Cache-Friendly Implementations of Transitive Closure*

Michael Penner and Viktor K Prasanna
University of Southern California
(mipenner@usc.edu, prasanna@usc.edu)
http://advisor.usc.edu

Abstract

In this paper we show cache-friendly implementations of the Floyd-Warshall algorithm for the All-Pairs Shortest-Path problem. We first compare the best commercial compiler optimizations available with standard cache-friendly optimizations and a simple improvement involving a block layout, which reduces TLB misses. We show approximately 15% improvements using these optimizations. We also develop a general representation, the Unidirectional Space Time Representation, which can be used to generate cache-friendly implementations for a large class of algorithms. We show analytically and experimentally that this representation can be used to minimize level-1 and level-2 cache misses and TLB misses and therefore exhibits the best overall performance. Using this representation we show a 2x improvement in performance with respect to the compiler optimized implementation. Experiments were conducted on Pentium III, Alpha, and MIPS R12000 machines using problem sizes between 1024 and 2048 vertices. We used the Simplescalar simulator to demonstrate improved cache performance.

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 [3, 17]. While the topic of cache performance has been well studied, much of the focus has been on dense linear algebra problems, such as matrix multiplication and FFT [3, 10, 14, 21]. All of these problems possess very regular access patterns that are known at compile time. In this paper, we take a unique approach to this topic by focusing on the fundamental irregular problem of transitive closure.

Optimizing cache performance to achieve better overall performance 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 [16]. 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 experimental results.

The All-Pairs Shortest-Path problem (hereafter referred to as transitive closure) is a fundamental problem in a wide variety of fields, most notably network routing and distributed computing. Transitive closure, as an irregular problem, poses unique challenges to improving cache performance, challenges that often cannot be handled by standard cache-friendly optimizations [8]. The Floyd-Warshall algorithm involves updating N^2 elements at each step. Simple tiling cannot be used to optimize these steps due to data dependencies from one step to the next.

In this paper we develop the *Unidirectional Space Time Representation* (USTR) and show that 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 [7]. However, such systolic array designs do not have the added challenge of dealing with cache conflicts and multiple levels of memory hierarchy. We also 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 a state of the art cache-friendly optimization, including those available in a research compiler [12].

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 optimization and compiler optimizations. In Section 3 we discuss each optimization that we consider and give Simplescalar results to substantiate our claims. In Section 4 we present

* Supported by the US DARPA Data Intensive Systems Program under contract F33615-99-1-1483 monitored by Wright Patterson Airforce Base and in part by an equipment grant from Intel Corporation.

experimental data gathered from the three machines we used. In Section 5 we draw conclusions and gives some direction for future work.

2. Background and Related Works

In this section we give the background information required in 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. 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, NxN matrices where D^k is the k^{th} matrix and is defined as follows: $D^k_{(i,j)}$ = shortest path from vertex i to vertex 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 k^{th} step, we compute D^k using the data from D^{k-1} in the manner shown in Figure 1[6].

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 NxN array D^k before moving on to the $(k+1)^{th}$ step. This data dependency from one k^{th} loop to the next eliminates the ability of any commercial or research compiler to improve data reuse. We have explored 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 [8]. These challenges mean that although the computational complexity of the Floyd-Warshall algorithm is $O(N^3)$, equivalent to matrix multiply, often transitive closure displays much longer running times.

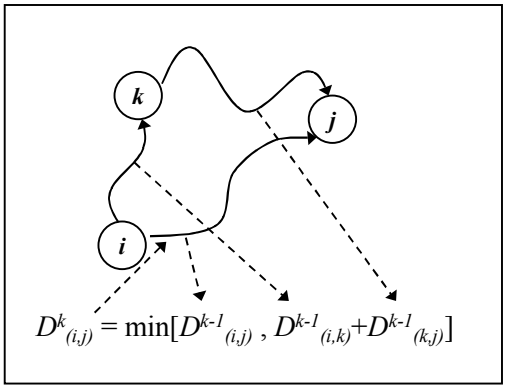

Figure 1: k^{th} step of Floyd-Warshall Algorithm

2.3. Related Work

A number of groups have done research in the area of cache performance analysis in implementing algorithms in recent years. Detailed cache models have been developed by Weikle, McKee, and Wulf in [20] and Sen and Chatterjee in [16]. Instead of eliminating cache misses, some groups develop methods to tolerate these misses. Multithreading has been discussed as one method of accomplishing this. Kwak and others discuss the effects of multithreading on cache performance in [11].

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 others discuss optimizing the widely used Basic Linear Algebra Subroutines (BLAS) in [21]. Chatterjee and Sen discuss a cache efficient matrix transpose in [4]. Frigo and others discuss the cache performance of cache oblivious algorithms for matrix transpose, FFT, and sorting in [9]. Park and Prasanna discuss dynamic data remapping to improve cache performance for the DFT in [13]. 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 [15, 16, 26]). Optimizing blocked algorithms has been extensively studied (see for example [12]). 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 3.4 without user intervention [8].

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 discusses making pointer-based data structures

cache-conscious in [5]. He focuses on providing structure layouts and structure definitions to make tree structures cache-conscious. Gao has also looked at optimizations for a heap data structures in [18]. The difference between this work and ours is that we focus on optimizing an algorithm instead of a data structure.

3. 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 Section 3 for our analysis. In Section 3.2 we show a data layout optimization that is used to compliment the compiler optimization. Finally, in Section 3.3 we introduce the Unidirectional Space Time Representation and how it can be used to generate cache-friendly optimizations. Throughout the sections we use result from the Simplescalar simulator to verify our analytical analysis. We show actual running times of the optimizations on our three machines in Section 4.

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. This 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 do perform such optimizations as inline functions and 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

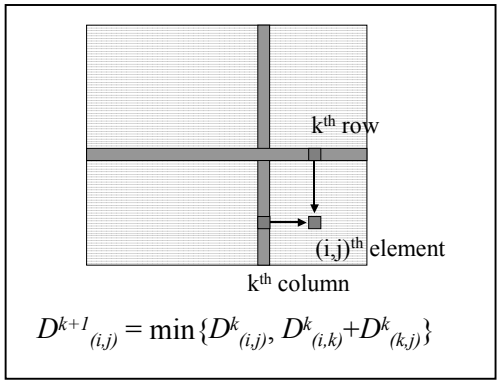

Figure 2: Basic step (k^{th} loop) in Floyd-Warshall algorithm

processor-memory traffic as the traffic 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 ($1024 \le N \le 2048$). Each data element is 8 bytes. Many processors currently on the market have in the range of 16 to 64 KB of level-1 cache 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 8KB. All of these parameters match closely with our assumptions.

Let us first examine the usual implementation of the Floyd-Warshall algorithm. The basic step (k^{th} loop) in this algorithm is to take the outer product of the k^{th} row and the k^{th} column and update the entire matrix. 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 k^{th} 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 i^{th} row during some k^{th} loop. In the worst case, this could conflict exactly with the k^{th} row of the matrix and cause an extra $O(N)$ conflict misses for that k^{th} loop. 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.* Then the adjacency matrix sits inside N/l different pages. Each one of these must be accessed during every k^{th} loop and all of them will not fit into the TLB. So, we will generate $O(N/l)$ TLB misses

* 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.

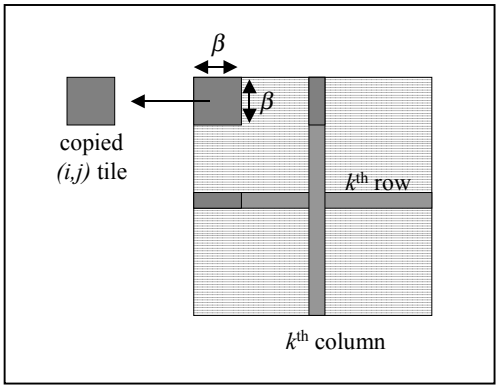

Figure 3: Tiling plus copying for Floyd-Warshall algorithm

during each k^{th} 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 3). 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. Due to 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 βxβ, where β^2 < cache size. Since the dependencies still require updating all N^2 elements in each k^{th} loop ($1 \le k \le N$), 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 i^{th} row conflicted with the k^{th} row, there is now an extra $O(\beta)$ traffic when a row of the tile conflicts with the k^{th} row. This reduction in conflict misses can be seen in the level-1 cache misses from Simplescalar (see Table 1).

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 2 rows of the matrix. If the TLB is smaller than $\beta + 1$, we will have $O(\beta)$ misses per tile, and $O(N^3/\beta)$ total TLB misses. In fact, this increase in TLB misses can be seen in our results from Simplescalar (see Table 1). In our experiments, this optimization gave performance improvements ranging from 0% to 40% over the baseline.

Data level-1 cache misses

N	Baseline	Tiled, β=32
1024	0.81	0.63
1536	2.72	2.13

(billions)

Data TLB misses

N	Baseline	Tiled, β=32
1024	5.29	86.71
1536	17.76	218.08

(millions)

Table 1: Simplescalar results for tiled and copied Floyd-Warshall algorithm. Architectural parameters used were from Pentium III architecture; see Section 4 for specific parameter values.

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 4). 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.

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 k^{th} 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 2. 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 total of $O(N^3/\beta^2)$ TLB misses and 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

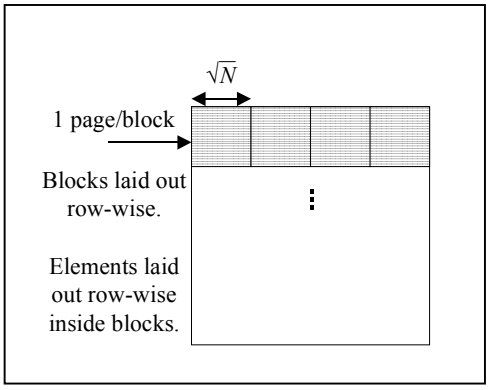

Figure 4: The Block Data Layout

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)

Table 2: Simplescalar results for BDL optimization of Floyd-Warshall algorithm. Architectural parameters used were from Pentium III architecture; see Section 4 for specific parameter values.

the tiled version, we showed the working set of pages was $O(\beta)$. This difference can be seen in the Simplescalar simulation results for TLB misses (see Table 2). The experimental results for the BDL optimization showed performance increases in the range of 5% to 15% on the Pentium III and approximately 40% on the Alpha (see Section 4)

3.3. 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 cache-friendly implementations of many algorithms. In Section 3.3.1 we introduce the basic idea of a space-time representation and the difference between this representation and the iteration space. In Section 3.3.2 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 3.3.3 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 4. 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.

3.3.1. 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 ([7, 19]). Consider a problem in which the result is an *N*x*N* matrix. We divide the problem in space by representing the computation required to calculate each result as a computational element (CE) in an *N*x*N* array, for example, the multiply-add operations required in a matrix multiply. Referring to Figure 6, each circle in the space represents the computation required for the $(i,j)^{th}$ result. The notion of time comes from the data flowing through this *N*x*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*x*N* systolic array [19]. 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.

Consider, for example, the simple systolic array implementation for multiplying 2, 4x4 matrices (see Figure 5). During t=1, the CE (1,1) receives A_{11} from the left and B_{11} from the right and computes $C_{11} = A_{11}*B_{11}$. During times t=2, 3, & 4, the CE will receive A_{1t} and B_{1t}, and will compute $C_{11} = A_{1t}*B_{1t} + C_{11}$. 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 $C_{44}=C_{44} + A_{44}*B_{44}$ [19].

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 [15]. As we will show in the next section, this fact allows us to generate implementations that are cache-friendly.

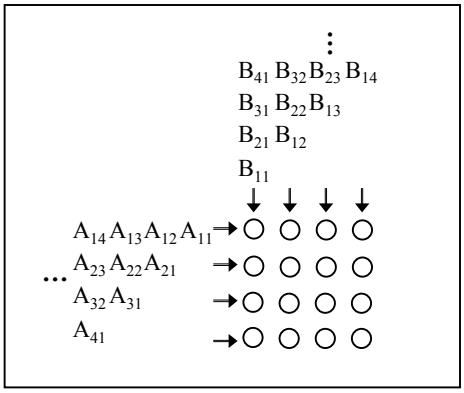

Figure 5: USTR for 4x4 matrix multiply

In summary, what we mean by a USTR is an *N*x*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 cooresponding with a usual 3-level perfectly nested loop.

3.3.2. 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

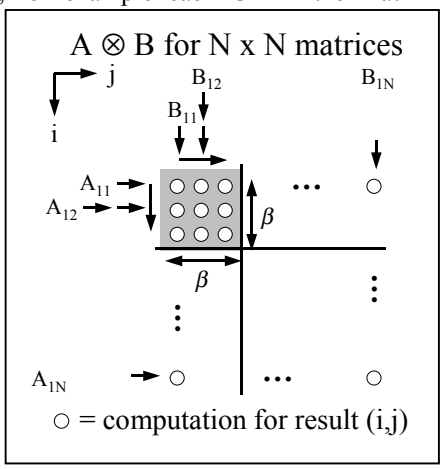

Figure 6: Unidirectional Space Time Representation. Note: ⊗ refers to a generic matrix operation.

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 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 2 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 βxβ (see Figure 6). 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.

Another method of tiling would be to first tile the array of CEs into tiles of size βxβ. 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≤β. This then would 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 the 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 6, 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/β)x(N/β) 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 6 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:

- When computing tile (i,j), all tiles (k,l), where $\{k \le i$ and $l < j\}$ or $\{k < i$ and $l \le j\}$, 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. One could also use a more complex schedule such as a wavefront. ■

When faced with a multi-level memory hierarchy, one could consider a multi-level tiling method for both the schedule and the data layout in the USTR. Consider a multi-level tiling method such as the method shown in Figure 7. In this method β would be chosen to minimize the traffic between level-1 and level-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 level-2 cache. We could use a simple row-wise layout of tiles within this larger β'xβ' tile. This could be repeated until we reach a level that is larger than our problem size. Using this multi-level tiling method, we can gain an improvement of $\sqrt{c_i}$ in traffic at each level of the memory hierarchy, where c_i is the size of the memory at the corresponding level of the memory hierarchy. In this case the schedule of βxβ tiles and β'x β' tiles becomes important. In order to take advantage of the most data reuse possible the schedule of operations must match the data layout while still satisfying the unidirectional data flow properties of the USTR.

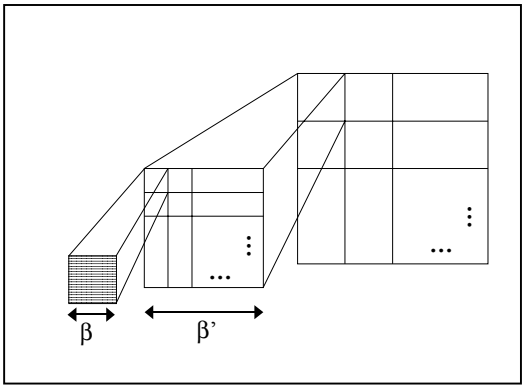

Figure 7: Multi-level tiling for USTR schedule and/or layout.

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 [21]. 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 βxβ tiles of data. We can classify cache misses into three categories; compulsory misses, 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.

- Use the 2:1 rule of thumb from [14] (see below) to adjust the cache size to that of an equivalent 4-way set associative cache. This minimizes conflict misses since our working set consists of 3 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.

- Choose β by Equation 1, where d is the size of one element and C is the adjusted cache size. This minimizes capacity misses.

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

The 2:1 rule of thumb states that a direct mapped cache of size C has approximately the same miss ratio as a 2-way set associative cache of size $C/2$. Based on the results published in [14] this rule of thumb holds loosely for any k and $2*k$ way set associative caches. For example, if the cache is a 2-way set associative cache of size C, the equation to solve would be $3*\beta^2*d = C/2$. Also 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, 8-way set associative cache. Use the 2:1 rule of thumb and base the calculations on a 512 KB, 4-way set associative cache. The element size d is 8 bytes. Solving Equation 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 $\beta = 140$.

3.3.3. A Cache-Friendly Algorithm for Transitive Closure. As we stated in Section 3.4.1, 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 [19].

- Given a graph with N vertices in the adjacency matrix representation, feed the matrix A into an NxN systolic array of processing elements (PEs) both row-wise from the top and column-wise from the left as shown in Figure 8.
- At each PE *(i,j)*, update the local variable $C_{(i,j)}$ by the following formula:

$$C_{(i,j)} = \min(C_{(i,j)}, A_{(i,k)} + A_{(k,j)}) \qquad \textbf{2}$$

  Where $A_{(i,k)}$ is the value received from the top and $A_{(k,j)}$ is the value received from the left.
- 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.
- 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 8).

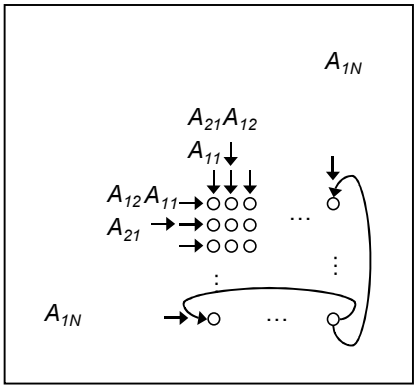

Figure 8: Systolic Array implementation of Floyd-Warshall algorithm

Lemma 1 [19]: The above computation results in the transitive closure of the input once all input data elements have been passed through the entire array exactly 3 times.

Without a transformation, this implementation does not fit in the USTR due to 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 3.4.1). 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 NxN arrays of PEs. Make a connection from the end of the i^{th} row in one array to the beginning of the i^{th} row in the next and from the end of the j^{th} column in one array to the beginning of the j^{th} column in the next as shown in Figure 9. These connections replace the loop around connections in the original systolic array implementation (see Figures 8 & 9).

This new representation qualifies as unidirectional and therefore is an USTR of the Floyd-Warshall algorithm.

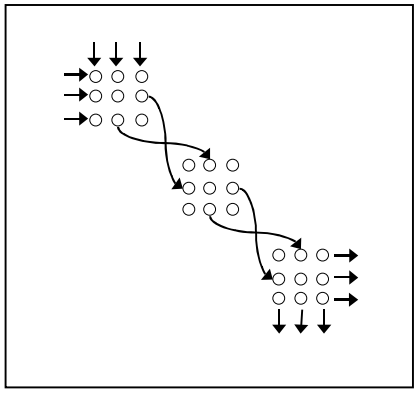

Figure 9: Unidirectional Space Time Representation of Systolic Array algorithm for transitive closure.

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 3.3.2 we can execute each CE on a uniprocessor architecture. We can also tile the computation in the manner shown in Section 3.4.2 and based on Theorem 1 we have:

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 [10]. 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 with respect to processor memory traffic.

To illustrate this reduction in processor-memory traffic we show results from Simplescalar experiments for the number of cache misses (see Table 3). 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 cache misses. This is due to 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 4.

Data level-1 cache misses

N	Baseline	USTR
1024	0.81	8.16
1536	2.72	2.76

(billions)

Data level-2 cache misses

N	Baseline	USTR
1024	538	18
1536	1,814	57

(millions)

Data TLB misses

N	Baseline	USTR
1024	5.29	4.08
1536	17.76	15.61

(millions)

Table 3: Example Simplescalar results for USTR Floyd-Warshall algorithm, β = 140. Architectural parameters used were from Pentium III architecture; see Section 4 for specific parameter values.

For our experiments we used two 933 MHz Pentium III machines. These have separate instruction and data level-1 caches, each 16 Kilobytes (KB), 4-way set associative with 32 Byte (B) lines. The processors have a unified on-chip level-2 cache, which is 256 KB, 8-way set associative with 32 B lines. The TLB is split for data and instructions. The instruction TLB has 32 entries and is 4-way set associative with LRU replacement. The data TLB has 64 entries and is 4-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 (used gcc compiler, version 2.95.2).

3.4. Summary

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

Summary of analytical and simulation results

	Baseline	Tiled	BDL	USTR
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

(billions)

Table 4: Summary of results from Section 3. Architectural parameters used were from Pentium III architecture; see Section 4 for specific parameter values.

4. Experimental Results

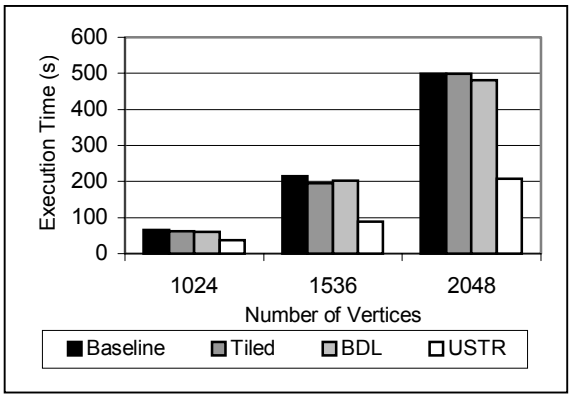

Figure 10: Execution times for implementations on Pentium III running Windows 2000.

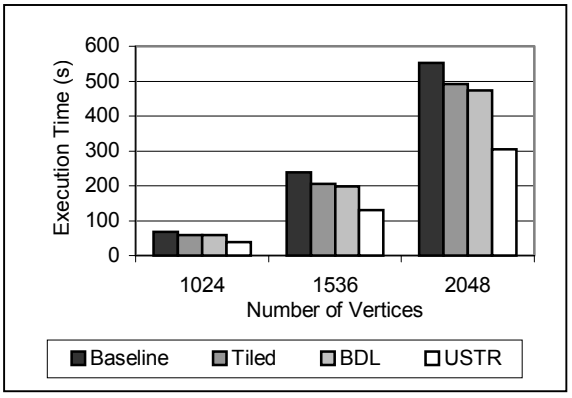

Figure 11: Execution times for implementations on Pentium III running Linux.

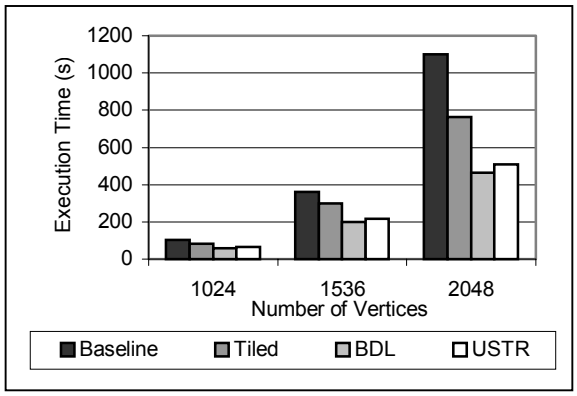

Figure 12: Execution times for implementations on Alpha running Linux.

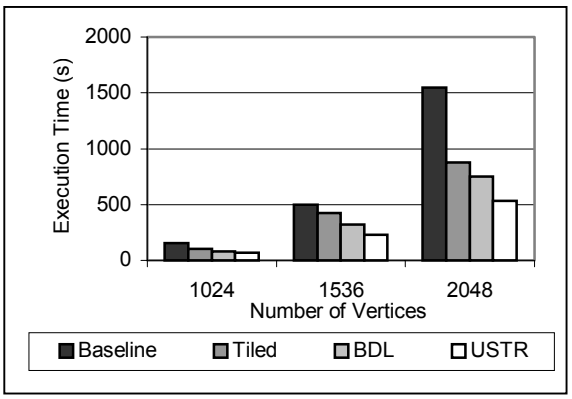

Figure 13: Execution times for implementations on MIPS R12000 running IRIX64.

We also used a 500 MHz Alpha machine for our experiments. This machine has split data and instruction level-1 caches each 64 KB, 2-way set associative with 64 B lines. The level-2 cache is a unified off-chip cache of size 4 Megabytes (MB), direct mapped with 64 B lines. Along with these, the Alpha also has an 8-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, 2-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).

The simulator that we used was from the Simplescalar Architectural Research Toolkit, version 2.0 [3]. 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 level-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.

Figures 10-13 show the actual running times of the 4 implementations on the 4 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 10&11). This is quite consistent with the simulation results presented in earlier sections (see Table 4). 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 difference to note is the difference in execution time for the baseline, relative to the tiled and copied and the BDL, on the two machines. This difference is probably due to the different compilers being used and the level of optimization done by those compilers. 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 due to 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. 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 3.3.2, Figure 7). 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 Figure 14 and Table 5. For the USTR optimization, we expanded our search space based on the results from our block size selection heuristic (see Section 3.4.2, equation 1). We experimented with block sizes in the range of 30 to 180 (see Figure 14). The best block sizes for each machine and optimization are given in Table 5.

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)

Table 5: Optimal tile sizes for tiled algorithms for each machine and range given by tile size heuristic

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 3 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 USTR will address these new opportunities by developing multi-level

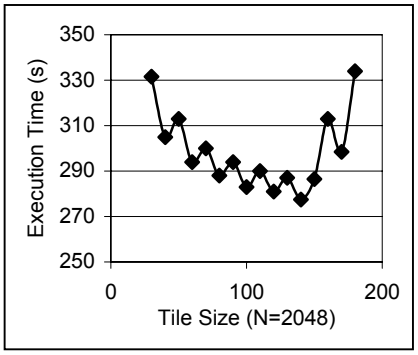

Figure 14: USTR Optimization, tile size selection Pentium III, Linux

5. Conclusions and Future Work

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 [1]. 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.

6. References

[1] ADVISOR Project. http://advisor.usc.edu/.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Menlo Park, California, 1974.

[3] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June, 1997.

[4] S. Chatterjee and S. Sen. Cache Efficient Matrix Transposition. In *Proc. of International Symposium on High Performance Computer Architecture*, January 2000.

[5] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999.

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

[7] M. Cosnard, P. Quinton, Y. Robert, and M. Tchuente (editors). *Parallel Algorithms and Architectures*. North Holland, 1986.

[8] P. Diniz. USC ISI, Personal Communication, March, 2001.

[9] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *Proc. of 40th Annual Symposium on Foundations of Computer Science*, 17-18, New York, NY, USA, October, 1999.

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

[11] H. Kwak, B. Lee, A. R. Hurson, S. Yoon and W. Hahn. Effects of Multithreading on Cache Performance. *IEEE Transactions on Computers*, Vol. 48, No. 2, February 1999.

[12] M. S. Lam, E. E. Rothberg, and M. E. Wolf. 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, California, April, 1991.

[13] N. Park, D Kang, K Bondalapati, and V. K. Prasanna. Dynamic Data Layouts for Cache-conscious Factorization of the DFT. In *Proc. of International Parallel and Distributed Processing Symposium*, May 2000.

[14] D. A. Patterson and J. L. Hennessy. *Computer Architecture A Quantitative Approach*. 2nd Ed., Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[15] F. Rastello and Y. Robert. Loop Partitioning Versus Tiling for Cache-Based Multiprocessor. In *Proc. of International Conference Parallel and Distributed Computing and Systems*, Las Vegas, Nevada, 1998.

[16] S. Sen, S. Chatterjee. Towards a Theory of Cache-Efficient Algorithms. In *Proc. of Symposium on Discrete Algorithms*, 2000.

[17] SPIRAL Project. http://www.ece.cmu.edu/~spiral/.

[18] X. Tang, R. Ghiya, L. J. Hendren, and G. R. Gao. Heap Analysis and Optimizations for Threaded Programs. In *Proc. of International Conference on Parallel Architectures and Compilation Techniques*, pages 14--25, San Francisco, Calif., November 1997.

[19] J. D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, Rockville, Maryland, 1983.

[20] D. A. B. Weikle, S. A. McKee, and Wm.A. Wulf. Caches As Filters: A New Approach To Cache Analysis. In *Proc. of Grace Murray Hopper Conference*, September 2000.

[21] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. *High Performance Computing and Networking*, November 1998.