associativity problem is particularly acute for `Transpose` on large arrays (where the number of rows or columns is a multiple of the cache size), or for `BitReverse` on even modest-sized arrays. Having an associativity problem means that all cachelines of `A` that contain data destined for contiguous locations in `B` are in the same "associativity class", and the architecture allows only some small number (depending on the machine) of these cachelines of `A` to be in cache at the same time.

- There may actually be two or three levels of cache. We don't consider multiple levels of cache in this paper.

- The processor has a number of *registers* (typically 32) and perhaps half can hold array elements (the remainder are needed for addresses, system pointers, etc.) One or two array elements can move between cache and registers per cycle.

4. Lower bounds on permutations

In this section, we use RoCol to derive lower bounds on the number of times data must be brought into TLB, cache, and registers. We will focus on programs that implement `Transpose`, since the exposition is easier, but the theorems apply to equally to `BitReverse`.

We always assume that `N1` and `N2`, the dimensions of `A`, are multiples of b_2, the size of a page. This implies they are also multiples of b_1 the size of a cacheline, since all hardware parameters are powers of two. We also assume the data are "nicely aligned", that is, every cacheline and page lies entirely within a single row of `A` or `B`.

Partition `A` and `B` into subarrays of size $b_1 \times b_1$. A performance problem with `Transpose` is that (for sufficiently large arrays) the b_1 cachelines that hold one such subarray of `A` are all in the same associativity class and cannot all co-reside in cache. Yet these lines hold elements that are destined to be moved to a single cacheline of `B`. In fact, each of the b_1 cachelines in the corresponding subarray of `B` gets one element from each of the lines in `A`'s subarray. It gets worse: the cachelines of the subarray of `B` are in the same associativity class.[8]

This situation leads to a trade-off between how often data is moved into cache and how often it is moved into registers. Suppose we restrict our implementation of transpose to programs that move each array element into a register only once during the execution of the program. The following theorem is applicable where `A` now represents a subarray whose elements are all in a single cache associativity class, and `B`'s elements are also in a single associativity class (though `B`'s class can be different from `A`'s.) The parameter Z is the "cache associativity". The theorem applies to each pair of subarrays separately, and then carries forward to the original (large) arrays.

Theorem 3: Given a computer with K registers and a cache that allows at most Z cachelines of `A` and Z cachelines of `B` in cache at a time. Let N be the size of `A`. Then any program that computes `Transpose(A,B)` by bringing each element into a register only once during the program must have at least $N/(Z + T^{-1}(K)/2)$ cachelines moves during its execution.

Proof: Corresponding to the given program is a *schedule* of data movement. This schedule is a sequence of operations of the following types:

- `Encache(X)` (where `X` is either `A[i,j]` or `B[j,i]`, for some `i` and `j`), which brings the cacheline containing the element `X` into cache.

- `Evict(X)`, which moves the cacheline containing `X` out of cache, freeing up space for a different cacheline to be cached.

- `Load(A[i,j],Rk)` (where $0 \leq \mathtt{k} < K$), which moves `A[i,j]` into the register `Rk`. The cacheline containing `A[i,j]` must currently reside in cache.

- `Store(Rk,B[j,i])` (where $0 \leq \mathtt{k} < K$), which moves the element in `Rk` into its final position in the `B` array. The cacheline containing `B[j,i]` must currently reside in cache.

- `Copy(A[i,j],B[j,i])`, which has the effect of a `Load` immediately followed by a `Store`, but doesn't require naming the register. Both `A[i,j]`'s and `B[j,i]`'s cachelines must currently reside in cache.

[7] They include the cache associativity, whether cache is physically or virtually addressed, and even how memory was allocated at runtime. You really don't want to know.

[8] Occasionally it will be the same associativity class as `A`'s subarray, making things a little worse still.

We assume the initial schedule doesn't contain any `Copy`'s; they are introduced to facilitate our analysis. By the theorem's assumptions, for each `i` and `j`, there is exactly one `Load(A[i,j])` and one `Store(B[j,i])` operation.

We will rearrange the schedule into a canonical form. First, we search the schedule for any instant of time when, for some `i` and `j`, both `A[i,j]`'s and `B[j,i]`'s cacheline are in cache. For each such occurrence, we remove the `Load(A[i,j],Rk)` and the `Store(Rk,B[j,i])` from the schedule, and instead, at the earliest point in the schedule where both lines are in cache, insert a `Copy(A[i,j],B[j,i])` operation. Note that this will be immediately after a `Encache` operation. For accounting purposes, we "charge" the `Copy` to the `Encache` operation it immediately follows.

After introducing as many `Copy`'s as possible, each `Encache` operation will have at most Z `Copy`'s charged to it, since there can be at most one for each cacheline of the opposite array that that is resident at the time of the `Encache` operation.

We now show how the schedule (ignoring the `Copy`'s) corresponds to a game of RoCol, where there is one pebble for each register, where the `(i,j)` pairs are the positions on the playing board, and where each `Encache` operation is a RoCol move.

Consider an `Encache(A[i,j])` operation and the set of subsequent `Load`'s from the cacheline containing `A[i,j]` that occur before the corresponding `Evict` operation. For each such `Load`, the corresponding `Store` must occur *after* the `Evict`, since otherwise we would have replaced the `Load-Store` pair with a `Copy`. Thus, it is legal to move all such `Load`'s down to just before the `Evict`, without needing to change any register names. We will bundle this set of `Load`'s together and call it a ROW move.

Similarly, for each `Encache(B[j,i])` operation, we consider the `Store` operations on that cacheline before it is next `Evict`-ed. These `Store`'s can be moved up to just after the `Encache`, and designated a COLUMN move.

Let c be the number of `Encache` operations. By our accounting method, we know that most cZ elements will be transposed by `Copy` moves. There remain $N - cZ$ elements to be moved in the RoCol game. Theorem 1 says that $c \geq 2(N - cZ)/T^{-1}(K)$. The theorem follows after a little algebra.

$\diamondsuit$

The significance of this theorem is more intuitive when expressed in terms of "roundtrips per element". A roundtrip means moving a 4-Byte array element into the smaller memory and back out again. For instance, a register roundtrip requires a load and a store, while a cache roundtrip involves two cache misses, one on each of the `A` and `B` arrays. Using this terminology, we can restate Theorem 3 as:

Corollary: Given a computer with K registers, a cacheline length of L and a cache associativity of Z, then `Transpose` with one register roundtrip per element will require at least $L/(2Z + T^{-1}(K))$ cache roundtrips per element.

Proof: Each cacheline move corresponds to a one-way trip for one L elements. To get the average number of roundtrips made by each element, we multiply the result of the theorem by $L/(2N)$

$\diamondsuit$

The 66-MHz IBM Power2 processors used in our later experiments has $Z = 4$ and $L = 32$ (measured in 4-Byte elements). The corollary says that even if we could use all $K = 32$ registers (so $T^{-1}(K) \approx 7.5$), an algorithm with one register roundtrip per element must average at least $32/15.5 \approx 2.06$ cache roundtrips per element.

An important observation is, **exactly the same analysis applies to the TLB**. For the Power2's TLB, $Z = 2$ and $L = 1024$; hence an algorithm with one register roundtrip per element must average at least $1024/11.5 \approx 89$ TLB roundtrips per element.

Is it worth having this much cache and TLB traffic, just so an algorithm can make optimum usage of register traffic? The answer depends on the relative costs, and the possible alternative algorithms. Table 1 provides the answer for the Power2. The cost of a register roundtrip is 1, since the Power2 can execute two memory operations (load's or store's) per cycle. The cache cost assumes that 25-cycles are required for a 32-element cacheline, i.e. a steady-state of 25 cycles for 16 roundtrips, or 1.56 cycles per roundtrip. The TLB on the Power2 is uncommonly fast – 15 cycles per 1024-element page. An alternative algorithm, shown in the last column, is COBRA, the `BitReverse` program[9] described in the next section. COBRA has two register roundtrips per element, but only one cache roundtrip and (on the Power2) 16 TLB roundtrips.

This figure suggest that the COBRA algorithm has a significant advantage. In practice, it is impossible to design a register-efficient algorithm that has only 2.06 cache roundtrips per element; 4 is a more realistic number. Nevertheless, the projected costs of the COBRA algorithm are close to those of our actual experiment.

Thus, it is better to have a cache-efficient algorithm (one that makes only one cache roundtrip per element), than a register-efficient one.

[9] Recall our assertion that in theory, bit-reversals and transpose have the same analysis. In practice, bit-reversals have more overhead and other problems.

Memory level (cycles/rt)	Register-efficient rt's	cost	COBRA rt's	cost
TLB (0.03)	89	2.60	16	0.47
Cache (1.56)	2.06	3.21	1	1.56
Reg (1.00)	1.00	1.00	2	2.00
Total cost		6.81		4.03

Table 1. Comparison of the theoretical lower bound cost, in cycles/element, for a `BitReverse` that is constrained to one register roundtrip (rt) per element, compared to the data movement cost of the COBRA algorithm, which has two register roundtrips per element. All costs are computed for the IBM Power2 processor.

We now focus on lower bounds for cache-efficient transpose and bit-reversal algorithms. First we give a lower bound on the number of register roundtrips that must be made.

Theorem 4: Suppose our computer has K registers and a cache that allows at most Z cachelines of `A` and Z cachelines of `B` in cache at a time, where each cacheline is of length L. Then any program that computes `Transpose(A,B)` with one cache roundtrip per element must have an average of at least $2 - (2Z + T^{-1}(K))/L$ register roundtrips per element.

Proof: Since the program is cache-efficient, it has exactly $2M/L$ cacheline moves, where M is the size of each array. Consider the schedule of data movement as described in the proof of Theorem 3. If we remove from this schedule all `Load`'s and `Store`'s of elements that make more than one roundtrip into registers, what remains will be the schedule of a program that transposes a sparse array. Theorem 3 asserts that this program (which we know has $2M/L$ cacheline moves) has at least $N/(Z + T^{-1}(K)/2)$ cachelines moves, where N is the number of elements in the sparse array. Thus, $2M/L \geq N/(Z + T^{-1}(K)/2)$, and so $N/M \leq (2Z + T^{-1}(K))/L$. Note that N/M is the fraction of elements that require only one register roundtrip; the remaining $1 - N/M$ require at least two. Thus, the average number of register roundtrips per element is at least $2 - (2Z + T^{-1}(K))/L$.

$\diamondsuit$

When applied to COBRA on the Power2, Theorem 4 says there must be at least $2 - (8 + 7.5)/32 = 1.52$ register roundtrips per element on average. Thus, it might be possible to save a half-cycle per element, thought at the cost of making the program more complicated.

Finally, we give a lower bound on the number of TLB roundtrips required by a cache-efficient program.

Theorem 5: Suppose our computer can hold K elements in cache and registers combined, and it has TLB associativity of Z and page length L elements. Assuming that at most Z pages of `A` and Z pages of `B` can be in the TLB at a time, any program that computes `Transpose(A,B)` with only one cache roundtrip per element must have an average of at least $L/(2Z + T^{-1}(K))$ TLB roundtrips per element.

Proof: The proof is essentially the same as the proof of Theorem 3 and its corollary, with "TLB pages" playing the role of cachelines, and with "cache" playing the role of registers.

We construct a schedule with `Encache` and `Evict` operations corresponding to pages moved into and out of the TLB. When a cacheline of `A` is moved from memory into cache, a `Load` operation is put into the schedule for each element of the cacheline. When a cacheline of `B` is moved out of cache, a `Store` for each element of the line is put into the schedule. All other data movement into registers or from one location to another in cache are ignored in the schedule. Reasoning as in Theorem 3, at least $N/(Z + T^{-1}(K)/2)$ pages must be moved into TLB during the execution of the program, where N is the number of elements in `A`. To get the number of roundtrips per element, we multiply by $L/(2N)$.

$\diamondsuit$

Applying Theorem 5 to the Power2 (which can hold 32800 elements in cache and registers combined), there must be at least $1024/(4 + T^{-1}(32800)) = 1024/259.6 = 3.94$ TLB roundtrips per element on average. Combined with the result from Theorem 4, the theoretical lower bound on any cache-efficient `Transpose` or `BitReverse` on the Power2 is 3.20 cycles/element.

5. An efficient bit-reversal

We now give a BitReverse algorithm that is cache-efficient and also has good TLB efficiency. It uses the square strategy of Figure 2, which is easier to implement and almost as efficient as the triangle strategy of Figure 3. The Cache Optimal BitReverse Algorithm (COBRA) that we present will also be the subject of the experiments presented in section 6.

The length of a binary string a is denoted $|a|$ (e.g. $|0100| = 4$). We represent the indices of the array we wish to `BitReverse`, `A`, with binary strings of the form `abc`, where `a` and `c` are of length q, where q is chosen to be at least lg of the size of a cacheline. Thus if $|\texttt{abc}| = \lg N$ then $|\texttt{b}| = \lg N - 2q$.

The pseudocode for COBRA is in figure 4. In COBRA,

```
for b = 0 to 2^(lgN-2q)-1
   b' = r(b)
   for a = 0 to 2^q-1
   a' = r(a)
      for c = 0 to 2^q-1            [1]
         T[a'c] = A[abc]

   for c = 0 to 2^q-1
      c' = r(a)
      for a' = 0 to 2^q-1           [2]
         B[c'b'a'] = T[a'c]
```

Figure 4. Pseudocode for COBRA.

for a given b, all data of the form A[*b*] is copied,[10] one cacheline at a time, to the temporary array T, which should reside completely in cache (this is all done in loop [1]). To ensure that T remains in cache, 2^{2q} should be smaller than the size of cache (in most cases making it $50\% - 75\%$ of the size of cache is effective to minimize conflict with other data that also sits in cache). In loop [2], the data are moved from T to the destination array B. Again, the data are moved into a cacheline of B at a time, avoiding associativity problems on B. The non-sequential references to T don't hurt, since the entire T array remains in cache.

Note that the inner loops of COBRA are very efficient. In loop [1], iterating on c results in "stride one" (i.e. sequential) accesses over A and T. In loop [2] iterating on a' gives stride one accesses over B and stride 2^q over T. The constant stride accesses make indexing efficient.

Thrashing can occur on loop [1] for certain values of b where A[abc] and T[a'c] become cache aligned. We minimize this effect in our implementation by unrolling the inner loop four times and scalarizing (loading four elements into registers and then storing the elements from the four registers into memory locations) in the unrolled loop.

Lastly, we note that the above pseudocode is for an out-of-place BitReverse. An in-place BitReverse (one that rearranges the elements but leaves the results in the same array) using the same principles is not hard to design.

6. Experimental results

COBRA was implemented on two computers, an IBM Power2 and a DEC Alpha 21164. The Power2 was described earlier. The Alpha has a one-way set associative 8-KByte primary (L1) data cache, where each cacheline holds 32 Bytes. It also has a 3-way set associative 96-KByte secondary (L2) cache with 64-Byte cachelines. The L1 cache has a 6-cycle miss penalty when the data are in L2, and an approximately 50-cycle additional cost (according to our experiments) when the data must be fetched from memory. The TLB has associativity 64 (it is fully-associative), each page is 8 KBytes, and a TLB miss costs about 75 cycles.

Figures 5 and 6 compare COBRA to the Hybrid program developed by Alan Karp. Karp's experiments [K96] demonstrate that on a wide variety of architectures, Hybrid is consistently either the best performing or near the best performing code, compared to 29 other methods he found in a thorough literature search.

The size we selected for $|a|$ and $|c|$ were dependent on the size of cache. On the IBM Power2 with its large cache, we found $|a| = |c| = 6$ was best. On Alpha, $|a| = |c| = 5$ was better, probably due to the smaller L1 cache.

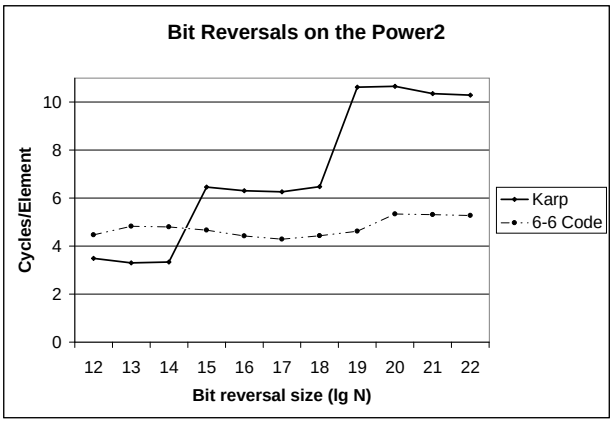

Figure 5. Performance of COBRA and Alan Karp's Hybrid bit-reversal programs on the Power2.

These figures show the near-linear performance of COBRA on both machines. COBRA is faster than Hybrid for almost all problem sizes. On the Power2, COBRA is nearly twice as fast as Hybrid on large arrays, and on the Alpha, it is more than 4 times as fast. We suspect the poorer scalability of Hybrid on the Alpha (compared to the Power2) is due to two factors: the Alpha has a smaller L1 cache than the Power2, requiring Hybrid to make more passes over the data, and the miss penalties on the Alpha are significantly greater than on the Power2.

[10] Each '*' represents all binary strings of length q.

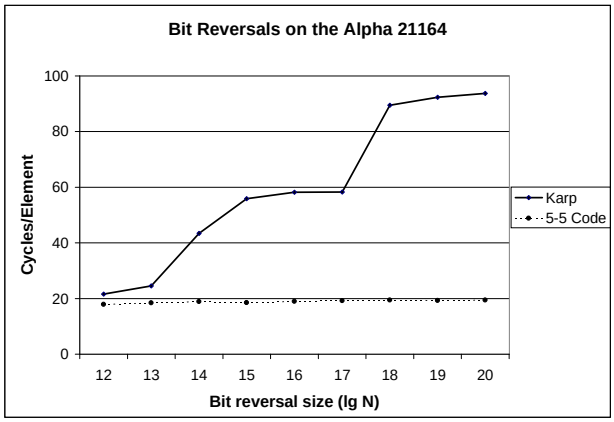

Figure 6. Comparison of COBRA and Hybrid on the Digital Alpha 21164.

7. Future work

Our goal is to eliminate the gap between the time required by FFT programs and the analyses that give lower bounds on this time. Having a nearly optimal `BitReverse` program gives us hope that the communication-efficient FFT of [ACFS94] can be made to out-perform existing implementations of other algorithms. What is needed next is a study of the "messy details" not modeled in by the UMH (particularly cache associativity) that are important to the performance of the remaining steps of the FFT algorithm.

8. Acknowledgements

Alan Karp kindly provided a copy of the programs used in his experiments. This greatly facilitated our experiments.

We also thank Ashok Chandra for a suggestion that sharpened Lemma 1 and cleaned up some messy details.

References

[ACS87] Aggarwal, A., A. K. Chandra, and M. Snir, "Hierarchical Memory with Block Transfer," *Proc 28th Symp. on Foundations of Comp. Sci.*, October 1987, pp. 204-216.

[AC88] Aggarwal, A., A. K. Chandra, "Virtual Memory Algorithms," *Proc. 20th. Symp. on Theory of Comp.*, May 1988, pp. 173-185.

[AV88] Aggarwal, A. and J. Vitter, "IO Complexity of Sorting and Related Problems," *CACM*, September 1988, pp. 305-314.

[AHU74] Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

[ACF90] Alpern, B., L. Carter, and E. Feig, "Uniform Memory Hierarchies," *IEEE Conference on Foundations of Computer Science*, October, 1990.

[ACFS94] Alpern, B., L. Carter, E. Feig, and T. Selker, "The Uniform Memory Hierarchy Model of Computation," *Algorithmica*, Volume 12, Number 2-3 (August–September 1994).

[B90] Bailey, D. H., "FFTs in External or Hierarchical Memory," *The Journal of Supercomputing*, v. 4, pp. 23-35, 1990.

[F72] Floyd, R. W., "Permuting Information in Idealized Two-Level Storage," *Complexity of Computer Computations*, Plenum Press, New York, 1972, pp. 105-109.

[GS66] Gentleman, W.M. and Sande, G., "Fast Fourier Transforms - For Fun and Profit," AFIPS Proceedings, vol. 29, pp. 298-309 (1966).

[HK81] Hong, J-W. and H. T. Kung, "I/O Complexity: The Red-Blue Pebble Game," *Proc. 13th. Symp. on Theory of Comp.*, May 1981.

[JJ97] Johnson, J.R. and R.W. Johnson, "Challenges of Computing the Fast Fourier Transform," DARPA Conference, June 1997.

[K96] Karp, A.H., "Bit Reversal on Uniprocessors," SIAM Review, Vol 38, No. 1, pp 1-26, March 1996.

[S95] Savage, J.E., "Extending the Hong-Kung Model to Memory Hierarchies," *Computing and Combinatorics* (Proceedings from COCOON '95), LNCS 959, pp.270-281, Springer-Verlag, 1995.

[V-L92] Van Loan, C., *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philidelphia, 1992.