

# Algorithms to Take Advantage of Hardware Prefetching

Shen Pan\*      Cary Cherng†      Kevin Dick‡      Richard E. Ladner§

## Abstract

Cache-oblivious and cache-aware algorithms have been developed to minimize cache misses. Some of the newest processors have hardware prefetching where cache misses are avoided by predicting ahead of time what memory will be needed in the future and bringing that memory into the cache before it is used. It is shown that hardware prefetching permits the standard Floyd-Warshall algorithm for all-pairs shortest paths to outperform cache-oblivious and cache-aware algorithms. A simple improvement to the standard simple dynamic programming algorithm yields an algorithm that takes advantage of prefetching, and outperforms cache-oblivious and cache-aware algorithms. Finally, it is shown that variants of standard FFT algorithms exhibit good prefetching performance.

## 1 Introduction.

The memory subsystem on modern computers is ubiquitously structured in a hierarchy with registers in the lowest level followed by the L1 cache, L2 cache, main memory and external memory such as hard disks, with memory access time increasing quickly from lower levels to higher levels. For the sake of discussion we consider a two-level model that consists of a cache of size  $M$  and an arbitrarily large main memory partitioned into blocks of size  $B$ . If the byte is not stored in the cache, the entire memory block where it resides is brought into the cache, and we call this a *cache miss*. The *I/O complexity* of an algorithm therefore becomes the number of blocks transferred upon cache misses between these two levels. *Cache Oblivious* algorithms are algorithms that do not use knowledge of  $M$  and  $B$ , yet still have good cache performance. On the other hand, *Cache Aware* algorithms do use knowledge of  $M$  and  $B$  of the host machine to optimize their cache performance. Together,

they are called *Cache Efficient* algorithms.

Many cache efficient algorithms for problems have been developed that have superior performance to standard algorithms for the same problems. However, a recent development in processor design, hardware prefetching, raises the question as to whether some of these custom cache efficient algorithms are always needed to reduce cache misses. With hardware prefetching, cache misses are avoided by predicting ahead of time what data in memory will be needed in the future and bringing that data into the cache before it is used. In this paper we explore this question and discover that hardware prefetching can be exploited to yield fast algorithms for the all-pairs shortest paths problem, simple dynamic programming, and the Fast Fourier Transform (FFT).

## 2 Hardware Prefetcher in the Pentium 4.

In the Pentium 4 processor, associated with the L2 cache is a hardware prefetcher [7] that monitors data access patterns and prefetches data automatically into the L2 cache. It attempts to stay 256 bytes ahead of the current data access locations. This prefetcher remembers the history of cache misses to detect concurrent, independent streams of data that it tries to prefetch ahead of use in the program. It follows one stream per 4KB page (either load or store) and can prefetch up to 8 simultaneous independent streams from eight different 4KB regions. The hardware prefetcher also has a few weaknesses. First of all, it requires rather regular memory access patterns. Moreover, start-up penalty applies before the hardware prefetcher triggers, and there might be unnecessary fetches after the end of an array is reached. For short arrays this overhead can reduce the effectiveness of the hardware prefetcher.

To understand the range and efficiency of the prefetcher, we timed sequences of array accesses with and without the prefetcher enabled. Prefetcher activation is controlled by setting bits 9 and 19 of the IA32\_MISC\_ENABLE model-specific register. More information can be found in Appendix B of Volume 3B of the Intel 64 and IA-32 Architectures Software Developer's Manual [6].

This study was performed on a machine running Linux 2.6.16-16 using a 3.4 GHz Pentium 4 processor

\*Amazon, 605 5th Ave. South, Seattle, WA 98104. shenpan@amazon.com

†Google Kirkland, 720 4th Ave, Suite 400, Kirkland WA 98003. cary@google.com.

‡Computer Science Department, California Institute of Technology, 1200 E. California Boulevard, MC 256-80, Pasadena, CA 91125. kdick@caltech.edu.

§Department of Computer Science and Engineering, Box 352350, University of Washington, Seattle, WA 98195. ladner@cs.washington.edu.

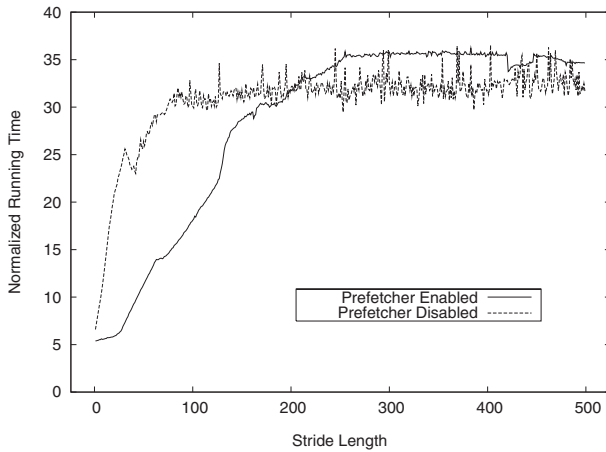


Figure 1: Normalized running time in seconds of sequential array accesses.

with an 8 KB L1 cache (8-way set associative with 64 B lines) and a 2 MB L2 cache (8-way set associative with 64 B lines). It had 1 GB of main memory and used g++ version 4.1.1 20060525 (Red Hat 4.1.1-1 with optimization -o). The program first allocates a large array and then traverses it ten times, each time reading from then writing to every  $n$ -th byte, where  $n$  is the stride length. The results, shown in Figure 1, use an array of forty million bytes with a stride varying from one to five hundred. The normalized time is reported, meaning the stated values are proportional to the time needed per array access. The given measurements are the medians of seven trials. Running this experiment for other large array sizes gave similar results.

With the prefetcher disabled, we expect the normalized time to depend heavily on the number of cache misses. The L1 and L2 caches use blocks of 64 bytes, so for strides of 63 and under accesses to elements already brought into the cache by previous operations come at a low cost. When the stride length is at least 64, we are effectively measuring the time taken (without normalization) for  $10 * (40 * 10^6) / n$  cache misses.

When the prefetcher is enabled, when  $n \leq 256$ , elements of the array will be brought into the L2 cache. This gives us some improvement, since many elements that would have been drawn from the main memory are instead pulled from the L2 cache. However, the hardware prefetcher requires a few initial misses before it can start prefetching, and it only prefetches from main memory into the L2 cache [7]. Furthermore, the overhead required by the prefetcher actually slows down the array accesses for large strides which are out of range of the prefetcher. These results suggest that hardware

prefetching can give significant speedup with sequential accesses to memory that are close together, but that prefetching can actually slow down accesses that are spaced far apart.

### 3 Cache Efficient Algorithms.

Expert programmers have known for many years that reducing the number of cache misses can significantly improve the running time of programs. Much effort has been put into designing the cache efficient versions of various dynamic programming algorithms. These algorithms work by reducing the constant factor in the complexity incurred by the cache misses. One major approach to improving the performance of the cache is to design cache-oblivious algorithms.

The cache-oblivious approach is explored by Frigo *et al.* in [4], which discusses the cache performance of cache-oblivious algorithms for matrix transpose, FFT and sorting. Park *et al.* [10] presented a cache-oblivious implementation of the Floyd-Warshall algorithm for the fundamental graph problem of all-pairs shortest paths by relaxing some dependencies in the iterative version. The cache-oblivious algorithm runs roughly 7 times faster than the Floyd-Warshall algorithm on a Pentium 3 machine. Chowdhury *et al.* [2] gave a new cache-oblivious framework called the *Gaussian Elimination Paradigm (GEP)* for Gaussian elimination without pivoting that also gives cache-oblivious algorithms for Floyd-Warshall all-pairs shortest paths in graphs and matrix multiplication, among other problems. New cache-oblivious and cache-aware algorithms for simple dynamic programming based on Valiant's context-free language recognition algorithm are designed, implemented, analyzed and empirically evaluated with timing studies and cache simulations by Cherng *et al.* [1].

A major technique in designing cache aware algorithms is *blocking*, that is partitioning the problem into cache size subproblems and solving each subproblem while its data is in the cache. In the cache-oblivious techniques quite often the problem partitions itself naturally into smaller subproblems in a recursive way. Unfortunately, if the recursion is continued all the way to bottom, then there is a lot of overhead from the recursion. A blocking technique that stops the recursion when the subproblem size reaches the size of the cache (say L2 cache), then solving the problem using a standard iterative approach, often yields a significantly faster program. On the negative side, the blocking technique only works if the cache size can be communicated to the program.

#### 4 All-Pairs Shortest Paths.

In the all-pairs shortest paths problem we are given a directed graph  $G$  with vertices indexed  $\{1, 2, \dots, n\}$  and for each directed edge  $(i, j)$  an associated non-negative cost  $c(i, j)$ . For each  $i$  and  $j$  we wish to find the lowest cost of all paths from  $i$  to  $j$ , where the cost of a path is the sum of cost of the edges on the path. The Floyd-Warshall algorithm (shown in Algorithm 1) is the standard iterative dynamic programming solution to the all-pairs shortest paths problem [3]. It runs in  $O(n^3)$  time and works by looking at paths with successively more and more possible interior vertices until all vertices are exhausted. The work is divided into  $n$  iterations. Initially,  $X[i, j] = c(i, j)$  if  $(i, j)$  is an edge,  $X[i, j] = \infty$  if  $(i, j)$  is not an edge, and  $X[i, i] = 0$ .

```

for  $k = 1$  to  $n$  do
  for  $i = 1$  to  $n$  do
    for  $j = 1$  to  $n$  do
       $X[i, j] := \min(X[i, j], X[i, k] + X[k, j])$ 
    end for
  end for
end for

```

Algorithm 1: The Floyd-Warshall Algorithm

**4.1 Cache Efficient Algorithms for All-Pairs Shortest Paths.** The two cache-oblivious algorithms described below are only defined for problems of size  $2^k$  for some  $k$ . To solve problems that are not the size a power of two the array can be padded appropriately.

The *Gaussian Elimination Paradigm* or *GEP*, introduced in [2], is a general cache-oblivious framework for problems. When specialized to the all-pairs shortest paths problem we obtain the recursive formulation described in Algorithm 2. If the input array  $X$  has size larger than  $1 \times 1$  then it is subdivided into four equal size matrices:

$$X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix}$$

In the algorithm, when the base case is called  $k_1 = k_2$  and the array  $X$  is  $1 \times 1$  and  $(i_1, j_1)$  is the index of the one element in the array. All-pairs shortest paths is then solved by calling  $F(X, 1, n)$ .

Another cache-oblivious technique is derived from the reduction of path problems to matrix multiplication [5, 9]. For this formulation of matrix multiplication, addition is the min operation and multiplication is the  $+$  operation. The recursive step of the *Matrix Multiply Paradigm* or *MMP* algorithm can be described elegantly by Algorithm 3. If  $X$  is not  $1 \times 1$  then the result  $X^*$  is computed recursively by dividing  $X$  into submatrices of

```

Procedure  $F(X, k_1, k_2)$ 
if  $k_1 = k_2$  then
   $X[i_1, j_1] := \min(X[i_1, j_1], X[i_1, k_1] + X[k_1, j_1])$ 
else
   $k_m := \lfloor \frac{k_1 + k_2}{2} \rfloor$ 
   $F(X_{11}, k_1, k_m)$ 
   $F(X_{12}, k_1, k_m)$ 
   $F(X_{21}, k_1, k_m)$ 
   $F(X_{22}, k_1, k_m)$ 
   $F(X_{22}, k_m + 1, k_2)$ 
   $F(X_{21}, k_m + 1, k_2)$ 
   $F(X_{12}, k_m + 1, k_2)$ 
   $F(X_{11}, k_m + 1, k_2)$ 
end if

```

Algorithm 2: GEP Algorithm

half the dimension as in the case of the GEP algorithm. The matrix multiply and accumulate operation is also done recursively using divide-and-conquer.

$$\begin{aligned}
X_{22} &:= X_{22}^* \\
X_{12} &:= X_{12} \cdot X_{22} \\
X_{11} &:= X_{11} + X_{12} \cdot X_{21} \\
X_{11} &:= X_{11}^* \\
X_{12} &:= X_{11} \cdot X_{12} \\
X_{21} &:= X_{22} \cdot X_{21} \\
X_{21} &:= X_{21} \cdot X_{11} \\
X_{22} &:= X_{22} + X_{21} \cdot X_{12}
\end{aligned}$$

Algorithm 3: MMP Algorithm

Cache-aware algorithms for the all-pairs shortest paths algorithm can be defined using *blocked* versions of the cache-oblivious algorithms. The blocked GEP algorithm has a parameter  $S$  such that if the subproblem size  $n \leq S$ , then the cost submatrix is computed using the Floyd-Warshall algorithm directly. The blocked MMP algorithm has two parameters  $S$  and  $M$ . The parameter  $S$  is such that if the subproblem size  $n \leq S$  then  $X^*$  is computed using the standard iterative dynamic program (Floyd-Warshall's Algorithm). The parameter  $M$  is such that if  $n \leq M$  then the matrix multiply and accumulate operations are done in the standard way, not with recursive divide-and-conquer.

**4.2 Experimental Results for All-Pairs Shortest Paths.** We implemented the Floyd-Warshall algorithm, the GEP algorithm, the MMP algorithm, the

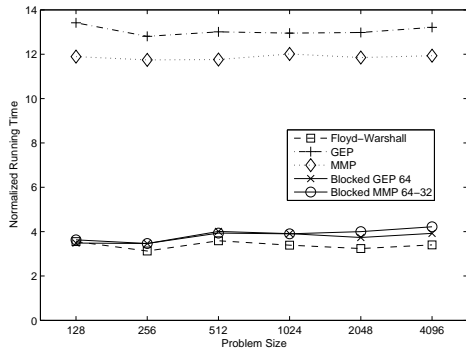


Figure 2: Normalized running time in nanoseconds of the five algorithms for the All-Pairs Shortest Paths problem on Pentium 4.

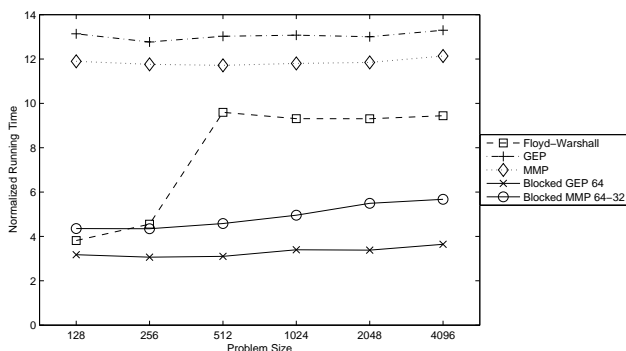


Figure 3: Normalized running time in nanoseconds of the five algorithms for the All-Pairs Shortest Paths problem on Pentium 4 with the hardware prefetcher turned off.

Blocked GEP algorithm and the Blocked MMP algorithm and conducted various running time experiments. In our implementation we chose to store the matrix, which used 4 Byte integers chosen randomly, in row-major order, that is, the rows of the matrix are stored in linear memory by storing row 1, then row 2, and so on. These experiments were run under Red Hat Fedora Core 4 on a 2.8 GHz Pentium 4 with 8 KB L1 data cache (4-way associative with 64 B lines) and 512 KB L2 data cache (8-way associative with 64 B lines). The machine on where the processor resides has 4 GB of main memory. All algorithms were implemented in C++. The compiler used was g++ 4.0.2 20051125 (Red Hat 4.0.2-8, with optimization -O3). In our studies of the all-pairs shortest paths algorithms the normalized time is the average of ten experiments divided by  $n^3$ .

Figure 2 shows the running time results for the five algorithms on the Pentium 4. The best block size for the

Blocked GEP algorithm ( $S = 64$ ) and the Blocked MMP algorithm ( $S = 64, M = 32$ ) are determined experimentally on a problem size of 2048. In contrast to results from prior studies [10], the Floyd-Warshall algorithm clearly out performs all the other algorithms. This is certainly an unexpected result because the Floyd-Warshall algorithm does not have the strong temporal locality exhibited by the cache-oblivious and cache-aware algorithms. Therefore, the Pentium 4 must have something that dramatically changes the performance characteristics of the Floyd-Warshall algorithm. Indeed, the Pentium 4 has hardware prefetching, that appears to obviate the need for special algorithms to help cache performance. Figure 3 shows the running time for the five algorithms on the same Pentium 4 machine, only with the hardware prefetcher turned off. Without hardware prefetching, the Floyd-Warshall algorithm is less than half as fast as the best cache-aware algorithm, the blocked GEP 64. This is a result more consistent with the prior results [10]. Examining the Floyd-Warshall Algorithm (Algorithm 1) closely shows that the inner loop accesses two rows, the  $i$ -th and  $k$ -th, simultaneously. Thus, we have two access streams with a stride of 4 bytes each, which is very amenable to hardware prefetching.

## 5 Simple Dynamic Programming.

Another form of dynamic programming is called *Simple Dynamic Programming problems* in [1]. Input elements  $x_1, \dots, x_n$  of a simple dynamic program of size  $n$  come from a set  $X$  which is the domain of a *non-associative semi-ring*  $(U, +, \cdot, 0)$ , where  $+$  is an associative  $(x + (y + z) = (x + y) + z)$ , commutative  $(x + y = y + x)$ , idempotent  $(x + x = x)$ , binary operator and  $\cdot$  is a nonassociative, noncommutative, binary operator. The value 0 is  $+$ -identity  $(x + 0 = x)$  and  $\cdot$ -annihilator  $(x \cdot 0 = 0 \cdot x = 0)$ . Finally, the operators satisfy the distributive laws  $(x \cdot (y + z) = x \cdot y + x \cdot z)$  and  $(y + z) \cdot x = y \cdot x + z \cdot x$ . The objective is to compute the sum ( $+$ ) of all ways to generate the product ( $\cdot$ ) of  $x_1, \dots, x_n$  in this order and under all possible groupings for the product.

The simple dynamic programming problem can be solved in  $O(n^3)$  time by Algorithm 4, which is the Cocke-Kasami-Younger (CKY) algorithm [8, 13], the standard iterative algorithm to solve this problem. Initialize the  $n \times n$  array  $D$  by  $D[i, i] = x_i$  and  $D[i, j] = 0$  if  $j \neq i$ . The algorithm proceeds to fill the upper-right of the matrix  $D$  one column at a time to yield the final result in  $D[1, n]$ . We call this algorithm the *Vertical algorithm* [1].

Two other iterative algorithms, the Horizontal Algorithm and Diagonal Algorithm, have the subproblems



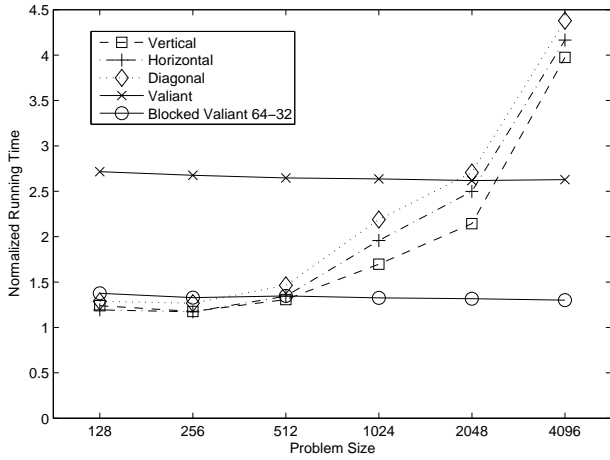


Figure 4: Normalized running time in nanoseconds of the five algorithms for Simple Dynamic Programming on Pentium 4.

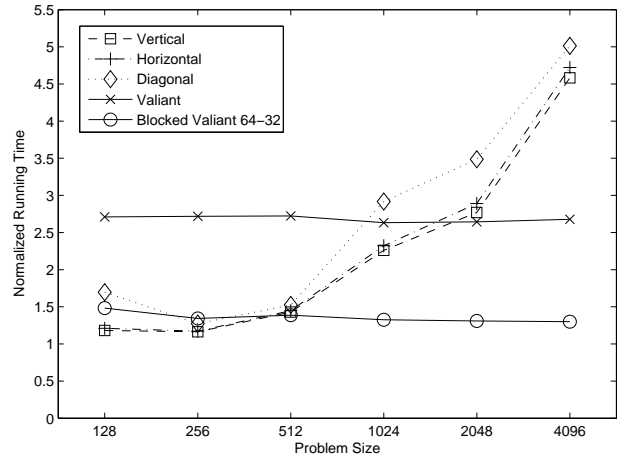


Figure 5: Normalized running time in nanoseconds of the five algorithms for Simple Dynamic Programming with the hardware prefetcher turned off on Pentium 4.

4 is not effective for this algorithm. The same is true of the horizontal and diagonal algorithms. To eliminate the column access stream of the CKY algorithms we use a form of data redundancy: for  $i \leq j$  the value  $D[i, j]$  is also stored in  $D[j, i]$ . This enables the CKY algorithm to be implemented with two row streams rather than a row and column stream. The Data Redundant Vertical Algorithm is described in Algorithm 7.

```

for  $j = 2$  to  $n$  do
  for  $i = j - 1$  to  $1$  do
    for  $k = i$  to  $j - 1$  do
       $D[i, j] := D[i, j] + D[i, k] \cdot D[j, k + 1]$ 
       $D[j, i] := D[i, j]$ 
    end for
  end for
end for

```

Algorithm 7: Data Redundant Vertical Algorithm for Simple Dynamic Programming

Data redundant algorithms based on the horizontal and diagonal algorithms can be defined similarly. Figure 6 shows the results from implementing the data redundant algorithms. The bottom two curves are from the Data Redundant Vertical and Horizontal Algorithms. On the negative side, if memory is a constraint then the data redundant algorithm, which uses twice as much memory as the standard CKY-Algorithm, will suffer page faults.

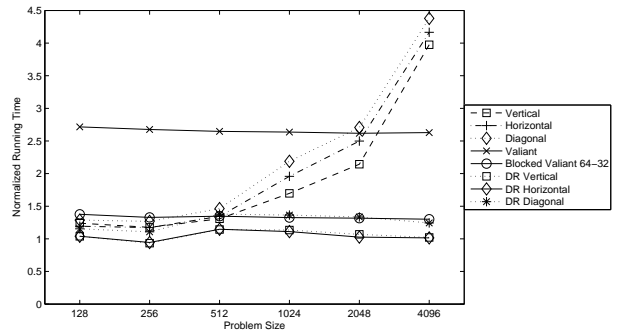


Figure 6: Normalized running time in nanoseconds of the Data Redundancy algorithms, the standard algorithms and the cache efficient algorithms for simple dynamic programming on Pentium 4.

## 6 Fast Fourier Transform.

The sequential accesses needed in running the Fast Fourier Transform (FFT) make it an ideal candidate for analysis under hardware prefetching. The FFT is an algorithm used to compute the Discrete Fourier Transform (DFT) of an input array  $A$  of  $n$  complex numbers in  $O(n \log n)$  time. This is given by the output array  $D[0..n-1]$ , where  $D[k] = \sum_{j=0}^{n-1} A[j] \omega_n^{kj}$ . Here  $\omega_n$  denotes the  $n$ th complex root of unity. We assume that  $n$  is a power of two. The first step of the FFT is to rearrange the input array  $A$  by taking the *bit-reversal permutation* [3]. Whether or not the prefetcher was enabled had little effect on the speed of the bit-reversal permutation, so its details are omitted. The remainder of the FFT's computation involves several

*butterfly operations* [3]. Each butterfly operation is just a few steps of complex arithmetic, and only the ordering of the butterfly operations is significant for study under prefetching.

```

m := 2
while m ≤ n do
  for j = 0 to m/2 - 1 do
    for k = 0 to n - 1 by m do
      butterfly(A[k + j], A[k + j + m/2], j, m)
    end for
  end for
  m = 2 * m
end while

```

Algorithm 8: Downwards Fast Fourier Transform

```

m := 2
while m ≤ n do
  for k = 0 to n - 1 by m do
    for j = 0 to m/2 - 1 do
      butterfly(A[k + j], A[k + j + m/2], j, m)
    end for
  end for
  m = 2 * m
end while

```

Algorithm 9: Across Fast Fourier Transform

Two standard FFT algorithms are presented, with their only difference being the ordering of the butterfly operations. The downwards method in Algorithm 8 is based on the FFT implementation of Numerical Recipes in C [11] while the across method in Algorithm 9 is based on one of Cormen *et al.* [3].

**6.1 Prefetcher-Friendly FFT Algorithm.** Long sequences of array accesses are desirable for hardware prefetching. In the downwards method, a longer-lasting  $k$  loop gives longer sequences of accesses, while in the across method, a long  $j$  loop is preferred. This translates into small and large values of  $m$ , respectively. We can see some of the benefits of both by having the first few executions of the  $m$  loop use the downwards method then having the remaining executions use the across method. This combination of the two standard approaches, described in Algorithm 10, requires a parameter  $s$ . This value is expected to be some power of two specifying how many iterations of the  $m$  loop will be done with the downwards method before switching.

**6.2 Cache-Efficient FFT Algorithm.** Another variant of this algorithm, described in Algorithm 11, is designed to be cache-aware. After applying bit-reversal,

```

m := 2
while m ≤ s do
  for j = 0 to m/2 - 1 do
    for k = 0 to n - 1 by m do
      butterfly(A[k + j], A[k + j + m/2], j, m)
    end for
  end for
  m = 2 * m
end while
while m ≤ n do
  for k = 0 to n - 1 by m do
    for j = 0 to m/2 - 1 do
      butterfly(A[k + j], A[k + j + m/2], j, m)
    end for
  end for
  m = 2 * m
end while

```

Algorithm 10: Prefetcher-Friendly Fast Fourier Transform

the downwards FFT is applied individually to the arrays  $A[0..l-1]$ ,  $A[l..2l-1]$ ,  $\dots$ ,  $A[n-l..n-1]$ . Here  $l$  is some power of two less than or equal to  $n$ . The remainder of the needed butterfly operations are done by the across method operating on the entire array. The appeal of this approach is that, for a well selected value of  $l$ , we can fill the cache (either the L1 or L2) with elements of the array and then perform much of our arithmetic without having cache misses.

```

for i = 0 to n - 1 by l do
  m := 2
  while m ≤ l do
    for j = 0 to m/2 - 1 do
      for k = 0 to n - 1 by m do
        butterfly(A[i+k+j], A[i+k+j+m/2], j, m)
      end for
    end for
    m = 2 * m
  end while
end for
while m ≤ n do
  for k = 0 to n - 1 by m do
    for j = 0 to m/2 - 1 do
      butterfly(A[k + j], A[k + j + m/2], j, m)
    end for
  end for
  m = 2 * m
end while

```

Algorithm 11: Cache-Efficient Fast Fourier Transform

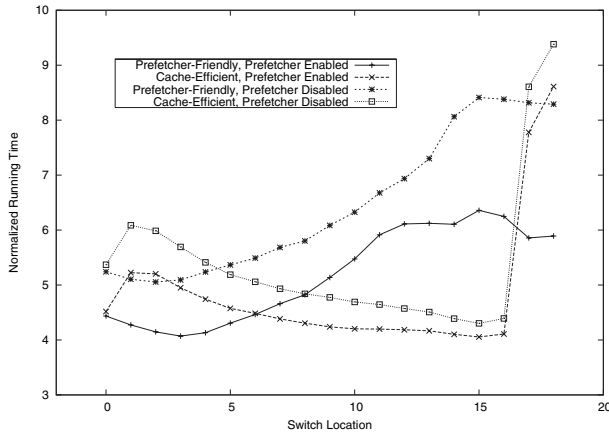


Figure 7: Normalized running time in seconds of different FFT implementations.

**6.3 Experimental Results for the FFT.** The FFT implementations of Algorithms 10 and 11 were timed with and without the hardware prefetcher enabled using the same experimental setup as in Section 4.2. The arrays had  $2^{18}$  randomly generated pairs of floats, with each pair corresponding to real and imaginary parts. Resulting times (in seconds) are multiplied by  $10^8 / (2^{18} * \log 2^{18})$  for normalization. The results are shown in Figure 7. For the prefetcher-friendly implementation, the  $x$ -axis denotes how many of the  $m$  loop iterations are done with the downwards method before switching to the across method. Our results suggest that a few iterations of the downwards method followed iterations of the across method gives the fastest times.

As expected, the cache-efficient approach is fastest when we can fill or nearly fill the cache with several elements to be used repeatedly, but is slower when we apply the downwards method to arrays larger than the cache. Here, the  $x$ -axis is the number of  $m$  loops performed by the downwards method to each subarray (whose sizes also depend on the value of the  $x$  coordinate) before the across method is employed. We see a substantial jump in time when we go from 16  $m$  loop iterations using the downwards method to 17. Since 16 iterations uses  $2^{16}$  pairs of floats, the needed array takes up  $2^{16} * 2 * 4$  Bytes which is precisely the size of the L2 cache (512 KB) used in the experiment.

Using either the prefetcher-friendly or the cache-efficient implementation would require tuning, that is choosing  $s$  or  $l$  so as to minimize the running time. Without prefetching enabled, the cache-efficient method gives significant improvement over the prefetcher-friendly method. With the prefetcher enabled, however, both methods give nearly the same minimum running

times. This suggests that hardware prefetching eliminates some of the need to design cache-efficient implementations.

## 7 Acknowledgments.

We would like to thank Jean-Loup Baer, Steve Swanson and Jan Sanislo for providing us with information and insight about hardware prefetchers.

## References

- [1] C. Cherng and R. E. Ladner, *Cache efficient simple dynamic programming*, Proceedings of the International Conference on the Analysis of Algorithms, 2005, pp. 49–58.
- [2] R. A. Chowdhury and V. Ramachandran, *Cache-oblivious dynamic programming*, Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2006, pp. 591–600.
- [3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, 2nd ed., 2001.
- [4] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, *Cache-oblivious algorithms*, Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS), 1999, pp. 17–18.
- [5] M. E. Furman, *Application of fast multiplication of matrices in the problem of finding the transitive closure of a graph*, Dokl. Akad. Nauk SSSR, 194:524 (Russian), Soviet Math. Dokl., 11(5):1252, 1970.
- [6] *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, <http://www.intel.com/design/processor/manuals/253669.pdf>.
- [7] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, *The Microarchitecture of the Pentium 4 Processor*, <http://www.intel.com/>.
- [8] T. Kasami, *An efficient recognition and syntax algorithm for context-free languages*, Scientific Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass., 1965.
- [9] I. Munro, *Efficient determination of the transitive closure of a directed graph*, Information Processing Letters, 1(2):56–58, 1971.
- [10] J.-S. Park, M. Penner and V. K. Prasanna, *Optimizing graph algorithms for improved cache performance*, IEEE Transactions on Parallel and Distributed Systems, vol. 15(9), pp. 769–782, 2004.
- [11] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*, Cambridge University Press, Cambridge, UK, 1992.
- [12] L. G. Valiant, *General context-free recognition in less than cubic time*, Journal of Computer and Systems Sciences, 10:308–315, 1975.
- [13] D. H. Younger, *Recognition of context-free languages in time  $n^3$* , Information and Control, 10(2):189–208, 1967.