Cache-Friendly Implementations of Transitive Closure

MICHAEL PENNER and VIKTOR K. PRASANNA
University of Southern California

The topic of cache performance has been well studied in recent years. Compiler optimizations exist and optimizations have been done for many problems. Much of this work has focused on dense linear algebra problems. At first glance, the Floyd–Warshall algorithm appears to fall into this category. In this paper, we begin by applying two standard cache-friendly optimizations to the Floyd–Warshall algorithm and show limited performance improvements. We then discuss the unidirectional space time representation (USTR). We show analytically that the USTR can be used to reduce the amount of processor-memory traffic by a factor of O($\sqrt{C}$), where C is the cache size, for a large class of algorithms. Since the USTR leads to a tiled implementation, we develop a tile size selection heuristic to intelligently narrow the search space for the tile size that minimizes total execution time. Using the USTR, we develop a cache-friendly implementation of the Floyd–Warshall algorithm. We show experimentally that this implementation minimizes the level-1 and level-2 cache misses and TLB misses and, therefore, exhibits the best overall performance. Using this implementation, we show a 2x improvement in performance over the best compiler optimized implementation on three different architectures. Finally, we show analytically that our implementation of the Floyd–Warshall algorithm is asymptotically optimal with respect to processor-memory traffic. We show experimental results for the Pentium III, Alpha, and MIPS R12000 machines using problem sizes between 1024 and 2048 vertices. We demonstrate improved cache performance using the Simplescalar simulator.

Categories and Subject Descriptors: F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problem Complexity; F.2.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*Computations on discrete structures*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Data structures, systolic array algorithms, Floyd–Warshall algorithm

1. INTRODUCTION

The topic of cache performance has been well studied in recent years. It has been clearly shown that the amount of processor-memory traffic is the bottleneck for achieving high performance in most applications [Sen Chatterjee 2000]. While this problem has been well studied, much of the focus has been on dense linear

Authors' address: Michael Penner and Viktor K. Prasanna, University of Southern California, Los Angeles, California 90007.

algebra problems, such as matrix multiplication and FFT [Chame et al. 2000; Frigo et al. 1999; Park et al. 2000; Whaley and Dongarra 1998]. All of these problems possess very regular access patterns that are known at compile time. In this paper, we focus on the fundamental graph problem of all-pairs shortest path, specifically, the Floyd–Warshall algorithm.

Optimizing cache performance to achieve faster overall execution time is a difficult problem. Modern microprocessors are including deeper and deeper memory hierarchies to hide the cost of cache misses. The performance of these deep memory hierarchies has been shown to differ significantly from predictions based on a single level of cache [Sen and Chatterjee 2000]. Different miss penalties for each level of the memory hierarchy, as well as the TLB, also play an important role in the effectiveness of cache-friendly optimizations. These miss penalties vary from processor to processor and can cause large variations in performance.

The all-pairs shortest-path problem (hereafter referred to as APSP) is a fundamental problem in a wide variety of fields, most notably network routing and distributed computing. APSP poses unique challenges to improving cache performance—challenges that often cannot be handled by standard cache-friendly optimizations [Hall 1999]. For example, the Floyd–Warshall algorithm involves updating N^2 elements at each step, where N is the number of vertices in the input graph. Simple tiling cannot be used to optimize these steps because of the data dependencies from one step to the next.

In this paper, we begin by examining the performance, both analytically and experimentally, of three standard implementations of the Floyd–Warshall algorithm. The first is the straightforward implementation compiled with the best available compiler optimizations in gcc or MS Visual C++. We then apply two optimizations, tiling with copying and tiling with the block data layout. These optimizations represent the best available compiler optimizations in research compilers. We show analytically and through SimpleScalar [Burger and Austin 1997] simulations that these optimizations do not improve the cache-performance of the Floyd–Warshall algorithm. We also show limited performance gains on our three architectures.

In order to improve cache performance, we use the *unidirectional space–time representation* (USTR). We show analytically that, by using this representation, we can develop cache-friendly implementations for a large class of algorithms. This representation is very similar to the space-time representation used in systolic array design, which also deals with partitioning the space as we do [Cosnard et al. 1986]. However, such systolic array designs do not have the added challenge of dealing with cache conflicts and multiple levels of memory hierarchy. We show how this representation can be used to uniquely face the challenges posed by the transitive closure problem. Using this representation, we show up to a factor of 2 improvement over the state of the art cache-friendly optimization, including those available in a research compiler [Lam et al. 1991], on our three architectures.

In the context of the USTR, we also show the importance of choosing the optimal tile size. We present a simple heuristic that can be used to intelligently narrow the experimental search space for the optimal tile size.

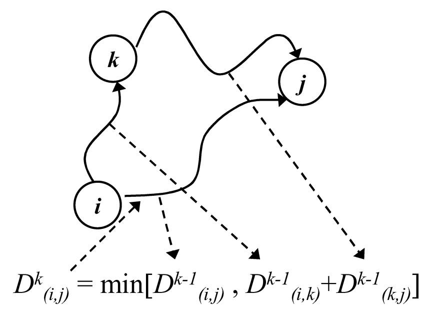

k
j
i
$D^k_{(i,j)} = \min[D^{k-1}_{(i,j)}, D^{k-1}_{(i,k)} + D^{k-1}_{(k,j)}]$

Fig. 1. *k*th step of Floyd–Warshall Algorithm.

The remainder of this paper is organized as follows. In Section 2, we give the background and briefly summarize some related work in the areas of cache optimizations and compiler optimizations. In Section 3, we discuss some basic cache optimizations that we consider and give Simplescalar (a widely used architecture simulator [Burger and Austin 1997]) results to substantiate our claims. In Section 4, we introduce our optimizations. Section 5 summarizes the optimizations. In Section 6, we present experimental data gathered from the three machines we used. In Section 7, we draw conclusions and give some directions for future work.

2. BACKGROUND AND RELATED WORKS

In this section we give the background information required for our discussion of various optimizations in Section 3. In Section 2.1, we give a brief outline of the Floyd–Warshall algorithm. Those readers comfortable with this algorithm can skip this section. In Section 2.2 we discuss some of the challenges that are faced in making the transitive closure problem cache friendly. Finally, in Section 2.3, we give some information regarding other work in the fields of cache analysis, cache-friendly optimizations, and compiler optimizations and how they relate to our work.

2.1 The Floyd–Warshall Algorithm

For the sake of discussion, suppose we have a directed graph G with N vertices labeled 1 to N and E edges. The Floyd–Warshall algorithm is a dynamic programming algorithm, which computes a series of N, $N \times N$ matrices, where D^k is the kth matrix and is defined as follows: $D^k_{(i,j)}$ = shortest path from vertex i to j composed of the subset of vertices labeled 1 to k. The matrix D^0 is the original graph G. We can think of the algorithm as composed of N steps. At each kth step, we compute D^k using the data from D^{k-1} in the manner shown in Figure 1. The pseudocode for this algorithm is given in Figure 2 [Cormen et al. 1990]. It should be noted that the Floyd–Warshall algorithm can be implemented using only a single matrix. The multiple matrix implementation is shown here for clarity and our optimizations are applicable to either implementation.

```
Floyd-Warshall($W$)
1. $n \leftarrow rows[W]$
2. $D^0 \leftarrow W$
3. for $k \leftarrow 1$ to $n$
4.     for $i \leftarrow 1$ to $n$
5.         for $j \leftarrow 1$ to $n$
6.             $D_{ij}^{k} \leftarrow \min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1})$
7. return $D^n$
```

Fig. 2. Pseudocode for the Floyd–Warshall algorithm.

2.2 Challenges

Transitive closure presents a very different set of challenges from those present in dense linear algebra problems, such as matrix multiply and FFT. In the Floyd–Warshall algorithm, the operations involved are comparison and add operations. There are no floating-point operations as in matrix multiply and FFT. We are also faced with dependencies that require us to update the entire $N \times N$ array D^k before moving on to the $(k+1)$th step. This data dependency from one kth loop to the next eliminates the ability of any commercial or research compiler to improve data reuse. We have explored this using the research compiler SUIF to optimize transitive closure and found that the optimization discussed in Section 3.1, namely tiling of the i and j loops, is the best it can perform without user-provided knowledge of the algorithm [Hall 1999]. These challenges, specifically the dependency from one kth loop to the next, mean that although the computational complexity of the Floyd–Warshall algorithm is $O(N^3)$, equivalent to the "usual" matrix multiply, transitive closure implementation displays much longer running times.

The model that we use for our research is that of a uniprocessor, cache-based system. We refer to the cache closest to the processor as $L1$ with size C_1 and subsequent levels as Li with size C_i. Throughout this paper we refer to the amount of *processor-memory traffic*. This is defined as the amount of data transferred between the last level of the memory hierarchy that is smaller than the problem size and the first level of the memory hierarchy that is larger than the problem size. In most cases we refer to these as cache and memory, respectively. Finally, we assume an internal TLB with a fixed number of entries.

It is well known that data is transferred between the cache and memory in blocks. This fact is taken advantage of best when the data layout matches the data access pattern as in the block data layout with the tiled Floyd–Warshall algorithm. However, this does not address the challenges of the Floyd–Warshall algorithm mentioned above. In order to reuse an individual element without incurring additional data transfer, novel techniques are required.

2.3 Related Work

In recent years, a number of groups have done research in the area of cache performance analysis. Detailed cache models have been developed by Weikle, McKee, and Wulf in [2000] and Sen and Chatterjee [2000]. Instead of eliminating cache misses, some groups develop methods to tolerate these misses. Multithreading has been discussed as one method of accomplishing this. Kwak et al. [1999] discuss the effects of multithreading on cache performance. Varman et al. [Kallahalla and Varman 2001; Varman and Verma 1999] have discussed similar caching issues and bounds for prefetching in the area of parallel I/O systems.

A number of papers have discussed the optimization of specific problems with respect to cache performance. The majority of these problems are in the area of dense linear algebra problems. Whaley and Dongarra [1998] discuss optimizing the widely used basic linear algebra subroutines (BLAS). Chatterjee and Sen [2000] discuss a cache efficient matrix transpose. Frigo et al. [1999] discuss the cache performance of cache oblivious algorithms for matrix transpose, FFT, and sorting. Park et al. [2000] discuss dynamic data remapping to improve cache performance for the DFT. One characteristic that these problems share is a very regular memory accesses that are known at compile time.

Another area that has been studied is the area of compiler optimizations (see, for example, [Rastello and Robert 1998; Sen and Chatterjee 2000; Whaley and Dongarra 1998]). Optimizing blocked algorithms has been extensively studied (see, for example, Lam et al. [1991]). The SUIF compiler framework includes libraries for performing data-dependency analysis and loop transformations, among other things. In this context, it is important to note that SUIF does not handle the data dependencies present in the Floyd–Warshall algorithm in a manner that improves the processor-memory traffic. It will perform the tiling optimization discussed in Section 3.1; however, it will not perform the transformation discussed in Section 4.2 without user intervention [Diniz 2001].

Although much of the focus of cache optimization has been on dense linear algebra problems, there has been some work that focuses on irregular data structures. Chilimbi et al. [1999a, 1999b] discusses making pointer-based data structures cache conscious. He focuses on providing structure layouts and structure definitions to make tree structures cache conscious. Tang et al. [1997] has also looked at optimizations for a heap data structures. The difference between this work and ours is that we focus on optimizing an algorithm instead of a specific data structure.

3. BASIC CACHE-FRIENDLY OPTIMIZATIONS

In this section, we explore three different optimizations of transitive closure. In Section 3.1, we show the usual implementation of the Floyd–Warshall algorithm, as well as a standard compiler technique for optimizing loop nests. We use these throughout the paper as our baseline. Section 3.1 also includes many of the definitions and assumptions that we use throughout Sections 3 and 4 for our analysis. In Section 3.2, we show a data layout optimization that is used to further improve the gains achieved in the compiler optimization.

Throughout Sections 3 and 4 we use result from the Simplescalar simulator from the Simplescalar Architectural Research Toolkit, version 2.0 [Burger and Austin 1997] as a companion to our theoretical analysis. The Simplescalar architecture is derived from the MIPS-IV ISA. The tool we used was sim-cache, which simulates the cache performance of a given executable. Parameters that are customizable include level-1 and -2 instruction and data cache parameters, as well as instruction and data TLB parameters. Parameters for these include the number of sets, block size, associativity, and replacement policy. We show running times of the optimizations on three widely used machines in Section 6.

3.1 Standard Optimization of the Floyd–Warshall Algorithm (Baseline Implementation)

As stated earlier, improving cache performance has been well studied in recent years in the area of dense linear algebra problems. Most of the optimizations developed deal with dense array structures. This dense array is present in the standard Floyd–Warshall algorithm. The purpose of this section is to introduce and analyze the baseline implementation as well as a fairly standard optimizations to improve cache performance. These optimizations produced less than 20% improvement over the baseline. The baseline that we use throughout our discussion is a usual implementation that is compiled using the state-of-the-art compiler optimizations available. The compilers we used for our experiments were GNU C++ (gcc) and Microsoft Visual C++ (MS VC++). We have verified that these compilers do not do loop transformation or copying. They only perform optimizations such as inline functions and local code reordering to hide miss latency.

In order to simplify the analysis we make a few assumptions. Suppose we have a graph with N vertices. The size of the adjacency matrix is then N^2. We are interested in optimizing performance as the problem size increases; the problem and intermediate data do not fit in the cache. We assume that the cache size is less than N^2 and the TLB is much smaller than N. We define processor-memory traffic as the amount of data transfer between the last level of the memory hierarchy that cannot contain the problem size (referred to as the cache) and the first level of the memory hierarchy that can contain the problem data (referred to as memory). On most traditional architectures, this would be between the level-2 cache and main memory. We also assume that the problem fits into some level of the main memory hierarchy. To experimentally validate our approaches and their analysis, the actual problem sizes that we are working with are between 1024 and 2048 nodes ($1024 \leq N \leq 2048$). Each data element is eight bytes. Many processors currently on the market have in the range of 16 to 64 KB of level-1 and between 256 KB and 4 MB of level-2 cache. Many processors have a TLB with approximately 64 entries and a page size of 4 to 8 KB. All of these parameters close match our assumptions.

Let us first examine the usual implementation of the Floyd–Warshall algorithm. The basic step (kth loop) in this algorithm is to take the outer product of the kth row and the kth column and update the entire matrix (see Figure 3). We assume the matrix is laid out in row major order. By definition of the algorithm then we are going to update N^2 elements in each kth loop. Since our cache is strictly less than N^2, this will generate $\Theta(N^3)$ total processor-memory traffic. Now suppose we want to update the entire ith row during some kth loop. In

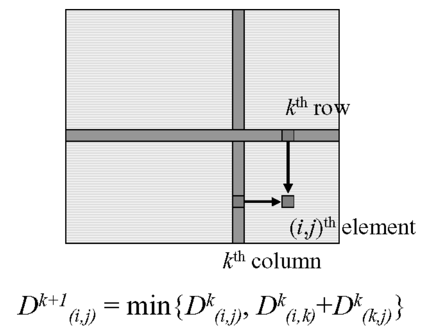

k^{th} row
$(i,j)^{th}$ element
k^{th} column
$D^{k+1}_{(i,j)} = \min\{D^k_{(i,j)}, D^k_{(i,k)}+D^k_{(k,j)}\}$

Fig. 3. Basic step (kth loop) in Floyd–Warshall algorithm.

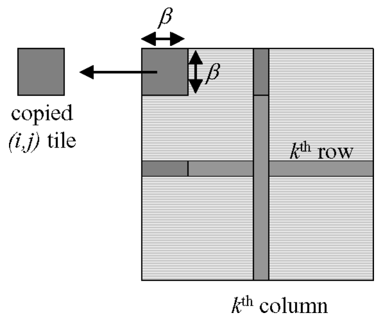


Fig. 4. Tiling plus copying for Floyd–Warshall algorithm.

the worst case, the location of the ith and kth rows in the cache could conflict exactly so that with each access to row i, the data required from row k would be removed from the cache. This could result in an extra O(N) conflict misses for that kth loop. Note that this phenomenon is avoided by associativity in the cache. We also want to consider TLB misses. In order to understand the TLB issues, suppose our page size is $N*l$ for some small l, possibly less than 1.[1] The adjacency matrix then sits inside N/l different pages. Each one of these must be accessed during every kth loop and all of them will not fit into the TLB. Thus, we will generate O(N/l) TLB misses during each kth loop. Therefore, the total number of TLB misses will be O(N^2/l).

The first optimization that we examine is a basic tiling approach combined with copying (Figure 4). Tiling is a loop transformation that attempts to reduce the working set size. It solves many small problems and combines the solutions into the solution for the original problem. Copying is used to reduce conflict misses within the tile by placing all the elements in contiguous memory locations. Because of data dependencies, the Floyd–Warshall algorithm can only be tiled for the i and j loops. In order to find the optimal tile size for each architecture, it is best to experiment with various tile sizes (see Section 4). For the sake of analysis, suppose that the tile size is $\beta \times \beta$, where β^2 < cache size. Since the dependencies still require updating all N^2 elements in each kth loop ($1 \le k \le N$),

[1]The Pentium III page size is 4 KB = 512 * d, where d is our data element size. The Alpha page size is 8 KB = 1024 * d.

Table I. Simplescalar Results for Tiled and Copied Floyd–Warshall Algorithm[a]

Data level-1 cache misses		
N	Baseline	Tiled, $\beta = 32$
1024	0.81	0.63
1536	2.72	2.13
		(billions)

Data TLB misses		
N	Baseline	Tiled, $\beta = 32$
1024	5.29	86.71
1536	17.76	218.08
		(millions)

[a]Architectural parameters used were from Pentium III architecture; see Section 4 for specific parameter values.

as in the original case, we will have O(N^3) overall processor-memory traffic. However, the tiled computation does reduce the working set size. Where we used to have an extra O(N) traffic when the ith row conflicted with the kth row, there is now an extra O(β) traffic when a row of the tile conflicts with the kth row. This reduction in conflict misses can be seen in the level-1 cache misses from Simplescalar (see Table I).

In order to understand the number of TLB misses, examine the problem of solving a single tile. Since the elements are laid out row-wise for the matrix, each row is on a different page; recall that page size is approximately N. This is true even with copying, since the tile in the original matrix must be accessed in order to copy it into contiguous locations. Therefore, this requires $\beta + 1$ pages to update each tile. For the baseline, the TLB working set is O(1), exactly two rows of the matrix. If the TLB is smaller than $\beta + 1$, we will have O(β) misses per tile and O(N^3/β) total TLB misses. In fact, this increase in TLB misses can be seen in our results from Simplescalar (see Table I). In our experiments, this optimization gave performance improvements ranging from 0 to 40% over the baseline.

3.2 Data Layout Optimization of the Floyd–Warshall Algorithm

The first optimization that we propose is a change in data layout. The theory behind this change in data layout is that in order to show spatial locality and, therefore, good cache performance, the data layout must match the data access pattern. In our tiled optimization, the access is naturally tile-by-tile, row-wise through the matrix. Within each tile, the data is also accessed row-wise. In order to match this data access pattern, the block data layout (BDL) should be used. The BDL is a two-level mapping that maps a tile of data, instead of a row, into contiguous memory. These blocks are laid out row-wise in the matrix and data is laid out row-wise within the block (see Figure 5). When the block size is equal to the tile size in the tiled computation, the data layout will exactly match the data access pattern. Also note that with this layout, copying is not necessary, since the elements in the tile are already in contiguous memory locations.

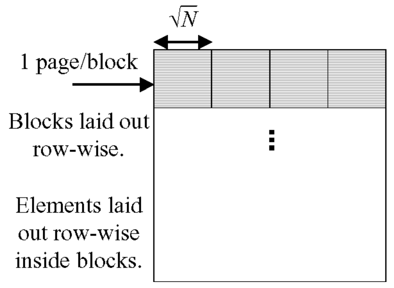

$\sqrt{N}$
1 page/block
Blocks laid out row-wise.
Elements laid out row-wise inside blocks.

Fig. 5. The block data layout.

Table II. Simplescalar Results for BDL Optimization of Floyd–Warshall Algorithm[a]

Data level-1 cache misses		
N	Baseline	BDL
1024	0.81	0.58
1536	2.72	1.95
		(billions)

Data TLB misses			
N	Baseline	Tiled	BDL
1024	5.29	86.71	5.80
1536	17.76	218.08	19.20
			(millions)

[a]Architectural parameters used were from Pentium III architecture; see Section 4 for specific parameter values.

The analysis of this optimization is very similar to that of the tiled and copied optimization. Since the dependencies still require updating the entire matrix in each kth loop, the total processor-memory traffic will be O(N^3). However, the working set is reduced by the tiled computation and the level-1 cache misses are reduced as shown in Table II. This is the same phenomenon that was shown in the tiling with copying optimization. Since each tile is in contiguous memory locations and is equal to O(1) TLB pages, this only requires O(1) TLB misses for each tile of computation. This gives a working set of O(1) pages. Recall that in the usual implementation, the working set was a row of the adjacency matrix. This was laid out in contiguous memory locations, so the working set of pages is O(1). In the tiled version, we showed the working set of pages was O(β). This difference can be seen in the Simplescalar simulation results for TLB misses (see Table II). The experimental results for the BDL optimization showed performance increases in the range of 5 to 40% over the baseline. These results are very similar to the results seen for the tiling with copying optimization (see Section 6).

4. THE UNIDIRECTIONAL SPACE–TIME REPRESENTATION AND CACHE-FRIENDLY ALGORITHMS

In this section, we introduce the unidirectional space-time representation (USTR). We show that this representation can be used to generate

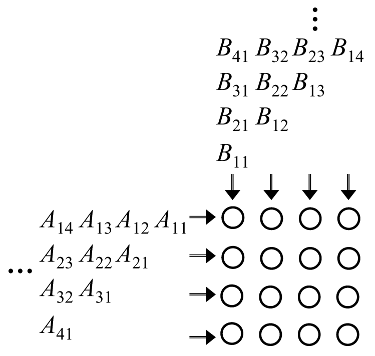
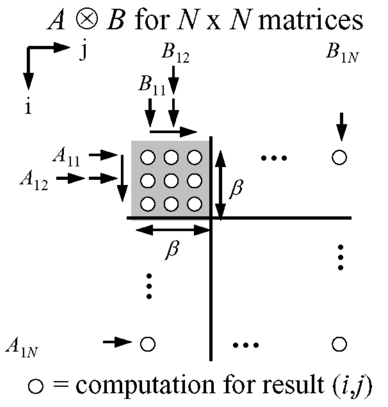

B_{41} B_{32} B_{23} B_{14}
B_{31} B_{22} B_{13}
B_{21} B_{12}
B_{11}
A_{14} A_{13} A_{12} A_{11}
A_{23} A_{22} A_{21}
A_{32} A_{31}
A_{41}

Fig. 6. USTR for 4×4 matrix multiply.

cache-friendly implementations of many algorithms. In Section 4.1, we introduce the basic idea of a space-time representation and the difference between this representation and the iteration space. In Section 4.2, we show how the USTR can be used to generate cache-friendly implementations. We also show analytical bounds on processor-memory traffic and show a technique to find an optimal partition size. Finally, in Section 4.3, we show one instance of how the USTR can be applied to transitive closure using results from Simplescalar to illustrate performance gains. Running times for this optimization can be found in Section 6. Throughout this section, we use matrix multiply as an example application; however, these techniques can be applied to many algorithms. For the sake of clarity, we will skip a formal definition of the USTR and focus on the key ideas.

4.1 Unidirectional Space–Time Representation

Let us first explain what we mean by a space-time representation. Similar notions have been used by the systolic array designs and VLSI signal processing community ([Cosnard et al. 1986; Ullman 1983]). Consider a problem in which the result is an $N \times N$ matrix. We divide the problem in space by representing the computation required to calculate each (i, j)th element in the result matrix as a computational element (CE) in an $N \times N$ array. For example, $\mathrm{CE}_{(i,j)}$ in a matrix multiply would represent the multiply-add operations required to compute $D_{(i,j)}$, where $D = A \times B$. Referring to Figure 6, each circle represents the computation required for the (i, j)th result. The notion of time comes from the data flowing through this $N \times N$ array of CEs. Referring to Figure 6 again, the data A would flow row-wise into the array from the left and the data B would flow column-wise into the array from the top. As the data flows through the array, each element does some simple computation on the data inside it and passes on the data. Once the data has flowed completely through the array, the (i, j) result lies in the corresponding CE. The space–time representation is much like a systolic array design. If each CE were viewed as a processor, the result would be an $N \times N$ systolic array [Ullman 1983]. The distinction that we add is the notion of unidirectional data flow. We only allow data to flow in the forward direction, either down or to the right. This allows us to generate a cache-friendly implementation.



$A \otimes B$ for N x N matrices
j
i
B_{12}
B_{11}
B_{1N}
A_{11}
A_{12}
A_{1N}
β
β
○ = computation for result (i,j)

Fig. 7. Unidirectional space–time representation. Note that $\otimes$ refers to a generic matrix operation.

Consider, for example, the simple systolic array implementation for multiplying two, 4×4 matrices (see Figure 7). During $t = 1$, the CE (1,1) receives A_{11} from the left and B_{11} from the top and computes $C_{11} = A_{11} * B_{11}$. During times $t = 2$, 3, and 4, the CE will receive A_{1t} and B_{t1}, and will compute $C_{11} = A_{1t} * B_{t1} + C_{11}$. In general, CE$(i, j)$ will receive data elements A_{ik} and B_{kj} at time $[(i-1)+(j-1)+k]$ and will compute $C_{ij} = C_{ij} + A_{ik} * B_{kj}$. The computation will be complete at time $t = 12$, when element (4,4) updates $C_{44} = C_{44} + A_{44} * B_{44}$ [24].

The key difference between this and the iteration space is the idea of scheduling operations in space. The iteration space actually deals only with scheduling operations in time, whereas the USTR represents operations divided in space as well as time [Rastello and Robert 1998]. As we will show in the next section, this fact allows us to generate implementations that are cache-friendly.

In summary, what we mean by a USTR is an $N \times N$ array of computational elements (CEs), where each element performs $O(N)$ computations. Thus, when implemented on a uniprocessor the algorithm requires $O(N^3)$ time. If the CEs are scheduled in a row-wise fashion, this would produce the *baseline* implementation corresponding with a usual 3-level perfectly nested loop.

4.2 From the USTR to a Cache-Friendly Implementation

In order to predict cache performance when we implement the above representation on a uniprocessor, we need to make a few assumptions regarding the CEs. We first assume that a fixed amount of computation is done at each CE during each time and the amount is relatively small. For the sake of simplicity, we also assume that each CE is performing exactly the same computation. We refer to this as a single operation. In the matrix multiply example, each element performed one multiply and add during each time unit. Finally, we assume that the local memory required within each CE is constant, for example, each CE in the matrix multiply array required local storage for one accumulated value. These assumptions are common to most systolic array designs. Note that the cache performance analysis does not depend on the type of operations being

```
USTR_Alg($A$, $B$, $C$, $N$, $b$)

1.  for $bi \leftarrow 1$ to $N/b$
2.      for $bj \leftarrow 1$ to $N/b$
3.          load $bxb$ elements of C at ($bi$,$bj$)
4.          for $bk \leftarrow 1$ to $N/b$
5.              load $bxb$ elements of A at ($bi$,$bk$)
6.              load $bxb$ elements of B at ($bk$,$bj$)
7.              for $i \leftarrow bi$ to $bi + (b-1)$
8.                  for $j \leftarrow bj$ to $bj + (b-1)$
9.                      for $k \leftarrow bk$ to $bk + (b-1)$
10.                         $C_{(i,j)} \leftarrow F(A_{(i,k)}, B_{(k,j)}, C_{(i,j)})$
11.     return $C$
```

Fig. 8. Pseudocode for the cache-friendly USTR. A and B are input matrices, C is the output matrix, $N \times N$ is the size of all matrices, b is the tile size, and F denotes multiply accumulate.

performed, making it applicable to any algorithm expressed in a USTR. All assumptions regarding cache size and problem size from Section 3.1 still hold. Recall that data flow has been limited to the forward direction, i.e., either down or to the right. Again, for the sake of clarity, we will skip formal proofs and focus on the key ideas.

Examining a single CE, note that the computation required is N operations. In the matrix multiply example, each CE required four operations to compute the final result. Each operation requires two new data elements as well as any locally stored values. This will subsequently result in 2^*N processor-memory traffic on a traditional architecture. In a usual implementation, each CE could be executed in a row-wise fashion. For the matrix multiply USTR, this corresponds to the usual 3-level nested loop code (without tiling). Based on the above calculation, this would result in $\Theta(N^3)$ processor-memory traffic.

Now let us define a tiled order of computations as follows. First tile the array of CEs into tiles of size $\beta \times \beta$ (see Figure 7). Within each tile, operate on CEs in a row-wise fashion. Within each CE, process β elements of the row and column that will pass through it before moving on to the next CE. We define a pass through a tile as executing each CE for β elements. Repeatedly pass through each CE in the tile until all input data has been processed. Returning to the matrix multiply example, this implementation would match with a 6-level nested loop implementation of matrix multiply (see Figure 8).

Another method of tiling would be to first tile the array of CEs into tiles of size $\beta \times \beta$. Within each tile, instead of processing β elements at each CE at a time, process the entire array for $t = 1$, then process it for $t = 2$, and so for $t \leq \beta$. This would then be defined as a single pass through the tile.

Between each CE and between tiles, we place a first-in-first-out (FIFO) buffer. When the adjacent CE or tile begins, it receives data from this buffer in the same manner as if all CEs were processing data simultaneously.

As we saw in Section 3.2, it is also beneficial to match our data layout to the data access pattern. Recall that we demonstrated large improvements in

TLB misses when we used the BDL on a tiled access pattern compared with a row-wise data layout for the same access pattern. Since access to the input data in the USTR is also in a tiled fashion, it is beneficial to again use the BDL to minimize TLB misses. Throughout this section, we assume a BDL when implementing the USTR to eliminate self-interference misses and minimize cross-interference misses between blocks of data.

When the computation is tiled, as shown earlier in Figure 7, we can take advantage of data locality and reduce the processor-memory traffic. Examining the first pass through a tile of the array of CEs, each CE performs β operations, requiring the first β data elements of one row and one column of the input, as well as its locally stored value. Note that the CE directly below it requires exactly the same column elements and β data elements from the next row. When this is extended to the entire tile, it requires $2^*\beta^2$ data elements of the input, β^2 locally stored values, and performs β^3 operations. In order to complete each tile, it must be passed through N/β times. This requires $2 * (N/\beta) * \beta^2$ data elements of the input, β^2 locally stored values, and performs $(N/\beta) * \beta^3$ total operations. From this discussion we have the following theorem.

THEOREM 1. *Given an USTR of an algorithm, we can reduce the amount of processor-memory traffic by a factor of β, where cache size is $O(\beta^2)$, compared with a baseline implementation.*

PROOF SKETCH. Each pass through a tile requires $2^*\beta^2$ elements of the input and β^2 locally stored elements and performs β^3 operations. If we choose β^2 to be O(C), where C is the cache size, all locally stored values will reside in the cache. Also, the current $2 * \beta^2$ tiles of the input will remain in the cache for the duration of the pass. Each pass through a tile then results in $2 * \beta^2$ processor-memory traffic. There is a total of $(N/\beta) \times (N/\beta)$ tiles. Each tile requires N/β passes. The total number of operations is given by:

$$\left(\frac{N}{\beta}\right) * \left(\frac{N}{\beta}\right) * \left(\frac{N}{\beta}\right) * \beta^3 = N^3$$

The total amount of processor-memory traffic is given by:

$$\left(\frac{N}{\beta}\right) * \left(\frac{N}{\beta}\right) * \left(\frac{N}{\beta}\right) * 2\beta^2 = 2 * \left(\frac{N^3}{\beta}\right)$$

Therefore the processor-memory traffic is reduced by a factor of β.

In order to implement the USTR, we must also consider the schedule for computing each tile. Recall from Figure 7 that in the USTR all data flow is in the forward direction. Therefore, in order to satisfy these data dependencies, a valid schedule will have the following characteristic:

1. When computing tile (i, j), all tiles (k, l), where $\{k \leq i$ and $l < j\}$ or $\{k < i$ and $l \leq j\}$, must have already been computed; where the tile (1,1) is the upper left-most tile.

For example, a row-wise schedule of tiles would satisfy this requirement. The pseudocode for this schedule is shown in Figure 8. One could also use a more complex schedule such as a wavefront. □

One of the key factors in Theorem 1 holding is that β^2 is chosen to be on the order of cache size. The simplest and possibly the most accurate method of choosing β is to experiment with various tile sizes. This is the method that the automatically tuned linear algebra subroutines (ATLAS) project employs [Whaley and Dongarra 1998]. However, it is beneficial to find an estimate of the optimal tile size. The following is a method to generate approximate bounds on the optimal tile size.

Note that the working set is composed of 3 $\beta \times \beta$ tiles of data. We can classify cache into three categories; compulsory, conflict misses, and capacity misses. Compulsory misses, by definition, cannot be avoided. Here we provide a heuristic for choosing a tile size, such that conflict and capacity misses are minimized.

The heuristic we provide makes use of the 2:1 rule of thumb from Patterson and Hennessy [1996]. The 2:1 rule of thumb states that a direct mapped cache of size C has approximately the same miss ratio as a two-way set associative cache of size $C/2$. Based on the results published in Patterson and Hennessy [1996] this rule of thumb holds loosely for any k and 2*$k$way set associative caches. For example, if the cache is a two-way set associative cache of size C, the equation to solve would be $3 * \beta^2 * d = C/2$.

1. Use the 2:1 rule of thumb to adjust the cache size to that of an equivalent four-way set associative cache. This minimizes conflict misses since our working set consists of three contiguous tiles of data. Self-interference misses are eliminated by the data being in contiguous locations and cross-interference misses are eliminated by the associativity.
2. Choose β by Eq. 1, where d is the size of one element and C is the adjusted cache size. This minimizes capacity misses.

$$3 * \beta^2 * d = C \tag{1}$$

Note that this does not calculate an exact value for the optimal β; it simply finds a loose bound on the desired search space.

It is also important to note that the search space must take into account each level of cache, as well as the size of the TLB. Given these various solutions for β the best tile size can be found experimentally. In order to validate this method, calculate the best tile size for the Pentium III machine based on the level-2 cache. The level-2 cache is a 256 KB, eight-way set associative cache. Use the 2:1 rule of thumb and base the calculations on a 512 KB, four-way set associative cache. The element size d is 8 bytes. Solving Eq. 1 gives $\beta = 147.8$. Experimentally, the best tile size for the USTR optimization of transitive closure on our Pentium III was found to be $\beta = 140$.

4.3 A Cache-Friendly Algorithm for Transitive Closure

As we stated in Section 4.1, the USTR is similar to notations used in the systolic array and VLSI signal-processing communities. A standard systolic array implementation of the Floyd–Warshall algorithm is as follows [Ullman 1983].

1. Given a graph with N vertices in the adjacency matrix representation, feed the matrix A into an $N \times N$ systolic array of processing elements (PEs) both row-wise from the top and column-wise from the left as shown in Figure 9.

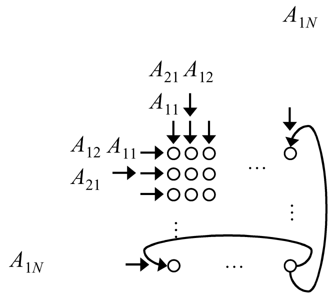


Fig. 9. Systolic array implementation of Floyd–Warshall algorithm.

2. At each PE (i, j), update the local variable $C_{(i,j)}$ by the following formula:
$$C_{(i,j)} = \min(C_{(i,j)}, A_{(i,k)} + A_{(k,j)}) \tag{2}$$
where $A_{(i,k)}$ is the value received from the left and $A_{(k,j)}$ is the value received from the top. $C_{(i,j)}$ is initialized to infinity. This equation comes directly from the Floyd–Warshall algorithm.
3. If $i = k$, pass the value $C_{(i,j)}$ down, otherwise pass $A_{(k,j)}$ down. If $j = k$, pass the value $C_{(i,j)}$ to the right, otherwise pass $A_{(i,k)}$ to the right.
4. Finally, when data elements reach the edge of the matrix, a loop around connection should be made, such that $A_{(i,N)}$ passes data to $A_{(i,1)}$ and $A_{(N,j)}$ passes data to $A_{(1,j)}$ (see Figure 9).

LEMMA 1 (ULLMAN 1983). *The above computation results in the transitive closure of the input once all input data elements have been passed through the entire array exactly three times.*

Without a transformation, this implementation does not fit in the USTR because of the loop around connections. Recall that in order to fit in our USTR, all data must flow in the forward direction, namely either down or to the right (see Section 4.1). However, based on the above Lemma 1, we can expand the original representation in the following manner.

Copy the entire array twice so that we have three $N \times N$ arrays of PEs. Make a connection from the end of the ith row in one array to the beginning of the ith row in the next and from the end of the jth column in one array to the beginning of the jth column in the next, as shown later in Figure 10. These connections replace the loop around connections in the original systolic array implementation (see Figure 9).

This new representation qualifies as unidirectional and, therefore, is an USTR of the Floyd–Warshall algorithm. Note that each PE in the systolic array implementation becomes a computational element (CE) in our USTR. Also note, that although the representation visually requires $3{*}N^2$ space, no additional memory is required compared with the baseline implementation. Based on the results in Section 4.2, we can execute each CE on a uniprocessor architecture. We can also tile the computation in the manner shown in Section 4.2. The pseudocode for this algorithm is given in Figure 11. Based on Theorem 1 we have:

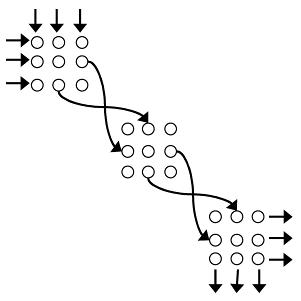


Fig. 10. Unidirectional space–time representation of systolic array algorithm for transitive closure.

```
Cache-Friendly_FW($A$, $C$, $N$, $b$)

1.  for $l \leftarrow$ 1 to 3
2.      for $bi \leftarrow$ 1 to $N/b$
3.          for $bj \leftarrow$ 1 to $N/b$
4.              load $bxb$ elements of C at ($bi$,$bj$)
5.              for $bk \leftarrow$ 1 to $N/b$
6.                  load $bxb$ elements of $A$ at ($bi$,$bk$)
7.                  load $bxb$ elements of $A$ at ($bk$,$bj$)
8.                  for $i \leftarrow bi$ to $bi + (b-1)$
9.                      for $j \leftarrow bj$ to $bj + (b-1)$
10.                         for $k \leftarrow bk$ to $bk + (b-1)$
11.                             $C_{(i,j)} \leftarrow \min(C_{(i,j)}, A_{(i,k)} + A_{(k,j)})$
12.return $C$
```

Fig. 11. Pseudocode for the cache-friendly implementation of the Floyd–Warshall algorithm. A is the input matrix, C is the output matrix, N is the dimension of all matrices, and b is the tile size.

THEOREM 2. *The Floyd–Warshall algorithm can be implemented on a uniprocessor such that the processor-memory traffic is reduced by a factor of β, where cache size is on the order of β^2 compared with the baseline implementation.*

The maximum reduction factor in processor-memory traffic to perform ordinary matrix multiplication given a limited internal memory is $O(\sqrt{M})$, where M is the size of the internal memory [Horowitz and Sahni 1978]. Using the structure of the Floyd–Warshall dependency graph, it can be shown:

THEOREM 3. *Our USTR implementation of the Floyd–Warshall algorithm is (asymptotically) optimal for all implementations of the Floyd–Warshall algorithm with respect to processor memory traffic. The proof of Theorem 3 is given in the Appendix.*

To illustrate this reduction in processor-memory traffic, we show results from Simplescalar experiments for the number of cache misses (see Table III). Even though this algorithm performs a total of 3^*N^3 operations, Simplescalar results show a 30x improvement in level-2 cache misses. Note that it was found experimentally that the best tile size for the USTR algorithm on the Pentium III architecture essentially ignores the level-1 cache and focuses on the level-2

Table III. Example Simplescalar Results for USTR Floyd–Warshall Algorithm, $\beta = 140$

Data level-1 cache misses (billions)		
N	Baseline	USTR
1024	0.81	8.16
1536	2.72	2.76

Data level-2 cache misses (millions)		
N	Baseline	USTR
1024	538	18
1536	1,814	57

Data TLB misses (millions)		
N	Baseline	USTR
1024	5.29	4.08
1536	17.76	15.61

Architectural parameters used were from Pentium III architecture; see Section 5 for specific parameter values.

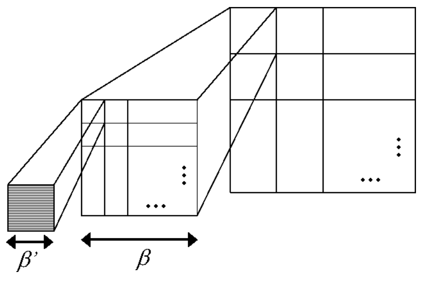


Fig. 12. Multilevel tiling for USTR schedule and/or layout.

cache misses. This is because of the level-2 cache being on-chip and, therefore, the miss penalty for a level-2 miss is much higher than a level-1 miss. For more information regarding experimental results see Section 6.

4.4 A Hierarchical Data Layout for Deep Memory Architectures

In order to hide miss latencies, modern processors are including deeper memory hierarchies. For the sake of this discussion, consider an m-level memory hierarchy, where the ith level is of size C_i and the first level is closest to the processor in terms of access cost. For simplicity sake, we assume $C_i \leq \mathrm{C}_{(i+1)}$ for all $i < m$. In order to optimize for this deep memory hierarchy, one could consider a multilevel tiling method for both the schedule and the data layout in the USTR. Consider a multilevel tiling method, such as the method shown in Figure 12. In this method, β' would be chosen to minimize the traffic between level-1 and -2 cache. This is exactly what we have shown thus far in our discussion. The traffic between the level-2 cache and the next level of the memory hierarchy would then be minimized by choosing β, such that β^2 is equal to the size of the

Table IV. Summary of Results from Sections 3 and 4[a]

Summary of analytical and simulation results				
	Baseline	Tiled	BDL	USTR (billions)
Computational complexity	N^3	N^3	N^3	N^3
Processor-memory traffic	N^3	N^3	N^3	N^3/β
Data Level-1 cache misses	2.72	2.13	1.95	2.76
Data Level-2 cache misses	1.81	1.85	1.84	0.057
Data TLB misses	0.018	0.218	0.019	0.016

[a]Architectural parameters used were from Pentium III architecture; see Section 6 for specific parameter values.

level-2 cache. This could be repeated until we reach a level that is larger than our problem size.

In order to take advantage of this layout, we must also consider the schedule of operations. Consider the original schedule with one level of blocking. In that schedule, we process β elements of data for each of the $\beta \times \beta$ computational elements in a tile in each step. In the case of a multilevel tiling, instead of directly executing these β^3 operations at a step, the step becomes similar to a recursive call. This original $\beta \times \beta$ tile is subdivided into $\beta' \times \beta'$ tiles and each computational element in these tiles processes β' elements of data in a substep. These recursive steps continue in the same way a usual recursive program would until we reach the final tile size corresponding with the level-1 cache. The difference between this and a usual recursive program is that the tile size is chosen to reflect an architectural parameter, not to subdivide the problem by a predetermined factor.

Using this multilevel tiling method, we have the following:

THEOREM 4. *The multilevel tiled data layout and schedule for the USTR reduces the traffic between the ith and the* $(i+1)$*th levels of cache by a factor of* β_i *at each level of the memory hierarchy, where* $\beta_i = O(\sqrt{C_i})$.

PROOF SKETCH. Recall from Theorem 1, that we can reduce the processor memory traffic by a factor of β, given a cache of size β^2. Based on the schedule of tiles discussed above and the formation of the USTR, it can be shown that each step where we perform β operations for each computation element in a $\beta \times \beta$ tile of elements is identical to the USTR for a complete problem where $N = \beta$. For this reason, Theorem 1 holds when we tile this $\beta \times \beta$ step into $\beta' \times \beta'$ substeps. Further, Theorem 1 holds for any $\beta_{(i+1)} \times \beta_{(i+1)}$ problem when we divide it into $\beta_i \times \beta_i$ steps. (Note that the schedule must follow the specification discussed above in order for this to hold.) Therefore, we can reduce the traffic between the ith and the $(i+1)$th levels of cache by a factor of β_i at each level of the memory hierarchy, where $\beta_i = O(\sqrt{C_i})$. □

5. SUMMARY

In summary, we compare the optimizations we have discussed in Sections 3 and 4 for computation complexity, processor-memory traffic, and Simplescalar results in Table IV. Cache size is less than N^2. Experimental results are shown in Section 6.

6. EXPERIMENTAL RESULTS

The experimental results section is divided into two subsections. In Section 6.1, we give details regarding the experimental setup. In Section 6.2, we discuss the experimental results.

6.1 Experimental Setup

For our experiments, we used two 933 MHz Pentium III machines. These have separate instruction and data level-1 caches, each 16 KB, four-way set associative with 32 byte (B) lines. The processors have a unified on-chip level-2 cache, which is 256 KB, eight-way set associative with 32 B lines. The TLB is split for data and instructions. The instruction TLB has 32 entries and is four-way set associative with LRU replacement. The data TLB has 64 entries and is four-way set associative with LRU replacement. The page size for both TLBs is 4 KB. The operating system was Windows 2000 professional (used MSVC++ compiler, version 6.0), on one, and Mandrake Linux, on the other (gcc compiler, version 2.95.2).

We also used a 500 MHz Alpha machine for our experiments. This machine has split data and instruction level-1 caches each 64 KB, two-way set associative with 64 B lines. The level-2 cache is a unified off-chip cache of size 4 MB, direct mapped with 64 B lines. Along with these, the Alpha also has an eight-element fully associative victim data buffer used for both instructions and data. The TLB on the Alpha has 128 entries and is fully associative. The page size is 8 KB. The operating system is Linux and we used the gcc compiler (version 2.91.66).

Finally, we used a 300 MHz MIPS R12000. This was part of a 64 processor SMP Origin 2000, although our implementations ran only on one processor. This processor also has split instruction and data level-1 cache; each 32 KB, two-way set associative, with 32 B lines. The level-2 cache is a unified 8 MB cache, direct mapped, with 64 B lines. The TLB has 64 entries, is fully associative, with a page size of 4 KB. The operating system was IRIX64 version 6.5 and we used the gcc compiler (version 2.8.1).

6.2 Experimental Results

Figures 13–17 show the actual running times of the four implementations on the four different machines; compiler-optimized, tiled and copied, block data layout (BDL), and the USTR optimization. On both Pentium III's, we show small improvements in the tiled optimization and the BDL, while the USTR implementation gave better than 2x improvement over the compiler optimized implementation (see Figures 13 and 14). This is quite consistent with the simulation results presented in earlier sections (see Table IV). The number of cache misses for the tiled and copied and the BDL optimization were both within 30% of the baseline for level-1 and within 2% for level-2. The BDL had the best level-1 cache performance and this shows up as the best execution time in all but one specific case ($N = 1536$ on the Pentium III running Windows). One notable difference is the amount of performance improvements seen in the tiled and copied and the BDL on the two machines. On the Linux machine, these algorithms showed significant performance improvements, whereas on the Windows machine, the

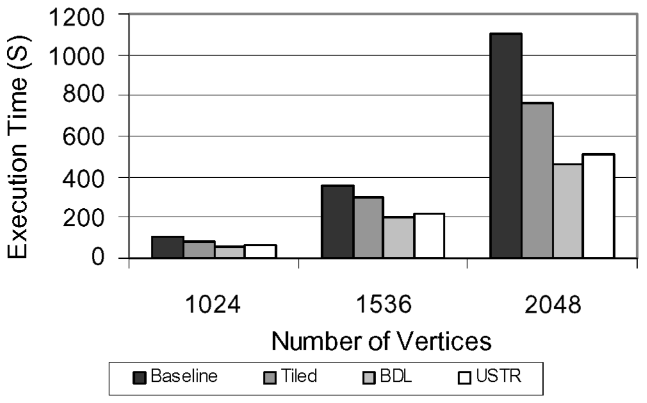


Fig. 13. Execution times for implementations on Pentium III running Linux.

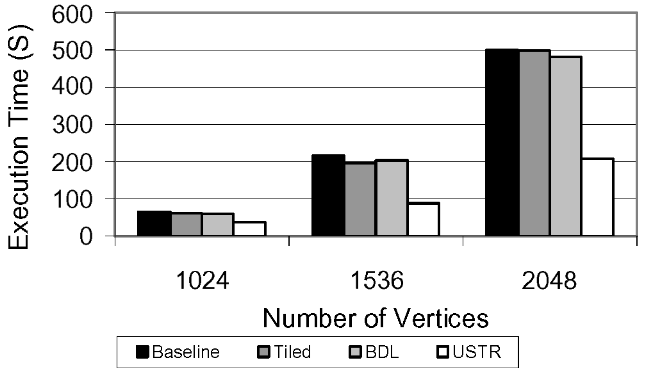


Fig. 14. Execution times for implementations on Pentium III running Windows 2000.

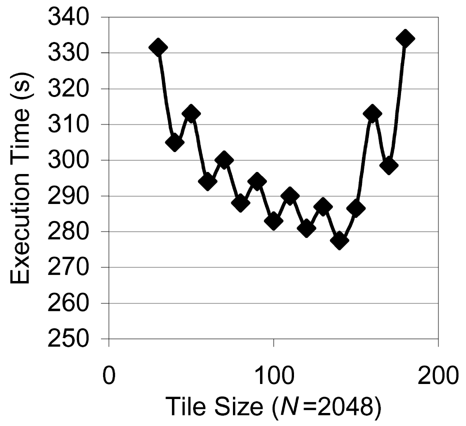


Fig. 15. USTR optimization, tile size selection Pentium III, Linux.

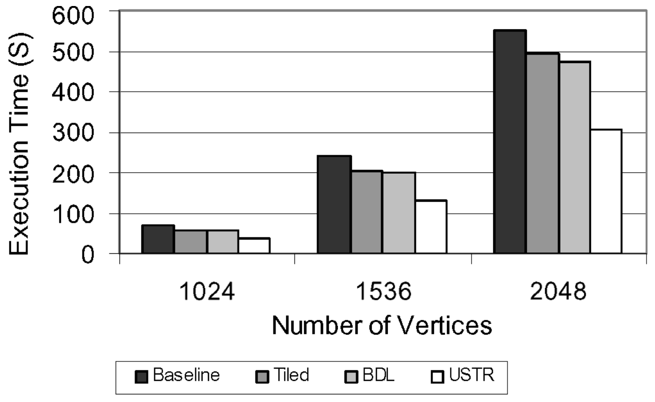


Fig. 16. Execution times for implementations on Alpha running Linux.

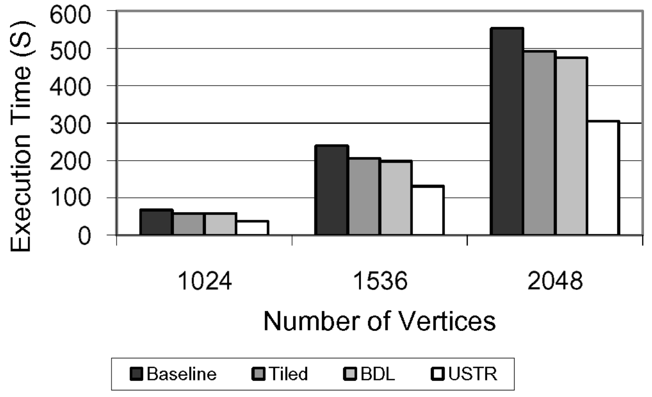


Fig. 17. Execution times for implementations on MIPS R12000 running IRIX64.

baseline, the tiled and copied, and the BDL all showed roughly equal runtimes. This difference is most likely due to the different compilers being used and the level of optimization done by those compilers as well as interaction with the operating system. The USTR optimization's improvement matches very nicely with the 97% decrease in level-2 cache misses. Recall that the memory hierarchy on the Pentium III behaves more like a two-level memory hierarchy because of the level-2 cache being on-chip. This performance led us to use a block size that essentially ignored the level-1 cache. In fact, our level-1 cache misses increased slightly from the baseline. This drastic decrease in level-2 cache misses as well as a slight decrease in TLB misses gave us an overall 2x improvement in performance.

The Alpha machine showed significantly different results. The tiled optimization and the BDL optimization showed much larger performance improvements, while the USTR implementation showed similar improvements as what we saw on the Pentium III's, approximately 2x improvement. One reason for this may be that the Alpha has an off-chip level-2 cache and a victim cache. This would show very different miss penalties, than we saw on the Pentium III.

Table V. Experimentally Optimal Tile Sizes for Tiled Algorithms for Each Machine and Range Given by Tile Size Heuristic

Optimal Tile Sizes				
	P III, W2K	PIII, Linux	Alpha	MIPS
Tiled and copied	36	32	42	42
BDL	38	40	40	40
USTR	140	140	70	70
USTR Range	(26,148)	(26,148)	(36,209)	(26,295)

In order to take full advantage of the two levels of cache on the Alpha a two level tiling of the USTR should be employed (see Section 4.4, Figure 12). At the time of this writing we have not performed these experiments.

The MIPS R12000 showed surprisingly poor performance for the baseline or compiler optimized code. This led to almost a 2x improvement for the tiled and copied optimization. The BDL optimization showed approximately 15% improvement over the tiled and copied optimization. The USTR optimization showed a 3x improvement over the baseline and almost a 2x improvement over the tiled and copied optimization. Apart from the poor performance of the baseline, these results match roughly with the results from our other architectures.

For each of the tiled optimizations (tiled and copied, BDL, and USTR), we used experimentation to find an optimal tile size for each machine. These results are shown in Table V. For the USTR optimization, we expanded our search space based on the results from our block size selection heuristic (see Section 4.2, Eq. 1). We experimented with block sizes in the range of 30 to 180 (see Figure 15). The best block sizes for each machine and optimization are given in Table V.

7. CONCLUSIONS AND FUTURE WORK

We examined a number of different optimizations for the Floyd–Warshall algorithm. We noted that this algorithm poses very different challenges from those seen in dense linear algebra problems. In order to address these challenges in a unique fashion, we proposed the *unidirectional space–time representation* (USTR). We showed analytically that this representation could be used to generate cache-friendly optimizations for a large class of algorithms and we demonstrated the improvements in cache performance for transitive closure using the Simplescalar simulator. Using this representation, we showed up to a 2x improvement in the performance of the Floyd–Warshall algorithm on three different architectures.

Using the USTR representation, it is also possible to generate cache-friendly implementations of both the Algebraic Path Problem and LU-Decomposition, without pivoting. The algebraic path problem is essentially a generalization of the Floyd–Warshall algorithm, so our USTR implementation can be generalized in the same fashion. For LU-decomposition, without pivoting the data dependencies, exist only in the forward direction and this, therefore, fits nicely in a USTR.

The deep memory hierarchy of modern uniprocessors poses new challenges and new opportunities for cache-friendly optimization. Future work on the

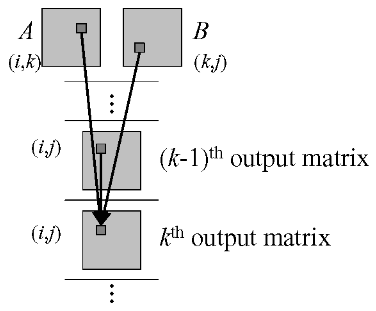


Fig. A1. Portion of dependency graph for outer product implementation of Matrix Multiply.

USTR will address these new opportunities by developing multilevel tiled data layouts and schedules that can be tuned to the multiple levels of cache memory.

This work is part of the Algorithms for Data IntensiVe Applications on Intelligent and Smart MemORies (ADVISOR) Project at USC. In this project, we focus on developing algorithmic design techniques for mapping applications to architectures. Through this we understand and create a framework for application developers to exploit features of advanced architectures to achieve high performance.

APPENDIX

We show the proof of the following Theorem:

THEOREM 3. *Our USTR implementation of the Floyd–Warshall algorithm is (asymptotically) optimal for all implementations of the Floyd–Warshall algorithm with respect to processor-memory traffic.*

PROOF. Hong and Kung [1981] develop the red-blue pebble game and the s-partition problem in order to show bounds on I/O complexity. One problem that they examine is ordinary matrix multiplication. Ordinary matrix multiplication refers to matrix multiplication in which exactly N^3 multiplications are performed when the input matrices are both $N \times N$. Using the s-partition problem on the dependency graph for this problem, they show a lower bound on I/O complexity of $\Omega(N^3/\sqrt{M})$, given $O(M)$ internal memory. In our case, the limited internal memory is the cache and the I/O complexity is the same as to the processor-memory traffic.

Consider the dependency graph obtained when computing $A * B = C$ by successively computing the outer product of the kth row and the kth column and updating the entire matrix. In this graph, the (i, j)th element in the kth output matrix depends on the (i, k)th element of A and the (k, j)th element of B and the (i, j)th element of the $(k - 1)$th output matrix (see Figure A1). Since this is an N^3 implementation of matrix multiplication, the processor-memory traffic will be bounded by $\Omega(N^3/\sqrt{M})$.

Now consider the dependency graph for the Floyd–Warshall algorithm given an $N \times N$ adjacency matrix A. At the kth step, use the kth row and the kth column of the $(k - 1)$th output matrix to generate the kth output matrix, for $1 \leq k \leq N$. In this case, the (i, j)th element of the kth output matrix depends on

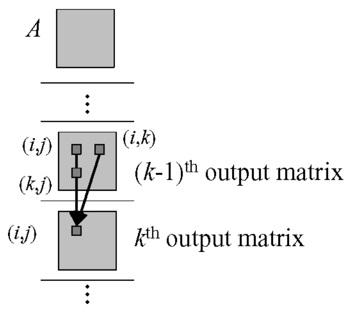


Fig. A2. Portion of dependency graph for the Floyd-Warshall algorithm.

the (i, k)th element, the (k, j)th element, and the (i, j)th element of the $(k-1)$th output matrix (see Figure A2).

In the dependency graph for the Floyd–Warshall algorithm, since the (i, j)th element of the kth output matrix depends on the (i, j)th element of the $(k-1)$th output matrix, then there exist a path in the dependency graph from the (i, j)th element of the input to the (i, j)th element of the kth output matrix. If, for all i,j, and k from 1 to N, all vertices on the path from the (i, k)th element of the input through the (i, k)th element of the $(k-1)$th output matrix to the (i, j)th element of the kth output matrix were ignored and similarly all vertices ignored on the path from the (k, j)th element of the input to the (i, j)th element of the kth output matrix, we would have the dependency graph for ordinary matrix multiplication. For this reason, matrix multiplication can be solved using the dependency graph for the Floyd–Warshall algorithm and, therefore, matrix multiplication reduces (with respect to processor-memory traffic) to the Floyd–Warshall algorithm.

Suppose that we can reduce the amount of processor-memory traffic for the Floyd–Warshall algorithm by a factor of S, where $S > \sqrt{M}$, where M is the cache size. Since matrix multiplication reduces (with respect to processor memory traffic) to transitive closure, we can also solve matrix multiplication and reduce the amount of processor-memory traffic by a factor of S, where $S > \sqrt{M}$. However, this violates the original bound on the reduction in processor-memory traffic for ordinary matrix multiplication. Therefore, the reduction in processor-memory traffic for the Floyd–Warshall algorithm is also bounded by $O(\sqrt{M})$.

Since the USTR implementation of the Floyd–Warshall algorithm has $O(N^3/\sqrt{M})$ processor-memory traffic, it is asymptotically optimal with respect to processor-memory traffic for all implementation of the Floyd–Warshall algorithm. □

ACKNOWLEDGMENTS

We are grateful to Jingzhao Ou for helpful comments and the editorial assistance of Vasanth Namasivayam in revising the manuscript.

REFERENCES

ADVISOR Project. http://advisor.usc.edu/.

BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar Tool Set, Version 2.0, University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June, 1997.

Chame, J., Hall, M., and Shin, J. 2000. Compiler transformations for exploiting bandwidth in PIM-based systems. In *Proc. of Solving the Memory Wall Workshop* (June).

Chatterjee, S. and Sen, S. 2000. Cache efficient matrix transposition. In *Proc. of International Symposium on High Performance Computer Architecture* (Jan.).

Chilimbi, T. M., Davidson, B., and Larus, J. R. 1999. Cache-conscious structure definition. *ACM SIGPLAN'99 Conference on Programming Language Design and Implementation* (May).

Chilimbi, T. M., Hill, M. D., and Larus, J. R. 1999. Cache-conscious structure layout. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation* (May).

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Cosnard, M., Quinton, P., Robert, Y., and Tchuente, M. (eds.) 1986. *Parallel Algorithms and Architectures*, North Holland, Amsterdam.

Diniz, P. 2001. USC ISI, Personal Communication (March).

Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. 1999. Cache-oblivious algorithms. In *Proc. of 40th Annual Symposium on Foundations of Computer Science*, 17–18, New York, (Oct.).

Hall, M. W., Kogge, P., Koller, J., Diniz, P., Chame, J., Draper, J., Lacoss, J., Brockman, J., Athas, W., Srivastava, A., Freeh, V., Shin, J., and Park, J. 1999. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proc. of International Conference on Supercomputing* (Nov.).

Horowitz, E. and Sahni, S. 1978. *Fundamentals of Computer Algorithms*. Computer Society Press.

Hong, J. and Kung, H. 1981. I/O Complexity: The Red Blue Pebble game. In *Proc. of ACM Symposium on Theory of Computing*.

Kallahalla, M. and Varman, P. J. 2001. Optimal prefetching and caching for parallel I/O systems. In *Proc. of 13th ACM Symposium on Parallel Algorithms and Architectures*.

Kwak, H., Lee, B., Hurson, A. R., Yoon, S., and Hahn, W. 1999. Effects of multithreading on cache performance. *IEEE Trans. Comput. 48*, 2 (Feb.).

Lam, M. S., Rothberg, E. E., and Wolf, M. E. 1991. The cache performance and optimizations of blocked algorithms. In *Proc. of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA (Apr.).

Park, N., Kang, D., Bondalapati, K., and Prasanna, V. K. 2000. Dynamic data layouts for cache-conscious factorization of the DFT. In *Proc. of International Parallel and Distributed Processing Symposium* (May).

Patterson, D. A. and Hennessy, J. L. 1996. *Computer Architecture: A Quantitative Approach*, 2nd Ed., Morgan Kaufmann, San Matis, CA.

Rastello, F. and Robert, Y. 1998. Loop partitioning versus tiling for cache-based multiprocessor. In *Proc. of International Conference Parallel and Distributed Computing and Systems*, Las Vegas, NV.

Sen, S. and Chatterjee, S. 2000. Towards a theory of cache-efficient algorithms. In *Proc. of Symposium on Discrete Algorithms*.

SPIRAL Project. http://www.ece.cmu.edu/∼spiral/.

Tang, X., Ghiya, R., Hendren, L. J. , and Gao, G. R. 1997. Heap analysis and optimizations for threaded programs. In *Proc. of International Conference on Parallel Architectures and Compilation Techniques*, San Francisco, CA (Nov.) 14–25.

Ullman, J. D. 1983. *Computational Aspects of VLSI*, Computer Science Press, Rockville, MD.

Varman, P. J. and Verma, R. M. 1999. Tight bounds for prefetching and buffer management algorithms for parallel I/O systems. *IEEE Trans. Parall. Distrib. Syst. 10*, 12, 1262–1275.

Weikle, D. A. B., Mckee, S. A., and Wulf, W. M. A. 2000. Caches as filters: A new approach to cache analysis. In *Proc. of Grace Murray Hopper Conference* (Sept.).

Whaley, R. C. and Dongarra, J. J. 1998. Automatically tuned linear algebra software. *High Performance Computing and Networking* (Nov.).

Received December 2001; accepted March 2006