# A Blocked All-Pairs Shortest-Paths Algorithm

**Gayathri Venkataraman** **Sartaj Sahni** **Srabani Mukhopadhyaya**
gvenkata@cise.ufl.edu sahni@cise.ufl.edu srabani@cise.ufl.edu
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611

We propose a blocked version of Floyd's all-pairs shortest-paths algorithm. The blocked algorithm makes better utilization of cache than does Floyd's original algorithm. Experiments indicate that the blocked algorithm delivers a speedup (relative to the unblocked Floyd's algorithm) between 1.6 and 1.9 on a Sun Ultra Enterprise 4000/5000 for graphs that have between 480 and 3200 vertices. The measured speedup on an SGI O2 for graphs with between 240 and 1200 vertices is between 1.6 and 2.

Additional Key Words and Phrases: All pairs shortest paths, blocking, cache, speedup.

## 1. INTRODUCTION

Traditionally, algorithms are developed, analyzed, and optimized for the RAM computer model in which a computer has a single uniformly accessible memory [10]. Contemporary computers, however, have multiple levels of memory and the memory access time varies significantly from one memory level to the next. For example, contemporary Sun and SGI workstations have an L1 cache, an L2 cache, and a main memory (Figure 1). The L1 cache in a Sun Ultra Enterprise 4000/5000 is 16 KB, the L2 cache is 4 MB, and main memory is in excess of 100 MB. Additionally, a contemporary computer has a limited number of registers—ten to twenty. Typically, it takes 1 cycle to access data from L1 cache. When the desired data is not in L1 cache, we experience an L1 *miss* and the data is brought from L2 cache to L1 cache using 6 to 10 cycles. If the desired data is not in L2 cache either, then we experience an L2 miss and data is fetched from main memory into L2 cache at a cost of (say) 50 cycles, and from there to L1 cache [17]. Computers with very fast processors may have an L2 miss penalty that is conderably larger than the typical 50 cycle penalty. We can reduce run time by organizing our computations so as to minimize the number of L1 and L2 cache misses.

Although several theoretical models for computers with multiple-level memories have been proposed [3; 2; 5], these models have not found wide application, and most of the work in the area of performance enhancement via cache optimization has been experimentally oriented. Trace driven simulators have been used to study the

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

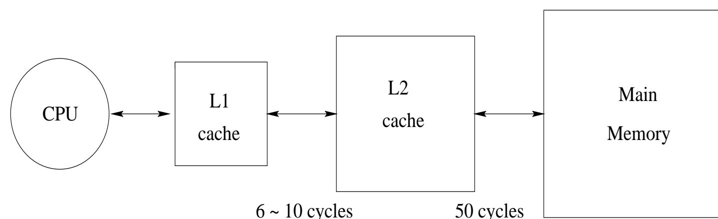



Fig. 1. A computer with two levels of cache

cache performance of a specific program running on a specific computer, determine the portions of the code or the data structures that result in a large fraction of the cache misses, and then optimize these code segments and/or data structures. Trace driven simulations have also been used to develop analytical models of cache behavior. See [4; 15; 19; 21; 22; 23], for example, for some ways in which trace driven simulators have been used in cache performance enhancement studies.

LaMarca and Ladner [12] develop a model for a single-level direct-mapped cache. They use this model to analyze the performance of binary heaps and cache-aligned $d$-heaps. Table 1 gives the speedups exhibited by heapsort when using a cache-aligned 4-heap vs. a binary heap. These speedups are from [12] and are for sorting 1,000,000 uniformly distributed 32-bit integers. The reported speedup generally increases as the number of elements to be sorted increases.

Table 1. Speedup of cache-aligned 4-heap relative to a binary heap

| Machine | Speedup |
|---|---|
| Pentium 90 | 1.70 |
| Power PC | 1.62 |
| Alphastation 250 | 1.46 |
| Alpha 3000/400 | 1.45 |
| Sparc 20 | 1.38 |

LaMarca and Ladner [13] optimize the cache performance of several sorting methods. Table 2 gives the measured speedups for sorting 1,000,000 uniformly distributed integers using cache optimized versions of heap sort and merge sort versus traditional implementations of these sort methods.

Xiao, Zhang, and Kubricht [25] use padding, partitioning, and buffering to improve the cache performance of tiled merge sort. Also, they use these techniques to obtain a version of quick sort that has improved cache utilization. Rahman and Raman [18] study cache effects in distribution sorting algorithms.

Lam, Rothberg, and Wolf [11] have considered the cache performance of a blocked matrix multiply code relative to a traditional matrix multiply code. Notcie that blocking in matrices is just the two-dimensional analog of tiling as used in tiled sort methods. For a size 300 matrix, their blocked matrix multiply code achieved a speedup of 4.3 on a DecStation 300 and a speedup of 3.0 on an IBM RS/6000.

Table 2. Speedups for cache-optimized heap sort and merge sort

| Machine | Speedup | |
|---|---|---|
| | Memory-tuned heapsort | Tiled mergesort |
| Sparc 10 | 1.85 | 1.38 |
| Power PC | 1.65 | 1.10 |
| Pentium PC | 1.75 | 1.15 |
| Dec Alphastation 250 | 1.58 | 1.90 |
| Dec Alpha 3000/400 | 1.62 | 1.32 |

This speedup is relative to the traditional matrix multiply algorithm. Eiron et al. [7] take another look at cache optimization strategies for matrix multiplication.

Ladner, Fix, and LaMarca [?] present a model that may be used to study the cache performance of algorithms on direct mapped caches. They use this model to analyze the cache performance of binary tree traversal and counting items in a large array. traversals and random accesses. In this paper we propose a blocked formulation of Floyd's dynamic programming algorithm to find the lengths of the shortest paths between all pairs of vertices in a graph [10]. Blocked (or tiled) computation methods have been used before (for example, [16; 9; 24; 11; 8; 1]). Our blocked algorithm provides a speedup (relative to the unblocked algorithm) between 1.6 and 1.9 on a Sun Ultra Enterprise 4000/5000 for graphs that have between 480 and 3200 vertices. The measured speedup on an SGI O2 for graphs with between 240 and 1200 vertices is between 1.6 and 2. These speedups are comparable to the speedups cited above for cache-optimized sorting and matrix multiplication codes on Sun platforms.

In Section 2 we give Floyd's all-pairs shortest-paths algorithm. Section 3 analyzes the potential speedup benefits from reorganizing Floyd's algorithm to make better use of cache. This analysis uses data gathered using the cache simulation tool *Shade* [20]. Our blocked version of Floyd's algorithm and a correctness proof are given in Section 4. Section 5 gives measured speedup results for our blocked algorithm.

## 2. FLOYD'S ALL-PAIRS SHORTEST-PATHS ALGORITHM

Let $G = (V, E)$ be a directed graph with $n$ vertices. Let $cost$ be the cost adjacency matrix for $G$. So $cost(i,i) = 0$, $1 \leq i \leq n$; $cost(i,j)$ is the length (or cost) of edge $(i,j)$ if $(i,j) \in E(G)$ and $cost(i,j) = \infty$ if $i \neq j$ and $(i,j) \notin E(G)$.

In the all-pairs shortest-paths problem we are to determine a matrix $A$ such that $A(i,j)$ is the length of a shortest path from $i$ to $j$. When $G$ has no cycle whose length (cost) is less than 0, the matrix $A$ may be computed using dynamic programming [10]. Let $A^k(i,j)$ be the length of a shortest path from $i$ to $j$ under the constraint that the path contain no intermediate vertex whose index is more than $k$. It is easy to see that $A(i,j) = A^n(i,j)$. When $G$ has no cycle with negative length, the following dynamic programming recurrence is valid:

$$A^0(i,j) = cost(i,j) \tag{1}$$

$$A^k(i,j) = \min\{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\}, k \geq 1 \tag{2}$$

```
function AllPairs(int A, int n)
{// A[i][j] = cost(i,j) initially
 // A[i][j] equals length of shortest i to j path on termination
  for (k = 1; k <= n; k++)
     for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
           A[i][j] = min(A[i][j], A[i][k] + A[k][j]);
}
```

Fig. 2. Floyd's shortest-paths algorithm

Equations 1 and 2 lead to the algorithm of Figure 2 to compute $A$. This algorithm is known as Floyd's algorithm. It may be shown [10] that `AllPairs` computes $A^k(i,j) = \mathtt{A[i][j]}$ in iteration `k` of the outermost `for` loop.

Although Floyd's algorithm has the same nested-three-loop structure as does the classical matrix multiplication algorithm, the two algorithms are quite different. For example, in the matrix multiplication algorithm, we can use any of the six possible permutations for the order of the three loops without affecting the computed result. In Floyd's algorithm, only the order of the innermost two loops may be changed; the outermost loop must remain outermost. The ability to freely permute the loops of the matrix multiplication algorithm allows us to decompose the loops so as to work with blocks of matrices. At any time, only two blocks are being worked on. When we attempt to decompose the loops of Floyd's algorithm so to work with blocks, the decomposition does not perform the same computations as the original. However, we can prove that the final computed results are the same. Further, as we shall see, the blocked version of Floyd's algorithm will mork on at most three blocks at a time.

## 3. UPPER BOUND ON ATTAINABLE SPEEDUP

We compute an upper bound on the maximum speedup attainable by rearranging the computation of Figure 2 so as to optimize cache useage. In computing this bound we assume that any rearrangement of the computation will not decrease the number of accesses made to the elements of the array `A`.

We first obtain an equation to estimate the execution/run time of Floyd's algorithm of Figure 2. The execution time of a program is given by the following equation [17]:

$$execution\ time = (CPU\ clock\ cycles + memory\ stall\ cycles) \times clock\ cycle\ time \quad (3)$$

where *memory stall cycles* is the number of cycles the CPU spends waiting for a memory reference to complete. The following equations are also from [17].

$$CPU\ clock\ cycles = CPI \times IC \quad (4)$$

$$memory\ stall\ cycles = number\ of\ L1\ misses \times L1\ miss\ penalty \quad (5)$$

$$number\ of\ L1\ misses = IC * L1\ misses\ per\ instruction \quad (6)$$

$$L1\ misses\ per\ instruction = memory\ references\ per\ instruction \times L1\ miss\ rate \quad (7)$$

where $IC$ is the *instruction count*, $CPI$ is the *clock cycles per instruction*, $L1$ *miss penalty* is the number of cycles the CPU waits when there is an L1 cache miss, and $L1$ *miss rate* is the number of L1 misses per memory reference.

Notice that Equation 5 ignores register misses. That is, when the data needed by the CPU is not in one of its registers, a register miss occurs. A register miss incurs a one-cycle penalty to get the required data from L1 cache. Therefore, Equation 5 actually underestimates the memory stall cycles. Like [17], we ignore the effect of register misses in our analysis.

From these equations we obtain:

$$execution\ time = (CPI \times IC\ +\ IC \times L1\ misses\ per\ instruction \times L1\ miss\ penalty) \times clock\ cycle\ time \quad (8)$$

We also see that

$$L1\ miss\ penalty = L2\ hit\ time + L2\ miss\ rate \times L2\ miss\ penalty \quad (9)$$

where $L2$ *hit time* is the number of cycles to load an L1 cache line from L2 cache, $L2\ miss\ penalty = memory\ hit\ time$ is the number of cycles needed to load an L2 cache line from main memory, and $L2$ *miss rate* is the number of L2 misses per L2 reference.

We use Equations 8 and 9 to estimate the run time of Floyd's algorithm. Since the L2 hit time and L2 miss penalty are architecture dependent and not available to us, we use typical numbers for these—the L2 hit time is assumed to be between 6 and 10 cycles and the L2 miss penalty is assumed to be 50 cycles. For the L1 misses per instruction and the L2 miss rate we use data obtained by using the cache simulator Shade on Floyd's algorithm. Table 3 gives this data for the SUN Enterprise 4000/5000. We note that processors such as the Sparc III have on-chip data collection for L1 cache misses. So, for these newer processors the L1 miss rate can be obtained by querying the appropriate register rather than by using a cache simulator such as Shade.

Table 3. Cache simulator data for algorithm of Figure 2

| **Matrix size** | **L1 misses per instruction in %** | **L2 miss rate in %** |
|---|---|---|
| 480 | 3.950 | 18.42 |
| 800 | 4.106 | 19.17 |
| 1600 | 4.133 | 19.42 |
| 2400 | 4.826 | 19.64 |
| 3200 | 5.553 | 20.07 |

Now we obtain a lower bound on the run time of a cache optimized version of Floyd's algorithm. Substituting Equation 7 into Equation 8 and making the reasonable assumption that cache optimization will not decrease the total number

Table 4. Cache characteristics of the Sun Enterprise 4000/5000

| Cache type | Associativity | Cache size | Line size |
|---|---|---|---|
| L1 | Direct mapped | 16KB | 32 bytes |
| L2 | Direct mapped | 4MB | 64 bytes |

of memory references (i.e., the number of memory references for the cache optimized code is at least $IC * memory\ references\ per\ instruction$ where $IC$ and $memory\ references\ per\ instruction$ are for `AllPairs`) yields

$$execution\ time \geq (CPI \times IC + IC \times memory\ references\ per\ instruction \times L1\ miss\ rate \times L1\ miss\ penalty) \times clock\ cycle\ time \quad (10)$$

The cache simulator gives 0.35 as the memory references per instruction for `AllPairs`. Substituting 0.35 for the number of memory references per instruction and the right side of Equation 9 for the L1 miss penalty into Equation 10, we get

$$execution\ time \geq (CPI \times IC + 0.35 \times IC \times L1\ miss\ rate \times (L2\ hit\ time + L2\ miss\ rate \times L2\ miss\ penalty)) \times clock\ cycle\ time \quad (11)$$

We may obtain a lower bound for the L1 and L2 miss rate by determining the minimum number of L1 and L2 misses that every reorganized version of Figure 2 must make. Since we intend to declare `i`, `j`, `k`, and `n` as register variables [6], references to these variables do not access cache and so do not cause any cache misses. Therefore, we focus on cache misses attributable to the array `A`. For our analysis we use the cache characteristics of the Sun Enterprise 4000/5000 that are shown in Table 4. By direct mapped we mean that each byte of main memory has exactly one byte of cache to which it may be mapped (i.e., cache associativity is 1). The line size of a cache gives the unit of memory transfer. So in the Sun Enterprise 4000/5000 an L1 cache miss results in a 32-byte block of data being transferred from L2 cache into L1 cache. The transferred block is one-half of an L2 line.

Since Floyd's algorithm accesses each of the $n^2$ elements of $A$, all $n^2$ elements of $A$ must get to L1 cache at some time. Let $N1$ be the number of elements of $A$ that fit in an L1 cache line. Since each L1 cache miss brings in exactly $N1$ elements of $A$, the number of L1 cache misses is at least $n^2/N1$. By a similar reasoning, the number of L2 cache misses is at least $n^2/N2$, where $N2$ is the number of $A$ elements that fit into an L2 cache line. Further, Floyd's algorithm makes $3n^3$ read accesses to $A$ (i.e., in the right side of the `min` statement of Figure 2) and $n^3$ write accesses (the left side of the `min` statement). We note that when the `min` statement of Figure 2 is coded as an `if` statement, write accesses are made only when the new `a[i][j]` value is smaller than the old one. In this case the number of write accesses ranges from 0 to $n^3$. To keep the analysis simple, we use $n^3$ as the write

access count. The total number of accesses to $A$ (read and write) is $4n^3$. Therefore,

$$\text{L1 miss rate} = \text{L1 misses per } A \text{ reference} \geq n^2/N1/(4n^3) = 1/(4 * N1) \quad (12)$$

$$\text{L2 miss rate} \geq \text{L2 misses per } A \text{ reference} \geq n^2/N2/(4n^3) = 1/(4 * N2) \quad (13)$$

The equality between the L1 miss rate and the misses per $A$ reference in Equation 12 follows from our assumption that variables other than $A$ will be register variables and so all memory references are to elements of $A$. Further, each reference to an element of $A$ is also a reference to L1 cache. The relation between the L2 miss rate and the L2 misses per $A$ reference in Equation 13 follows from the observation that the number of L2 references cannot exceed the number of L1 references, which, by our assumptions, equals the number of $A$ references.

Since we assume that cache optimization does not reduce the number of $A$ references, these bounds apply to all cache optimized versions of `AllPairs`.

Substituting the bounds of Equations 12 and 13 into Equation 11, we get the following lower bound on the run time of a cache optimized version of Floyd's algorithm.

$$\begin{aligned} \textit{execution time} \geq\ & (CPI \times IC + 0.35 \times IC \times 1/(4 * N1) \times \\ & (\textit{L2 hit time} + 1/(4 * N2) \times \textit{L2 miss penalty}) \\ & \times \textit{clock cycle time} \end{aligned} \quad (14)$$

Dividing Equation 8 by Equation 14 yields the following bound on the speedup obtainable by optimizing cache utilization.

$$\textit{speedup} \leq \frac{CPI + \textit{L1 misses per instruction} \times (\textit{L2 hit time} + \textit{L2 miss rate} \times \textit{L2 miss penalty})}{CPI + 0.35 \times 1/(4 * N1) \times (\textit{L2 hit time} + 1/(4 * N2) \times \textit{L2 miss penalty})} \quad (15)$$

Figure 3 plots the maximum value of the right side of Equation 15 when $CPI$ ranges between 1 and 2, L2 hit time ranges from 6 to 10 cycles, and L2 miss penalty is 50 cycles. The L1 misses per instruction and the L2 miss rate are taken from Table 3. For $N1$ and $N2$, we assume that $A$ is an integer array and that each integer is 4 bytes. Using an L1 cache line size of 32 bytes and L2 line size of 64 bytes (these sizes correspond to those for the Sun Enterprise 4000/5000), we get $N1 = 8$ and $N2 = 16$. Figure 3 gives the maximum speedup we can get by optimizing the cache usage of Floyd's algorithm on typical computers that have a two-level cache.

## 4. BLOCKED VERSION OF FLOYD'S ALGORITHM

### 4.1 The Algorithm

We partition the cost adjacency matrix into submatrices of size $B \times B$. $B$ is called the *blocking factor*. Although this is not necessary, we assume, for simplicity, that $B$ divides $n$. Figure 4(a) shows a *blocked* $12 \times 12$ matrix, the blocking factor is $B = 4$. In Figure 4(a) the matrix elements are numbered in row-major order. Figure 4(b) shows the numbering scheme we use for the blocks.

Our blocked version of Floyd's algorithm (Figure 2) will perform $B$ iterations of the outermost loop of Figure 2 on each $B \times B$ block of $A$ before advancing to the

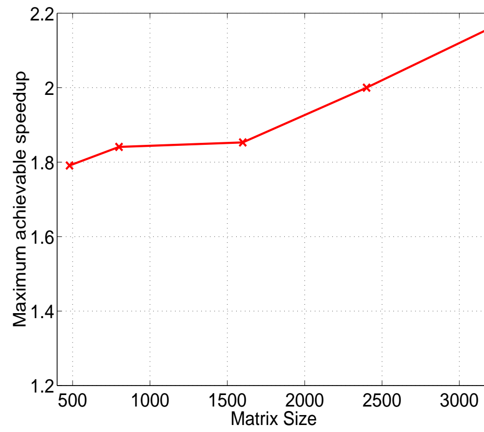



Fig. 3. Maximum achievable speedup for different matrix sizes

| | | |
|---|---|---|
| 1 2 3 4<br>13 14 15 16<br>25 26 27 28<br>37 38 39 40 | 5 6 7 8<br>17 18 19 20<br>29 30 31 32<br>41 42 43 44 | 9 10 11 12<br>21 22 23 24<br>33 34 35 36<br>45 46 47 48 |
| | | |
| | | |

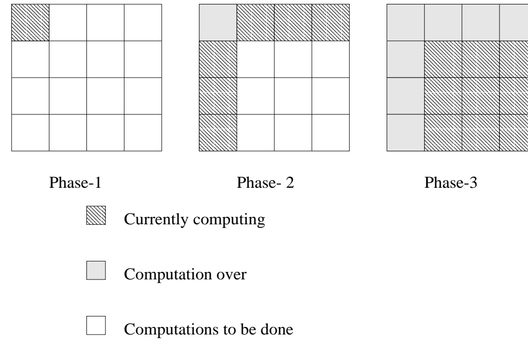
(a)

| | | |
|---|---|---|
| block(1,1) | block(1,2) | block(1,3) |
| block(2,1) | block(2,2) | block(2,3) |
| block(3,1) | block(3,2) | block(3,3) |

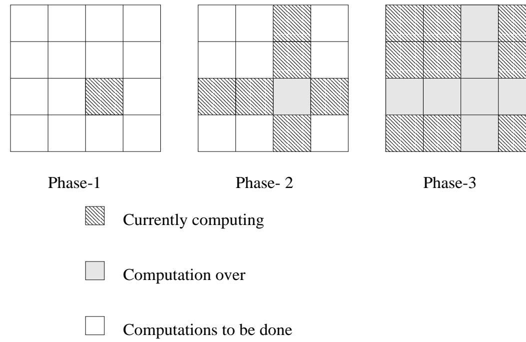
(b)

Fig. 4. Blocked matrix

next $B$ iterations. It is convenient to think of each set of $B$ iterations as divided into three phases. (Note that our implementation does not actually preform the computation in the three phase order described below.) For example, in phase 1 of the first set of $B$ iterations, Equation 2 is used to compute $D^k = A^k$, $1 \leq k \leq B$ for the elements in block (1,1) (see Figure 5(a)). Since these $B$ iterations access only the $A$ elements within block (1,1), we say that block (1,1) is a self-dependent block in the first $B$ iterations.




(a) Phases when (1,1) is the self-dependent block




(b) Phases when block $(t, t)$ is the self-dependent block

Fig. 5. Blocks computed in each phase

In phase 2 of the first $B$ iterations a modified Equation 2 is used to compute $D^k$, $1 \leq k \leq B$ for the remaining blocks $(1, *)$ and $(*, 1)$ that are on the same row or column as the self-dependent block (see Figure 5(a)). For the remaining (1,*) blocks the modified Equation 2 is

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^B(i, k) + D^{k-1}(k, j)\}, k \geq 1 \tag{16}$$

where $D^0(i,j) = A^0(i,j)$. For the remaining (*,1) blocks the modified Equation 2 is

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{k-1}(i,k) + D^B(k,j)\}, k \geq 1 \tag{17}$$

In phase 3 $D^k$, $1 \leq k \leq B$ is computed for the remaining blocks (i.e., for blocks that are not on the same row or column as the self-dependent block). This computation is done using Equation 18.

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^B(i,k) + D^B(k,j)\}, k \geq 1 \tag{18}$$

Phase 3 is followed by the next round of $B$ iterations. These are also done in three phases. This time block (2,2) is the self-dependent block. $D^k$, $B < k \leq 2B$ are computed for the self-dependent block in phase 1 using the equation

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{k-1}(i,k) + D^{k-1}(k,j)\} \tag{19}$$

In phase 2 $D^k$, $B < k \leq 2B$ are computed for the remaining blocks that are on the same row or column as the self-dependent block and in phase 3 $D^k$, $B < k \leq 2B$ is computed for the blocks that are not on the same row or column as the self-dependent block. The phase 2 computation uses the following equation for the (2,*) blocks

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{2B}(i,k) + D^{k-1}(k,j)\} \tag{20}$$

The (*,2) blocks use the following equation

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{k-1}(i,k) + D^{2B}(k,j)\} \tag{21}$$

and the phase 3 blocks use the equation

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{2B}(i,k) + D^{2B}(k,j)\} \tag{22}$$

Figure 5(b) shows the blocks computed in each phase when block $(t,t)$ is the self-dependent block. The following equations are used to compute the $(t,*)$, $(*,t)$, and phase 3 blocks, respectively.

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{tB}(i,k) + D^{k-1}(k,j)\} \tag{23}$$

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{k-1}(i,k) + D^{tB}(k,j)\} \tag{24}$$

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{tB}(i,k) + D^{tB}(k,j)\} \tag{25}$$

Figure 6 gives a high-level description of the blocked version of Floyd's algorithm. You may verify that $D^k(i,j) =$ `D[i][j]` is computed in any iteration of the `for k` loop.

### 4.2 Correctness of Blocked Algorithm

The $D^k(i,j)$ values computed by the blocked algorithm are not necessarily the same as the $A^k(i,j)$ values computed by the unblocked algorithm. For example, when $B = 4$, the unblocked algorithm computes $A^1(4,7) = \min\{A^0(4,7), A^0(4,1) + A^0(1,7)\}$, whereas the blocked algorithm computes $D^1(4,7) = \min\{D^0(4,7), D^4(4,1) + D^0(1,7)\} = \min\{A^0(4,7), D^4(4,1) + A^0(1,7)\}$. Since $D^4(4,1) = A^4(4,1)$ is $\leq A^0(4,1)$, $D^1(4,7) \leq A^1(4,7)$.

```
function BlockedAllPairs(int D, int n, int B)
{// D[i][j] = cost(i,j) initially
 // D[i][j] equals length of shortest i to j path on termination
  for (round = 1; round <= n / B; round++)
  {
     // self-dependent block
     for (k = (round - 1) * B + 1; k <= round * B; k++)
        for (all i and j in the self-dependent block)
           D[i][j] = min(D[i][j], D[i][k] + D[k][j]);

     // remaining blocks
     do the following for the remaining blocks, one block at a time and
           in the order: phase 2 blocks to the right of block (r,r),
           phase 2 blocks above (r,r), phase 3 blocks above and right of (r,r),
           phase 2 blocks left of (r,r), phase 3 blocks above and left of (r,r),
           phase 2 blocks below (r,r), phase 3 blocks below and left of (r,r),
           phase 3 blocks below and right of (r,r)
        for (k = (round - 1) * B + 1; k <= round * B; k++)
           for (all i and j in the current block)
              D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
  }
}
```

Fig. 6. Blocked version of Floyd's shortest-paths algorithm

To establish the correctness of the blocked algorithm we must show that $D^n(i,j) = A^n(i,j)$ for all $i$ and $j$. That is, even though $D^k(i,j)$ and $A^k(i,j)$ may not be equal for $k < n$, the values agree in the end when $k = n$. Actually we will show that $A$ and $D$ agree at the end of each set of $B$ iterations (i.e., at the end of each iteration of the outermost loop of Figure 6). That is, $D^k(i,j) = A^k(i,j)$ for all $i$ and $j$ whenever $k$ is a multiple of $B$. Hence $D^n(i,j) = A^n(i,j)$ for all $i$ and $j$.

Let $k = qB$. We use induction on $q$ to show that $D^k(i,j) = A^k(i,j)$ for all $i$ and $j$ for $0 \le q \le n/B$. In the induction base we have $q = 0$. So $k = 0$ and $D^0(i,j) = A^0(i,j) =$ cost of edge from $i$ to $j$.

In the induction hypothesis we assume that $D^k(i,j) = A^k(i,j)$ for all $i$ and $j$ when $k = sB$ for some arbitrary nonnegative integer $s$.

In the induction step we show that $D^k(i,j) = A^k(i,j)$ for all $i$ and $j$ when $k = (s+1)B$. First we show this for $i$ and $j$ in the self-dependent block.

$$\begin{aligned} D^{sB+1}(i,j) &= \min\{D^{sB}(i,j), D^{sB}(i,sB+1) + D^{sB}(sB+1,j)\} \\ &= \min\{A^{sB}(i,j), A^{sB}(i,sB+1) + A^{sB}(sB+1,j)\} \\ &= A^{sB+1}(i,j) \end{aligned}$$

Similarly $D^{sB+2}(i,j) = A^{sB+2}(i,j)$, $D^{sB+3}(i,j) = A^{sB+3}(i,j)$ and so on. Hence $D^{(s+1)B}(i,j) = A^{(s+1)B}(i,j)$ for all elements in the self-dependent block.

Next consider any element that is in a block that is in the same row as the self dependent block. For such an element,

$$D^{sB+1}(i,j) = \min\{D^{sB}(i,j), D^{(s+1)B}(i,sB+1) + D^{sB}(sB+1,j)\} \tag{26}$$

From the induction hypothesis and the result just proved for the self-dependent

block, Equation 26 becomes

$$D^{sB+1}(i,j) = \min\{A^{sB}(i,j), A^{(s+1)B}(i,sB+1) + A^{sB}(sB+1,j)\} \qquad (27)$$

In the next iteration, we compute

$$\begin{aligned}
D^{sB+2}(i,j) &= \min\{D^{sB+1}(i,j), \\
&\qquad D^{(s+1)B}(i,sB+2) + D^{sB+1}(sB+2,j)\} \\
&= \min\{A^{sB}(i,j), \\
&\qquad A^{(s+1)B}(i,sB+1) + A^{sB}(sB+1,j), \\
&\qquad D^{(s+1)B}(i,sB+2) + D^{sB+1}(sB+2,j)\} \\
&= \min\{A^{sB}(i,j), \\
&\qquad A^{(s+1)B}(i,sB+1) + A^{sB}(sB+1,j), \\
&\qquad A^{(s+1)B}(i,sB+2) + D^{sB+1}(sB+2,j)\} \\
&= \min\{A^{sB}(i,j), \\
&\qquad A^{(s+1)B}(i,sB+1) + A^{sB}(sB+1,j), \\
&\qquad A^{(s+1)B}(i,sB+2) + A^{sB}(sB+2,j), \\
&\qquad A^{(s+1)B}(i,sB+2) + A^{(s+1)B}(sB+2,sB+1) + A^{sB}(sB+1,j)\}
\end{aligned}$$

Consider the second and fourth terms in the last min expression. $A^{(s+1)B}(i,sB+1)$ is the length of the shortest path from vertex $i$ to vertex $sB+1$ when intermediate vertex indices can be at most $(s+1)B$ and $A^{(s+1)B}(i,sB+2)+A^{(s+1)B}(sB+2,sB+1)$ is the length of the shortest path from vertex $i$ to vertex $sB+1$ that passes through vertex $sB+2$ (which is $\leq (s+1)B$) but passes through no vertex whose index is more than $(s+1)B$. Hence $A^{(s+1)B}(i,sB+1) \leq A^{(s+1)B}(i,sB+2)+A^{(s+1)B}(sB+2,sB+1)$. Therefore, $\min\{A^{(s+1)B}(i,sB+1)+A^{sB}(sB+1,j), A^{(s+1)B}(i,sB+2)+A^{(s+1)B}(sB+2,sB+1)+A^{sB}(sB+1,j)\} = A^{(s+1)B}(i,sB+1)+A^{sB}(sB+1,j)$. So

$$\begin{aligned}
D^{sB+2}(i,j) &= \min\{A^{sB}(i,j), \\
&\qquad A^{(s+1)B}(i,sB+1) + A^{sB}(sB+1,j), \\
&\qquad A^{(s+1)B}(i,sB+2) + A^{sB}(sB+2,j)\}
\end{aligned}$$

In a similar manner we may show that

$$\begin{aligned}
D^{sB+3}(i,j) &= \min\{A^{sB}(i,j), \\
&\qquad A^{(s+1)B}(i,sB+1) + A^{sB}(sB+1,j), \\
&\qquad A^{(s+1)B}(i,sB+2) + A^{sB}(sB+2,j), \\
&\qquad A^{(s+1)B}(i,sB+3) + A^{sB}(sB+3,j)\}
\end{aligned}$$

and

$$\begin{aligned}
D^{(s+1)B}(i,j) &= \min\{A^{sB}(i,j), \\
&\qquad A^{(s+1)B}(i,sB+1) + A^{sB}(sB+1,j),
\end{aligned}$$

$$A^{(s+1)B}(i, sB+2) + A^{sB}(sB+2, j), \cdots,$$
$$A^{(s+1)B}(i, (s+1)B) + A^{sB}((s+1)B, j)\} \qquad (28)$$

Now consider the shortest path $P^{(s+1)B}(i,j)$ from $i$ to $j$ with intermediate vertex indices at most $(s+1)B$. The length of $P^{(s+1)B}(i,j)$ is $A^{(s+1)B}(i,j)$. $P^{(s+1)B}(i,j)$ may or may not contain any intermediate vertex whose index is greater than $sB$ but less than $(s+1)B$. If $P^{(s+1)B}(i,j)$ does not contain any such vertex, then the length of $P^{(s+1)B}(i,j)$ is $A^{sB}(i,j)$. Suppose that $P^{(s+1)B}(i,j)$ contains one or more intermediate vertices whose index is greater than $sB$ and less than $(s+1)B$. Let $x$ be the last intermediate vertex on this path such that $sB < x \leq (s+1)B$. The length of $P^{(s+1)B}(i,j)$ is $A^{(s+1)B}(i,x) + A^{sB}(x,j)$. Although we do not know the value of $x$, we may assert that the path length is $\min\{A^{(s+1)B}(i,r)+A^{sB}(r,j) | sB < r \leq (s+1)B\}$. Therefore $D^{(s+1)B}(i,j) = A^{(s+1)B}(i,j)$.

The proof for the remaining cases is similar. We omit the proof.

### 4.3 Optimal Blocking Factor

In computing the optimal blocking factor, we seek to minimize L1 cache misses. We focus on L1 misses alone because the L2 cache is typically quite large (4MB on the SUN Enterpise 4000/5000) and therefore able to accomodate fairly large matrices. Consequently, for matrix sizes of up to about 1000, the only L2 cache misses that occur are essential (or unavoidable) ones. The L1 cache, on the other hand, is quite small (16KB on our SUN Enterprise 4000/5000) resulting in a very large number of cache misses.

When computing the $D$ values in a block during any round (i.e., an iteration of the outermost loop) of function `BoundedAllPairs`, at most three blocks are active. The computation for the self-dependent block accesses elements only in the self-dependent block. So during the self-dependent block computation only 1 block is active. The computation for a block $R$ that is on the same row or column as the self-dependent block acceses elements in $R$ as well as elements in the self-dependent block. Therefore, 2 blocks are active during the computation for $R$. For a block $R$ that is not on the same row or column as the self dependent block, `BlockedAllPairs` accesses elements from 3 blocks—block $R$, the block that is in the same row as the self-dependent block and the same column as $R$, and the block that is in the same column as the self-dependent block and in the same row as $R$. Therefore, L1 cache misses are minimized by choosing the largest block size $B$ such that 3 block loads of the array $D$ fit into L1 cache. Suppose that the elements of $D$ are 4-byte integers and that our L1 cache capacity is $C$ bytes and that each L1 cache line is $S$ bytes. We must choose $B$ to be the largest integer such that $3B^2 * 4 \leq C$ (equivalently, $B \leq \sqrt{C/12}$) and $B$ is a multiple of $S/4$. The second requirement is necessary as the smallest unit of data brought into L1 cache is $S$ bytes and these $S$ bytes are contiguous bytes of memory.

For the Sun Ultra Enterprise 4000/5000 $C = 16K$ and $S = 32$. Therefore, the blocking factor should be the largest integer that is $\leq \sqrt{C/12} = 37$ and is a multiple of $32/4 = 8$. That is, we should use $B = 32$ as the blocking factor. For the SGI O2 $C = 32K$ and $S = 32$. The optimal blocking factor for the SGI O2 is the largest integer that is $\leq \sqrt{C/12} = 52$ and is a multiple of $32/4 = 8$. This optimal blocking factor is 48.

### 4.4 Crude Cache-Miss Analysis

The basic computation step in Floyd's shortest-paths algorithm is

$$A(i,j) = \min\{A(i,j), A(i,k) + A(k,j)\} \tag{29}$$

Assume that the optimized code for this step makes a read access of $A(i,j)$, $A(i,k)$, and $A(k,j)$ (in this order), followed by a write access of $A(i,j)$. Even though this write access is made only when $A(i,j) > A(i,k) + A(k,j)$, we shall assume that the write access is made each time the basic step (Equation 29) is executed. The analysis assumes a write through cache with a write buffer. That is, writes are done to L1 cache as well as to L2 cache; the write to L2 cache is done via a write buffer so that the CPU is not blocked waiting for the write to L2 cache to complete. Although the use of a write buffer does not guarantee that the CPU will not get blocked by a write, our analysis assumes that there is no CPU blocking caused by writes. For a write miss, we assume that the cache has a no allocate policy. That is, the data is written to L2 cache (via the write buffer) and the modified L2 cache block is not loaded into the L1 cache. These assumptions on the handling of writes are valid for the computers we use in our subsequent experimental evaluation. Since our analysis will count L1 misses only, the assumption of a no allocate policy means that we can ignore the writes from the analysis.

Performing an accurate analysis of the cache miss count is rather complicated, because the cache conflicts that take place in both Floyd's algorithm and ours are highly dependent on the value of $n$, the size of a cache line, and the number of cache lines. To see why this is so, let $S1$ be the number of lines in the L1 cache of our computer, and let $N1$ be the number of matrix elements that fit into one L1 cache line. Consider the computation of Equation 29 using Floyd's algorithm (Figure 2). If $\lceil (in+j)/N1 \rceil \bmod S1 = \lceil (n+j)/N1 \rceil \bmod S1$, then every iteration of the innermost loop results in an L1 cache miss when an attempt is made to read $A(i,j)$ (unless $A(i,j)$ is in the cache because of an earlier computation), and another miss when we read $A(k,j)$ (because $A(i,j)$ and $A(k,j)$ map into the same L1 cache line). This happens, for example, when $n$ is a power of 2. The situation is even more complex with respect to determining exactly when an access made by our blocked algorithm will result in a cache miss.

Instead of a cumbersome exact analysis, we do a rather crude but simple analysis. First consider Floyd's algorithm. For any fixed $k$, every $N1$ iterations of the innermost **for** loop results in a cache miss when $A(i,j)$ (Equation 29) is accessed. Ignoring the cache misses resulting from the accesses of $A(i,k)$ and $A(k,j)$, we obtain $n^2/N1$ as the cache miss count for each value of $k$. When the algorithm moves from one $k$ to the next, we assume that the number of cache misses is not reduced by any $A$ values left in the cache from the previous iteration of the outermost loop. Therefore, the total number of cache misses is approximately $n^3/N1$.

For our blocked algorithm, we assume that the elements in the up to 3 blocks active at any time are generally not in cache conflict. The number of cache misses to bring these up to 3 blocks into L1 cache is at most $3B^2/N1$. The number of iteration of the blocked algorithm is $(n/B)*(n^2/B^2) = n^3/B^3$. So, the total number of cache misses is bounded by $3n^3/(B*N1)$. Under the assumptions of our crude analysis, the blocked algorithm requires a factor of $O(B)$ fewer cache misses than required by Floyd's algorithm.

The actual reduction in the number of cache misses will depend on the particular value of $n$ as this value determines the cache conflicts that occur when the blocked algorithm works with its up to 3 blocks.

## 5. EXPERIMENTAL RESULTS

The speedup of our blocked shortest paths algorithm (Figure 6) relative to the unblocked algorithm (Figure 2) was measured by programming the two algorithms in C++ (the g++ compiler with optimization option o5 was used) and running the two programs on on a Sun Ultra Enterprise 4000/5000 and an SGI O2. Both programs were compiled using the highest-level of compiler optimization possible. For the SUN Enterprise experiments, we used matrix sizes of $n = 480$, 800, 1600, 2400, and 3200. The SGI experiments used $n = 240$, 320, 480, 640, 720, 960, and 1200. The actual data used to populate the inital distance matrix isn't too important as this data affects only the number of updates of `A` values (see Figure 2, an `A` value is updated only when `A[i][j]` is greater than `A[i][k] + A[k][j]`). As mentioned in Section 4.4, for computers that have a write through cache with a write buffer, write misses do not have a very significant impact on run time. Therefore, we expect the run time of Floyd's algorithm as well as that of our blocked version to be relatively insensitive to the actual values that poplulate the initial `A` matrix. For our tests, the initial `A` matrices represented undirected connected graphs with an edge density of 80% (i.e., 80% of the possible $n(n-1)/2$ edges of an $n$ vertex undirected graph are present). The edge weights (lengths) were randomly generated integers in the range [1, 1000]. For each graph size $n$, the measured statistics were averaged over 10 instances.

We first present the results for the SUN Ultra Enterprise. Figure 7 gives the measured speedups for different blocking factors and different $n$. In this, and other figures, when the shown block size does not divide $n$, a nearby block size that divides $n$ was used. For example, when $n = 3200$, a block size of 50 is used in place of the block size of 48. As predicted by our analysis, the optimal blocking factor is 32 for all $n$ (note that our experiments were limited to $n$ values that are divisible by 32).

Figure 8 compares the speedup obtained by `BlockedAllPairs` and the maximum speedup possible by optimizing cache utilization. The curve for maximum possible speedup is that of Figure 3.

The speedup obtained by `BlockedAllPairs` is fairly close to the maximum possible. One reason we do not achieve the predicted maximum speedup is that the total instruction count for `BlockedAllPairs` is more than that for `AllPairs`. Recall that in determining the maximum speedup curve of Figure 3 we assumed that the instruction count for the cache optimized algorithm is the same as that of `AllPairs`.

Figure 9 gives the *L1 misses per instruction* for the unblocked and blocked versions of Floyd's algorithm. The data for this figure were obtained using the cache simulator Shade. As expected the blocked code shows better cache utilization.

The L1 and L2 caches on an SGI O2 computer are both 2-way set associative. The L1 cache is 32KB with a line size of 32 bytes and the L2 cache is 1MB with a line size of 64 bytes. Figure 10 shows the speedup obtained by the blocked algorithm on an SGI O2. Except for one anomaly, maximum speedup is obtained when the

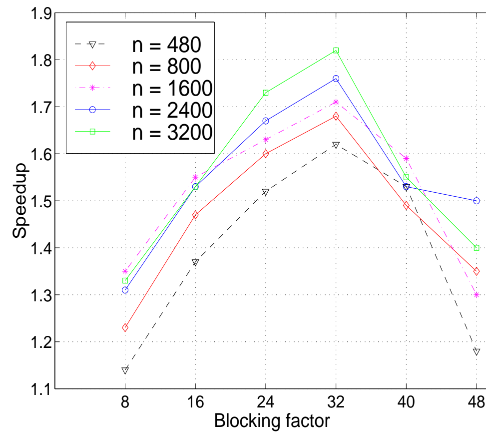



Fig. 7. Speedup of `BlockedAllPairs` on a Sun Ultra Enterprise

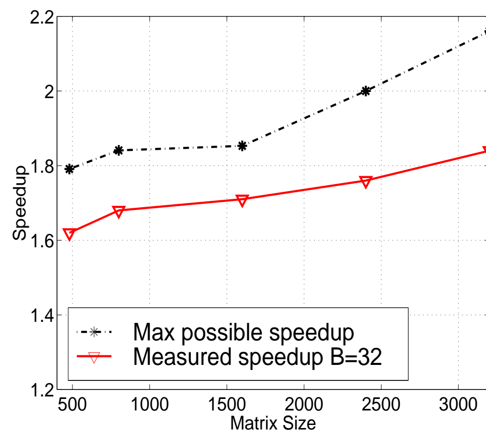



Fig. 8. Measured and maximum possible speedup

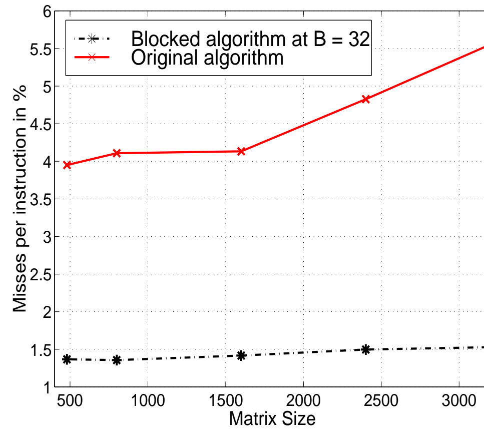



Fig. 9. Misses per instruction for unblocked and blocked algorithms

blocking factor is the predicted optimal factor of 48.

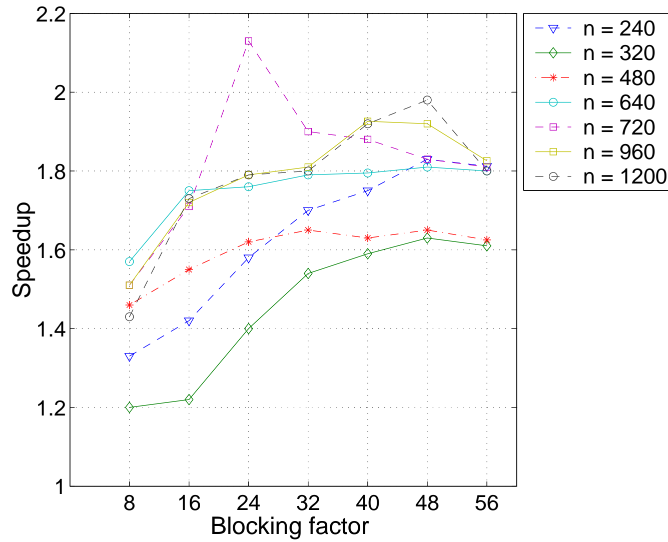



Fig. 10. Speedup obtained by `BlockedAllPairs` on an SGI O2

## 6. CONCLUSION

We have developed a blocked version of Floyd's all-pairs shortest-paths algorithm. Experimental results show that the blocked version obtains speedups close to the maximum possible for a cache optimized version of Floyd's algorithm.

REFERENCES

[1] W. AbuSufah, D. J. Kuck, and D. H. Lawrie. Automatic program transformation for virtual memory computers. In *Proc. of the 1979 National Computer Conference*, 969–974, New York, 1979.

[2] A. Aggarwal, K. Chandra, and M. Snir. A model for hierarchical memory. In *The 19th Annual ACM Symposium on Theory of Computing*, 305–314, NewYork, 1987.

[3] A. Aggarwal, K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *The 28th Annual IEEE Symposium on Foundations of Computer Science*, 204–216, LosAngeles, CA, 1987.

[4] A. Aggarwal, M. Horowitz, and Hennessey. An analytical cache model. *The ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

[5] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3):72–109, 1994.

[6] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, New York, 53–65, 1990.

[7] N. Eiron, M. Rodeh, and I. Steinwarts. Matrix multiplication: a case study of enhanced data cache utilization. *ACM JEA*, 4, Article 3, 1999.

[8] D. Gannon and W. Jalby. The influence of memory hierarchy on algorithm organization: Programming FFTs on a vector multiprocessor. *In The Characteristics of Parallel Algorithms, MIT Press, Cambridge*, 1987.

[9] G. H. Golub and C. F. Van Loan. Matrix Computations. *Johns Hopkins University Press, Baltimore*, 1989.

[10] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms*. Computer Science Press, New York, 1998.

[11] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *ACM*, 26:63–74, 1991.

[12] A. LaMarca and R. E. Ladner. The influences of caches on the performance of heaps. *The ACM Journal of Experimental Algorithms*, 1(4), 1996.

[13] A. LaMarca and R. E. Ladner. The influences of caches on the performance of sorting. *Journal of Algorithms*, 31, 66-104, 1999.

[14] R. Ladner, J. Fix, and A. LaMarca. The cache performance analysis of traversals and random accesses. In *The ACM-SIAM Symposium on Discrete Algorithms*, 613–622, 1999.

[15] M. Martonosi, A. Gupta, and T. Anderson. Memspy: analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1–12, Newport, Rhode Island, 1992.

[16] A. C. McKeller and E. G. Coffman. The organization of matrices and matrix operations in a paged multiprogramming environment. *CACM*, 12(3):153–165, 1969.

[17] D. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Analysis*. Morgan Kaufmann, San Mateo, CA, 1996.

[18] N. Rahman and R. Raman. Analysing cache effects in distribution sorting. *ACM JEA*, 5, Article 14, 2000.

[19] J. P. Singh, H. S. Stone, and D. F. Thiebaut. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Transactions on Computers*, 41(7):811–825, 1992.

[20] Sun Microsytems. Introduction to Shade. Manual, Sun Microsytems, Mountain View, CA, 1998.

[21] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the 1994 ACM SIGMETRICS: Conference on Measurement and Modelling of Computer Systems*, 261–271, Nashville, Tennessee, 1994.

[22] O. Temam, C. Fricker, and W. Jalby. Influence of cross-interference on blocked loops: A case study with matrix-vector multiply . *The ACM Transactions on Programming Languages and Systems*, 17(4):561–575, 1994.

[23] H. Wen and J. L. Baer. Efficient trace driven simulation methods for cache performance analysis. *The ACM Transactions on Computer Systems*, 9(3):222–241, 1991.

[24] M. E. Wolf and M. S. Lam. A data locality optimizing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, 30–44, Toronto, Ontario, Canada, 1991.

[25] L. Xiao, X. Zhang, and S. Kubricht. Improving memory performance of sorting algorithms, *ACM JEA*, 5, Article 3, 2000.