# AFRL-IF-WP-TR-2003-1541

# ALGORITHMS FOR DATA INTENSIVE APPLICATIONS ON INTELLIGENT AND SMART MEMORIES

**Viktor K. Prasanna**

**University of Southern California**
**Departments of Contracts and Grants**
**University Park**
**Los Angeles, CA 90089-1147**

**MARCH 2003**

**Final Report for 15 June 1999 – 31 January 2003**

**Approved for public release; distribution is unlimited.**

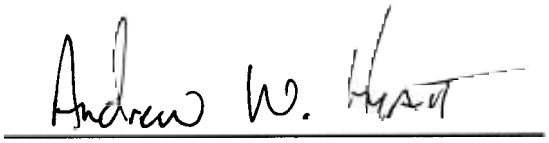
**INFORMATION DIRECTORATE**
**AIR FORCE RESEARCH LABORATORY**
**AIR FORCE MATERIEL COMMAND**
**WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334**

# NOTICE

USING GOVERNMENT DRAWINGS, SPECIFICATIONS, OR OTHER DATA INCLUDED IN THIS DOCUMENT FOR ANY PURPOSE OTHER THAN GOVERNMENT PROCUREMENT DOES NOT IN ANY WAY OBLIGATE THE US GOVERNMENT. THE FACT THAT THE GOVERNMENT FORMULATED OR SUPPLIED THE DRAWINGS, SPECIFICATIONS, OR OTHER DATA DOES NOT LICENSE THE HOLDER OR ANY OTHER PERSON OR CORPORATION; OR CONVEY ANY RIGHTS OR PERMISSION TO MANUFACTURE, USE, OR SELL ANY PATENTED INVENTION THAT MAY RELATE TO THEM.

THIS REPORT IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS, IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.



ANDREW W. HYATT, 2Lt, USAF
Program Monitor

ROBERT A. EHRET, Chief
Collaborative Simulation Technology & Applications Branch
Information Systems Division
Information Directorate

Do not return copies of this report unless contractual obligations or notice on a specific document require its return.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS**.

| 1. REPORT DATE *(DD-MM-YY)* | 2. REPORT TYPE | 3. DATES COVERED *(From - To)* |
|---|---|---|
| March 2003 | Final | 06/15/1999 – 01/31/2003 |

**4. TITLE AND SUBTITLE**
ALGORITHMS FOR DATA INTENSIVE APPLICATIONS ON INTELLIGENT AND SMART MEMORIES

**5a. CONTRACT NUMBER**
F33615-99-1-1483

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**
69199F

**6. AUTHOR(S)**
Viktor K. Prasanna

**5d. PROJECT NUMBER**
ARPI

**5e. TASK NUMBER**
FS

**5f. WORK UNIT NUMBER**
06

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
University of Southern California
Departments of Contracts and Grants
University Park
Los Angeles, CA 90089-1147

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

Information Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson Air Force Base, OH 45433-7334

DARPA / IPTO
3701 Fairfax Drive
Arlington, VA 22203-1714

Office of Naval Research
4520 Executive Drive, Suite 300
San Diego, CA

**10. SPONSORING/MONITORING AGENCY ACRONYM(S)**
AFRL/IFSD

**11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)**
AFRL-IF-WP-TR-2003-1541

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution is unlimited.

**13. SUPPLEMENTARY NOTES**
Report contains color. Report contains software or code.

**14. ABSTRACT** ***(Maximum 200 Words)***

During the course of this contract research was focused on techniques for improving processor-memory efficiency and performance on traditional, cache-based processors, and on state-of-the-art configurable cache and Processing-in-Memory designs. Work was done for matrix multiplication and the transitive closure stressmark, and cache performance optimizations were developed and shown to be effective. It was shown that these techniques can be used on a large class of algorithms. Significant contributions include the Unidirectional Space Time Representation (USTR) and a novel recursive implementation of the Floyd-Warshall algorithm. Using the USTR it is possible to quickly generate cache-friendly implementations of a large class of algorithms. The generalized split temporal/spatial cache architecture was defined as an abstraction of several application-controlled advanced cache architectures. Various graph algorithms were studied to identify the inefficiencies in the memory hierarchy and develop explicit cache management algorithms. Analytical performance estimations were derived for several problems, and simulations of optimized applications showed reduced memory traffic and improved average memory access times. A high-level simulator was implemented for Processing in Memory (PIM) architectures. The simulator allows faster development cycles and a better understanding of how an application will port to other PIM architectures. The simulator was verified against low-level simulation results.

**15. SUBJECT TERMS**
Performance, Modeling, Prediction, Algorithms, Optimization, Memory Hierarchy, Cache, Transitive Closure, Matrix Multiplication, PIM, Application Directed, Explicit Cache Management

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT: | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON (Monitor) |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | Lt. Andrew Hyatt |
| Unclassified | Unclassified | Unclassified | SAR | 164 | 19b. TELEPHONE NUMBER *(Include Area Code)* (937) 904-9162 |

**Standard Form 298 (Rev. 8-98)**
Prescribed by ANSI Std. Z39-18

## Contents

***Section*** ***Page.***

**1 Summary Report....1**
**1.1 Introduction....3**
**1.2 Summary of ADVISOR Project Accomplishments....4**
**1.2.1 Introduction to Memory Performance Optimizations....4**
**1.2.2 Performance Improvements....11**
**1.2.3 Modeling and Simulation of Flexible Memory Architectures....12**
**1.2.4 Performance Modeling of Processing-in-Memory (PIM) Architectures....15**
**1.2.5 Final Stressmark Performance Improvements Report....18**
**1.3 Lessons Learned....19**
**1.4 Technology Transition....20**
**1.5 Papers Acknowledging this Contract....22**

**2 Copies of Papers Acknowledging this Contract....24**
**2.1 Application Directed Explicit Management for Advanced Cache Architectures...25**
**2.2 Performance Modeling of Interpretive Simulation of PIM Architectures and Applications....41**
**2.3 Optimizing Graph Algorithms for Improved Cache Performance....46**
**2.4 Cache-Friendly Implementations of Transitive Closure....89**
**2.5 Analysis of Memory Hierarchy Performance of Block Data Layout....101**
**2.6 Tiling, Block Data Layout, and Memory Hierarchy Performance....111**

**3 Final Stressmark Results Submission....141**

**4 Appendix: Source Code Information....153**

# Final Project Report

## Introduction

The objective of this project was to develop an algorithmic framework that enables effective and efficient mapping of data intensive applications onto Intelligent and Smart memory architectures, as well as traditional cache architectures. Intelligent memories integrate processing logic on the same chip as memory and support high bandwidth and low latency memory access to on-chip memory. Smart memory architectures provide the ability to adapt the hardware behavior by modifying the memory controllers to enhance cache and memory performance. Effective use of these novel features requires innovative mapping techniques in addition to the utilization of higher bandwidth and/or lower latency offered by these advanced architectures.

This report details the research accomplished in this project. In Section 1 we present a summary of the work accomplished, highlighting some of the results and approaches investigated. Section 2 contains copies of all papers published that acknowledge this contract. Section 3 contains the final stressmark results and analysis of the methods used to optimize various data-intensive stressmarks. Section 4 contains information about the source code, including methods for building the code, and the platforms for which the code is intended.

The CD included with this binder contains all of the source code used in the project, with instructions for building the code, and a soft copy of the complete report.

# Summary of ADVISOR Project Accomplishments

## Contents


**1 Introduction** **1**

**2 Summary of Accomplishments** **2**
- 2.1 Introduction to Memory Performance Optimizations . . . 2
  - 2.1.1 Unidirectional Space Time Representation . . . 3
  - 2.1.2 Tiled Implementation of the Floyd-Warshall algorithm . . . 4
  - 2.1.3 Data Layouts for Recursive Programs . . . 5
  - 2.1.4 Graph Matching . . . 6
  - 2.1.5 Block Data Layout Optimization . . . 7
  - 2.1.6 Cache-conscious data layout based on Perfect Latin Squares . . . 8
- 2.2 Performance Improvements . . . 9
- 2.3 Modeling and Simulation of Flexible Memory Architectures . . . 10
- 2.4 Performance Modeling of Processing-in-Memory (PIM) Architectures . . . 13
- 2.5 Final Stressmark Performance Improvements Report . . . 16

**3 Lessons Learned** **17**

**4 Technology Transition** **18**
- 4.1 Impulse project (Univ. of Utah) . . . 18
- 4.2 SLIIC and DIVA projects (USC/ISI) . . . 19
- 4.3 ATLAS project interactions . . . 19

**5 Papers Acknowledging this Contract** **20**

# 1 Introduction

The motivation for this work is what is commonly referred to as the processor-memory gap [7, 13, 2, 10]. While memory density has been growing rapidly, the speed of memory has been far outpaced by the speed of modern processors. Current latencies to memory are on the order of 100 processor cycles [11]. This phenomenon has resulted in severe application level performance degradation on high-end systems. This problem has been well studied for many applications, such as dense linear algebra problems [18], including matrix multiplication and FFT.

Achieving better overall performance by optimizing cache performance is a difficult problem. The performance of deep memory hierarchies present in most modern processors has been shown to differ significantly from predictions based on a single level of cache. Different miss penalties for each level of the memory hierarchy, as well as the Translation Lookaside Buffer(TLB), affect the effectiveness of cache-friendly optimizations [11, 7]. These penalties vary among processors and cause large variations in the effectiveness of cache performance optimizations.

The area of graph problems is fundamental in a wide variety of fields, most notably network routing, distributed computing, and computer aided circuit design. Network routing in particular is a rapidly growing problem with the explosion of the Internet. Routing tables are growing in size and the frequency of updates is pushing the limits of current routers [16]. Graph problems pose unique challenges to improving cache performance due to their irregular data access patterns [5]. These challenges often cannot be handled using standard cache-friendly optimizations. The focus of this research is to develop methods of meeting these challenges.

The objective of this project was to develop an algorithmic framework that enables effective and efficient mapping of data intensive applications onto Intelligent and Smart memory architectures [1]. Intelligent memories integrate processing logic on the same chip as memory and support high bandwidth and low latency memory access to on-chip memory [13, 2, 17]. Smart memory architectures provide the ability to adapt the hardware behavior by modifying the memory controllers to enhance cache and memory performance [12, 14]. Effective use of these novel features requires innovative mapping techniques in addition to the utilization of higher bandwidth and/or lower latency offered by these advanced architectures.

A suite of data intensive kernels or stressmarks designed to stress the memory hierarchy was provided by the Data Intensive Systems Program [8]. We have explored various optimizations of these stressmarks, including cache-friendly data layouts such as block layouts and recursive layouts, and we have developed simulators that can model these applications on cutting-edge processor architectures.

During the course of this project the focus (Figure 1) has been on techniques for improving and understanding cache performance on traditional, cache-based processors. The majority of the work was done for the Data Intensive Systems stressmark package. It includes transitive closure stressmark, BiConjugate Gradient, FFT, pointer following, and others. It was also shown that these techniques can be used on a large class of algorithms.

Significant contributions include the Unidirectional Space Time Representation (USTR) and a tiled and a novel recursive implementation of the Floyd-Warshall algorithm. Using the USTR it is possible to quickly generate cache-friendly implementations of a large class of algorithms. Simulators and stressmark implementations were developed for flexible memory architectures (FMA) with split caches, Processor-Integrated-Memory (PIM) systems, and a recursive simulator for massively parallel systems such as the IBM BlueGene.

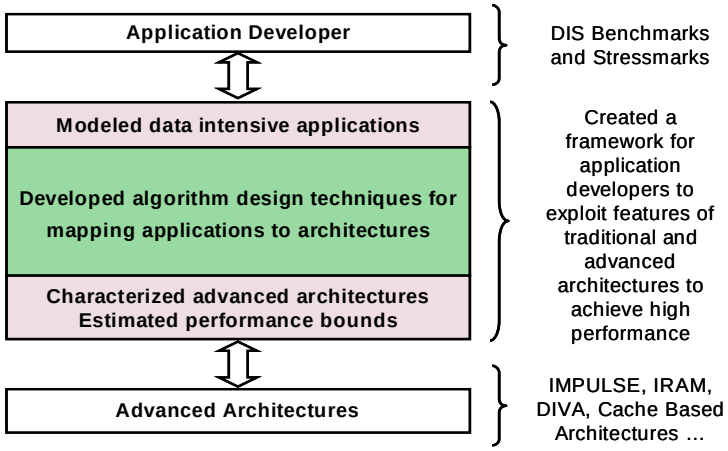



Figure 1: Accomplishments of the Advisor project

# 2 Summary of Accomplishments

## 2.1 Introduction to Memory Performance Optimizations

For most regular and irregular applications, cache and TLB behavior has a significant effect on performance. Memory in modern microprocessor systems is composed of a hierarchy of small, fast memories fed by large, slower memories (Figure 2). The Cache is a small memory area where recently and often used data is stored on the assumption that it will be used again soon. This prevents long delays in accessing the main memory. The TLB, or Translation Lookaside Buffer, is an even smaller, special purpose memory that holds a limited amount of information to help applications convert their memory references into addresses in the physical memory system [7]. State-of-the-art data layouts and control transformations attempt to minimize capacity misses and interference misses in the cache hierarchy. Control optimizations like tiling reduce the working set and improve cache behavior by reducing capacity misses.

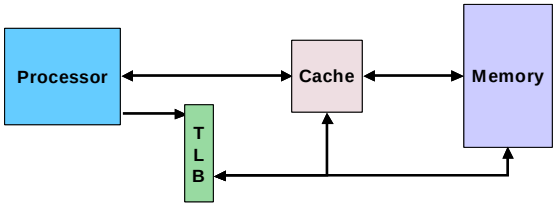



Figure 2: Memory hierarchy of a general microprocessor system

We have performed analysis of TLB, cache and DRAM behavior for various algorithms like matrix multiplication and a class of graph algorithms. Our work extends previous research on analyzing cache performance by taking a unified approach towards the tradeoffs involved in optimizing data and control transformations for various levels of the cache hierarchy. We have demonstrated speedups for these algorithms by selectively applying *novel optimizations* (in addition to a set of common optimizations) based on cache behavior analysis. We formalized the lessons learned in the form of rules for a compiler framework for loop optimizations in data intensive computations.

### 2.1.1 Unidirectional Space Time Representation

The Floyd-Warshall algorithm is a dynamic programming algorithm [5], which computes a series of $N$, $N \times N$ 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 for each $(i,j)$th value. Dijkstra's algorithm is designed to solve the single-source shortest path problem. It does this by repeatedly extracting from a priority queue $Q$ the nearest vertex $u$ to the source, given the distances known thus far in the computation (Extract-Min operation). Once this nearest vertex is selected, all vertices $v$ that neighbor $u$ are updated with a new distance from the source (Update operation).

Transitive closure, as an irregular problem, poses unique challenges to improving cache performance, challenges that often cannot be handled by standard cache-friendly optimizations.

A number of approaches have been taken to address the Transitive Closure stressmark. We have evaluated various approaches to optimize its performance with respect to processing time and processor-memory traffic. In this area we have also considered the single source shortest path problem, the minimum spanning tree problem and the problem of graph matching as these are related graph problems.

The *Unidirectional Space Time Representation* (USTR) was developed to uniquely address the complexities of transitive closure.

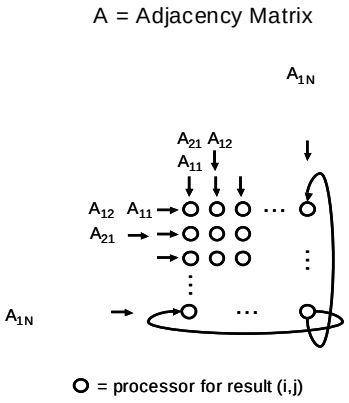



Figure 3: Simple USTR for Transitive Closure

First we will explain what we mean by a space-time representation. 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 result as a computational element (CE) in an $N \times N$ array, for example, the multiply-add operations required in a matrix multiply. Referring to Figure 3, 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 $N \times N$ array of CEs.

Referring to Figure 3 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 $(i,j)$th 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. 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.

In general the following to any problem that can be solved using an Unidirectional Space Time representation.

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

The USTR implementation of the Floyd-Warshall algorithm was shown in Simplescalar simulations to give a 30x decrease in level-2 cache misses (based on the Pentium III architecture). This implementation out-performed the best compiler optimizations by a factor of 2 on the Pentium III, Alpha, and MIPS R12000 architectures. It can also be shown that the USTR implementation of the Floyd-Warshall algorithm is asymptotically optimal with respect to processor-memory traffic. These results are detailed in "Cache Friendly Implementations of Transitive Closure" by Michael Penner and Viktor Prasanna, which appeared in the proceedings of PACT 2001, and is contained in Section 2. A novel recursive implementation as well as a new blocked implementation of the Floyd-Warshall algorithm have also been developed. In initial experiments these implementations show a 10x improvement over the compiler optimized implementation on the MIPS R12000. They also show between 3x and 6x for the Pentium III, Alpha, and SUN ULTRASPARC III. The source code for these optimizations is contained in Section 4.

### 2.1.2 Tiled Implementation of the Floyd-Warshall algorithm

The basic goal of tiling is to reduce the work set size so that the problem will fit into the cache. Through our technique we satisfy the data-dependencies by reordering the smaller problems or tiles. This technique showed up to 10x improvement for the Floyd-Warshall algorithm. This implementation was also shown to be asymptotically optimal with respect to processor-memory traffic.

Compiler groups have used tiling to achieve higher data reuse in looped code. Unfortunately, the data dependencies from one $k$-loop to the next in the Floyd-Warshall algorithm make it impossible for current compilers including research compilers to perform 3 levels of tiling. In order to tile the outermost loop we must cleverly reorder the tiles in such a way that satisfies data dependencies from one $k$-loop to the next as well as within each $k$-loop.

Consider the following tiled implementation of the Floyd-Warshall algorithm. Tile the problem into $B \times B$ tiles. During the $k$th block iteration, update first the $(k,k)$th tile, then the remainder of the k th row and $k$th column, then the rest of the matrix. Figure 4 shows an example matrix tiled into a 4x4 matrix of blocks. Each block is of size $B \times B$. During each outermost loop, we would update first the black tile representing the $(k,k)$th tile, then the grey tiles, then the white tiles. In this way we satisfy all dependencies from each $k$th loop to the next as well as all dependencies within each $k$th loop.

*Theorem: The new tiled implementation of the Floyd-Warshall algorithm reduces the processor memory traffic by a factor of $B$ where $B^2$ is on the order of the cache size.*

For more information, see "Optimizing Graph Algorithms for Improved Cache Performance," in Proceedings of the International Parallel and Distributed Processing Symposium, April 2002. Joon-Sang Park, Michael Penner, and Viktor K. Prasanna.

### 2.1.3 Data Layouts for Recursive Programs

Recursive implementations have recently been used to increase cache performance. However, Floyd-Warshall has proven to be difficult to implement recursively, because the Floyd-Warshall algorithm not only contains all the dependencies present in ordinary matrix multiplication, but also additional dependencies that can not be satisfied by the simple recursive implementation of matrix multiply. We have developed a novel recursive implementation of the Floyd-Warshall algorithm, which appears in Figure 4. We also proved the correctness of the implementation and showed analytically that it reaches an asymptotically optimal amount of processor memory traffic.

```
Floyd-Warshall (A)
        {
         A11 = min(A11, A11+A11);
         A12 = min(A12, A11+A12);
         A21 = min(A21, A21+A11);
         A22 = min(A22, A21+A12);
         A22 = min(A22, A22+A22);
         A21 = min(A21, A22+A21);
         A12 = min(A12, A12+A22);
         A11 = min(A11, A12+A21);
        }
```

(a)

```
FWR (A, B, C)
        {
          if (not base case)
                {
                FWR(A11, B11, C11);
                FWR(A12, B11, C12);
                FWR(A21, B21, C11);
                FWR(A22, B21, C12);
                FWR(A22, B22, C22);
                FWR(A21, B22, C21);
                FWR(A12, B12, C22);
                FWR(A11, B12, C21);
                }
           else
              {
        /* run standard Floyd-Warshall */
           ...
                }
         }
```

(b)

Figure 4: Floyd-Warshall Source (a) Base case algorithm (b) Recursive algorithm

The Z-Morton layout (Figure 5(c)) has been used to match the data layout to the access pattern. The Z-Morton ordering is a recursive layout defined as follows: Divide the original matrix into 4 quadrants and lay these tiles in memory in the order NW, NE, SW, SE. Recursively divide each quadrant until a limiting condition is reach. This smallest tile is typically laid out in either row or column major fashion Elements laid out row-wise inside blocks.

Through experiments on four different architectures we show that our Block Data Layout (Figure 5(b)) performs equally as well as the Z-Morton layout for recursive programs. The Z-Morton layout has also been used as a non-linear array layout for tiled applications. In this context, we show that our Block Data layout performs equally as well and significantly decreases the index computation costs.

We showed up to 2x improvement when we set the base case such that the base case would utilize more of the cache closest to the processor. Once we reached a problem size $B$, where $B^2$ is on the order of the cache size, we execute a standard iterative implementation of the Floyd-Warshall algorithm. This improvement varied from one machine to the next and is due to the decrease in the overhead of recursion. We have shown that the number of recursive calls in the recursive algorithm is reduced by a factor of $B^3$ when we stop the recursion at a problem of size $B$.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
| 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |

(a) Row-major layout

| 0 | 1 | 4 | 5 | 8 | 9 | 12 | 13 |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 7 | 10 | 11 | 14 | 15 |
| 16 | 17 | 20 | 21 | 24 | 25 | 28 | 29 |
| 18 | 19 | 22 | 23 | 26 | 27 | 30 | 31 |
| 32 | 33 | 36 | 37 | 40 | 41 | 44 | 45 |
| 34 | 35 | 38 | 39 | 42 | 43 | 46 | 47 |
| 48 | 49 | 52 | 53 | 56 | 57 | 60 | 61 |
| 50 | 51 | 54 | 55 | 58 | 59 | 62 | 63 |

(b) Block data layout

| 0 | 1 | 4 | 5 | 16 | 17 | 20 | 21 |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 7 | 18 | 19 | 22 | 23 |
| 8 | 9 | 12 | 13 | 24 | 25 | 28 | 29 |
| 10 | 11 | 14 | 15 | 26 | 27 | 30 | 31 |
| 32 | 33 | 36 | 37 | 48 | 49 | 52 | 53 |
| 34 | 35 | 38 | 39 | 50 | 51 | 54 | 55 |
| 40 | 41 | 44 | 45 | 56 | 57 | 60 | 61 |
| 42 | 43 | 46 | 47 | 58 | 59 | 62 | 63 |

(c) Morton data layout

Figure 5: Various data layouts: block size $2 \times 2$ for (b) and (c)

### 2.1.4 Graph Matching

Graph matching is an important problem that finds uses in pattern recognition for computer vision, face recognition and tracking of objects.

Graph matching forces algorithm designers to deal with dependencies that require possible examination of the entire graph during each step of the computation. By partitioning the graph into sub-graphs that will fit into the cache and finding the local maximum we can drastically reduce total amount of work for the entire graph (Figure 6). Experimental results show performance improvements are highly dependent on the density of the graph. For dense graphs a very good match can be found in the initial tiled phase and execution time can be improved by as much as 16x. Average performance improvements ranged between 2x and 6x (Figure 7).

```
CacheFriendlyMatching(G)
{
        Partition G into g[1], g[1], …, g[p];
        For i = 1  to p
                m[i] = FindMatching(g[i], Ø);
        M = MergeAll(m);
        M = FindMatching(G, M);
        return M;
}
```

(a)

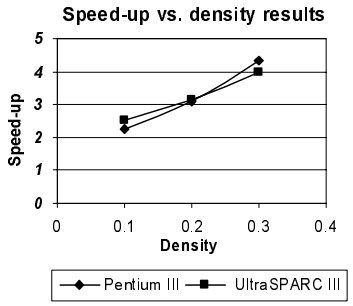
```
FindMatching(G, M)
{
    while (there exists an augmenting path)
    {
        increase |M| by one using the augmenting path;
    }
    return M;
}
```

(b)

Figure 6: Pseudocode for Graph Matching techniques

Our technique is not just an alternative way of finding a good starting matching, although it may seem like, but a general framework for improving performance of algorithms. For example, our technique allows additional enhancement in the presence of any greedy procedure for finding a good starting point by regarding the greedy procedure plus augmenting as a whole algorithm and applying our technique.

Roughly speaking, the complexity of our new implementation of matching algorithm depends on the graph-partitioning algorithm used in the first stage. The bound of the cardinality of a matching obtained at the first stage determines the overall complexity and the partitioning algorithm affects the bound. Also the complexity of partitioning algorithm itself affects the overall complexity our algorithm if it becomes the dominant factor.




| Cache Performance | | |
|---|---|---|
| | Baseline | New impl. |
| Accesses | 853 | 578 |
| Misses | 127 | 32 |
| Miss Rate | 14.86% | 5.56% |
| (Input: 8K nodes, 0.1 density) | | ($10^6$) |

Figure 7: Graph Matching Optimization Results

We presented a paper, "Optimizing Graph Algorithms for Improved Cache Performance", at the 16th annual IEEE & ACM International Parallel and Distributed Processing Symposium (IPDPS 2002). This paper can be found in Section 2. Our graph matching strategy used combinations of a number of different techniques.

### 2.1.5 Block Data Layout Optimization

Several experimental studies were conducted on block data layout as a data transformation technique used in conjunction with tiling to improve cache performance.

To support multi-dimensional array representations in current programming languages, the default data layout is row-major or column-major, denoted as canonical layouts. Both row-major and column-major layouts have similar drawbacks. For example, consider a large matrix stored in row-major layout. Due to large stride, column accesses can cause cache conflicts. Further, if every row in a matrix is larger than the size of a page, column accesses can cause TLB thrashing, resulting in drastic performance degradation. In block data layout, a large matrix is partitioned into sub-matrices. Each sub-matrix is a $B \times B$ matrix and all elements in the sub-matrix are mapped onto contiguous memory locations. The blocks are arranged in row-major order.

For standard matrix access patterns, we found an asymptotic lower bound on the number of TLB misses for any data layout:

$$\frac{2N^2}{\sqrt{P}S_{tlb}} \tag{1}$$

Block data layout achieves this bound. We have shown that block data layout improves TLB misses by a factor of $O(B)$ compared with conventional data layouts, where $B$ is the block size of the block data layout.

These results were published as "Analysis of Memory Hierarchy Performance of Block Data Layout" in the International Conference on Parallel Processing (ICPP 2002), August 2002. The paper can be found in Section 2.

We also applied the Block Data Layout to the transitive closure problem. 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 $O(N^3)$. 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 $O(\frac{N^3}{\beta^2})$ ) 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 the tiled version, we showed the working set of pages was $O(\beta)$.

The experimental results for the Block Data Layout optimization showed performance increases in the range of 5% to 15% on the Pentium III and approximately 40% on the Alpha.

The work on Block Data Layout for the transitive closure problem was published by Michael Penner and Viktor K. Prasanna as "Cache Friendly Implementations of Transitive Closure," in the Proceedings of International Conference on Parallel Architectures and Compilation Techniques, September 2001.

### 2.1.6 Cache-conscious data layout based on Perfect Latin Squares

The theory of Perfect Latin Squares (PLS) was originally co-developed by the PI in the context of parallel memory systems [9, 10]. Perfect Latin Squares were used as a mathematical framework for data distribution among parallel memory banks to minimize memory bank conflicts for array accesses. In the context of cache memories, we have applied the PLS methodologies to define data layouts to minimize cache conflicts in a uniprocessor memory hierarchy.

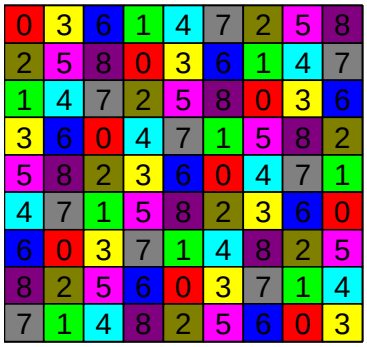



Figure 8: Perfect Latin Square

A Latin square of order $N$ is an $N \times N$ square composed with symbols from 0 to $n-1$ such that no symbol appears more than once in any row or in any column [9]. The rows are numbered from 0 to $N-l$, top to bottom. The columns are also numbered from 0 to $N-l$, left to right. The squares 9(a) and 9(b) shown below are examples of Latin squares of order 4.

We define a perfect Latin square of order $N^2$ as a diagonal Latin square of order $N^2$ such that no symbol appears more than once in any main subsquare. Hence, in a perfect Latin square, no symbol appears more than once in any row, in any column, in any diagonal or in any main subsquare.

For an $N^2 \times N^2$ matrix, a PLS-based mapping can be generated that provides conflict-free access to rows, columns, main diagonals, and major sub-squares. If the cache associativity is two or more, this mapping also generates conflict-free access to arbitrary sub-squares. Address computation is

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \\ 2 & 3 & 0 & 1 \\ 3 & 0 & 1 & 2 \end{pmatrix}$$

(a) Latin Square

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 2 & 3 & 0 & 1 \\ 3 & 2 & 1 & 0 \\ 1 & 0 & 3 & 2 \end{pmatrix}$$

(b) PLS

Figure 9: Latin Squares. In both squares, no symbol appears more than once in a single row or column. In (b) this is also true for diagonals and main subsquares

also an important issue in any irregular mapping technique. For the PLS based mapping, we have shown that address computation can be done in constant time.

PLS-based array layouts for a generic matrix access achieved up to $O(N^2)$ reduction in cache conflicts for column access and $O(N)$ reduction in cache conflicts for any major $N \times N$ subsquare access. The improvement is in comparison with the standard row-major layout. These improvements were demonstrated both theoretically and through simulations on the SimpleScalar architecture simulator.

## 2.2 Performance Improvements

We applied many optimization techniques for several problems over the course of our research. Figures 10,11,12,13,14,15 are some of our cache optimization results not presented elsewhere in this paper for transitive closure and graph matching.

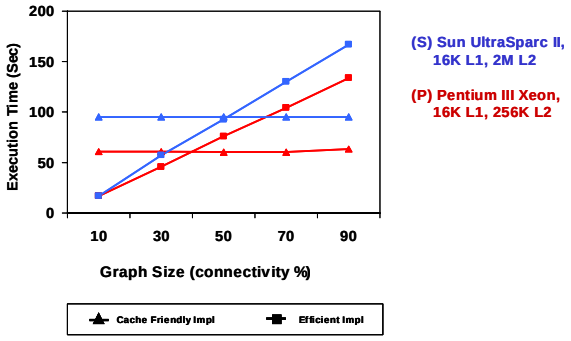



Figure 10: All-Pairs Shortest Path results comparing efficient implementations with our "cache-friendly" implementation

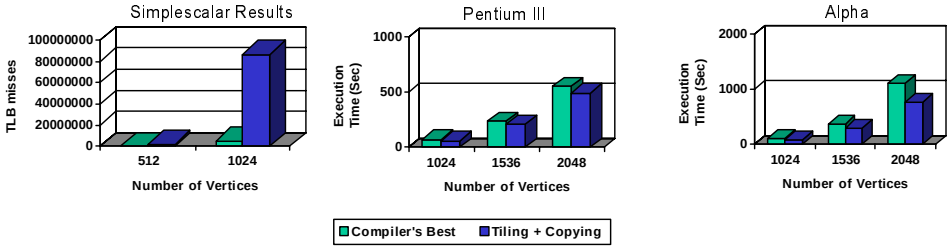



Figure 11: The Floyd-Warshall Algorithm: Tiling and Copying (Baseline)

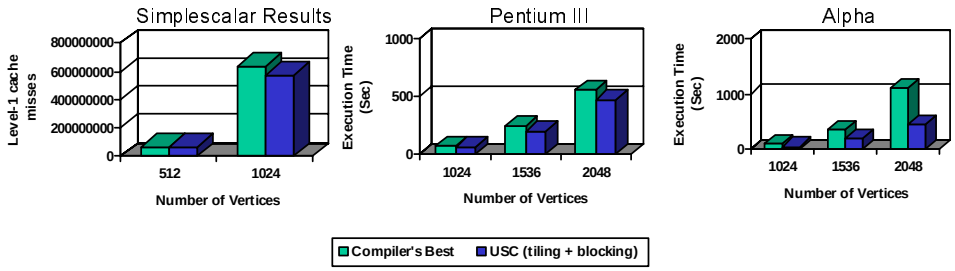



Figure 12: The Floyd-Warshall Algorithm: Tiling with Block Data Layout. Block layout reduces self-interference misses and TLB misses

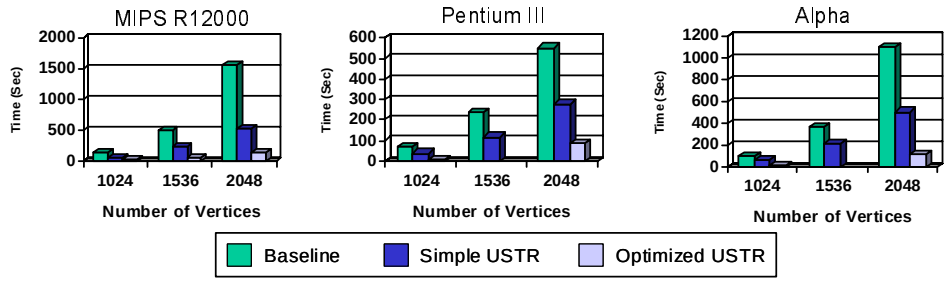



Figure 13: The Optimized USTR Implementation for the Floyd-Warshall Algorithm. The Optimized USTR provides approximately 6x performance improvement over the best standard Floyd-Warshall implementation

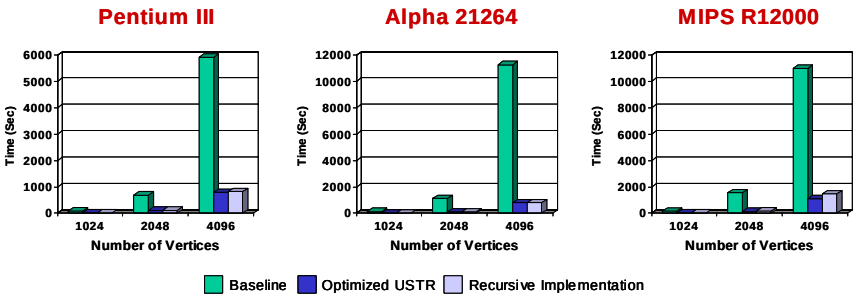



Figure 14: Floyd-Warshall Algorithm Optimizations. Replacing in-cache computation with standard Floyd-Warshall code gave 20% improvement over fully recursive implementation

## 2.3 Modeling and Simulation of Flexible Memory Architectures

Memory system performance is a key limiting factor in today's computer systems. Traditional cache replacement policies are often inefficient for modern application software. On the temporal side, data is not always placed in cache according to its temporal locality. In many applications, large data structures with low temporal reuse compete for cache space, although small data structures with high temporal reuse are desirable. Our work addressed inefficiencies directly through application software. On the spatial side, traditional architectures have difficulty dealing with data references of different spatial localities at the same time. Explicit management can solve this problem by separating data references into different caches.

The idea of explicit cache management as an architectural feature can be found in several modern

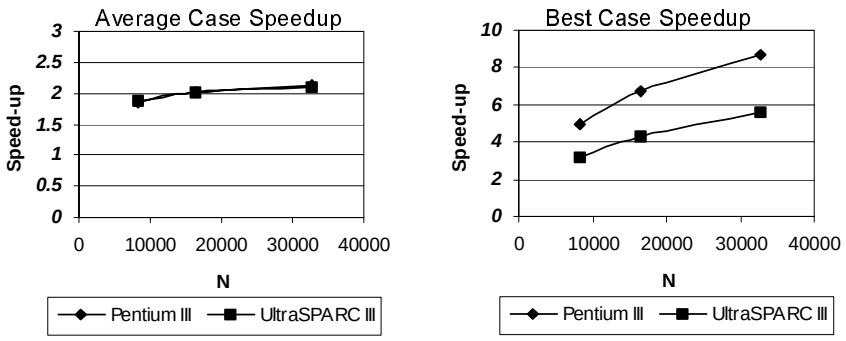



Figure 15: Graph Matching Optimizations: (a) Average Case: randomly generated graph (b) Best case: graph designed such that maximum matching is found in the first stage

processors: The cache in the Itanium architecture is divided into a "temporal structure" and a "spatial structure" at each level. A bit field in each load/store/prefetch instruction specifies which structure to use. Intel XScale has a 16K "Data Cache" and a 2K "Mini-Data Cache". A bit field in page table controls which cache to use. Intel StrongARM also has a similar design. UltraSPARC III Cu has a 2K prefetch cache in addition to the regular cache. A prefetch instruction can fetch data into one or both of the caches. HPL/PD, which is a reference architecture and simulated by Trimaran/IMPACT compiler infrastructure, has L1, L2 cache and a prefetch cache. It also uses a bit field in load/store instructions to control which cache to use. Similar architectures can also be found in several papers, such as Split Temporal/Spatial Cache and Dual Data Cache. [12, 14]

In these architectures, software can control hardware mechanisms of memory hierarchy directly. We call this explicit cache management. The name is used to distinguish from hardware only approaches, which are automatic (implicit). We define a generalized split temporal/spatial cache simulator architecture (Figure 16), to support explicit cache management algorithms in this paper. The idea of explicit cache management, however, is not limited to this type of architectures.

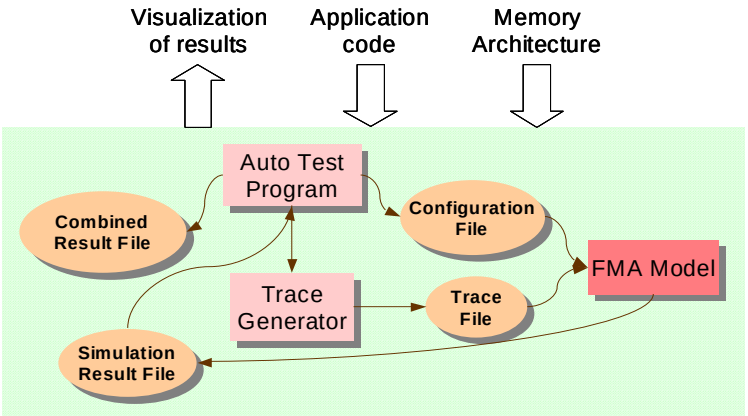



Figure 16: Flexible Memory Architecture (FMA) simulator architecture

We developed a unified Flexible Memory Architecture (FMA) model (Figure 17) that abstracts a flexible, parameterized memory hierarchy. The Intelligent and Smart memory architectures being developed by other Data Intensive Systems projects are a subset of the range of advanced memory architectures the model is capable of representing. The unified FMA model provides a framework

for efficient modeling, representation and manipulation of memory hierarchy parameters in an architecture-independent fashion. The uniqueness of our model (and the simulator based on it) lies in its ability to integrate the requirements of many DIS projects and provide a common platform for technology transfer.

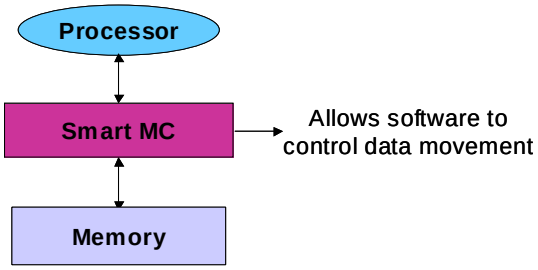



Figure 17: Smart Memory Architectures allow software to control data movement within memory hierarchy

In our work, we defined the generalized split temporal/spatial cache architecture as an abstraction of several advanced cache architectures. Individual problems were analyzed, inefficiencies in the memory hierarchy were identified and explicit cache management algorithms were developed. The problems include the sparse matrix vector multiplication problem from the conjugate gradient stressmark and problems from data structure and graph applications. According to our timing model, the average memory access time of the sparse matrix vector multiplication problem can be reduced by 21% to 62% over a broad range of cache configurations (Figure 18).

We published a technical report, "Application Directed Explicit Management for Advanced Cache Architectures", USC CENG October 2002.

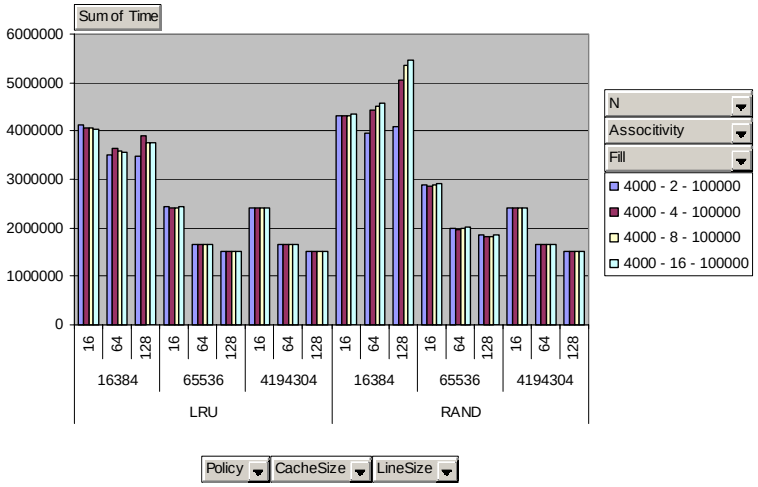



Figure 18: Flexible Memory Architecture (FMA) simulator results for the matrix stressmark

The unified FMA simulator based on the model is available, and the source is contained in the Appendix of this document. The purpose of the FMA simulator is to provide a first-order evaluation of the efficacy of algorithmic techniques to improve memory performance of (irregular) applications

on advanced memory architectures. Modeling a flexible, parameterized memory hierarchy provides a common platform for exploring the advanced memory architecture space without depending on availability of low-level simulators for the various memory architectures being developed in the DIS program. Also, from an algorithm designer's perspective, the simulator provides a rapid estimation of the relative performance of alternate data layouts, memory hierarchies, etc. without a time-consuming low-level simulation on a specific advanced memory architecture simulator. The current version of the FMA simulator allows the algorithm designer to specify the initial cache configurations and a set of alternate configurations (Figure 19). The alternate configurations are labeled with the position in the access string where the cache needs to be reconfigured. The parameters that can be dynamically varied are the line size, cache size, and associativity.

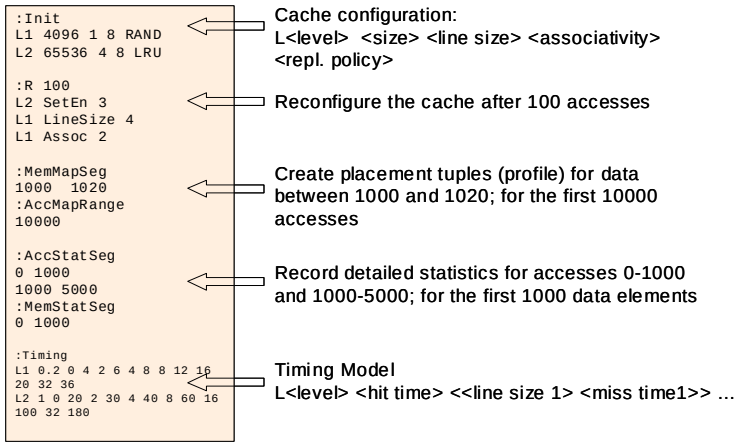



Figure 19: Sample FMA Configuration File

## 2.4 Performance Modeling of Processing-in-Memory (PIM) Architectures

Processing-In-Memory (PIM) systems achieve tremendous memory-processor bandwidth by combining microprocessors and memory together on the same chip substrate. Commodity microprocessors have steadily increased the size of their on-board caches– the speed and complexity of today's chips requires that instructions and data be immediately ready when requested by the processor. The PIM architecture attempts the opposite tactic; instead of pushing data to the processor, PIM moves the processors to the data. Instead of multiple levels of cache and a huge main memory, PIM places a cache-less, comparatively simple processor directly on each memory chip.

PIM architectures (Figure 20) offer an order of magnitude more processor-memory bandwidth, compared to traditional processor-memory architectures, without a cache hierarchy and the performance implications thereof. We defined a first-order PIM model, and a parameterized simulator for various applications on various PIM architecture are available. Applications modeled include 2-D FFT, BiConjugate Gradient, corner turn, matrix multiply, transitive closure, and a molecular dynamics simulation. We also provided verification of the accuracy of the simulator for several applications, a sample of which is in Figure 22. Architecture models include the Berkeley VIRAM, USC-ISI DIVA, and a recursive model of the IBM BlueGene system. [6, 13, 17]

Our high-level simulator predicts the performance (execution time) of an application over various PIM or application configurations. The parameters needed by the simulator characterize the simulation instance. These include the number of PIM nodes, on-chip and off-chip bandwidth for

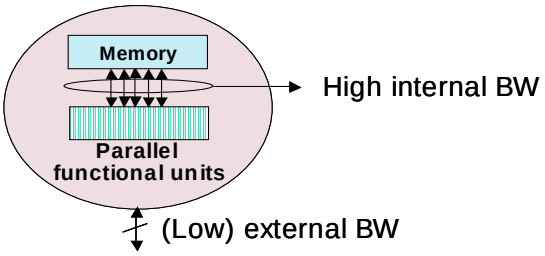



Figure 20: Processor-In-Memory (PIM) Architectures have processors and memory on a single chip.

each node, network topology, and application parameters such as block size, resolution, or problem size, and various replacement and coherence algorithms. The architecture modeled by the simulator is not restricted to any specific PIM implementation due to its completely parametrized nature. Our simulator is linked to a GUI which allows fast visualization of the effects of important parameters.

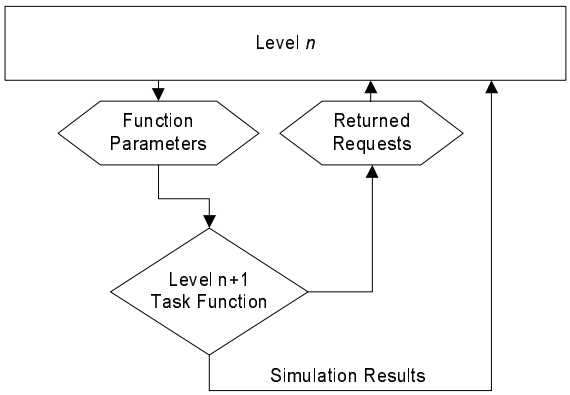



Figure 21: Recursive System Structure

Our paper, "Performance Modeling and Interpretive Simulation of PIM Architectures and Applications," was presented at the 8th annual ACM Euro-Par Parallel and Distributed Processing conference, 2002.

We continued with the PIM simulation framework and expanded it with support for massively parallel systems, such as the IBM BlueGene/L. Instead of determining the performance of a system as composed of discrete nodes at what abstracts to be a single level, we form the nodes into groups that are as homogeneous as possible. For each group, we create a parameter file which defines variables such as network topology, latencies in the network, and computational speeds. The file also defines the behavior of the system at that level, including what sort of computational/communications operations need to be executed by that node or level. The simulator then runs the parameter files "recursively," in that each parameter file describes a collection of nodes, each type of which have their own parameter file and simulation process (Figure 21).

The performance modeling system can capture a wide range of complex behaviors without defining distinct simulators for each level, and without defining away the variations that makes each level of hierarchy unique. Moreover, we do this with a simulator that functions at a high enough level so as to produce results quickly, on the order of seconds, and produce results sufficient for optimization of data placement, network topology, and overall structure of the system.

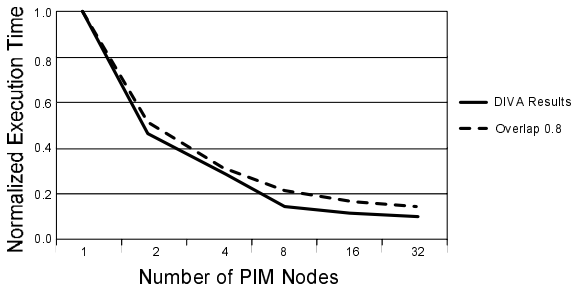



Figure 22: Comparison between high-level simulated BiConjugate Gradient on DIVA architecture and DIVA team's low-level simulation results

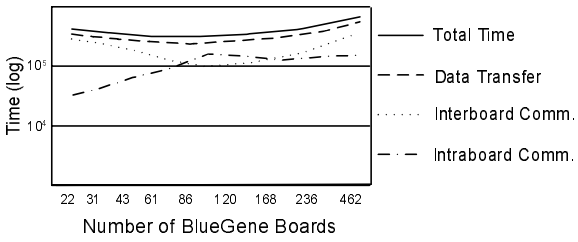



Figure 23: A sample of execution results for the molecular dynamics application on the BlueGene/L model

The host can send and receive data in a unicast or multicast fashion, either over a bus or a high-bandwidth, switched network. The bus is modeled as a single datapath with parameterized bus width, startup time, and per element transmission time.

The expanded recursive simulator offers robust topology and congestion management tools. Network topology is defined between lower-level simulation blocks. Congestion is handled at the network topology level. The simulator provides various ways of defining communication cost. The most detailed and most time-consuming of the choices is a least-cost path routing which also records the effect of congestion during a time step. The simulator collects all of the communication calls that are specified in the script to occur during a time interval and then adds them to the network in an arbitrary fashion. An 'n' deep queue is assigned the same cost as having to traverse an unutilized 'n' hop path. A message will follow the path through the network with the lowest hop number, as determined by application of Dijkstra's algorithm. In this setup, the first message sent will have a time proportional to the number of network links traversed, and that number will be the fastest possible path. The last message in a time step, conversely, may follow an unpredictable path and have a significantly higher time to completion.

A paper on this work, "Recursive Simulation for High-Level Performance Estimation of Massive Systems," was submitted to IPDPS 2003 and ICS03. This paper details the "recursive" method of high-level simulation for Massively Parallel systems.

The source code for the simulator, and both of the papers, are in Sections 2 and 4 of this binder.

## 2.5 Final Stressmark Performance Improvements Report

In June 2002 we submitted a report of our results to Joe Musmanno, the stressmark suite lead designer for the Data Intensive Systems program[1]. The report is in Section 2 of this binder. Significant contributions include the Unidirectional Space Time Representation (USTR) and a novel recursive implementation of the Floyd-Warshall algorithm. We found the keys to improve the performance of the memory system are as follows: increase data reuse, decrease cache conflicts, and decrease cache pollution. The techniques that we use to achieve these ends can be categorized as data layout optimizations and data access pattern optimizations. Some of the highlights of our work include tiling layouts that provide up to 10x improvement for the Floyd-Warshall algorithm, and the USTR that reduces level-2 cache misses by up to 30x (based on the Pentium III architecture).

We submitted stressmark results for several novel optimizations of the Floyd-Warshall algorithm and Dijkstra's algorithm:

| | |
|---|---|
| Floyd-Warshall | Baseline Approach |
| Floyd-Warshall | Tiling and Copying |
| Floyd-Warshall | Tiling and the Block Data Layout |
| Floyd-Warshall | Simple USTR |
| Floyd-Warshall | Optimized USTR |
| Floyd-Warshall | Recursive USTR |
| Dijkstra's | Baseline Approach |
| Dijkstra's | Cache-friendly Implementations |

Each of the previous stressmarks were run on the following architectures:

| | |
|---|---|
| Alpha 21264 | uniprocessor 500MHz with 512MB main memory, 64K L1 data cache, 4MB L2 |
| MIPS R12000 | 64 processor 300MHz with 16GB shared memory, 32K L1, 8 MB L2 |
| Pentium III | 4 processor 700MHz with 32K L1, 1MB L2 |
| UltraSPARC III | 2 processor 750 MHz SUN Blade 1000 with 1GB shared memory, 64K L1, 8MB L2 |

| | Efficient | Cache Friendly |
|---|---|---|
| Computational Complexity | $N*(N+E)*\lg(N)$ | $N^3$ |
| Processor-Memory traffic | $N*(N+E)*\lg(N)$ | $N^3$ |
| Data level-1 cache misses | 6,199,690,069 | 2,410,185,663 |
| Data level-2 cache misses | 2,760,091,657 | 907,142,166 |
| Data TLB misses | 32,402,464 | 8,414,663 |

Figure 24: Dijkstra's Algorithm Comparison: Theoretical bounds and SimpleScalar results based on Pentium III architecture. We observed significant improvements for dense graphs with density 50%

[1]The stressmark suite was developed by the Atlantic Aerospace Electronics Corporation, in conjunction with The Boeing Company and ERIM International, Inc. The suite is available at http://www.aaec.com/projectweb/dis/

| | Baseline | Tiled | Tiling + Block Data Layout | Simple USTR |
|---|---|---|---|---|
| Computational complexity | $N^3$ | $N^3$ | $N^3$ | $N^3$ |
| Processor-memory traffic | $N^3$ | $N^3$ | $N^3$ | $N^3/B$ |
| Data Level-1 cache misses | 2.72 | 2.12 | 1.94 | 2.76 |
| Data Level-2 cache misses | 1.81 | 1.85 | 1.84 | 0.057 |
| Data TLB misses | 0.018 | 0.22 | 0.019 | 0.016 |

Figure 25: Summary of Floyd-Warshall Results: Theoretical bounds and SimpleScalar results based on Pentium III architecture

In addition to the stressmark results we also presented a simulation tool for the high-level parameterized performance estimation of PIM architectures. Some of the results of our experiments on the tool for various architectures were included. The simulator currently has models of the Berkeley VIRAM and the ISI DIVA architectures, and can evaluate performance for the following applications:

BiConjugate Gradient
2-D FFT
1-D FFT
Corner Turn
Matrix Multiply
Transitive Closure

We also provided input to Joe Musmanno regarding the evaluation of the stressmark results. The stressmark results submitted are in Section 3 of this document.

# 3 Lessons Learned

The speed of modern processors is increasing at a rate of roughly 60% per year while the speed of memory is increasing at a rate of roughly 7% per year. This difference is often referred to as the processor-memory gap, and it causes the latency to memory as seen by the processor to increase significantly with each passing year. In order to hide this increasing latency, caches have been designed to take advantage of locality of reference; the fact that once an element is accessed there is a good chance that it and/or elements near will be accessed in the near future. The cache is much smaller than main memory and is placed much closer to the processor in terms of latency.

Modern processors are including more levels of cache, each level larger in size and farther from the processor in terms of latency. Invariably the processor will access data that is not in the cache and this will result in a cache miss. Cache misses can be categorized into one of three categories: cold misses, capacity misses, and conflict misses. A cold miss occurs the first time a data element is accessed. These misses are unavoidable. A capacity miss occurs if the working set of the application is larger than the cache. These misses can be avoided by either decreasing the working set or increasing the size of the cache. A conflict miss occurs if two or more data elements in the working set map to same place in the cache and the replacement of one results in a subsequent cache miss when that element is accessed. This type of miss can be avoided in a number

of ways including improved data access patterns, improved data layout, reducing the working set, etc.

Two other issues that should be addressed are cache pollution and TLB misses. TLB misses are similar to cache misses except that they refer to misses in the Translation Look-aside Buffer. They can be categorized the same as cache misses and reducing them follows a similar pattern. Cache pollution is a somewhat different issue. This refers to when a cache line is brought into the cache and only a small portion of it is used before it is pushed out of the cache. A large amount of cache pollution will increase the bandwidth requirement of the application, even though the application is not utilizing more data.

Based on this discussion, the keys to improve the performance of the memory system are as follows: increase data reuse, decrease cache conflicts, and decrease cache pollution. The techniques that we use to achieve these ends can be categorized as data layout optimizations and data access pattern optimizations. In our data layout optimizations we attempt to match the data layout to an existing data access pattern. For example, we use the Block Data Layout to match the access pattern of a tiled algorithm. In our data access pattern optimizations, we design both novel and trivial optimizations to the algorithm to improve the data access pattern. For example, we implemented both a tiled implementation and a novel recursive implementation of the Floyd-Warshall algorithm to improve the data access pattern. The techniques that we use are algorithmic in nature, meaning that we assume no control of the hardware or the operating system.

In Dijkstra's algorithm and Prim's algorithm, the largest data structure is the graph representation. An optimal representation, with respect to space, would be the adjacency-list representation. However, this involves pointer chasing when traversing the list. The priority queue has been highly optimized by various groups over the years. Unfortunately, the update operation is often excluded, as it is not necessary in such algorithms as sorting. The asymptotically optimal implementation that considers the update operation is the Fibonacci heap. Unfortunately this implementation includes large constant factors and did not perform well in our experiments.

Access to source code is obviously required for our optimizations. Changes to the source code are fairly minor and are most often isolated to the inner loop or to the loop structure of the transitive closure kernel. In some cases, such as when using the Block Data Layout or in the optimization to Dijkstra's algorithm, it may be necessary to change the data structure or data layout for the kernel. We achieved this by allocating additional space and copying the data into the correct format. Upon completion the result was copied back to the original format. Since transitive closure is an $O(N^3)$ complexity algorithm, copying $O(N^2)$ data required a very small amount of time relative to the total running time. For any optimization that requires copying, the running time given includes the time for copying. Possibly the most difficult task is choosing the appropriate block size for the tiled implementations. This was done experimentally on one problem size on each machine and the block size found was used for all problem size. ATLAS provides a technique for automatically performing this experimentation at compile time, and a similar approach could be developed for these implementations.

# 4 Technology Transition

## 4.1 Impulse project (Univ. of Utah)

We have interacted with the Impulse project with the eventual purpose of integrating our static/dynamic data layout techniques into the compiler framework being built for Impulse. The Impulse architecture currently supports only a few remapping functions that can be efficiently implemented in

a simple ALU. We have proposed new remapping functions for various applications that can also be implemented in Impulse.

### 4.2 SLIIC and DIVA projects (USC/ISI)

We have interacted with the SLIIC project with a view to collaborate in building their PIM simulator. Our interaction with the DIVA [6, 2] project at USC/ISI was ideally to obtain their PIM simulator for DIVA. We hoped also to gain an understanding of the issues that the DIVA team faces in their design, so that support can be incorporated into the structure of our parameterized PIM performance estimation tool. At the conclusion of our project, our high-level simulation was efficient and accurate, and the source has been released to the public domain.

### 4.3 ATLAS project interactions

ATLAS stands for Automatically Tuned Linear Algebra Software. ATLAS's purpose is to provide portably optimal linear algebra software. For all supported operations, ATLAS achieves performance on par with machine-specific tuned libraries [18].

We have studied block data layout as a data transformation technique used in conjunction with tiling to improve cache performance. In most cases, datalayout in ATLAS goes along the following way: After initialization, both matrix $A$ and $B$ are in column major. Before the computation begins, they copy matrix $A$ into a temporary buffer $pA$ and the data in $pA$ are in block data layout. All the following computations use data in $pA$ and the data layout of $pA$ is never changed again. For each column panel of matrix $B$, ATLAS copies the data into a temporary buffer, $pB$. (Data in $pB$ are in block data layout now). Then matrix $A$ is multiplied with this column panel. After the multiply, data in this column panel won't be needed again. Then ATLAS refills the buffer $pB$ with the next column panel of $B$ and goes on with the computation. Eventually both matrix $A$ and $B$ will be completely changed into block data layout.

We have provided a theoretical analysis for the TLB and cache performance of block data layout. Based on this analysis, we proposed an approach for block size selection that provides a tight range for optimal block size in using block data layout for dense linear algebra implementations on cache based machines. The key results of our work are as follows:

The optimal block size $B_{tc1}$ that minimizes the miss cost caused by L1 cache and TLB misses is given as

$$B_{tc1} \approx \sqrt{\frac{(\frac{2L_{1c}M_{tlb}}{P_v} + [2 + \frac{3L_{c1}+2L_{c1}^2}{S_{c1}}]H_2)S_{c1}}{4H_2}} \tag{2}$$

where $H_i$ is the cost of a hit in the $i^{th}$ level cache, $M_{tlb}$ is the penalty of a TLB miss, $S_{ci}$ is the size of the $i^{th}$ level cache, $L_{ci}$ is the line size of the $i^{th}$ level cache, and $P_v$ is the virtual page size.

The optimal block size for minimizing the total miss cost for all cache levels is given by

$$B_{tc1} \leq B_{opt} < \sqrt{S_{c1}} \tag{3}$$

This interval (shown in Figure 26 can effectively reduce the range of block sizes searched by the ATLAS optimization routines, greatly speeding up ATLAS optimization.

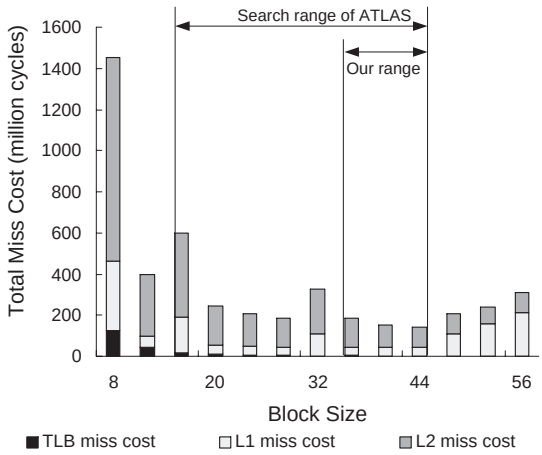



Figure 26: Total miss cost for 6-loop tiled matrix multiplication

# 5 Papers Acknowledging this Contract

Xi Wang and Viktor K. Prasanna, "Application Directed Explicit Management for Advanced Cache Architectures," Technical Report, Department of Electrical Engineering, University of Southern California, October 2002.

Z.K. Baker and Viktor K. Prasanna, "Performance Modeling and Interpretive Simulation of PIM Architectures and Applications," Euro-Par 2002, August 2002.

Joon-Sang Park, Michael Penner, and Viktor K. Prasanna, "Optimizing Graph Algorithms for Improved Cache Performance," In Proceedings of the International Parallel and Distributed Processing Symposium, April 2002.

Joon-Sang Park, Michael Penner, and Viktor K. Prasanna, "Optimizing Graph Algorithms for Improved Cache Performance," Technical Report, Department of Electrical Engineering, University of Southern California, 2002.

Michael Penner and Viktor K. Prasanna, "Cache Friendly Implementations of Transitive Closure," In Proceedings of International Conference on Parallel Architectures and Compilation Techniques, September 2001.

Neungsoo Park, Bo Hong, and Viktor K. Prasanna, "Analysis of Memory Hierarchy Performance of Block Data Layout," International Conference on Parallel Processing (ICPP 2002), August 2002.

Neungsoo Park, Bo Hong, and Viktor K. Prasanna, "Tiling, Block Data Layout, and Memory Hierarchy Performance" Technical Report, Department of Electrical Engineering, University of Southern California, November 2001.

# References

[1] ADVISOR project website. `http://advisor.usc.edu`.

[2] DIVA project website. `http://www.isi.edu/asd/diva/`.

[3] Impulse project website. `http://www.cs.utah.edu/impulse/`.

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

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Electrical and Computer Science Series. MIT Press, 1992. ISBN 0-262-03141-8.

[6] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, A. Srivastava, W. Athas, J. Brockman, V. Freeh, J. Park, and J. Shin. Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture. In *SC99*.

[7] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1996.

[8] Atlantic Aerospace Electronics Corporation in conjunction with The Boeing Company and ERIM International Inc. DIS stressmark suite. `http://www.aaec.com/projectweb/dis/`.

[9] K. Kim and V. K. Prasanna-Kumar. Perfect latin squares and parallel array access. In *Proceedings of the 16th annual international symposium on Computer architecture*. ACM Press, 1989.

[10] K. Kim and V. K. Prasanna-Kumar. Latin squares for parallel array access. In *IEEE Transactions on Parallel and Distributed Systems*, April 1993.

[11] Nihar R. Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck: Problems and solutions. *ACM Crossroads*, 1999.

[12] V. Milutinovic, M. Tomasevic, B. Markovi, and M. Tremblay. A new cache architecture concept: the split temporal/spatial cache. In *Electrotechnical Conference*, 1996.

[13] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, April 1997.

[14] M. Prvulovic, D. Marinov, Z. Dimitrijevic, and V. Milutinovic. Split temporal/spatial cache: A survey and reevaluation of performance. In *IEEE TCCA Newsletter*, July 1999.

[15] H. Sharangpani. Intel Itanium Processor Microarchitecture Overview. *Microprocessor Forum*, October 1999.

[16] William Stallings. *Data & Computer Communications*. Prentice Hall, 6th edition, 2000.

[17] IBM BlueGene Project Overview. `http://www.research.ibm.com/bluegene/`.

[18] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). *Proceedings of SC'98*, November 1998.

# 2 Papers Acknowledging this Contract

Xi Wang and Viktor K. Prasanna, *"Application Directed Explicit Management for Advanced Cache Architectures,"* Technical Report, Department of Electrical Engineering, University of Southern California, October 2002.

Z.K. Baker and Viktor K. Prasanna, *"Performance Modeling and Interpretive Simulation of PIM Architectures and Applications,"* In Proceedings of Euro-Par 2002, August 2002.

Joon-Sang Park, Michael Penner, and Viktor K. Prasanna, *"Optimizing Graph Algorithms for Improved Cache Performance,"* In Proceedings of the International Parallel and Distributed Processing Symposium, April 2002.

Joon-Sang Park, Michael Penner, and Viktor K. Prasanna, *"Optimizing Graph Algorithms for Improved Cache Performance,"* Technical Report, Department of Electrical Engineering, University of Southern California, 2002.

Michael Penner and Viktor K. Prasanna, *"Cache Friendly Implementations of Transitive Closure,"* In Proceedings of International Conference on Parallel Architectures and Compilation Techniques, September 2001.

Neungsoo Park, Bo Hong, and Viktor K. Prasanna, *"Analysis of Memory Hierarchy Performance of Block Data Layout,"* International Conference on Parallel Processing (ICPP 2002), August 2002.

Neungsoo Park, Bo Hong, and Viktor K. Prasanna, *"Tiling, Block Data Layout, and Memory Hierarchy Performance"*, Technical Report, Department of Electrical Engineering, University of Southern California, November 2001.

# Application Directed Explicit Management for Advanced Cache Architectures*

Xi Wang and Viktor K. Prasanna

University of Southern California, Los Angeles, USA
{xiw, prasanna}@usc.edu

Technical Report No. 02 - 09

Department of Electrical Engineering – Systems
University of Southern California
Los Angeles, California 90089-2562
213-740-4465

**Abstract:** In this paper, we demonstrate the effectiveness of application directed explicit cache management. We define the generalized split temporal/spatial cache architecture as an abstraction of several advanced cache architectures. We analyze individual problems, identify the inefficiencies in the memory hierarchy and develop explicit cache management algorithms. In our algorithms, the application software controls hardware mechanisms directly. To illustrate various optimizations, problems are chosen from regular, sparse, data structure and graph applications. Analytical performance estimations are derived for several problems. Simulations show reduced memory traffic and improved average memory access time. For example, in the sparse matrix vector multiplication problem, the average memory access time can be reduced by 21% to 62% over a broad range of cache configurations.

# 1. Introduction

Memory system performance is a key limiting factor in today's computer systems. Traditional cache replacement policies are often inefficient for modern application software. On the temporal side, data is not always placed in cache according to its temporal locality. In many applications, large data structures with low temporal reuse compete for cache space, although small data structures with high temporal reuse are desirable. Hardware [1][2][3][5][7][8] and compiler [4][6] based approaches have been proposed. This paper addresses inefficiencies directly from application software. On the spatial side, traditional architectures have difficulty dealing with data references of different spatial localities at the same time. Explicit management can solve this problem by separating data references into different caches.

When the performance of hardware is pushed to the limit, some burden is shifted to the software. EPIC architectures follow this path. Software based approaches can be further divided into two layers: compiler and application. Now compilers [4][6] are picking up the burden from hardware. However, compilers also have their limitations. This motivates us to go one step further to explore the application directed approach.

The idea of explicit cache management as an architectural feature can be found in several modern processors: The cache in the Itanium architecture [11] is divided into a "temporal structure" and a "spatial structure" at each level. A bit field in each load/store/prefetch instruction specifies which structure to use. Intel XScale [12] has a 16K "Data Cache" and a 2K "Mini-Data Cache". A bit field in page table controls which cache to use. Intel StrongARM also has a similar design. UltraSPARC III Cu [13] has a 2K prefetch cache in addition to the regular cache. A prefetch instruction can fetch data into one or both of them. HPL/PD [14], which is a reference architecture and simulated by Trimaran/IMPACT [15] compiler infrastructure, has L1, L2 cache and a prefetch cache. It also uses a bit field in load/store instructions to control which cache to use. Similar architectures can also be found in several papers, such as Split Temporal/Spatial Cache [1] and Dual Data Cache [6].

In these architectures, software can control hardware mechanisms of memory hierarchy directly. We call this explicit cache management. The name is used to distinguish from hardware only approaches, which are automatic (implicit). We define an abstract architecture, generalized split temporal/spatial cache architecture, to support explicit cache management algorithms in this paper. The idea of explicit cache management, however, is not limited to this type of architectures.

In the rest of this paper, we first define the generalized split temporal/spatial cache architecture in Section 2. Then we analyze individual problems, identify the inefficiencies and develop explicit management algorithms in Section 3. In our algorithms, application software controls hardware mechanisms directly. Simulation results and analysis are given in Section 4. More architectural issues are discussed in Section 5.

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

# 2. The Generalized Split Temporal/Spatial Cache Architecture

Our work is applicable to the architectures stated earlier in the introduction section. However, as they are not general-purpose, it is difficult to use them directly in an application directed approach. Architectures of real processors are often not formally defined. Architectures in literature are often specialized for a hardware approach. As we would like to make our explicit management algorithms applicable to a family of architectures and independent of specific architectural features, we define an abstract but realistic architecture: the generalized split temporal/spatial cache architecture (Figure 1). It is simple, free from implementation details and can cover many specific architectures.

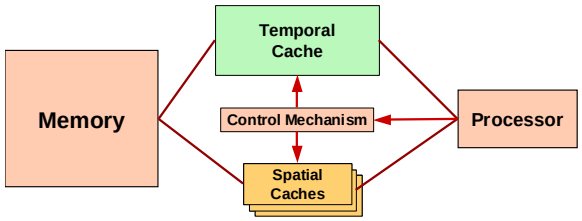



**Figure 1: Generalized split temporal/spatial cache architecture model**

**Architecture Model:** This architecture consists of one temporal cache and one or more spatial caches for data references. The function of the temporal cache is similar to that of a "regular" cache. It stores data with good temporal locality. The function of spatial caches is similar to prefetch or stream buffers. They are much smaller than the temporal cache and are used to handle data with poor temporal locality. They can have built-in prefetch mechanisms, which will be discussed in our algorithms.

In our architecture model, the temporal cache is large and simple; spatial caches are small but equipped with advanced prefetch mechanisms. A simple design of the temporal cache will make efficient use of physical resources for capacity and speed. On the other side, keeping spatial caches small would limit the side effects of complex designs.

**Cache Management:** A control mechanism is needed to determine which cache to use on each load/store operation. It can be either hardware controlled or software controlled. In this paper, software control is used, and we call this target cache control. For simplicity, we assume that data can be loaded into only one of the caches on a data reference, and a cache hit occurs only in the specified cache. We also assume write back with write allocation policy on all caches, and there is coherence protocol like in multi-processor systems.

# 3. Application Directed Explicit Management

## 3.1 Objectives of Our Optimization

Given the ability to control the generalized split temporal/spatial cache architecture, our application directed explicit management have three objectives:

**A. Tuning data placement in temporal cache toward optimal replacement:** In the application directed approach, an algorithm designer can predict the future. This prediction can be combined with a history based replacement policy. Specifically, like optimal replacement policy, if the predicted reuse distance is too large or there is no reuse at all, we can divert the data reference to a spatial cache. The predictions do not need to be precise or complete. Any imperfections can still be covered by traditional replacement policy in temporal cache. [3] gives an upper bound on hit ratio, which would be applicable here, although there are many differences. Better data placement will translate into reduced memory traffic. This can be measured by the miss rate defined in Section 3.2.

**B. Organizing spatial locality for prefetch:** The explicit management not only improves data placement in the temporal cache, but also creates opportunities for optimizations on spatial locality. As temporal locality and spatial locality are often associated, separating data references according to their temporal locality will often make the spatial locality in each cache more uniform. If multiple spatial caches are available, we can further group data references into different spatial caches according to spatial locality. Uniform spatial locality will help spatial locality based optimizations, such as aggressive prefetch.

**C. Overlapping the operations of different caches:** Different caches in a split cache architecture can work in parallel explicitly. We can improve the performance by restructuring programs to maximize the overlapping between cache operations.

## 3.2 Performance Metrics

The performance improvement from split cache architectures cannot be evaluated by miss rate directly, as there are multiple caches with different configurations. We use average memory access time and characteristic miss rate to measure performance improvements.

### 3.2.1 Average Memory Access Time

The average memory access time is highly architecture dependent. In our simulation, it is based on a representative cache and memory system assumption: A one-level cache connected to main memory via a 64-bit bus. We assume SDRAM with a 5-1-1-1 access cycle is used. On a cache line (block) fetch, the first 8 bytes take 5 memory cycles to complete, and the rest of the data transfers take 1 memory cycle for every 8 bytes. This results in a timing model, as shown in Table 1. We also assume the cost of a cache hit is 0.2 memory cycles.

| Line Size | Miss Penalty |
|---|---|
| 16 bytes | 6 memory cycles |
| 32 bytes | 8 memory cycles |
| 64 bytes | 12 memory cycles |
| 128 bytes | 20 memory cycles |

**Table 1: Relation between cache line size and cache miss penalty**

The purpose of using average memory access time is to evaluate the effects of our spatial locality optimizations, where we use spatial caches with different line sizes. The performance numbers will be different on other systems, but our optimizations will be effective as long as cache miss penalty increases with line size.

### 3.2.2 Characteristic Miss Rate

We define the characteristics miss rate as a timing model independent performance metric. It is measured on a reference split cache architecture, in which the cache line size of all caches are the same. This makes the cache miss penalty comparable. No prefetch techniques are applied. The size of all spatial caches is one cache line, thus the performance improvement from temporal reuse in spatial caches is not included. Cache misses from all caches are counted. The characteristic miss rate is defined as:

*characteristic_miss_rate = sum_of_all_cache_misses / total_number_of_accesses*

The main purpose of this definition is to make the meaning of the characteristic miss rate similar to the meaning of the miss rate on a traditional architecture. As prefetch techniques are excluded, it is a measure of the effectiveness of data placement in the temporal cache, closely related to memory traffic. It can be used in two ways: compare the performance of an application with or without explicit management, or compare the performance between a split cache architecture and a traditional architecture. Another advantage of this metric is that it can often be derived from an algorithm analytically.

## 3.3 Explicit Cache Management Algorithms for Selected Problems

### 3.3.1 Circular Data Block Access – An Illustrative Example

This problem is an abstraction of a simple but common access pattern. A data block is brought into a processor sequentially several times during the execution of an algorithm.

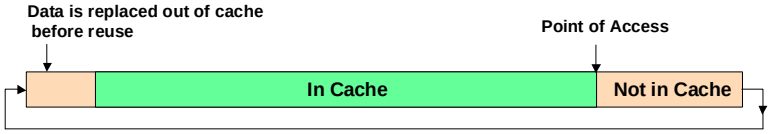



**Figure 2: Circular data block access on traditional architecture**

If the data block is larger than the cache, only recently accessed data is available in cache. Figure 2 shows the situation on a direct mapped cache or a fully associative cache with LRU replacement policy. The performance is poor in this scenario. There is zero temporal reuse exploited, as data is replaced out of cache before it is accessed again. Even if the data block is just slightly larger than the cache, all temporal reuse is gone. The cache behaves like a FIFO buffer. The situation is more complex for caches with limited associativity or random replacement policy, but the reuse will still decrease quickly as the data block size exceeds the cache size. This situation can be improved by the following target cache control (Figure 3):

- *Temporal Cache <= A region in the data block*[*]
- *Spatial Cache <= Rest of the data block*

[*] *size_of_the_region = size_ of_ the_temporal_cache*

The symbol "<=" is used to indicate that the cache on the left side will handle the data on the right side.

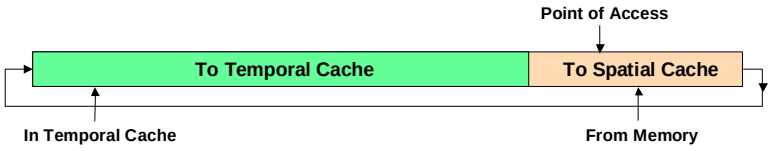



**Figure 3: Circular data block access with explicit management**

With explicit management, all data in the temporal cache will be reused. The rest of the data block will be loaded from memory each time. After the first iteration, characteristic miss rate (as well as the memory traffic) can be reduced by:

*cache_size/data_size (cache_size<data_size)*

*data_size is the size of the data block.*

For example, when the data block is just a little larger than cache, memory traffic can be reduced by nearly 100%; when the data block is 5 times as large as cache, the memory traffic can be reduced by 20%.

The data placement produced by the above explicit management is equivalent to the data placement produced by the optimal (MIN) replacement policy. This placement can also be considered as a global resource allocation. It is interesting to notice that all the data have exactly the same access pattern, but we treat them differently to improve performance. This type of problems would be difficult to handle by access history based hardware approaches.

Performance can be further improved by interleaving the operation of the temporal cache and the spatial cache, as shown in Figure 4. Some latency will be hidden if the two caches can work concurrently.

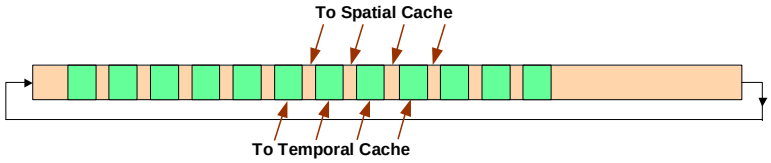



**Figure 4: Interleaving the operations of temporal cache and spatial cache**

### 3.3.2 Sparse Matrix Vector Multiplication

The sparse matrix vector multiplication problem is a widely used numerical kernel. It is included in many benchmark suites for high-end systems, as its performance is a key indication of a system's ability to handle irregular numerical problems.

**Optimization A**

The access patterns of the matrix and the vector are quite different. For the matrix, if the multiplication is performed just once, there is no temporal reuse. The spatial locality is usually good, depending on the data structure used. For the vector, the temporal locality is good, but the spatial locality is usually poor. A cache miss is more costly for a vector reference because of the poor spatial locality. Thus, the vector should have priority for cache space. However, in traditional architectures, matrix references will compete with vector references and evict vector data out of cache, due to its larger size. We can also consider this situation as cache pollution. Performance can be improved by the following target cache control:

- *Temporal Cache <= Vector*
- *Spatial Cache <= Matrix*

If the vector is smaller than the cache, the cache miss rates have a simple analytical form. The vector will be effectively locked in the temporal cache. All cache misses are compulsory. We can choose the data structure for the matrix such that the accesses are sequential to maximize spatial locality. Suppose there are two variables for each matrix element (index and data), two variables for each row of the matrix (row index and the number of nonzero elements in the row) and each data element is 4 bytes. We can calculate the characteristic miss rate as follows:

*spatial_misses = matrix_size / line_size = (F*2+N*2)*4/line_size*
*temporal_misses = vector_size /line_size = N*4/line_size*

$$characteristic_miss_rate = \frac{4(2F+3N)}{line_size \cdot (3F+2N)}$$

*vector_size: The size of the vector data in bytes*
*matrix_size: The size of the matrix data in bytes*
*line_size: The cache line size in bytes*
*spatial_misses, temporal_misses: The number of misses in each cache*
*N: The dimension of the matrix (square matrix)*
*F: The number of nonzero elements in the matrix*

The spatial locality of the matrix reference can be exploited to further improve performance. We can sequentially prefetch the matrix data into the spatial cache. The performance is limited only by memory bandwidth.

**Optimization B (Vector smaller than Cache)**

In problems such as finding the solutions of linear systems, the multiplication will be repeated many times with the same matrix. There is also temporal reuse for the matrix data, although the reuse frequency is still lower than that of the vector. If we consider the references to the matrix data separately, it leads to a "Circular Data Block Access" problem, which is discussed in Section 3.3.1. If the temporal cache is larger than the vector, we can incorporate the algorithm of that problem into optimization A. The revised target cache control is:

- *Temporal Cache*
  *<= Vector*
  *<= A region in the Matrix*[*]
- *Spatial Cache <= Rest of the Matrix*

[*] *size_of_the_region = size_of_the_temporal_cache – size_of_the_vector*

**Optimization C (Vector larger than cache)**

When the vector is larger than the cache, it cannot be cached efficiently, even if we direct the matrix data to the spatial cache. Most of the memory access time can be attributed to the cache misses for the vector. Although it is difficult to further reduce the number of cache misses, we can reduce the cache miss penalty. As there is little chance for spatial reuse, we can use a small data transfer unit between the cache and the memory to lower the cache miss penalty of vector references, by the following target cache control:

- *Temporal Cache <= A region in the Vector*[*]
- *Spatial Cache A <= Rest of the Vector*
- *Spatial Cache B <= Matrix*

[*] *size_of_the_region = size_of_the_temporal_cache*

Spatial cache A has a small data transfer unit, this can be accomplished by using a small line size or partial line fill. Spatial cache B has a large line size or prefetch mechanism to exploit the spatial locality of the matrix, as described in optimization A. The temporal cache is still used to store part of the vector data.

### 3.3.3 Random Tree Search

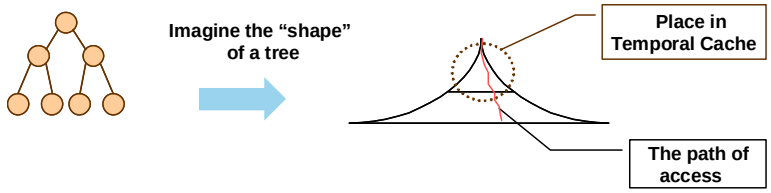



**Figure 5: Binary tree search**

We use the random tree search problem as an example of data structure applications. A search operation on a tree results in a series of data accesses, one access at each level. The access frequency of the nodes in the tree is not uniform. Nodes closer to the root are much more frequently accessed, due to the exponential growth of the number of nodes at each level (Figure 5). Thus, nodes closer to the root should have priority for cache space. However, in traditional cache architectures, lower level nodes will compete with higher level nodes for cache space. We use the following target cache control to improve data placement:

- *Temporal Cache <= Top Tree Nodes, up to the capacity of the Temporal Cache*[*]
- *Spatial Cache <= Rest of the Tree Nodes*

[*]*Select tree nodes level-by-level from the root, until the cache capacity is reached. The last level may not be fully directed to the temporal cache.*

After enough search operations have been performed, the memory system will reach a stable state. All accesses to the nodes in the temporal cache will be cache hits. All cache misses will come from the accesses to the nodes directed to the spatial cache. This is a better data placement, as we will show in the simulation section.

We make the following approximations to calculate the analytical estimation of characteristic miss rate: The temporal cache only holds complete tree levels. An access to a node in the temporal cache will cause two cache hits, assuming there are one access to the key and one access to one of the pointers. An access to the spatial cache will cause one cache miss and one cache hit, assuming the first access will cause a cache miss and the second access will cause a hit due to spatial locality. The derivation is skipped here and the stable state characteristic miss rate is given below:

$$characteristic_miss_rate = \frac{2^c + 2^L(L-c-1)}{2(1+2^L(L-1))}$$

$\boldsymbol{c} = \lfloor \log_2(temporal_size / node_size) \rfloor$ *(The number of levels that can fit in temporal cache.)*

***L:*** *The number of levels of the tree*

The data layout of this problem also needs to be adjusted for efficient target cache control. The nodes directed to the temporal cache and the nodes directed to the spatial cache should be stored in different areas.

### 3.3.4 Structure and Payload Problems

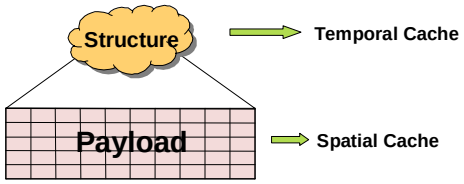



**Figure 6: Structure and payload problems**

In pointer based data structures, data can often be classified into two categories: structure data and payload. Pointers, indexes and keys can be considered as structure data, and the attached data can be considered as payload. (Figure 6) Structure data is essential for the operation of a data structure, while payload is the data manipulated by the data structure.

Structure data tends to be small, has good temporal locality and poor spatial locality. Payload tends to be large, has good spatial locality and poor temporal locality. For example, suppose the above Random Tree Search problem appears during the queries of an image database. In this case, an image is attached to each tree node. The difference between the tree structure and the image data is obvious. Therefore, structure data should have priority for cache space. In general, we can use the following target cache control for this kind of problems:

- *Temporal Cache <= Structure*
- *Spatial Cache <= Payload*

Special data layouts are also needed for this optimization. The structure data and payload data should be stored separately. As the payload usually has good spatial locality, we can use prefetch to further improve performance.

### 3.3.5 Dijkstra's Shortest-Path Algorithm

Dijkstra's shortest-path algorithm is an effective graph algorithm. It is often used in network routing, CAD and many other science and engineering applications. It is also a test the MiBench [18] benchmark suite. Usually two

major data structures are used in this algorithm, the graph data and the priority queue. The graph data stores information about graph edges and their costs. The priority queue is used to extract the lowest-cost node efficiently.

For the graph data, if we run the algorithm from a single source, there is no temporal reuse. There is some spatial locality, however, as all the edges going out from a node will be accessed consecutively during a relax operation. For the priority queue, there is temporal reuse, but the spatial locality is poor. Therefore, we can use the temporal cache to store the priority queue and divert the graph data to the spatial cache:

- *Temporal Cache <= Priority queue*
- *Spatial Cache <= Graph data*

This problem is a good example of straightforward explicit management optimizations for relatively complex problems. There are many opportunities for the optimizations of the priority queue, which are left for future study.

# 4. Simulation Results

## 4.1 Simulator

We developed a trace simulator to simulate the generalized split cache architecture. The simulated architecture consists of one temporal cache and multiple spatial caches. The cache size, line size, replacement policy and associativity of all caches can be configured to simulate various architectures. For simplicity, there is no prefetch mechanism included in the simulator. We use spatial caches with different line sizes to exploit spatial locality. Both miss rate and average memory access time are measured by this simulator. The timing model described in Section 3.2 is used to calculate the total memory cycles. The average memory access time is the total number of memory cycles divided by the number of references.

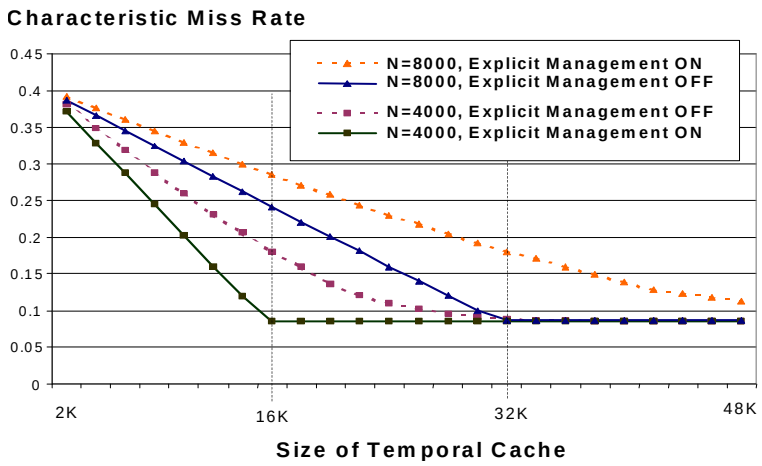
The trace is generated by inserting patches into program source code directly. To include explicit management information, an integer pair is generated for each memory access. One integer indicates the address. The other integer indicates which cache to use and if the access is a read or a write. This resembles the instruction embedded method (see Section 5).

## 4.2 Results

### 4.2.1 Sparse Matrix Vector Multiplication

Figure 7 shows the characteristic miss rate (defined in Section 3.2). The Optimization A in Section 3.3.2 is used. The sparse matrix is generated by inserting nonzero elements randomly. Its data structure groups the index and the data element together, so the access to the matrix is sequential.

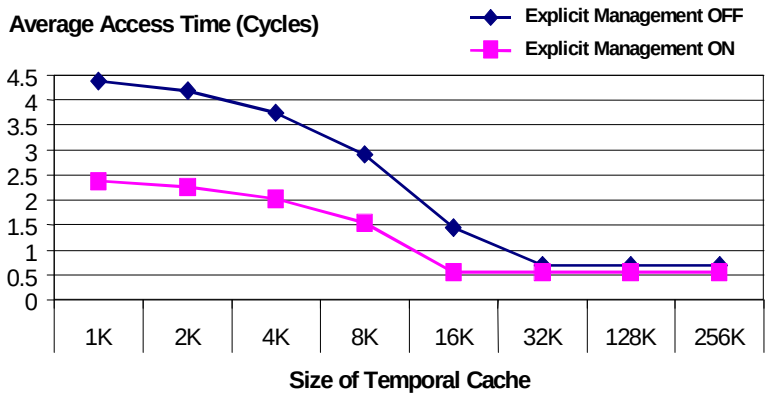
We can see the performance improvement is connected to the relative size of the vector and the temporal cache. Maximum characteristic miss rate reduction is 52% for both N=4000 and N=8000. Each data element is 4 bytes in this simulation, thus maximum reductions occur when the size of the vector is close to the size of the temporal cache. For other cache sizes, there are noticeable improvements when the size of the temporal cache falls between 1/4 to 2 times of the size of the vector. Other simulations results show the number of nonzero data elements in the matrix does not have a significant influence.




**N:** Dimension of matrix (square matrix)
**Nonzero data elements in matrix:** 160,000
**Spatial Cache:** 32-byte cache line, the cache size is 1 line
**Temporal Cache:** 32-byte cache line, 4-way associative, LRU replacement policy

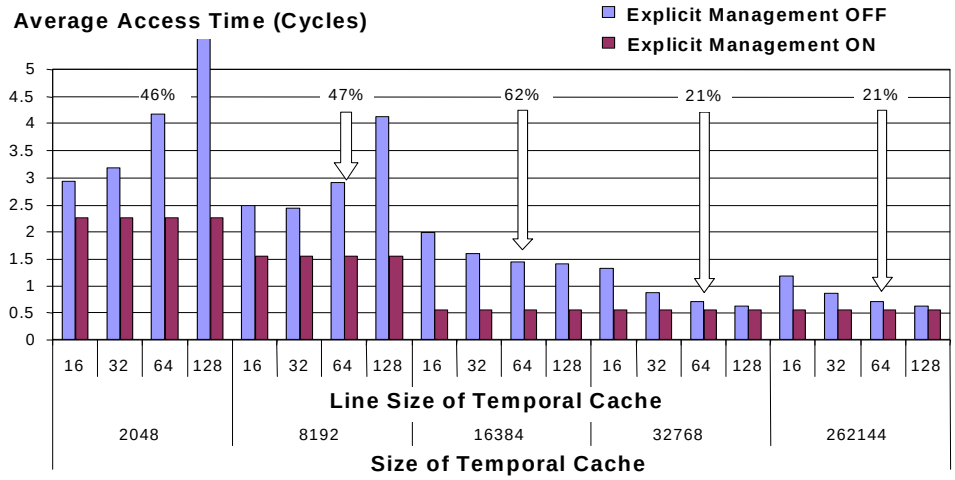
**Figure 7: Characteristic miss rate of sparse matrix vector multiplication**

Figure 8 and Figure 9 show the average memory access time of this problem. Figure 8 focuses on the relation between performance and cache size. Figure 9 focuses on the relation between performance and line size. The optimization C in Section 3.3.2 is used in this simulation. There are now two spatial caches, one for the vector and the other for the matrix. Different line sizes are used to exploit spatial locality. We can see the performance is further improved. This improvement is also observed over a broader range of cache sizes than the improvement in Figure 7.




**Matrix size:** 4,000*4,000
**Nonzero data elements in matrix:** 160,000
**Spatial Cache A:** 16-byte cache line, the cache size is 1 line
**Spatial Cache B:** 256-byte cache line, the cache size is 1 line
**Temporal Cache:** 64-byte cache line, 4-way associative, LRU replacement policy

**Figure 8: Average memory access time of sparse matrix vector multiplication**




**Parameters same as Figure 8**
**Reduction in average access time marked in percentage**

**Figure 9: Average memory access time of sparse matrix vector multiplication**

The performance improvement in Figure 8 and Figure 9 comes from three different sources. The contribution of each source is determined by the relative size of the vector and the temporal cache. This is explained in Table 2.

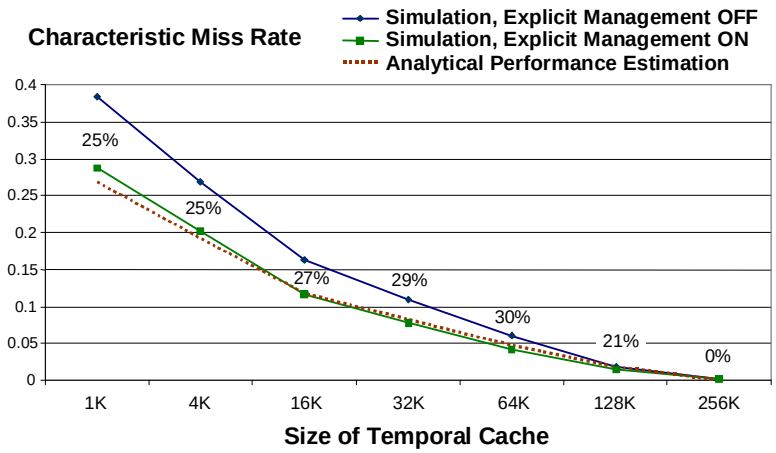
A. Better data placement in temporal cache. Memory traffic is also reduced.
B. Reduced cache miss penalty of vector references by using a small line size on the spatial cache A. Memory traffic is also reduced.
C. Improved memory bandwidth of matrix references by using a large line size on the spatial cache B.

| Relative Size | Performance Contributors | | | Comments |
|---|---|---|---|---|
| | A | B | C | |
| Cache << Vector | Some | Main | Some | Cache misses from vector references dominate memory access time. |
| Cache ~ Vector | Main | Some | Some | Explicit management works efficiently on data placement, with explicit management, vector is locked into cache, without explicit management, there is severe pollution from matrix data. |
| Cache >> Vector | Little | None | Main | There is enough space for both vector and matrix. Data placement does not have a significant influence. B is not applicable, as the whole vector can be locked in temporal cache. |

**Table 2: Analysis of different sources of performance improvement**

### 4.2.2 Binary Tree Search

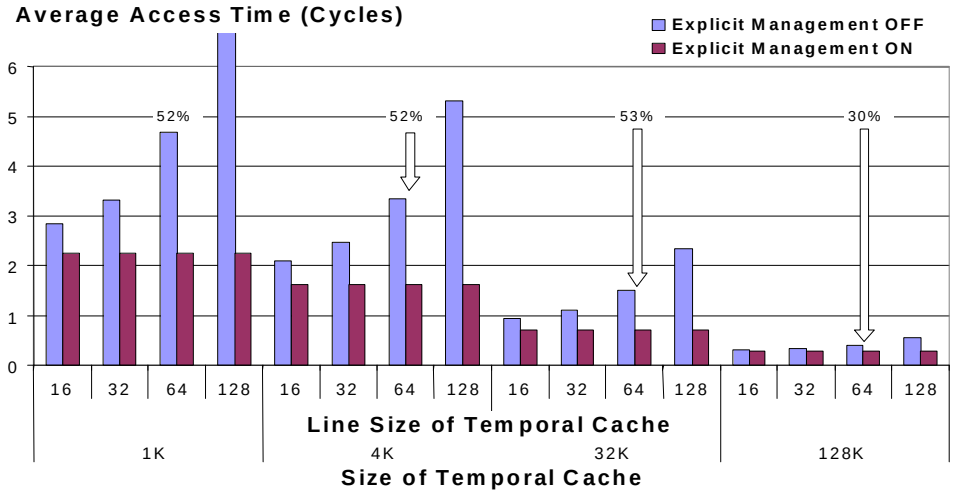
Figure 10 shows the characteristic miss rate for the binary tree search problem related to the size of the temporal cache, including the simulation result and the analytical performance estimation described in Section 3.3.3. For the simulation result, we can see there are similar reductions across a wide range of temporal cache sizes. When the cache is larger than the tree (The tree is 192K bytes), there is no improvement, as the complete tree can fit into the temporal cache. For the analytical performance estimation, we can see it matches the simulation result. The difference is caused by approximations introduced during the derivation.




**Tree:** 14-level full binary search tree
**Node size:** 12 bytes
**Tree size:** 192K bytes (number_of_nodes * node_size)
**Number of search operation:** 4 * number_of_tree_nodes
**Temporal Cache:** 64-byte cache line, 4-way associative, LRU replacement policy
**Spatial Cache:** 64-byte cache line, the cache size is 1 line
**Reduction in characteristic miss rate marked in percentage (between simulation results)**

**Figure 10: Characteristic miss rate of binary tree search**

Figure 11 shows the average access time for the problem. The line size of the spatial cache is now set to 16 bytes to reduce cache miss penalty. The performance is further improved when the temporal cache has a medium to large line size.

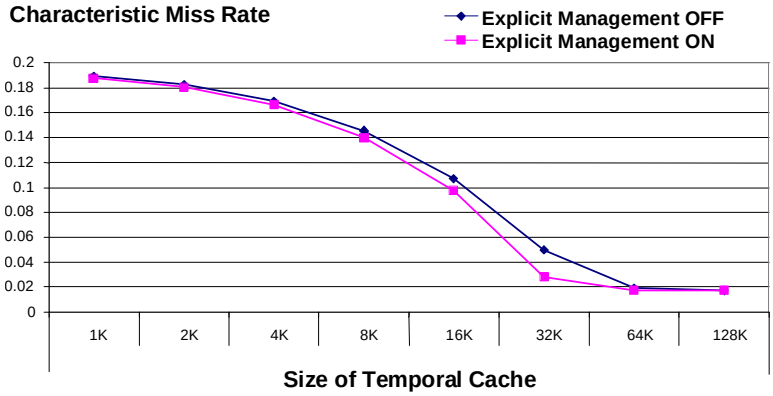



**Spatial Cache:** 64-byte cache line, the cache size is 1 line
**Other parameters same as Figure 10**
**Reduction in average memory access time marked in percentage**

**Figure 11: Average memory access time of binary tree search**

### 4.2.3 Dijkstra's Algorithm

Figure 12 shows the characteristic miss rate for the problem. We can see the effect of explicit management on this problem is similar to the sparse matrix vector multiplication problem. The peak miss rate reduction is 42%. This occurs when the size of the temporal cache is 32K. This result also suggests that a smaller cache with explicit management can behave like a larger cache without explicit management.




**Number of nodes:** 2,000
**Number of edges:** 200,000
**Temporal Cache:** 64-byte cache line, 4-way associative, LRU replacement policy
**Spatial Cache:** 64-byte cache line, the cache size is 1 line

**Figure 12: Characteristic miss rate of Dijkstra's algorithm**

# 5. Discussion

The explicit cache management is a rather new and broad area. We focused on explicit management algorithms in previous sections. In this section, we will briefly address some architectural issues.

**Multilevel Cache:** The Generalized Split Cache Architecture we defined before can be extended to multilevel cache architectures. There are two available methods:

We can simply replace the single level temporal cache with a multilevel traditional cache. This is the scheme used by HPL-PD [14]. The algorithms for single level cache can be used directly.

The split cache can also be built at each level recursively. This is the scheme used by Itanium [11]. We can get better performance by applying different optimizations at different levels. For example, for the sparse matrix vector multiplication problem, we can use L1 cache for vector data only and L2 cache for both vector and matrix data (see Section 3.3.2).

**Implementation of Target Cache Control:** In split cache architectures, on a data reference, software needs to pass some extra information to hardware to indicate which cache to use. There are two available implementations:

**Instruction Embedded Method:** We can add a bit field to each load/store instruction. Both Itanium [11] and HPL-PD [14] use this method. It is inherently a compile time solution, as compiler will generate all the machine

code. It is not very flexible for run-time adaptations, as “if” may be need. The amount of embedded information is limited.

**Page Table Embedded Method:** The page table is another good place to embed control information. This method is used by Intel XScale [12]. It is inherently a run time operating system solution, as the OS is in charge of the page table. Although it may not be as convenient as the Instruction Embedded Method, there are several advantages: A large amount of information can be included in the page table, which is based on main memory and cached by TLB. We may need a large space for prefetch information. The instruction set architecture is not affected by this method, so it can be easily incorporated into existing architectures. Run-time adaptations can also be more efficient.

**Cache Line Size in Split Cache Architectures:** There are at least two reasons for organizing cache memory into cache lines. One is efficiency: The amount of tag memory and parallel search circuit is proportional to the number of cache lines, the larger cache line, the lower hardware cost. The other benefit is prefetch. Cache line fetch can be considered as an implicit prefetch, which can hide memory latency and increase memory throughput. The cache line size is always a tradeoff. Although a cache with flexible line size can improve performance, this may not be very practical, as the hardware complexity would be high, and proportional to the smallest possible line size.

The situation is much better in split cache architectures, as duties are distributed into temporal and spatial caches. We suggest a relatively large fixed line size for the temporal cache for efficiency. Data with poor spatial locality can be handled by spatial caches. A possible design is to employ multiple spatial caches with different (and fixed) line sizes. This way, no special cache design is needed. In the sparse matrix vector multiplication problem, this can provide very good performance.

**Resolving Cache Conflict:** The explicit management on split cache architectures can also be used to resolve cache conflict. We can simply leave only one of the conflicting references to the temporal cache and divert all other references to spatial caches. The advantage is that data layout and program structure do not need to be changed.

**Better Performance Prediction:** In our problems, we found it is much easier to estimate performance after the explicit management is applied. The characteristic miss rate of several problems can be expressed in analytical forms. Due to better data placement control from explicit management, we can not only improve performance, but also make the memory access time more predictable. This would be helpful for real time applications.

**Relation to Hardware and Compiler Based Approaches:** Compared with hardware and compiler approaches, application directed approach is more precise. It also requires much less hardware support than hardware approaches. The advantage of hardware and compiler approaches is that they are automatic. Application directed approach works best for performance sensitive kernel applications. The three approaches can also be used together.

Our approach is also a good reference for compiler and hardware optimizations, as it can reveal the mechanism of performance improvement. Our explicit management algorithms are directly applicable to compiler optimizations. They are also good references for hardware optimizations.

## 6. Related Work

**Architecture Definitions:** Various architectures are defined in [1][4][6][7]. In these architectures, there are multiple caches for data with different temporal and spatial localities. Although there are many differences among them, the ideas are similar.

**Hardware Adaptive Approaches:** [1][2][3][5][7][8] can be classified as hardware adaptive approaches. In most of these approaches, the control decision is based on access history. Static control based on profiling information is also used in some papers.

**Compiler Driven Approaches:** [4][6] are compiler driven approaches, where the compiler analyzes source code and generates control information. They are focused on regular numerical codes.

**The Address Mapping Approach:** [9] is a interesting approach, in which applications control the virtual/physical address mapping to get better data placement.

**Software Prefetch:** Software prefetch techniques such as [16] also have some similarities to our approach, in that they are also combined hardware and software efforts. They are different, however, as our approach addresses data placement and memory traffic besides access latency.

## 7. Conclusion

The performance improvements in our problems are attractive. The explicit management algorithms are straightforward. The underlying architecture, the Generalized Split Temporal/Spatial Cache architecture, is realistic. All of these factors make us think our approach is an efficient next step to improve memory system performance.

## 8. References


[1] *A new cache architecture concept: the split temporal/spatial cache,* Milutinovic, V. Tomasevic, M. Markovi, B. Tremblay, M., *Electrotechnical Conference, 1996*

[2] *Split Temporal/Spatial Cache: A Survey and Reevaluation of Performance*, M. Prvulovic D. Marinov Z. Dimitrijevic and V. Milutinovic, *IEEE TCCA Newsletter, July 1999*

[3] *Run-Time Cache Bypassing,* T. L. Johnson, D. A. Connors, M. C. Merten, and W. W. Hwu, *IEEE Transactions on Computers, Vol. 48, No. 12, December 1999, pp. 1338-1354*

[4] *A Locality Sensitive Multi-Module Cache with Explicit Management*, Jesús Sánchez and Antonio González, *Proc. of the ACM International Conference on Supercomputing (ICS-99), Rhodes (Greece)*

[5] *A modified approach to data cache management,* G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, *Proceedings of the 28th Annual International Symposium on Microarchitecture, December 1995*

[6] *Software Management of Selective and Dual Data Caches,* Sanchez, F. J., Gonzalez, A., Valero, M., *IEEE TCCA NEWSLETTERS, March 97*

[7] *A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality*, Gonzalez, A., Aliagas, C. and Valero, M., *Proceedings of the International Conference on Supercomputing (ICS'95), Barcelona, Spain*

[8] *Annotated Memory References: A Mechanism for Informed Cache Management,* A. R. Lebeck, D. R. Raymond, C. Yang, M. S. Thottethodi, *Euro-Par '99*

[9] *Cache-Conscious Structure Layout*, Trishul M. Chilimbi, Mark D. Hill, and James R. Larus, *Programming Language Design and Implementation (PLDI), 1999*

[10] (Removed for blind review)

[11] *Intel® Itanium™ Architecture Software Developer's Manual Vol. 1~3 rev. 2.0*, Intel *http://developer.intel.com*

[12] *Intel® XScale™ Core Developer's Manual*, Intel, *http://developer.intel.com*

[13] *UltraSPARC III Cu User's Manual*, SUN

[14] *HPL-PD architecture specification: Version 1.1*,Vinod Kathail, Michael S. Schlansker, and B. Ramakrishna Rau, *Tech. Rep. HPL-93-80(R.1), Hewlett Packard Company, Feb. 2000*

[15] *Trimaran website, http://trimaran.org*

[16] *Compiler-Based Prefetching for Recursive Data Structures*, C.-K. Luk and T. C. Mowry, *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, 1996.*

[17] *DIS Stressmark Suite Version 1.0*, Titan Systems Corporation Atlantic Aerospace Division, *http://www.aaec.com/projectweb/dis, 2000*

[18] *MiBench: A Free, Commercially Representative Embedded Benchmark Suite*, Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, Richard B. Brown, *IEEE 4th Annual Workshop on Workload Characterization, 2001*

# Performance Modeling and Interpretive Simulation of PIM Architectures and Applications

Zachary K. Baker and Viktor K. Prasanna

University of Southern California, Los Angeles, CA USA
zbaker@halcyon.usc.edu, prasanna@ganges.usc.edu
http://advisor.usc.edu

**Abstract.** Processing-in-Memory systems that combine processing power and system memory chips present unique algorithmic challenges in the search for optimal system efficiency. This paper presents a tool which allows algorithm designers to quickly understand the performance of their application on a parameterized, highly configurable PIM system model. This tool is not a cycle-accurate simulator, which can take days to run, but a fast and flexible performance estimation tool. Some of the results from our performance analysis of 2-D FFT and biConjugate gradient are shown, and possible ways of using the tool to improve the effectiveness of PIM applications and architectures are given.

## 1 Introduction

The von Neumann bottleneck is a central problem in computer architecture today. Instructions and data must enter the processing core before execution can proceed, but memory and data bus speeds are many times slower than the data requirements of the processor. Processing-In-Memory (PIM) systems propose to solve this problem by achieving tremendous memory-processor bandwidth by combining processors and memory together on the same chip substrate. Notre Dame, USC ISI, Berkeley, IBM, and others are developing PIM systems and have presented papers demonstrating the performance and optimization of several benchmarks on their architectures. While excellent for design verification, the proprietary nature and the time required to run their simulators are the biggest detractors of their tools for application optimization. A cycle-accurate, architecture-specific simulator, requiring several hours to run, is not suitable for iterative development or experiments on novel ideas. We provide a simulator which will allow faster development cycles and a better understanding of how an application will port to other PIM architectures [4, 7]. For more details and further results, see [2].

[1] 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. The PIM Simulator is available for download at http://advisor.usc.edu

## 2 The Simulator

The simulator is a wrapper around a set of models. It is written in Perl, because the language's powerful run-time interpreter allows us to easily define complex models. The simulator is modular; external libraries, visualization routines, or other simulators can be added as needed. The simulator is composed of various interacting components. The most important component is the data flow model, which keeps track of the application data as it flows through the host and the PIM nodes. We assume a host with a separate, large memory. Note that as the PIM nodes make up the main memory of the host system in some PIM implementations. The host can send and receive data in a unicast or multicast fashion, either over a bus or a non-contending, high-bandwidth, switched network. The bus is modeled as a single datapath with parameterized bus width, startup time and per element transmission time. Transmissions over the network are assumed to be scheduled by the application to handle potential collisions. The switched network is also modeled with the same parameters but with collisions defined as whenever any given node attempts to communicate with more than one other node(or host), except where multicast is allowed. Again, the application is responsible for managing the scheduling of data transmission. Communication can be modeled as a stream or as packets.

Computation time can be modeled at an algorithmic level, e.g. $n \lg(n)$ based on application parameters, or in terms of basic arithmetic operations. The accuracy of the computation time is dependent entirely on the application model used. We assume that the simulator will be commonly used to model kernel operations such as benchmarks and stressmarks, where the computation is well understood, and can be distilled into a few expressions. This assumption allows us to avoid the more complex issues of the PIM processor design and focus more on the interactions of the system as a whole.

## 3 Performance Results

### 3.1 Conjugate Gradient Results

Figure 1 shows the overall speedup of the biConjugate Gradient stressmark with respect to the number of active PIM elements. It compares results produced by our tool using a DIVA parameterized architecture to the cycle-accurate simulation results in [4]. Time is normalized to a simulator standard. The label of our results, "Overlap 0.8", denotes that 80% of the data transfer time is hidden underneath the computation time, via prefetching or other latency hiding techniques. The concept of overlap is discussed later in this paper.

BiConjugate Gradient is a DARPA DIS stressmark [1]. It is used in matrix arithmetic to find the solution of $y = Ax$, given $y$ and $A$. The complex matrices in question tend to be sparse, which makes the representation and manipulation of data significantly different than in regular data layout of FFT. The application model uses a compressed sparse row matrix representation of $A$, and load balances based on the number of elements filling a row. This assumes that the number of rows is significantly higher than the number of processors. All PIM nodes are sent the vector y and can thus execute on their sparse elements independently of the other PIM nodes.

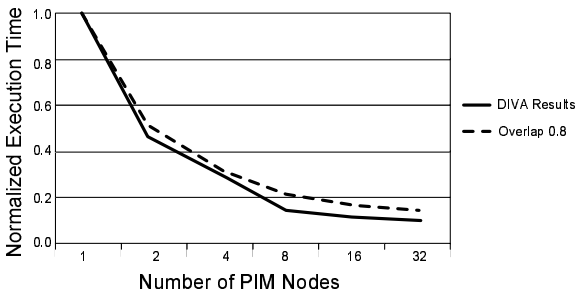



**Fig. 1.** Speedup from one processor to n processors with DIVA model

Figure 2 is a graph of the simulator output for a BiCG application with parameters similar to that of the DIVA architecture with a parallel, non-contending network model, application parameters of n(row/column size of the matrix)=14000 and nz(non zero elements)=14 elements/row. Figure 2(left) shows the PIM-to-PIM transfer cost, Host-to-PIM transfer costs, computation time, and total execution time(total) as the number of PIM nodes increases under a DIVA model. The complete simulation required 0.21 seconds of user time on a Sun Ultra250 with 1024 MB of memory.

The graph shows that the computation time decreases linearly with the number of PIM nodes, and the data transfer time increases non-linearly. We see in the graph that PIM-to-PIM transfer time is constant– this is because the number of PIM nodes in the system does not dramatically affect the amount of data (a vector of size n in each iteration) sent by the BiCG model. Host-to-PIM communication increases logarithmically with number of PIM; the model is dependent mostly on initial setup of the matrices and final collection of the solution vectors. The Host-to-PIM communication increases toward the end as the communications setup time for each PIM becomes non-negligible compared to the total data transferred. Figure 2(right) shows a rescaled version of the total execution time for the same parameters. Here, the optimal number of PIM under the BiCG model and architectural parameters is clear– this particular application seems suited to a machine of 64 to 128 PIM nodes most optimally in this architecture model.

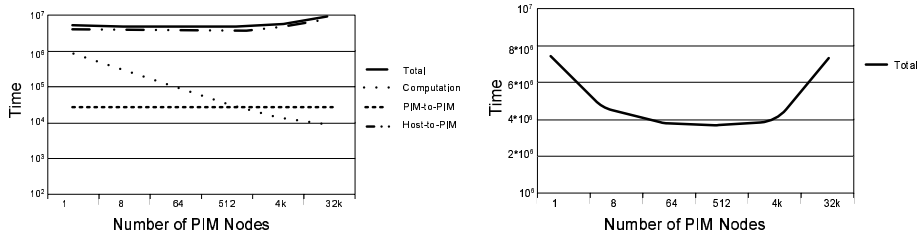



**Fig. 2.** BiConjugate Gradient Results; unit-less timings for various amounts of PIM nodes. (left: all results, right: total execution time only)

### 3.2 FFT

Another stressmark modeled is the 2-D FFT. Figure 3 shows execution time versus the number of FFT points for the Berkeley VIRAM architecture, comparing our results against their published simulation results [8]. This simulation,

for all points, required 0.22 seconds of user time. The 2-D FFT is composed of a one dimensional FFT, a matrix transpose or 'corner-turn', and another FFT, preceded and followed by heavy communication with the host for setup and cleanup. Corner turn, which can be run independently of the FFT application, is a DARPA DIS stressmark [1]. Figure 3 shows the VIRAM speedup results against various overlap factors– a measure of how much of the data exchange can overlap with actual operations on the data. Prefetching and prediction are highly architecture dependent; thus the simulator provides a parameter for the user to specify the magnitude of these effects.

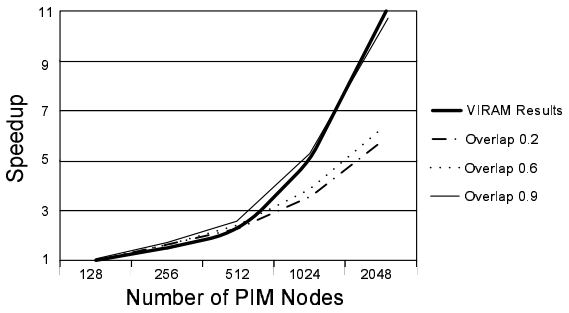



**Fig. 3.** Speedup versus number of FFT Points for various fetch overlaps, normalized to 128 points.

In the graph we see that the VIRAM results match most closely with an overlap of 0.9; that is, virtually all of the data transfer is hidden by overlapping with the computation time. This 'overlap' method is similar to the 'clock multiplier factor N' used by Rsim in that it depends on the application and the system and cannot to determined without experimentation [5].

Inspecting the VIRAM architecture documentation, we see that it includes a vector pipeline explicitly to hide the DRAM latency [6]. Thus our simulation results suggest the objective of the design has been achieved.

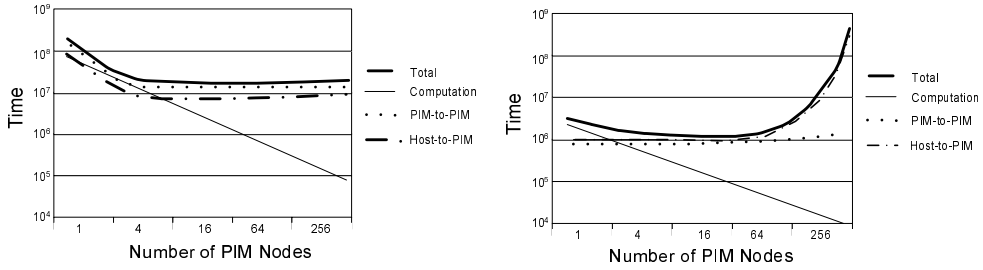



**Fig. 4.** 2-D FFT Results (left: Small memory size, right: Small problem size)

The simulator can be used to understand the performance of a PIM system under varying application parameters, and the architecture's effect on optimizing those parameters. A graph of the simulator output in Figure 4(left) and 4(right) show a generic PIM system interconnected by a single wide bus. The

FFT problem size is $2^{20}$ points, and the memory size of any individual node is 256K. The change in slope in Figure 4(left) occurs because the problem fits completely within the PIM memory after the number of nodes exceeds four. Until the problem size is below the node memory capacity, bandwidth is occupied by swapping blocks back and forth between the node and the host memory. Looking toward increasing numbers of PIM, we see that the total time has a minimum at 128, and then slowly starts to increase. Thus it could be concluded that an optimal amount of PIM nodes for an FFT of size $2^{20}$ is 128.

## 4 Conclusions

In this paper we have presented a tool for high-level modeling of Processing-In-Memory systems and its uses in optimization and evaluation of algorithms and architectures. We have focused on the use of the tool for algorithm optimization, and in the process have given validation of the simulator's models of DIVA and VIRAM. We have given a sketch of the hardware abstraction, and some of the modeling choices made to provide an easier-to-use system. We have shown some of the application space we have modeled, and presented validation for those models against simulation data from real systems, namely DIVA from USC ISI and VIRAM from Berkeley.

This work is part of the Algorithms for Data IntensiVe Applications on Intelligent and Smart MemORies (ADVISOR) Project at USC [3]. 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.

## References


1. Titan Systems Corporation Atlantic Aerospace Division. DIS Stressmark Suite. http://www.aaec.com/projectweb/dis/, 2000.
2. Z. Baker and V.K. Prasanna. Technical report: Performance Modeling and Interpretive Simulation of PIM Architectures and Applications. In preparation.
3. V.K. Prasanna et al. ADVISOR project website. http://advisor.usc.edu.
4. M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, A. Srivastava, W. Athas, J. Brockman, V. Freeh, J. Park, and J. Shin. Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture. In *SC99*.
5. C.J. Hughes, V.S. Pai, P. Ranganathan, and S.V. Adve. Rsim: Simulating Shared-Memory Muliprocessors with ILP Processors, Feb 2002.
6. Christoforos Kozyrakis. A Media-Enhanced Vector Architecture for Embedded Memory Systems Technical Report UCB//CSD-99- 1059, July 1999.
7. D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM: IRAM, 1997.
8. Randi Thomas. An Architectural Performance Study of the Fast Fourier Transform on Vector IRAM. Master's thesis, University of California, Berkeley, 2000.

# Optimizing Graph Algorithms for Improved Cache Performance*

Joon-Sang Park, Michael Penner, and Viktor K Prasanna
University of Southern California
{jsp, mipenner, prasanna} @usc.edu
http://advisor.usc.edu

**Abstract**

*Tiling has long been used to improve cache performance. Recursion has recently been used as a cache-oblivious method of improving cache performance. Both of these techniques are normally applied to dense linear algebra problems. We develop new implementations by means of these two techniques for the fundamental irregular problem of Transitive Closure, namely the Floyd-Warshall Algorithm, and prove their optimality with respect to processor-memory traffic. Using these implementations we show up to 10x improvement in execution time. In this context we also compare the performance of a nonlinear array layout with that of the block data layout. We also address Dijkstra's algorithm for the single-source shortest-path problem and Prim's algorithm for Minimum Spanning Trees, for which neither tiling nor recursion can be directly applied. For these algorithms, we demonstrate up to a 2x improvement by using a cache friendly graph representation. We also demonstrate improvements in cache performance for two cache friendly implementations of the heap compared with the asymptotically optimal implementation, with respect to time complexity. Experimental results are shown for the Pentium III, UltraSPARC III, Alpha 21264, and MIPS R12000 machines using problem sizes between 1024 and 4096 vertices. We demonstrate improved cache performance using the Simplescalar simulator.*

## 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 many applications [5][25]. While cache performance has been well studied, much of the focus has been on dense linear algebra problems, such as matrix multiplication and FFT [5][12][20][30]. All of these problems possess very regular access patterns that are known at compile time. In this paper, we take a different approach to this topic by focusing on some fundamental irregular graph problems.

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 [25]. 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 penalties vary among processors and cause large variations in execution time.

The area of graph problems are fundamental in a wide variety of fields, most notably network routing, distributed computing, and computer aided circuit design. Graph problems, as irregular, pose unique challenges to improving cache performance, challenges that often cannot be handled using standard cache-friendly optimizations [9]. The focus of this research is to develop methods of meeting these challenges.

* 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.

In this paper we present a number of optimizations to the Floyd-Warshall algorithm, Dijkstra's algorithm, and Prim's algorithm. For the Floyd-Warshall algorithm we present a recursive implementation that achieves a 6x improvement over the baseline implementation. We also show that by tuning the base case for the recursion, we can further improve performance by approximately 20%. We also show a novel approach to tiling for the Floyd-Warshall algorithm that achieves performance very close to that of the recursive implementation. Also note that today's state of the art research compilers cannot generate this implementation [9].

There are some natural combinations of implementation and data layout that decrease overhead costs, such as index computation, and yield performance advantage. In this paper, we show that our implementations of the Floyd-Warshall algorithm perform roughly equal with either the Morton layout or the Block Data Layout.

For Dijkstra's algorithm and Prim's algorithm, to which tiling and recursion are not directly applicable, we present a cache-friendly graph representation. By matching the data layout of the representation to the access pattern we show up to a 2x improvement in execution time. We also discuss the optimization of the heap. We discuss in detail the unique challenges posed by this dynamic data structure, and present two cache-friendly optimizations for the heap. Using these optimizations we show significant improvements in cache performance compared with the Fibonacci heap, which represents the asymptotically optimal implementation of the heap for these algorithms.

The remainder of this paper is organized as follows: In Section 2 we give the background needed and briefly summarize some related work in the areas of cache optimization and compiler optimizations. In Section 3 we discuss optimizing the Floyd-Warshall algorithm. In Section 4 we discuss optimizing the heap data structure and Dijkstra's algorithm. In Section 5 we apply the heap optimizations to Prim's algorithm. In Section 6 we draw conclusions.

## 2. Background and Related Work

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 graph algorithms. Those readers comfortable with the algorithms can skip this. In Section 2.2 we discuss some of the challenges that are faced in making the transitive closure problem cache-friendly. We also discuss the model that we use to analyze cache performance and the four architectures that we use for experimentation throughout the paper. 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. Overview of Graph Algorithms

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 $N$, $N$x$N$ 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 for each $(i, j)^{th}$ value [7].

Dijkstra's algorithm is designed to solve the single-source shortest path problem. It does this by repeatedly extracting from a priority queue Q the nearest vertex $u$ to the source, given the distances known thus far in the computation (Extract-Min operation). Once this nearest vertex is selected, all vertices $v$ that neighbor $u$ are updated with a new distance from the source (Update operation). The

pseudo-code for the algorithm is given in Figure 2. The optimal implementation of Dijkstra's algorithm utilizes the Fibonacci heap and has complexity $O(N\lg(N) + E)$ [7].

Prim's algorithm for Minimum Spanning Tree is very similar to Dijkstra's algorithm for the single-source shortest path problem. In both cases a root node or source node is chosen and all other nodes reside in the priority queue. Nodes are extracted using an Extract-min operation and all neighbors of the extracted vertex are updated. The difference in Prim's algorithm is that nodes are updated with the weight of the edge from the extracted node instead of the weight from the source or root node [7].

### 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 $N$x$N$ array $D^k$ before moving on to the $(k+1)^{th}$ step (see Figure 2). 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 SUIF research compiler and found that it cannot perform the optimizations discussed in Section 3 without user provided knowledge of the algorithm [9]. 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.

In Dijkstra's algorithm and Prim's algorithm, the most efficient implementation uses a Fibonacci heap structure for the priority queue (see Section 4.2). This involves pointer manipulation and irregular accesses, which are inherently cache-unfriendly. In fact, we can show significant improvements in cache miss rate by using a cache-friendly implementation of the heap instead of the asymptotically optimal Fibonacci heap.

The model that we use for our research is that of a uni-processor, 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 traffic 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 (see Figure 4). Finally, we assume an internal TLB with a fixed number of entries.

We use four different architectures for our experiments. The Pentium III Zeon running Windows 2000 is a 700 MHz, 4 processor shared memory machine with 4 GB of main memory. Each processor has 32 KB of level-1 data cache and 1 MB of level-2 cache on-chip. The level-1 cache is 4-way set associative with 32 B lines and the level-2 cache is 8-way set associative with 32 B lines. The UltraSPARC III machine is a 750 MHz SUN Blade 1000 shared memory machine running Solaris 8. It has 2 processors and 1 GB of main memory. Each processor has 64 KB of level-1 data cache and 8 MB of level-2 cache. The level-1 cache is 4-way set associative with 32 B lines and the level-2 cache is direct mapped with 64 B lines. The MIPS machine is a 300 MHz R12000, 64 processor, shared memory machine with 16 GB of main memory. Each processor has 32 KB of level-1 data cache and 8 MB of level-2 cache. The level-1 cache is 2-way set associative with 32 B lines and the level-2 cache is direct mapped with 64 B lines. The Alpha 21264 is a 500 MHz uniprocessor machine with 512 MB of main memory. It has 64 KB of level-1 data cache and 4 MB of level-2 cache. The level-1 cache is 2-way set associative with 64 B lines and the level-2 cache is direct mapped with 64 B lines. It also has an 8 element fully-associative victim cache. Unless otherwise specified the Simplescalar simulations are done using 16 KB of level-1 data cache and 256 KB of level-2 cache parameters.

### 2.3. Related Work

A number of groups have done research in the area of cache performance analysis in recent years. Detailed cache models have been developed by Weikle, McKee, and Wulf in [29] and Sen and Chatterjee in [25]. XOR-based data layouts to eliminate cache misses have been explored by Valero and others in [13]. 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 [15].

A number of papers have discussed the optimization of specific dense linear algebra problems with respect to cache performance. Whaley and others discuss optimizing the widely used Basic Linear Algebra Subroutines (BLAS) in [30]. Chatterjee and Sen discuss a cache efficient matrix transpose in [5]. Frigo and others discuss the cache performance of cache oblivious algorithms for matrix transpose, FFT, and sorting in [12]. Park and Prasanna discuss dynamic data remapping to improve cache performance for the DFT in [20]. One characteristic that all 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 [18], [19], [24], [27]). Optimizing blocked algorithms has been extensively studied (see for example [16]). 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 not perform the transformations discussed in Section 3 without user intervention [9].

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. discusses making pointer-based data structures cache-conscious in [6]. He focuses on providing structure layouts to make tree structures cache-conscious. The difference between this work and ours is that we are focusing on the dynamic heap data structure, instead of a more static tree structure such as a binary tree. As we discuss in Section 4.2, this dynamic nature presents some unique challenges. LaMarca and Ladner developed analytical models and showed simulation results predicting the number of cache misses for the heap in [17]. However, the predictions they made were for an isolated heap, and the model they used was the hold model, in which the heap is static for the majority of operations. In our work, we assume a very dynamic nature for the heap, and we conduct experiments for complete algorithms as opposed to isolating the heap.

We have recently published work on the Floyd-Warshall algorithm in [22] that showed a 2x improvement using the Unidirectional Space Time Representation. Compared with [22], this paper represents a new approach to optimizing the Floyd-Warshall algorithm, namely recursion and a novel tiled implementation. We also expand our scope of algorithms to include Dijkstra's algorithm for the single source shortest path problem and Prim's algorithm for the minimum spanning tree problem.

## 3. Optimizing the Floyd-Warshall Algorithm

In this section we address the challenges of the Floyd-Warshall algorithm. In Section 3.1 we introduce and prove the correctness of a recursive implementation for the Floyd-Warshall algorithm. We also analyze the cache performance and show experimental results for this implementation compared with the baseline. We also show that by tuning the recursive algorithm to the cache size, we can improve its performance by roughly 10%. In Section 3.2, we present a novel tiled implementation

of the Floyd-Warshall algorithm. Finally, in Section 3.3, we address the issue of data layout for both the blocked implementation and the recursive implementation.

Throughout this section we make use of the following assumptions. We assume a directed graph with $N$ vertices and $E$ edges. We assume the cache model described in Section 2.2, where $Ci < N^2$ for some $i$ and the TLB size is much less than $N$. To experimentally validate our approaches and their analysis, the actual problem sizes that we are working with are between 1024 and 4096 nodes ($1024 \leq N \leq 4096$). 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.

In [14] it was shown that the lower bound on processor-memory traffic was $\Omega(N^3/\sqrt{C})$ for the usual implementation of matrix multiply. By examining data dependency graphs for both matrix multiplication and the Floyd-Warshall algorithm, it can be shown that matrix multiplication reduces to the Floyd-Warshall algorithm with respect to processor-memory traffic. Therefore, we have the following:

**Lemma 3.1:** The lower bound on processor-memory traffic for the Floyd-Warshall algorithm, given a fixed cache size $C$, is $\Omega(N^3/\sqrt{C})$, where $N$ is the number of vertices in the input graph.

### 3.1. A Recursive Implementation of the Floyd-Warshall Algorithm

As stated earlier, recursive implementations have recently been used to increase cache performance. It was stated in [11] that recursive implementations perform automatic blocking at every level of the memory hierarchy. To the authors' knowledge, there does not exist a recursive implementation of the Floyd-Warshall algorithm. The reason for this, is that the Floyd-Warshall algorithm not only contains all the dependencies present in ordinary matrix multiplication, but also additional dependencies that can not be satisfied by the simple recursive implementation of matrix multiply. What is shown here is a novel recursive implementation of the Floyd-Warshall algorithm. We also prove the correctness of the implementation and show analytically that it reaches an asymptotically optimal amount of processor memory traffic.

In order to design a recursive implementation of the Floyd-Warshall algorithm, first examine the standard implementation when applied to a 2x2 matrix. The code for this is shown in Figure 5a. Notice that 8 calls are made to the min() operation and each call requires 3 data values from the matrix. Convert this into a recursive program by replacing the call to the min() function with a recursive call. Instead of passing 3 data values, pass 3 sub-matrices corresponding to quadrants of the input matrix. This code is shown in Figure 5b. The initial call to the recursive algorithm passes the entire input matrix as each argument. Subsequent calls pass quadrants of their input arguments as shown in Figure 5b. Code similar to Figure 5a calling the min() operation is used as the base case for when the input matrices are of size 2x2.

**Theorem 3.1:** The recursive implementation of the Floyd-Warshall algorithm detailed above satisfies all dependencies in the Floyd-Warshall algorithm and computes the correct result.

**Proof:**

The correctness of this algorithm is proven using induction on the depth of the recursion tree. At the bottom of the recursion tree, an ordinary implementation of the Floyd-Warshall algorithm is used.

*Base cases:*

If the depth of the recursion tree is 0, then the entire problem is solved using an ordinary implementation of the Floyd-Warshall algorithm. This has been proven correct.

When the depth of the recursion tree is 1, the matrix is divided into four quadrants. The top level of recursion will make the 8 recursive calls shown in Figure 5b, where $A = B = C$. Each of these functions will then use an ordinary implementation of the Floyd-Warshall algorithm.

The first call, step 1, passes the Northwest quadrant as each argument. Since the function then uses an ordinary implementation of the Floyd-Warshall algorithm, this will correctly compute the Northwest quadrant of $D^k$ for $1 \le k \le N/2$.

The second call, step 2, computes the Northeast quadrants of $D^k$ for $1 \le k \le N/2$. Examining the dependencies for this computation shows that the data in the Northwest quadrant of $D^{k-1}$ is required in order to compute the Northeast quadrant of $D^k$. This dependency is satisfied by passing the Northwest quadrant as input to the function in step 2 and by the fact that the Northwest quadrant of $D^{k-1}$ was computed in step 1. In the same fashion, the third call, step 3, computes the Southwest quadrant of $D^k$ for $1 \le k \le N/2$ using data from the Northwest quadrant of $D^{k-1}$ computed in step 1.

The fourth call, step 4, requires data from both the Northeast and the Southwest quadrants of $D^{k-1}$. These quadrants are passed as input to the function and were computed in steps 2 and 3. Using these first 4 steps, we compute the complete $D^k$ for $1 \le k \le N/2$.

Figure 4b shows that steps 5 – 8 are the reverse of steps 1 – 4. In each step we compute the values for one quadrant of $D^k$ for $N/2 < k \le N$. All data required in each step is computed either in that step or in a previous step.

*Inductive step:*

Assume that the algorithm correctly computes the output when the depth of the recursion is d. When we consider the problem when the depth of recursion is d+1, each recursive call at the first level (Figure 4b), is a call to a problem in which the depth of recursion is d. Each of these calls has been assumed to run correctly, given that all data required is available at the time of execution. It was already shown that all data required for each of the 8 recursive calls at the top level is computed either during that step or in a previous step. Therefore, the algorithm runs correctly when the depth of recursion is d+1, and by induction, the algorithm runs correctly for all recursion depths. ■

**Theorem 3.2:** The recursive implementation reduces the processor-memory traffic by a factor of B, where $B = \mathrm{O}\left(\sqrt{C}\right)$. This is accomplished without any machine dependant setup cost, such as tuning of the block size.

**Proof:**

Note that the running time of this algorithm is given by

$$T(N) = 8 * T\left(\frac{N}{2}\right) = N^3 \qquad 1$$

Define the amount of processor memory traffic by the function D($x$). Without considering cache, the function behaves exactly as the running time.

$$D(N) = 8 * D\left(\frac{N}{2}\right) = N^3 \qquad 2$$

Consider the problem after $k$ recursive calls. At this point the problem size is $N/2^k$. There exists some $k$ such that $N/2^k = \mathrm{O}\left(\sqrt{C}\right)$, where $C$ = cache size. For simplicity we set $B = N/2^k$. At this point, all data will fit in the cache and no further traffic will occur for recursive calls below this point. Therefore:

$$D(B) = O\left(B^2\right) \qquad 3$$

By combining Equation 2 and Equation 3 it can be shown that:

$$D(N) = \frac{N^3}{B^3} * D(B) = \frac{N^3}{B} \quad (4)$$

Therefore, the processor-memory traffic is reduced by a factor of $B$. ■

**Theorem 3.3:** The recursive implementation reduces the traffic between the $i^{th}$ and the $(i\text{-}1)^{th}$ level of cache by a factor of $B_i$ at each level of the memory hierarchy, where $B_i = O(\sqrt{C_i})$.

**Proof:**

Note first of all, that no tuning was assumed when calculating the amount of processor-memory traffic in the proof of Theorem 3.2. Namely, Equation 3 holds for any $N$ and any $B$ where $B = O(\sqrt{C})$.

In order to prove Theorem 3.3, first consider the entire problem and the traffic between main memory and the $m^{th}$ level of cache (size $C_m$). By Theorem 3.2, the traffic will be reduced by $B_m$ where $B_m = O(\sqrt{C_m})$. Next consider each problem of size $B_m$ and the traffic between the $m^{th}$ level of cache and the $(m\text{-}1)^{th}$ level of cache (size $C_{m-1}$). By replacing $N$ in Theorem 3.2 by $B_m$, it can be shown that this traffic is reduced by a factor of $B_{m-1}$ where $B_{m-1} = O(\sqrt{C_{m-1}})$.

This simple extension of Theorem 3.2 can be done for each level of the memory hierarchy, and therefore the processor-memory traffic between the $i^{th}$ and the $(i\text{-}1)^{th}$ level of cache will be reduced by a factor of $B_i$, where $B_i = O(\sqrt{C_i})$. ■

Finally, recall from Lemma 3.1 that the lower bound on processor-memory traffic for the Floyd-Warshall algorithm is given by $\Omega(N^3/\sqrt{C})$, where $C$ is the cache size. Also recall from Theorem 3.2 the upper bound on processor-memory traffic that was shown for the recursive implementation was $O(N^3/B)$, where $B^2 = O(C)$. Given this information we have the following Theorem.

**Theorem 3.4:** Our recursive implementation is asymptotically optimal among all implementations of the Floyd-Warshall algorithm with respect to processor-memory traffic.

As a final note in the recursive implementation, we show up to 2x improvement when we set the base case such that the base case would utilize more of the cache closest to the processor. Once we reached a problem size $B$, where $B^2$ is on the order of the cache size, we execute a standard iterative implementation of the Floyd-Warshall algorithm. This improvement varied from one machine to the next and is due to the decrease in the overhead of recursion. It can be shown that the number of recursive calls in the recursive algorithm is reduced by a factor of $B^3$ when we stop the recursion at a problem of size $B$. A comparison of full recursion and recursion stopped at a larger block size is shown for the Pentium III and the UltraSPARC III in Figures 11 and 12.

In order to improve performance, $B^2$ must be chosen to be on the order of the L1 cache size. The simplest and possibly the most accurate method of choosing $B$ is to experiment with various tile sizes. This is the method that the Automatically Tuned Linear Algebra Subroutines (ATLAS) project employs [30]. However, it is beneficial to find an estimate of the optimal tile size. A block size selection heuristic for finding this estimate is discussed in [22], and outlined here.

- Use the 2:1 rule of thumb from [14] 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 tiles of data. Self-interference misses are eliminated by the data being in contiguous locations within each tile and cross interference misses are eliminated by the associativity.
- Choose $B$ by Equation 5, where $d$ is the size of one element and $C$ is the adjusted cache size. This minimizes capacity misses.

$$3 * B^2 * d = C \quad (5)$$

The baseline we use for our experiments is a straightforward implementation of the Floyd-Warshall algorithm. It was shown in [22] that standard optimizations yield limited performance increases on

most machines. The Simplescalar results in Table 1 for the recursive implementation show a 30% decrease in level-1 cache misses and a 2x decrease in level-2 cache misses for problem sizes of 1024 and 2048. In order to verify the improvements on real machines, we compare the recursive implementation of the Floyd-Warshall algorithm with the baseline. For these experiments the best block size was found experimentally. The results show a 10x improvement in overall execution time on the Alpha, better than 7x improvement on the Pentium III and the MIPS, and almost a 3x improvement on the UltraSPARC III. These results are shown in Figures 7-10. Figures 11 and 12 show that a 2x improvement in execution time on the UltraSPARC III can be gained by choosing the optimal base block size. Likewise, a 30% improvement can be gained on the Pentium III. Differences in performance gains between machines are expected, due to the wide variance in cache parameters and miss penalties.

### 3.2. A Tiled Implementation for the Floyd-Warshall Algorithm

Compiler groups have used tiling to achieve higher data reuse in looped code. Unfortunately, the data dependencies from one k-loop to the next in the Floyd-Warshall algorithm make it impossible for current compilers including research compilers to perform 3 levels of tiling. In order to tile the outermost loop we must cleverly reorder the tiles in such a way that satisfies data dependencies from one *k*-loop to the next as well as within each k-loop.

Consider the following tiled implementation of the Floyd-Warshall algorithm. Tile the problem into *B*x*B* tiles. During the $k^{th}$ block iteration, update first the $(k,k)^{th}$ tile, then the remainder of the $k^{th}$ row and $k^{th}$ column, then the rest of the matrix. Figure 5 shows an example matrix tiled into a 4x4 matrix of blocks. Each block is of size *B*x*B*. During each outermost loop, we would update first the black tile representing the $(k,k)^{th}$ tile, then the grey tiles, then the white tiles. In this way we satisfy all dependencies from each $k^{th}$ loop to the next as well as all dependencies within each $k^{th}$ loop.

**Theorem 3.5:** The new tiled implementation of the Floyd-Warshall algorithm reduces the processor memory traffic by a factor of $B$ where $B^2$ is on the order of the cache size.

**Proof sketch:** At each block we perform $B^3$ operations. There are $N/B$ x $N/B$ blocks in the array and we pass through each block $N/B$ times. This gives us a total of $N^3$ operations. In order to process each block we require only $3*B^2$ elements. This gives us a total of $N^3/B$ total processor-memory traffic. ■

Given this upper bound on traffic for the tiled implementation and the lower bound shown in Lemma 3.1, we have.

**Theorem 3.6:** The new tiled implementation is asymptotically optimal among all implementations of the Floyd-Warshall algorithm with respect to processor-memory traffic.

When implementing the tiled implementation of the Floyd-Warshall algorithm, it is important to use the best possible block size. As mentioned in Section 3.1, the best block size should be found experimentally, and the block size selection heuristic discussed in Section 3.1 can be used to give a rough bound on the best block size. However, when implementing the tiled implementation, 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 *B* the search space should be expanded accordingly.

Simplescalar results for the tiled implementation are shown in Table 2. These results show a 2x improvement in level-2 cache misses and a 30% improvement in level-1 cache misses. Experimental results show a 10x improvement in execution time for the Alpha, better than 7x improvement for the Pentium III and the MIPS and roughly a 3x improvement for the UltraSPARC III (See Figures 13-16).

### 3.3. Data Layout Issues

It is also important to consider data layout when implementing any algorithm. It has been shown by a number of groups that data layouts tuned to the data access pattern of the algorithm can reduce both TLB and cache misses (see for example [20], [22], and [4]). In the case of the recursive algorithm, the access pattern is matched by a Z-Morton data layout. The Z-Morton ordering is a recursive layout defined as follows: Divide the original matrix into 4 quadrants and lay these tiles in memory in the order NW, NE, SW, SE. Recursively divide each quadrant until a limiting condition is reach. This smallest tile is typically laid out in either row or column major fashion (see Figure 18). See [5] for a more formal definition of the Morton ordering.

In the case of the tiled implementation, the Block Data Layout (BDL) matches the access pattern. 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 17). By setting the block size equal to the tile size in the tiled computation, the data layout will exactly match the data access pattern.

We experimented with both of these data layouts for each of the algorithms. The results are shown in Tables 3 and 4. All of the execution times were within 15% of each other with the Z-Morton data layout winning slightly for the recursive implementation and the BDL winning slightly for the tiled implementation. The fact that the Z-Morton was slightly better for the recursive implementation and likewise the BDL for the tiled implementation was exactly as expected, since they match the data access pattern most closely. The closeness of the results is mostly likely due to the fact that the majority of the data reuse is within the final block. Since both of these data layouts have the final block laid out in contiguous memory locations, they perform equally well.

It is also important to note that the Z-Morton data layout has a very complex index computation, which can only be hidden in a recursive algorithm. The BDL has a very simple index computation in comparison. Therefore it is significant to show that for non-recursive algorithms, the BDL performs just as well or better, while avoiding the overhead of a complex index computation.

## 4. Optimizing the Single-Source Shortest Path Problem

In this section we discuss cache-friendly optimizations of Dijkstra's algorithm for the single-source shortest path problem. We first consider the input graph representation in Section 4.1. For small problem sizes, the graph representation represents the majority of the processor memory traffic. We present a cache friendly representation that improves performance by 20% to 2x. In Section 4.2, we discuss optimizing the priority queue. As the heap is the most common and the most optimal implementation of the priority queue, we focus on designing a cache-friendly heap data structure. We compare our implementation to the asymptotically optimal implementation of the heap in Dijkstra's algorithm with respect to time complexity, the Fibonacci heap.

### 4.1. Optimizing the Graph Representation

For the problem sizes that we have considered in Section 3, i.e. 1K to 4K nodes, the graph representation represents the majority of traffic in Dijkstra's algorithm. This is due to the fact that the priority queue is very small and fits entirely within the cache. Cache conflicts between the graph representation and the priority queue can pose a problem even at these small sizes, but we assume for this section that the conflicts are minimal. For all problem sizes, the size of the priority queue starts at $N$ and decreases throughout the computation. In contrast, the graph representation will be of size $O(N + E)$, where $E = O(N^2)$ for dense graphs.

One difficulty we face when optimizing the graph representation is the access pattern. Each element in the representation is accessed exactly once. For each node that is extracted from the heap, the corresponding list of adjacent nodes is read from the graph representation. Once each node is extracted from the heap, the computation is complete. In this context, we can take advantage of two things. The first is prefetching. Modern processors perform aggressive prefetching in order to hide memory latencies. The second is to optimize at the cache line level. In this case, a single miss would bring in multiple elements that would subsequently be accessed and result in cache hits. This is known as minimizing cache pollution.

There are two commonly used graph representations. The adjacency matrix is an $N$x$N$ matrix, where the $(i,j)^{th}$ element is the cost from the $i^{th}$ element to the $j^{th}$ element. This representation is of size $O(N^2)$. It has the nice property that elements are accessed in a contiguous fashion and therefore, cache pollution will be minimized and prefetching will be maximized. However, for sparse graphs, the size of this representation is inefficient. The adjacency list representation is a pointer-based representation where a list of adjacent nodes is stored for each node in the graph. Each node in the list includes the cost of the edge from the given node to the adjacent node. This representation has the property of being of optimal size for all graphs, namely $O(N+E)$. However, the fact that it is pointer based, leads to cache pollution and difficulties in prefetching.

Consider a simple combination of these two representations. For each node in the graph, we have a corresponding array of adjacent nodes. The size of this array is exactly the out-degree of the given node. There are simple methods to construct this representation when the out-degree is not known until run time. For this representation, the elements at each point in the array look similar to the elements stored in the adjacency list. Each element must store both the cost of the path and the index of the adjacent node. Since the size of each array is exactly the out-degree of the corresponding node, the size of this representation is exactly $O(N+E)$. This makes it optimal with respect to size. Also, since the elements are stored in arrays and therefore in contiguous memory locations, the cache pollution will be minimized and prefetching will be maximized. Subsequently this representation will be referred to as the *adjacency array representation*.

In order to demonstrate the performance improvements using our graph representation, we performed Simplescalar simulations as well as experiments on two different machines, the Pentium III and UltraSPARC III, for Dijkstra's algorithm. The Simplescalar simulations show a significant improvement in level-2 cache misses for the adjacency array representation compared with the adjacency list representation (see Table 5). This is due to the reduction in cache pollution and increase in prefetching that was predicted. The experimental results also demonstrate improved performance. Figures 20 - 23 show a 2x improvement for Dijkstra's algorithm on the Pentium III and a 20% improvement on the UltraSPARC III This significant difference in performance is due in part to the difference in the memory hierarchy of these two architectures.

A second comparison to observe is between the Floyd-Warshall algorithm and Dijkstra's algorithm for very sparse graphs, i.e. edge densities less than 20%. For these graphs, Dijkstra's algorithm is more efficient for the all pairs shortest path problem. By using the adjacency array representation of the graph in Dijkstra's algorithm, the range of graphs over which Dijkstra's algorithm outperforms the Floyd-Warshall algorithm, can be increased. Figures 24 and 25 show a comparison of the best Floyd-Warshall algorithm with Dijkstra's algorithm for sparse graphs. On the Pentium III, we were able to increase the range for Dijkstra's algorithm from densities up to 5% to densities up to 20%. On the UltraSPARC III we increased the range from densities up to 20% to densities up to 35%.

### 4.2. Optimizing the Priority Queue

For very large problems, those in which $N$ is much larger than the cache size, the priority queue can generate a large number of cache misses. For this reason, we discuss optimizing the priority queue. Due to the length of time required to simulate or execute either Dijkstra's algorithm or Prim's algorithm for very large problem sizes, we have defined an architecture for which small problem sizes will stress the memory hierarchy. This architecture has a 4KB level-1 cache and 16KB level-2 cache. The problem size we used was 4096 nodes with a density of 90%. Using these parameters we show Simplescalar results to demonstrate improved cache performance.

Optimizing the priority queue to improve cache performance presents a very unique set of challenges. The optimal implementation with respect to running time uses the Fibonacci heap. This implementation is a pointer based data structure that is extremely dynamic. Nodes are moved within each tree and among trees in almost every operation. In [6], Chilimbi et al. discusses the optimization of various pointer based data structures. However, he focuses on very *static* structures where the main access pattern is a root to leaf path traversal. This work cannot be directly applied to the heap for two reasons. The first is the dynamic nature of the heap that was just mentioned. The second is the fact that the access pattern is often not a simple root to leaf path. For example an update operation will start by accessing an element in the middle of the tree and then traverse up the tree for some time. In [17], LaMarca et. al. discusses the analysis and optimization of the heap. The heap analysis and experiments that are discussed here differ significantly from that work. We assume that the heap is very dynamic, while the analysis done by LaMarca is done in the Hold model, one in which the heap is static for most of the operations. LaMarca also isolates the heap for analysis and experimentation, whereas we conduct all of our simulations on the complete Dijkstra's algorithm.

In order to establish a baseline, we examine the performance of the Fibonacci heap in Dijkstra's algorithm. The Fibonacci heap represents the optimal implementation of the priority queue with respect to time complexity. The Extract-min operation requires O(lg $V$) time and the Update operation takes O(1), constant time when amortized over all operations. The total time complexity for Dijkstra's single source shortest path problem using the Fibonacci heap is O(($V$lg$V$ + $E$).

In contrast to the Fibonacci heap consider the clustered heap, a more cache-friendly implementation of the priority queue. In [6], Chilimbi et. al. presents a simple ancestor-descendant clustering within cache lines for static trees. By placing $d$ element subtrees into cache lines, you will incur only one miss to access a path through any given subtree. When compared with a breadth-first mapping this will decrease processor-memory traffic by a factor of lg($d$), where lg($d$) is the height of the subtree in a single cache line and $d$ is the number of elements that fit into a cache line. When applying this technique to the heap, there are a few factors that must be considered. The first is the dynamic nature of the heap in the context of Dijkstra's algorithm. We are able to decrease the dynamic nature some by applying the clustering to a standard implementation of the heap. This presents much less data movement than in the Fibonacci heap. One nice property of the heap is its simple index computation for ancestors and descendants. Unfortunately, this property does not remain once we apply the clustering layout. In order to realize the clustering layout, while sustaining the property of simple index computation, we place an indirection layer between the access and the data. This layer is actually a standard heap containing pointers to the actual data. The size of this indirection heap is much smaller than the real heap and therefore, presents few problems with respect to the cache. Figure 19 gives a graphical example of using the indirection heap to access the clustered data layout. In this fashion we can decrease the cost of access by a factor of lg($d$) and retain the property of a simple index computation.

As a second cache friendly implementation of the priority queue, consider a partitioned heap. In this case the original large heap would be partitioned into $k$ independent heaps, each of size $c$, where $c$

is the on the order of the cache size. If we divide the original heap by placing the first $N/k$ nodes into the first heap and so on, we can order the updates following each extract-min, such that each independent heap will be brought into the cache at most once. In this way, no node in the partitioned heap will be brought into the cache more than once, and the amount of traffic is bounded by $V$. Since we perform $V$ extract-min's and subsequent updates, the total traffic for the Update operations will be bounded by $V^2$ compared with a possible $E*\lg(V)$, where $E = O(V^2)$ for dense graphs. In order to find the minimum, we must examine the minimum of each of the $k$ independent heaps rather than just extracting the root of our original large heap. Once the minimum is located, there is a small reduction in the cost for the actual extraction. The total cost for the extract-min will then be $O(k + \lg(V/k))$.

In order to demonstrate the cache performance of the clustered heap and the partitioned heap, we define an architecture in which the cache will be stressed as mentioned earlier and perform Simplescalar simulations for Dijkstra's algorithm. The result is shown in Table 6. Notice the reduction in the level-2 cache miss rate for the clustered heap and the partitioned heap vs. the Fibonacci heap. The results also show that the partitioned heap performs slightly better than the clustered heap in both level-1 and level-2 cache miss rate.

## 5. Optimizing the Minimum Spanning Tree Problem

As mentioned in Section 2, Prim's algorithm for minimum spanning tree is very similar to Dijkstra's algorithm for the single source shortest path problem. For this reason the optimizations applicable to Dijkstra's algorithm are applicable to Prim's algorithm. Figures 26 - 29 show the result of applying the optimization to the graph representation discussed in Section 4.1 to Prim's algorithm. Recall that this was an optimization to the graph representation replacing the adjacency list representation with the adjacency array representation. Our results show a 2x improvement on the Pentium III running Windows 2000 and 20% for the UltraSPARC III. These results are for problem sizes 2048 and 4096. This result is very similar to the results we saw for the same comparison in Dijkstra's algorithm. Recall that our Simplescalar results for Dijkstra's algorithm showed an improvement in the level-2 cache misses. Based on the similarity between Dijkstra's algorithm and Prim's algorithm, we could expect similar cache performance improvements for Prim's algorithm.

## 6. Conclusion

Using various optimizations for graph algorithms, we have showed a 3x to 10x improvement for the Floyd-Warshall algorithm and a 20% to 2x improvement for Dijkstra's algorithm and Prim's algorithm. Our optimizations to the Floyd-Warshall algorithm represent a novel recursive implementation as well as a novel tiled implementation of the algorithm. We also compared the performance of a non-linear data layout with that of a simple block data layout. For Dijkstra's algorithm and Prim's algorithm, we presented a cache-friendly graph representation that gave significant performance improvements. We also discussed optimization of the priority queue and showed significant improvements in cache miss rate for our clustered heap and our partitioned heap compared with the Fibonacci heap.

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.

## 7. 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, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. *ACM Symposium on Parallel Algorithms and Architectures*, 1999.

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

[6] 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.

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

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

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

[10] M. Fredman and R. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the Association for Computing Machinery*, vol. 34, no. 3, July 1987.

[11] Jeremy D. Frens and David S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proc. of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.

[12] 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.

[13] A. Gonzalez, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating Cache Conflict Misses through XOR-Based Placement Functions. In *Proc. of 1997 International Conference on Supercomputing*, Vienne, Austria, July, 1997.

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

[15] 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.

[16] 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.

[17] A. LaMarca and R. E. Ladner. The Influence of Caches on the Performance of Heaps. *ACM Journal of Experimental Algorithmics*, 1, 1996.

[18] D. Padua. The Fortran I Compiler. *Computing in Science & Engineering*, January/February 2000.

[19] D. A. Padua. Outline of a Roadmap for Compiler Technology. *IEEE Computational Science & Engineering*, Fall 1996.

[20] 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.

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

[22] M. Penner and V. K. Prasanna. Cache-Friendly Implementations of Transitive Closure. In *Proc. of International Conference on Parallel Architectures and Compiler Techniques*, Barcelona, Spain, September 2001.

[23] G. Rivera and C. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[24] 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.

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

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

[27] 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.

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

[29] 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.

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

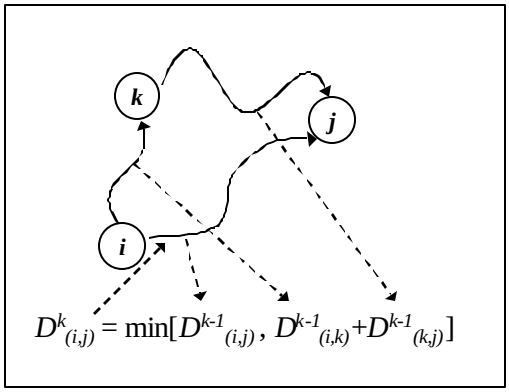



**Figure 1: $k^{th}$ step of Floyd-Warshall Algorithm**

```
S = Ø
Q = V[G]
While Q ≠ Ø
      u = Extract-Min(Q)
      S = S U {u}
      For each vertex v ∈ Adj[u]
            Update d[v]
```

**Figure 2: Dijkstra's algorithm**

**Definitions: V[G] is the set of vertices, Adj[u] is the adjacency list for vertex u, d[v] is the distance from the source to v**

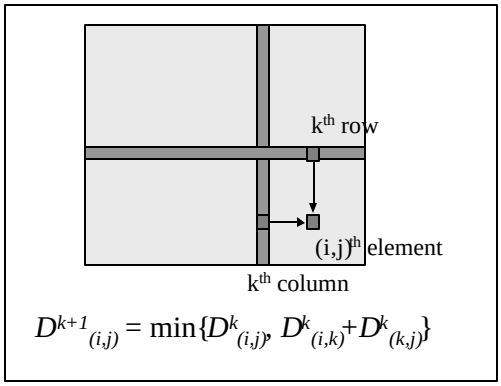



**Figure 3: $k^{th}$ iteration of outer loop in Floyd-Warshall Algorithm**

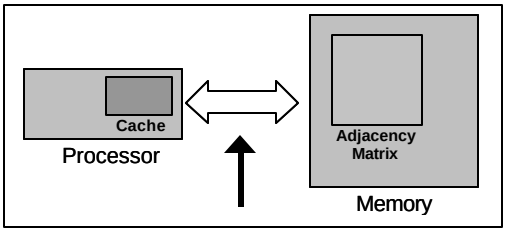



**Figure 4: Processor-Memory traffic diagram**

```
Floyd-Warshall (A)
{
        A11 = min(A11, A11+A11);
        A12 = min(A12, A11+A12);
        A21 = min(A21, A21+A11);
        A22 = min(A22, A21+A12);
        A22 = min(A22, A22+A22);
        A21 = min(A21, A22+A21);
        A12 = min(A12, A12+A22);
        A11 = min(A11, A12+A21);
}
```

```
FWR (A, B, C)
{
      if (not base case) {
            FWR(A11, B11, C11);
            FWR(A12, B11, C12);
            FWR(A21, B21, C11);
            FWR(A22, B21, C12);
            FWR(A22, B22, C22);
            FWR(A21, B22, C21);
            FWR(A12, B12, C22);
            FWR(A11, B12, C21);
      }
      else {
            /* run standard Floyd-
               Warshall */
            …
      }
}
```

**Figure 5, a&b: Recursive implementation of the Floyd-Warshall algorithm (FWR)**

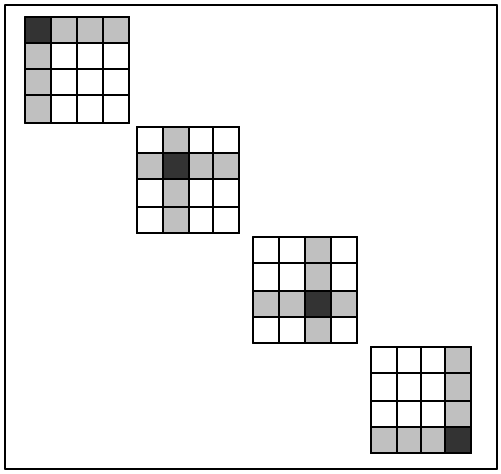


**Figure 6: Diagram for tiled implementation of the Floyd-Warshall algorithm**

**Data level-1 cache misses**

| N | Baseline | Recursive |
|---|---|---|
| **1024** | 0.806 | 0.546 |
| **2048** | 6.442 | 4.362 |

(billions)

**Data level-2 cache misses**

| N | Baseline | Recursive |
|---|---|---|
| **1024** | 0.537 | 0.280 |
| **2048** | 4.294 | 2.232 |

(millions)

**Table 1: Simplescalar result (Recursive Floyd-Warshall)**

**Data level-1 cache misses**

| N | Baseline | Tiled |
|---|---|---|
| **1024** | 0.806 | 0.542 |
| **2048** | 6.442 | 4.326 |

(billions)

**Data level-2 cache misses**

| N | Baseline | Tiled |
|---|---|---|
| **1024** | 0.537 | 0.276 |
| **2048** | 4.294 | 2.195 |

(millions)

**Table 2: Simplescalar result (Tiled Floyd-Warshall)**

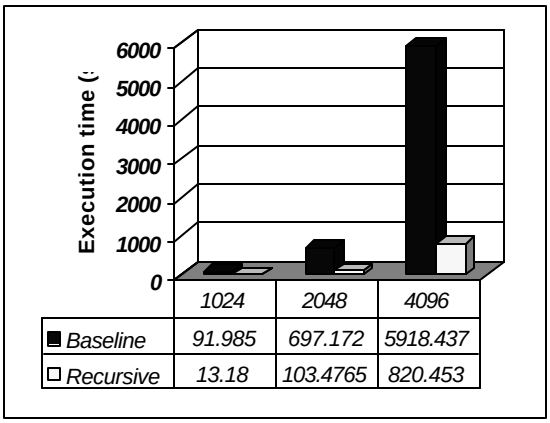



**Figure 7: Baseline vs. Recursive implementation on Pentium III**

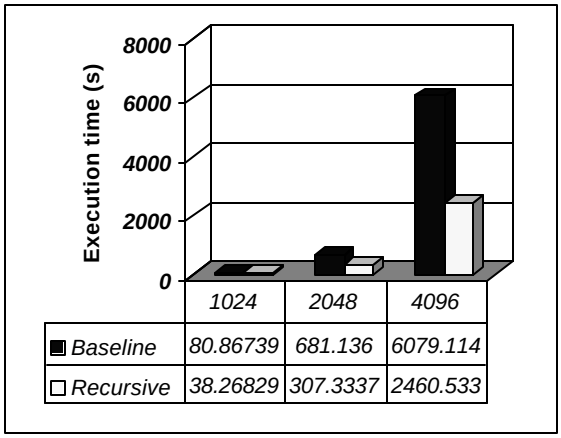



**Figure 8: Baseline vs. Recursive implementation on UltraSPARC III**

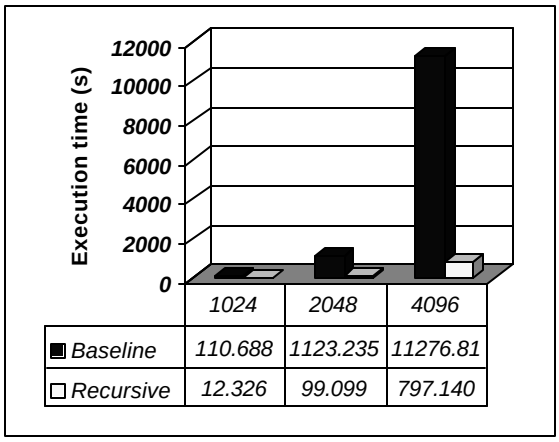



**Figure 9: Baseline vs. Recursive implementation on Alpha**

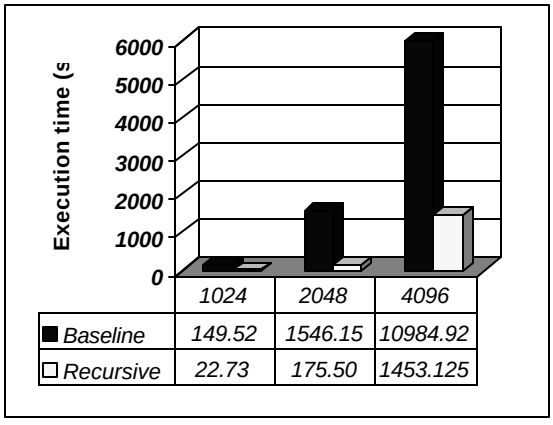



**Figure 10: Baseline vs. Recursive implementation on MIPS**

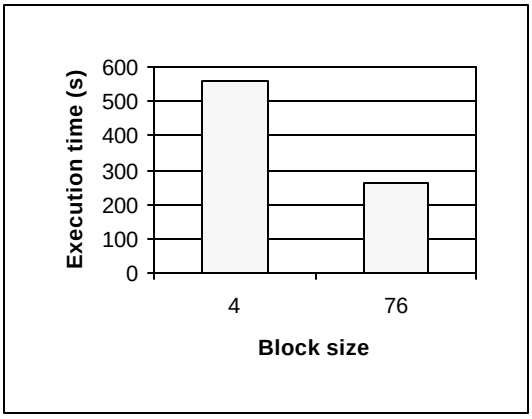



**Figure 11: Execution times for different base block sizes (Full recursion vs. Optimal block size) on UltraSPARC III, *N* = 2048**

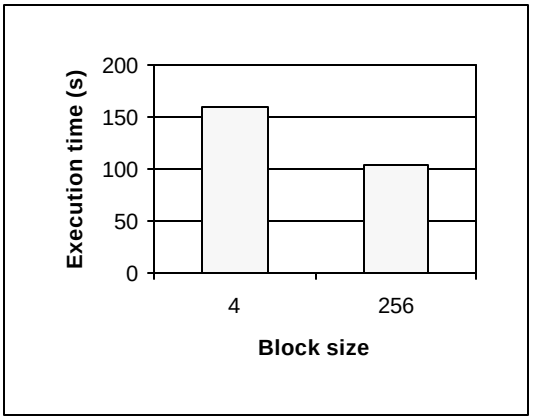



**Figure 12: Execution times for different base block sizes (Full recursion vs. Optimal block size) on Pentium III, *N* = 2048**

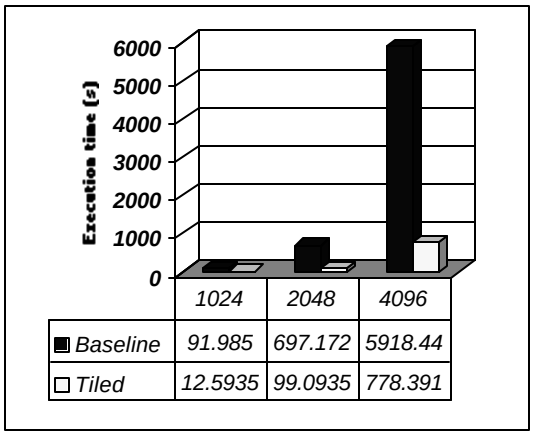



**Figure 13: Baseline vs. Tiled implementation on Pentium III**

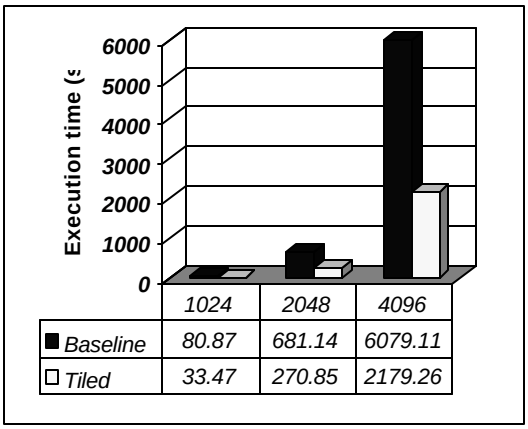



**Figure 14: Baseline vs. Tiled implementation on UltraSPARC III**

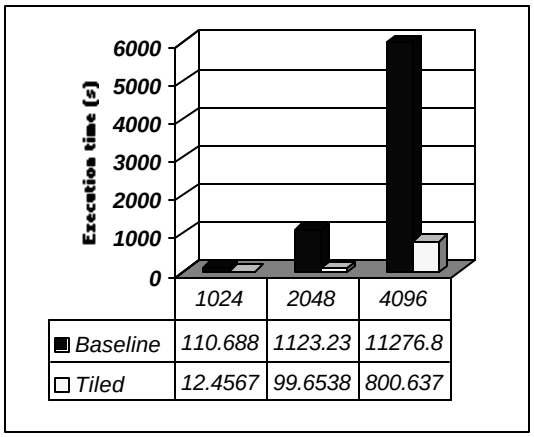



**Figure 15: Baseline vs. Tiled implementation on Alpha**

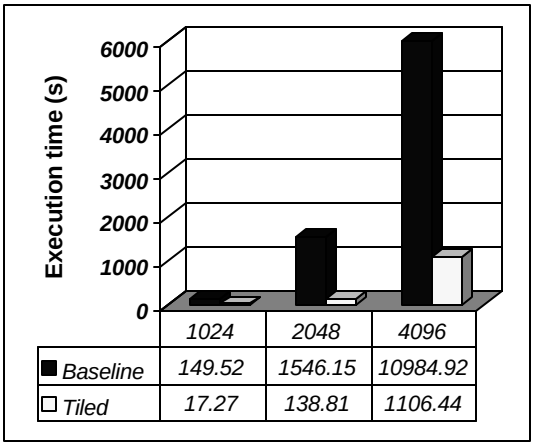



**Figure 16: Baseline vs. Tiled implementation on MIPS**

**Recursive Implementation**

| N | Morton Layout | Block Data Layout |
|---|---|---|
| **2048** | 103.48 | 111.42 |
| **4096** | 820.45 | 878.89 |

(sec)

**Tiled Implementation**

| N | Morton Layout | Block Data Layout |
|---|---|---|
| **2048** | 99.25 | 99.39 |
| **4096** | 779.53 | 780.41 |

(sec)

**Table 3: Pentium III results for data layout comparison.**

**Recursive Implementation**

| N | Morton Layout | Block Data Layout |
|---|---|---|
| **2048** | 307.33 | 311.26 |
| **4096** | 2460.53 | 2488.88 |

(sec)

**Tiled Implementation**

| N | Morton Layout | Block Data Layout |
|---|---|---|
| **2048** | 278.48 | 271.35 |
| **4096** | 2248.20 | 2184.09 |

(sec)

**Table 4: Ultrasparc III results for data layout comparison.**

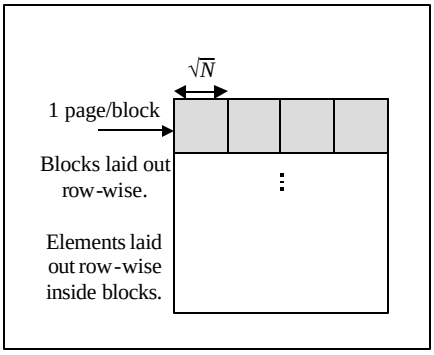



**Figure 17: The Block Data Layout**

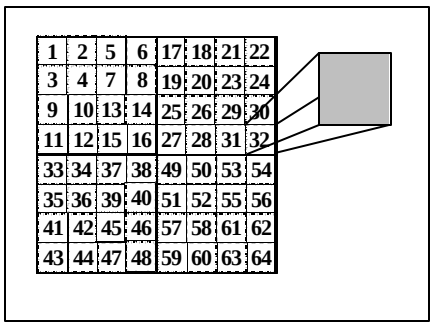



**Figure 18: The Morton Layout**

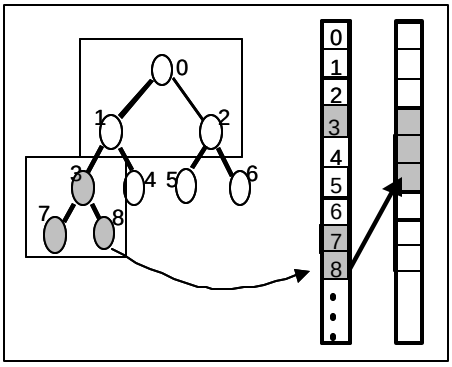



**Figure 19: Clustered Heap Operation**

**Cache miss rates**

| | Linked-List | Array |
|---|---|---|
| **D-Level 1** | 0.2936 | 0.2622 |
| **D-Level 2** | 0.4242 | 0.3545 |

(DL1:16k, DL2:256k, Input: 2048 nodes, 0.9 density)

**Table 5: Simplescalar results for Linked-List and Array graph representation**

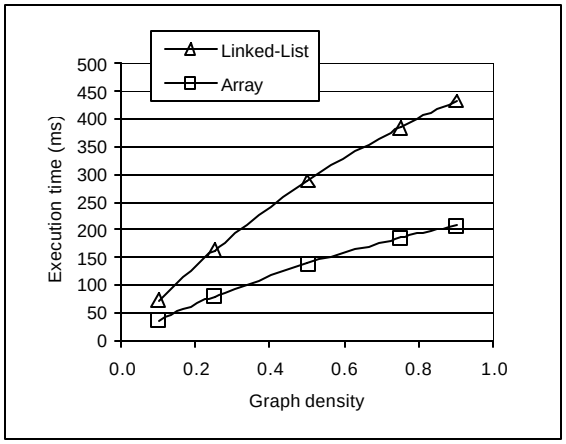



**Figure 20: Linked-List vs Array graph representation for Dijkstra's alg. on Pentium III, *N* = 2048**

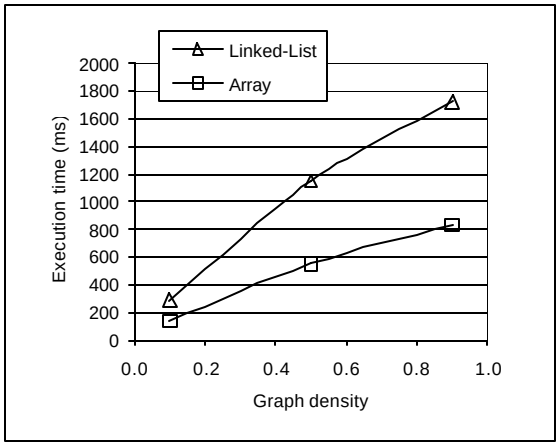



**Figure 21: Linked-List vs Array graph representation for Dijkstra's alg. on Pentium III, *N* = 4096**

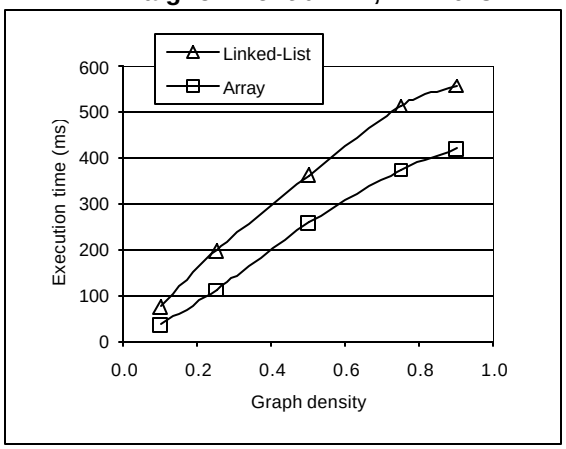



**Figure 22: Linked-List vs Array graph representation for Dijkstra's alg. on UltraSPARC III, *N* = 2048**

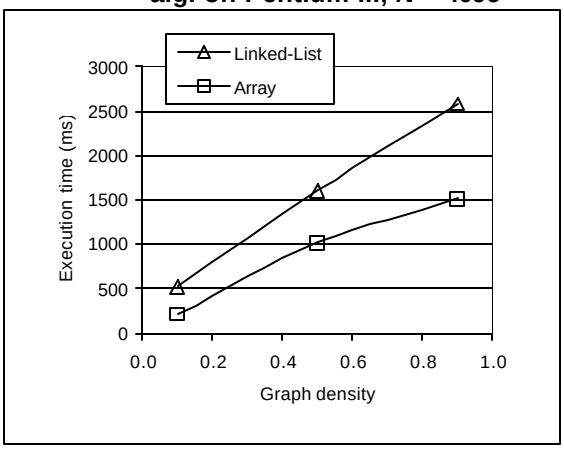



**Figure 23: Linked-List vs Array graph representation for Dijkstra's alg. on UltraSPARC III, *N* = 4096**

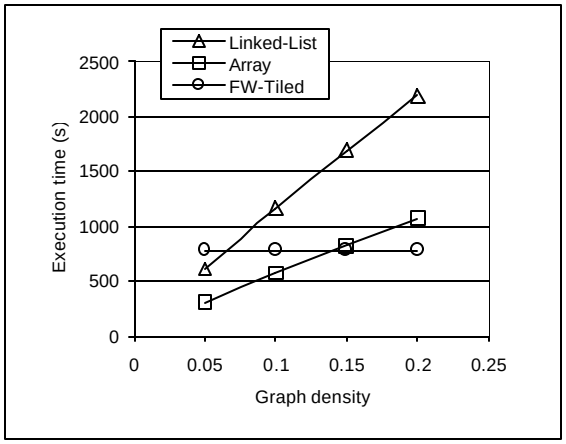



**Figure 24: Linked-List and Array graph representation for Dijkstra's algorithm vs. best Floyd-Warshall on Pentium III, *N* = 2048**

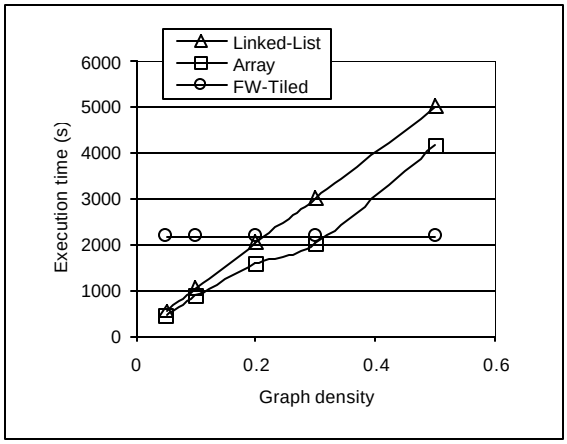



**Figure 25: Linked-List and Array graph representation for Dijkstra's algorithm vs. best Floyd-Warshall on UltraSPARC III, *N* = 2048**

| | Baseline (Fibonacci heap) | Clustered Heap | *k*-Partitioned Heap |
|---|---|---|---|
| Computational complexity | O((VlgV + E) | O((V+E)lgV) | O(Vk+(V+E)lg(V/k)) |
| Data Level-1 cache miss rate | 0.0427 | 0.0479 | 0.0455 |
| Data Level-2 cache miss rate | 0.5927 | 0.3138 | 0.2914 |
| Data Level-1 cache misses | 59521364 | 62833311 | 61504500 |
| Data Level-2 cache misses | 51973614 | 54356226 | 50013594 |

(DL1:2k, DL2:8k, Input graph: 4096 nodes with 0.9 Density)

**Table 6: Simplescalar results for Dijkstra's algorithm with various heap implementations**

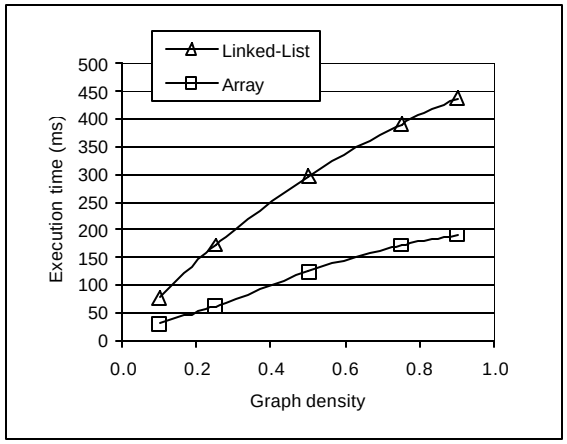



**Figure 26: Linked-List vs Array graph representation for Prim's alg. on Pentium III, *N* = 2048**

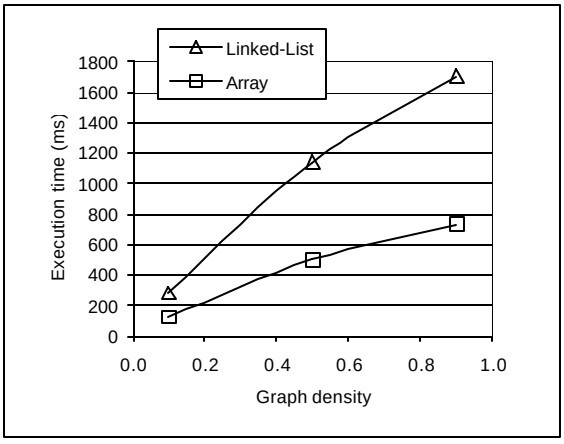



**Figure 27: Linked-List vs Array graph representation for Prim's alg. on Pentium III, *N* = 4096**

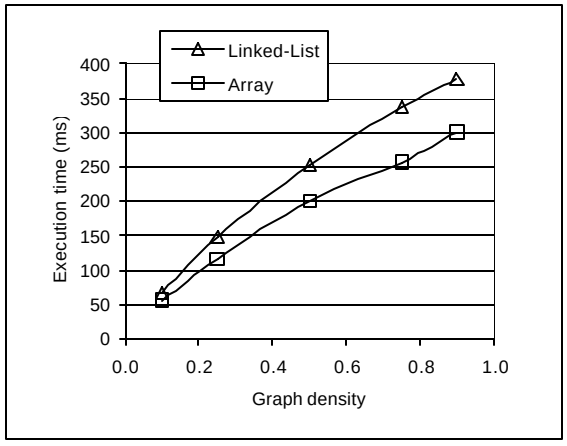



**Figure 28: Linked-List vs Array graph representation for Prim's alg. on UltraSPARC III, *N* = 2048**

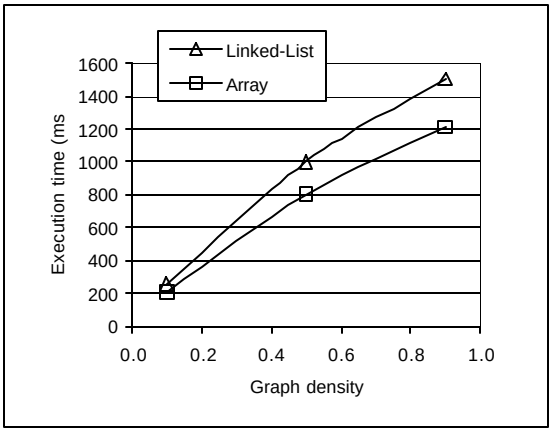



**Figure 29: Linked-List vs Array graph representation for Prim's alg. on UltraSPARC III, *N* = 2048**

# Optimizing Graph Algorithms for Improved Cache Performance

Joon-Sang Park, Michael Penner,
and Viktor K. Prasanna

CENG 02-01

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213-740-4483)

# Optimizing Graph Algorithms for Improved Cache Performance[*+]

Joon-Sang Park, Michael Penner, and Viktor K. Prasanna
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
{jsp, mipenner, prasanna}@usc.edu
http://advisor.usc.edu

**Abstract**

*Graph algorithms are fundamental in a wide variety of fields, and while much focus has been on optimizing various algorithms for improved cache performance, little focus has been on the area of graph algorithms. The reasons for this are varied, but at the core is that graph algorithms pose a very different and complex challenge to improving cache performance. In this paper, we present a new recursive implementation for the fundamental graph problem of Transitive Closure, namely the Floyd-Warshall Algorithm, and prove its optimality with respect to processor-memory traffic. Using this cache-oblivious implementation we show more than a* 6x *improvement in execution time on three different architectures. We also discuss the impact of data layout on cache performance in the context of a tiled implementation of the Floyd-Warshall algorithm. Secondly, we address Dijkstra's algorithm for the single-source shortest-path problem and Prim's algorithm for Minimum Spanning Tree, for which neither tiling nor recursion can be directly applied. For these algorithms, we demonstrate up to a* 2x *improvement by using a cache-friendly graph representation. Finally, we apply both the cache friendly graph representation and the basic idea of tiling to the problem of graph matching. Using these techniques we show performance improvements of* 2x – 3x. *Experimental results are shown for the Pentium III, UltraSPARC III, Alpha 21264, and MIPS R12000 machines. Problem sizes ranged from 1024 to 4096 vertices for the Floyd-Warshall algorithm and up to 65536 vertices for Dijkstra's algorithm, Prim's algorithm, and graph matching. We demonstrate improved cache performance using the SimpleScalar simulator.*

[*] 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.
[+] A previous version of this paper appears in *Proceedings of the International Parallel and Distributed Processing Symposium,* April 2002.

## 1. Introduction

The motivation for this work is what is commonly referred to as the processor-memory gap. While memory density has been growing rapidly, the speed of memory has been far outpaced by the speed of modern processors. Current latencies to memory are on the order of 100 processor cycles. This phenomenon has resulted in severe application level performance degradation on high-end systems. This problem has been well studied for many dense linear algebra problems, such as matrix multiplication and FFT (see for example, [23][32][36]). A similar problem is also present and well studied in I/O systems (see for example, [17][33]).

A number of groups are attempting to improve performance by performing computations in memory. Smart memory or processing in memory takes advantage of the high on chip bandwidth of memory to perform data intensive operations (see for example, [4][20][37]). Other groups are attacking the problem in software; either in the compiler through reordering instructions and prefetching (see for example, [16][18][27]) or through complex data layouts to improve cache performance (see for example, [6][10][13]).

Achieving better overall performance by optimizing cache performance is a difficult problem. The performance of deep memory hierarchies present in most modern processors has been shown to differ significantly from predictions based on a single level of cache. 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 penalties vary among processors and cause large variations in the effectiveness of cache performance optimizations.

The area of graph problems is fundamental in a wide variety of fields, most notably network routing, distributed computing, and computer aided circuit design. Network routing in particular is a rapidly growing problem with the explosion of the Internet. Routing tables are growing in size and the frequency of updates is pushing the limits of current routers. Graph problems pose unique challenges to improving cache performance due to their irregular data access patterns. These challenges often cannot be handled using standard cache-friendly optimizations [9]. The focus of this research is to develop methods of meeting these challenges. A suite of data intensive kernels or stressmarks designed to stress the memory hierarchy is discussed in [21] & [22]. The transitive closure problem discussed in this paper is from the stressmark suite.

In this paper we present a number of cache-friendly optimizations to the Floyd-Warshall algorithm, Dijkstra's algorithm, Prim's algorithm, and graph matching. For the Floyd-Warshall algorithm we present a *cache-oblivious* recursive implementation that achieves more than a 6x improvement over the baseline implementation on three different architectures. We also show that by tuning the base case for the recursion, we can further improve performance by up to 2x. We also show analysis and discuss the impact of data layout on cache performance in the context of a tiled implementation of the Floyd-Warshall algorithm. While these techniques are well known for dense linear algebra problems such as matrix multiply, their application to transitive closure faces a significantly different set of challenges. Note that today's state of the art research compilers cannot generate these implementations [9].

There are some natural combinations of implementation and data layout that decrease overhead costs, such as index computation, and yield performance advantage. In this paper, we show that the recursive and tiled implementations of the Floyd-Warshall algorithm perform roughly equal with either the Morton layout or the Block Data Layout.

For Dijkstra's algorithm and Prim's algorithm, to which tiling and recursion are not directly applicable, we use a known cache-friendly graph representation. By using a data layout for the graph representation that matches the access pattern we show up to a 2x improvement in execution time.

Finally, we use the techniques discussed with respect to the Floyd-Warshall algorithm and Dijkstra's algorithm to optimize cache performance for the problem of graph matching. The algorithm we use is a primitive graph matching algorithm for bipartite graphs. We first apply the cache friendly graph representation used for Dijkstra's algorithm and Prim's algorithm, since the data access pattern to the graph is similar. We then use the idea of tiling to reduce the working set size. Performance improvements were in the range of 2x to 3x depending on the density of the graph and the quality of the partitioning done to accomplish tiling.

The remainder of this paper is organized as follows: In Section 2 we give the background needed and briefly summarize some related work in the areas of cache optimization and compiler optimizations. In Section 3 we discuss optimizing the Floyd-Warshall algorithm. In Section 4 we discuss optimizing Dijkstra's algorithm. In Section 5 we apply the optimizations discussed in Section 4 to Prim's algorithm. In Section 6 we discuss applying the techniques to the problem of graph matching. Finally, in Section 7 we draw conclusions.

## 2. Background and Related Work

In this section we give the background information required in our discussion of various optimizations in Section 3 - 6. In Section 2.1 we give a brief outline of the graph algorithms. Those readers comfortable with the algorithms can skip this. For more details of these algorithms see [7] or [14]. In Section 2.2 we give some background on cache-based architectures and optimizing algorithms for improved cache performance. In Section 2.3 we discuss some of the challenges that are faced in making the transitive closure problem cache-friendly. We also discuss the model that we use to analyze cache performance and the four architectures that we use for experimentation throughout the paper. 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. Overview of Key Graph Algorithms

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 $N$, $N$x$N$ matrices where $D^k$ is the $k^{\text{th}}$ 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 cost matrix for the given graph $G$. We can think of the algorithm as composed of $N$ steps. At each $k^{\text{th}}$ step, we compute $D^k$ using the data from $D^{k-1}$ in the manner shown below for each $(i, j)^{\text{th}}$ value. Pseudo-code is given in Figure 1.

$$D^k{}_{(i,j)} = \min\left(D^{k-1}{}_{(i,j)}, D^{k-1}{}_{(i,k)} + D^{k-1}{}_{(k,j)}\right)$$

Dijkstra's algorithm is designed to solve the single-source shortest path problem. It does this by repeatedly extracting from a priority queue Q

```
Floyd-Warshall(W)

1. n ← rows[W]
2. D(0) ← W
3. for k ← 1 to n
4.       for i ← 1 to n
5.             for j ← 1 to n
6.                   dij(k) ← min(dij(k-1), dik(k-1) + dkj(k-1))
7. return D(n)
```

**Figure 1: Pseudo code for the Floyd-Warshall algorithm**

the nearest vertex *u* to the source, given the distances known thus far in the computation (Extract-Min operation). Once this nearest vertex is selected, all vertices *v* that neighbor *u* are updated with a new distance from the source (Update operation). The pseudo-code for the algorithm is given in Figure 2. The optimal implementation of Dijkstra's algorithm utilizes the Fibonacci heap and has complexity O($N$ lg($N$) + $E$), although the Fibonacci heap may only be interesting in theory due to large constant factors.

```
Dijkstra's(V)

1. S = Ø
2. Q = V[G]
3. while Q ≠ Ø
4.       u = Extract-Min(Q)
5.       S = S U {u}
6.       for each vertex v ∈ Adj[u]
7.             Update d[v]
8. return S
```

**Figure 2: Pseudo code for Dijkstra's algorithm**

Prim's algorithm for Minimum Spanning Tree is very similar to Dijkstra's algorithm for the single-source shortest path problem. In both cases a root node or source node is chosen and all other nodes reside in the priority queue. Nodes are extracted using an Extract-min operation and all neighbors of the extracted vertex are updated. The difference in Prim's algorithm is that nodes are updated with the weight of the edge from the extracted node instead of the weight from the source or root node.

```
Find_Match(G, M)

1. while (there exists an augmenting path)
2. {
3.       increase |M| by one using the augmenting
           path;
4. }

5. return M;
```

**Figure 3: Pseudo code for primitive graph matching algorithm**

For the sake of graph matching a subset *M* of *E* is considered a matching if no vertex is incident on more than one edge in *M*. A matching is considered maximal if it is not a subset of any other matching. A vertex is considered free if no edge in *M* is incident upon it. Using these definitions a primitive matching algorithm can be defined as follows [29]. Beginning at a free vertex use a breadth first search to find a path *P* from that free vertex to another free vertex alternating between edges in *M* and edges not in *M*. This is considered an augmenting path. Update the matching *M* by taking the symmetric difference of the edge sets of *M* and *P*. The algorithm is complete when no augmenting path can be found. The running time of this algorithm has been shown to be O(*N***E*). Pseudo-code is given in Figure 3. A more detailed explanation of this primitive matching algorithm is given in [29].

### 2.2. Overview of Cache Based Architectures and Optimizing Algorithms for Improved Cache Performance

It is a well-known fact that the speed of modern processors is increasing at a rate of roughly 60% per year while the speed of memory is increasing at a rate of roughly 7% per year. This difference is often referred to as the processor-memory gap, and it causes the latency to memory as seen by the processor to increase significantly with each passing year. In order to hide this increasing latency, caches have been designed to take advantage of locality of reference; the fact that once an element is accessed there is a good chance that it and/or elements near will be accessed in the near future. The cache is much smaller than main memory and is placed much closer to the processor in terms of latency. Modern processors are including more levels of cache, each level larger in size and farther from the processor in terms of latency.

Invariably the processor will access data that is not in the cache and this will result in a cache miss. Cache misses can be categorized into one of three categories: cold misses, capacity misses, and conflict misses. A cold miss occurs the first time a data element is accessed. These misses are unavoidable. A capacity miss occurs if the working set of the application is larger than the cache. These misses can be avoided by either decreasing the working set or increasing the size of the cache. A conflict miss occurs if two or more data elements in the working set map to same place in the cache and the replacement of one results in a subsequent cache miss when that element is accessed. This type of miss can be avoided in a number of ways including improved data access patterns, improved data layout, reducing the working set, etc [24].

Two other issues that should be addressed are cache pollution and TLB misses. TLB misses are similar to cache misses except that they refer to misses in the Translation Look-aside Buffer. They can be categorized the same as cache misses and reducing them follows a similar pattern. Cache pollution is a somewhat different issue. This refers to when a cache line is brought into the cache and only a small portion of it is used before it is pushed out of the cache. A large amount of cache pollution will increase the bandwidth requirement of the application, even though the application is not utilizing more data.

Based on this discussion, the keys to improve the performance of the memory system are as follows: increase data reuse, decrease cache conflicts, and decrease cache pollution. The techniques that we use to achieve these ends can be categorized as data layout optimizations and data access pattern optimizations. In our data layout optimizations we attempt to match the data layout to an existing data access pattern. For example, we use the Block Data Layout to match the access pattern of a tiled algorithm (see Section 3), or we an adjacency array to match the access pattern of Dijkstra's algorithm and Prim's algorithm (see Section 4 & 5). In our data access pattern optimizations, we design both novel and trivial optimizations to the algorithm to improve the data access pattern. For example, we implemented both a novel tiled implementation and a novel recursive implementation of the Floyd-Warshall algorithm to improve the data access pattern.

A different approach to improving the performance of the cache is to design cache-oblivious algorithms. This is explored in by Frigo, et. al. in [12]. In this article, the algorithms do not ignore the presence of a cache, but rather they use recursion to improve performance regardless of the size or organization of the cache. By doing this, they can improve the performance of the algorithm without tuning the application to the specifics of the host machine. In our work we develop a cache-oblivious implementation of the Floyd-Warshall algorithm. One difference between this work and ours is that they assume a fully associative cache when developing and analyzing the techniques. For this reason, they do not consider any data layout optimizations to avoid cache conflicts. They assume that at some point in the recursion the problem will fit into the cache and all work done following this point will be of optimal cost. In fact we show between 20% and 2x performance improvements by optimizing what is done once we reach a problem size that fits into the cache.

### 2.3. 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 data dependences that require us to update the entire $N$x$N$ array $D^k$ before moving on to the $(k+1)^{th}$ step (see Figure 4). This data dependence 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 SUIF research compiler and found that it cannot perform the optimizations discussed in Section 3 without user provided knowledge of the algorithm [9]. 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.

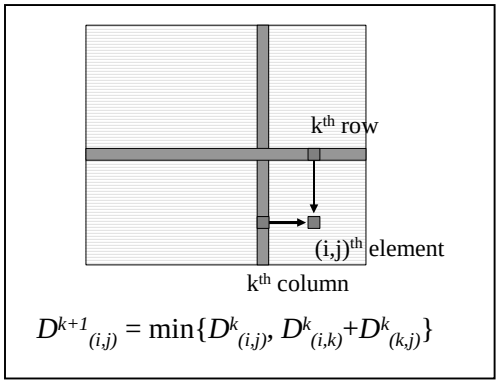



**Figure 4: $k^{th}$ iteration of outer loop in Floyd-Warshall Algorithm**

In Dijkstra's algorithm and Prim's algorithm, the largest data structure is the graph representation. An optimal representation, with respect to space, would be the adjacency-list representation. However, this involves pointer chasing when traversing the list. The priority queue has been highly optimized by various groups over the years. Unfortunately, the update operation is often excluded, as it is not necessary in such algorithms as sorting. The asymptotically optimal implementation that considers the update operation is the Fibonacci heap. Unfortunately this implementation includes large constant factors and did not perform well in our experiments.

The primitive graph matching algorithm poses challenges that resemble challenges in both the Floyd-Warshall algorithm and Dijkstra's algorithm. As in the Floyd-Warshall algorithm, each breadth first search to find an augmenting path could examine any part or the entire input graph. Recall that the Floyd-Warshall algorithm requires updating the entire graph at each step. Unlike the Floyd-Warshall algorithm, tiling and recursion cannot be applied, even with clever reordering, since the search cannot be limited to a small part of the graph and still find a maximal matching for the entire graph. We also have the situation as in Dijkstra's algorithm where the size of the graph representation can affect performance and, although optimal with respect to size, the adjacency list representation could cause a degradation of cache performance due to pointer chasing when traversing the list.

The model that we use in this paper is that of a uniprocessor, cache-based system. We refer to the cache closest to the processor as $L_1$ with size $C_1$, and subsequent levels as $L_i$ with size $C_i$. Throughout this paper we refer to the amount of *processor-memory traffic*. This is defined as the amount of traffic 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 or equal to 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.

We use four different architectures for our experiments. The Pentium III Xeon running Windows 2000 is a 700 MHz, 4 processor shared memory machine with 4 GB of main memory. Each processor has 32 KB of level-1 data cache and 1 MB of level-2 cache on-chip. The level-1 cache is 4-way set associative with 32 B lines and the level-2 cache is 8-way set associative with 32 B lines. The UltraSPARC III machine is a 750 MHz SUN Blade 1000 shared memory machine running Solaris 8. It has 2 processors and 1 GB of main memory. Each processor has 64 KB of level-1 data cache and 8 MB of level-2 cache. The level-1 cache is 4-way set associative with 32 B lines and the level-2 cache is direct mapped with 64 B lines. The MIPS machine is a 300 MHz R12000, 64 processor, shared memory machine with 16 GB of main memory. Each processor has 32 KB of level-1 data cache and 8 MB of level-2 cache. The level-1 cache is 2-way set associative with 32 B lines and the level-2 cache is direct mapped with 64 B lines. The Alpha 21264 is a 500 MHz uniprocessor machine with 512 MB of main memory. It has 64 KB of level-1 data cache and 4 MB of level-2 cache. The level-1 cache is 2-way set associative with 64 B lines and the level-2 cache is direct mapped with 64 B lines. It also

has an 8 element fully-associative victim cache. All experiments are run on a uniprocessor or on a single node of a multiprocessor system. Unless otherwise specified simulations are performed using the SimpleScalar simulator with a 16 KB, 4-way set associative level-1 data cache and a 256 KB, 8-way set associative level-2 cache.

### 2.4. Related Work

A number of groups have done research in the area of cache performance analysis and optimizations in recent years. Detailed cache models have been developed by Weikle, McKee, and Wulf in [35] and Sen and Chatterjee in[31]. XOR-based data layouts to eliminate cache misses have been explored by Valero and others in [13]. Data layouts for improving cache performance of embedded processor applications have been explored in [10].

A number of papers have discussed the optimization of specific dense linear algebra problems with respect to cache performance. Whaley and others discuss optimizing the widely used Basic Linear Algebra Subroutines (BLAS) in [36]. Chatterjee, et. al. discuss layout optimizations for a suite of dense matrix kernels in [5]. Frigo and others discuss the cache performance of cache oblivious algorithms for matrix transpose, FFT, and sorting in [12]. Park and Prasanna discuss dynamic data remapping to improve cache performance for the DFT in [23]. One characteristic that all 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 [27]). Optimizing blocked algorithms has been extensively studied (see for example [18]). The SUIF compiler framework includes a large set of libraries including libraries for performing data dependence analysis and loop transformations. In this context, it is important to note that SUIF does not handle the data dependences present in the Floyd-Warshall algorithm in a manner that improves the processor-memory traffic. It will not perform the transformations discussed in Section 3 without user intervention [9].

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. discusses making pointer-based data structures cache-conscious in [6]. He focuses on providing structure layouts to make tree structures cache-conscious. LaMarca and Ladner developed analytical models and showed simulation results predicting the number of cache misses for the heap in [19]. However, the predictions they made were for an isolated heap, and the model they used was the *hold model*, in which the heap is static for the majority of operations. In our work, we consider Dijkstra's algorithm and Prim's algorithm in which the heap is very dynamic. In both Dijkstra's algorithm and Prim's algorithm O($N$) Extract-Mins are performed and O($E$) Updates are performed. Finally in [30], Sanders discusses a highly optimized heap with respect to cache performance. He shows significant performance improvement using his *sequential heap*. The sequential heap does support Insert and Delete-min very efficiently, however the Update operation is not supported.

In the presence of the Update operation, the asymptotically optimal implementation of the priority queue, with respect to time complexity, is the Fibonacci heap. This implementation performs O($N$*lg($N$) + $E$) operations in both Dijkstra's algorithm and Prim's algorithm. In our experiments the large constant factors present in the Fibonacci heap caused it to perform very poorly. For this reason, we focus our work on the graph representation and the interaction between the graph representation and the priority queue.

In [34],Venkataraman, et. al. present a tiled implementation of the Floyd-Warshall algorithm that is essentially the same as the tiled implementation shown in this paper. In this paper, we consider a

wider range of architectures and also analyze the cache performance with respect to processor memory traffic. We also consider data layouts to avoid conflict misses in the cache, which is not discussed in [34]. Due to the fact that we use a blocked data layout we are able to relax the constraint that the blocking factor should be a multiple of the number of elements that fit into a cache line. This allows us to use a larger block size and show more speedup. In [34], they derive an upper bound on achievable speed-up of 2 for state-of-the-art architectures. Our optimizations lead to more than a 6x improvement in performance on three different state-of-the-art architectures.

We have recently published work on the Floyd-Warshall algorithm in [25] that showed a 2x improvement using the Unidirectional Space Time Representation. Compared with [25], this paper represents a new approach to optimizing the Floyd-Warshall algorithm, namely recursion and tiling, which gives up to an additional 3x improvement in execution time. Further, we expand our scope of algorithms to include Dijkstra's algorithm for the single source shortest path problem, Prim's algorithm for the minimum spanning tree problem, and graph matching.

## 3. Optimizing FW

In this section we address the challenges of the Floyd-Warshall algorithm. In Section 3.1 we introduce and prove the correctness of a recursive implementation for the Floyd-Warshall algorithm. We analyze the cache performance and show experimental results for this implementation compared with the baseline. We also show that by tuning the recursive algorithm to the cache size, we can improve its performance by up to 2x. In Section 3.2, we perform some analysis and discuss the impact of data layout on cache performance in the context of a tiled implementation of the Floyd-Warshall algorithm. Finally, in Section 3.3, we address the issue of data layout for both the tiled implementation and the recursive implementation.

Throughout this section we make the following assumptions. We assume a directed graph with $N$ vertices and $E$ edges. We assume the cache model described in Section 2.3, where $C_i < N^2$ for some $i$ and the TLB size is much less than $N$. To experimentally validate our approaches and their analysis, the actual problem sizes that we are working with are between 1024 and 4096 nodes ($1024 \leq N \leq 4096$). 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 8 KB.

In [15], it was shown that the lower bound on processor-memory traffic was $\Omega(N^3/\sqrt{C})$ for the usual implementation of matrix multiply. By examining the data dependence graphs for both matrix multiplication and the Floyd-Warshall algorithm, it can be shown that matrix multiplication reduces to the Floyd-Warshall algorithm with respect to processor-memory traffic. Therefore, we have the following:

**Lemma 3.1:** The lower bound on processor-memory traffic for the Floyd-Warshall algorithm, given a fixed cache size $C$, is $\Omega(N^3/\sqrt{C})$, where $N$ is the number of vertices in the input graph.

### 3.1. A Recursive Implementation of FW

As stated earlier, recursive implementations have recently been used to increase cache performance. It was stated in [11] that recursive implementations perform automatic blocking at every level of the memory hierarchy. To the authors' knowledge, there does not exist a recursive implementation of the Floyd-Warshall algorithm. The reason for this, is that the Floyd-Warshall algorithm not only contains all the dependences present in ordinary matrix multiplication, but also

additional dependences that can not be satisfied by the simple recursive implementation of matrix multiply. What is shown here is a novel recursive implementation of the Floyd-Warshall algorithm. We also prove the correctness of the implementation and show analytically that it reaches the asymptotically optimal amount of processor memory traffic.

In order to design a recursive implementation of the Floyd-Warshall algorithm, first examine the standard implementation when applied to a 2x2 matrix. The standard implementation loops over the variables *k*, *i*, and *j* from 1 to *N*. When the 2x2 case is unrolled we have the code shown in Figure 5. Notice that 8 calls are made to the min() operation and each call requires 3 data values from the matrix. This is converted into a recursive program by replacing the call to the min() function with a recursive call. Instead of passing 3 data values, we pass 3 sub-matrices corresponding to quadrants of the input matrix. This code is shown in Figure 6. The initial call to the recursive algorithm passes the entire input matrix as each argument. Subsequent calls pass quadrants of their input arguments as shown in Figure 6. The code similar to Figure 5 calling the min() operation is used as the base case for when the input matrices are of size 2x2.

```
Floyd-Warshall-2b2-Unrolled(W)

2.    D^(0) ← W

3.    d11^(1) ← min(d11^(0), d11^(0) + d11^(0))
4.    d12^(1) ← min(d12^(0), d11^(0) + d12^(0))
5.    d21^(1) ← min(d21^(0), d21^(0) + d11^(0))
6.    d22^(1) ← min(d22^(0), d21^(0) + d12^(0))
7.    d22^(2) ← min(d22^(1), d22^(1) + d22^(1))
8.    d21^(2) ← min(d21^(1), d22^(1) + d21^(1))
9.    d12^(2) ← min(d12^(1), d12^(1) + d22^(1))
10.   d11^(2) ← min(d11^(1), d12^(1) + d21^(1))

11.   return D^(2)
```

**Figure 5: Pseudo code for the 2x2 unrolled version of the Floyd-Warshall algorithm**

```
Floyd-Warshall-Recursive(A, B, C)

1.    if (not base case) {
2.       A11 ← FWR(A11, B11, C11);
3.       A12 ← FWR(A12, B11, C12);
4.       A21 ← FWR(A21, B21, C11);
5.       A22 ← FWR(A22, B21, C12);
6.       A22 ← FWR(A22, B22, C22);
7.       A21 ← FWR(A21, B22, C21);
8.       A12 ← FWR(A12, B12, C22);
9.       A11 ← FWR(A11, B12, C21);
10.   }
11.   else {
12.     /* run base case */                    1
13.   }

14.   return A
```

**Figure 6: Pseudo code for the recursive version of the Floyd-Warshall algorithm**

In order to complete the proof of the correctness of the recursive implementation of the Floyd-Warshall algorithm we need the following claim.

**Claim 1:** When computing the following equation it is sufficient for the correctness of the Floyd-Warshall algorithm that $k' \geq k-1$.

$$D^{k}_{(i,j)} = \min\left(D^{k-1}_{(i,j)}, D^{k'}_{(i,k)} + D^{k'}_{(k,j)}\right)$$

**Proof:**

By virtue of the min operation, the values used for $D^{k'}_{(i,k)}$ & $D^{k'}_{(k,j)}$ will be ≤ to $D^{k-1}_{(i,k)}$ & $D^{k-1}_{(k,j)}$. Therefore, $D^{k'}_{(i,k)} + D^{k'}_{(k,j)} \leq D^{k-1}_{(i,k)} + D^{k-1}_{(k,j)}$, and $D^{k}_{(i,j)}$ using $k' \geq k-1$ will be ≤ the value computed using k-1. Since no values are used that are not representative of paths, there exists a path from the $i^{th}$ vertex to the $j^{th}$ vertex of cost given by Equation 1. Also, since the goal of the Floyd-Warshall algorithm is to find the shortest path, Equation 1 will give the correct final result. ■

As a final note, this does not claim that Equation 1 computes the shortest path from the $i^{th}$ vertex to the $j^{th}$ vertex using vertices up to $k'$. It merely computes a path from the $i^{th}$ vertex to the $j^{th}$ vertex that

is less than or equal in cost to the shortest path from the $i^{th}$ vertex to the $j^{th}$ vertex using vertices up to $k$-1.

**Theorem 3.1:** The recursive implementation of the Floyd-Warshall algorithm detailed above satisfies the dependences given by Equation 1 and correctly computes the transitive closure of the input graph.

**Proof:**

By definition the straightforward implementation of the Floyd-Warshall algorithm computes the outer product of the input matrix with addition replaced by minimum and multiplication replaced by addition. Subsequently, this is referred to as the FW outer product. Also, for the sake of simplicity, assume that the problem size ($N$) is a power of 2.

*Base case:*

When the number of vertices is equal to 2, the recursive implementation is identical to the original implementation of the Floyd-Warshall algorithm given in Figure 5.

*Induction Step:*

Assume that the recursive implementation correctly computes the FW outer product for problem sizes up to $N$/2. Then, for a problem of size $N$, the 8 recursive calls shown in Figure 6 will be made.

The first call, step 1, passes the Northwest quadrant as each argument. By assumption, this will correctly compute the Northwest quadrant of $D^{N/2}$. In other words, the shortest path will be found from $i$ to $j$ with all intermediate vertices in the set 1 to $k$, where $i$, $j$, and $k$ are in the set 1 to $N$/2.

The second call, step 2, computes the Northeast quadrants of $D^{N/2}$. By Claim 1, we can use the data from the Northwest quadrant of $D^{N/2}$ instead of $D^{k-1}$. This step finds the shortest path from $i$ to $j$ with all intermediate vertices in the set 1 to $k$, where $i$ and $k$ are in the set 1 to $N$/2 and $j$ is in the set $N$/2 + 1 to $N$.

In the same fashion, the third and fourth calls complete the computation of $D^{N/2}$ and after the first four recursive calls we have correctly computed the shortest path from from $i$ to $j$ with all intermediate vertices in the set 1 to $k$, where $i$ and $j$ are in the set 1 to $N$ and $k$ is in the set 1 to $N$/2.

The second set of four recursive calls works in the same way that the first set did and complete the computation of $D^N$, the last three using result from other quadrants of $D^N$ instead of $D^{k-1}$ by Claim 1. In this way, we correctly compute the shortest path from $i$ to $j$, and by induction the recursive implementation of the Floyd-Warshall algorithm is correct for all $N$, where $N$ is a power of 2. ■

**Theorem 3.2:** The recursive implementation reduces the processor-memory traffic by a factor of $B$, where $B = \mathrm{O}(\sqrt{C})$.

**Proof:**

Note that the running time of this algorithm is given by

$$T(N) = 8 * T\left(\frac{N}{2}\right) = \Theta(N^3) \qquad \textbf{2}$$

Define the amount of processor memory traffic by the function D($x$). Without considering cache, the function behaves exactly as the running time.

$$D(N) = 8 * D\left(\frac{N}{2}\right) = \Theta(N^3) \qquad \textbf{3}$$

Consider the problem after $k$ recursive calls. At this point the problem size is $N/2^k$. There exists some $k$ such that $N/2^k = \mathrm{O}(\sqrt{C})$, where $C$ = cache size. For simplicity we set $B = N/2^k$. At this point, all data will fit in the cache and no further traffic will occur for recursive calls below this point. Therefore:

$$D(B) = O(B^2) \quad (4)$$

By combining Equation 3 and Equation 4 it can be shown that:

$$D(N) = \frac{N^3}{B^3} * D(B) = O(\frac{N^3}{B}) \quad (5)$$

Therefore, the processor-memory traffic is reduced by a factor of $B$. ■

**Theorem 3.3:** The recursive implementation reduces the traffic between the $i^{th}$ and the $(i\text{-}1)^{th}$ level of cache by a factor of $B_i$ at each level of the memory hierarchy, where $B_i = \Omega(\sqrt{C_i})$.

**Proof:**

Note first of all, that no tuning was assumed when calculating the amount of processor-memory traffic in the proof of Theorem 3.2. Namely, Equation 5 holds for any $N$ and any $B$ where $B = \Omega(\sqrt{C})$.

In order to prove Theorem 3.3, first consider the entire problem and the traffic between main memory and the $m^{th}$ level of cache (size $C_m$). By Theorem 3.2, the traffic will be reduced by $B_m$ where $B_m = \Omega(\sqrt{C_m})$. Next consider each problem of size $B_m$ and the traffic between the $m^{th}$ level of cache and the $(m\text{-}1)^{th}$ level of cache (size $C_{m-1}$). By replacing $N$ in Theorem 3.2 by $B_m$, it can be shown that this traffic is reduced by a factor of $B_{m-1}$ where $B_{m-1} = \Omega(\sqrt{C_{m-1}})$.

This simple extension of Theorem 3.2 can be done for each level of the memory hierarchy, and therefore the processor-memory traffic between the $i^{th}$ and the $(i\text{-}1)^{th}$ level of cache will be reduced by a factor of $B_i$, where $B_i = \Omega(\sqrt{C_i})$. ■

Finally, recall from Lemma 3.1 that the lower bound on processor-memory traffic for the Floyd-Warshall algorithm is given by $\Omega(N^3/\sqrt{C})$, where $C$ is the cache size. Also recall from Theorem 3.2 the upper bound on processor-memory traffic that was shown for the recursive implementation was $O(N^3/B)$, where $B^2 = O(C)$. Given this information we have the following Theorem.

**Theorem 3.4:** Our recursive implementation is asymptotically optimal among all implementations of the Floyd-Warshall algorithm with respect to processor-memory traffic.

As a final note in the recursive implementation, we show up to 2x improvement when we set the base case such that the base case would utilize more of the cache closest to the processor. Once we reached a problem size $B$, where $B^2$ is on the order of the cache size, we execute a standard iterative implementation of the Floyd-Warshall algorithm. This improvement varied from one machine to the other and is due to the decrease in the overhead of recursion. It can be shown that the number of recursive calls in the recursive algorithm is reduced by a factor of $B^3$ when we stop the recursion at a problem of size $B$. A comparison of full recursion and recursion stopped at a larger block size showed a 30% improvement on the Pentium III and a 2x improvement on the UltraSPARC III.

In order to improve performance, $B^2$ must be chosen to be on the order of the L1 cache size. The simplest and possibly the most accurate method of choosing $B$ is to experiment with various tile sizes. This is the method that the Automatically Tuned Linear Algebra Subroutines (ATLAS) project employs [36]. However, it is beneficial to find an estimate of the optimal tile size. A block size selection heuristic for finding this estimate is discussed in [25], and outlined here.

- Use the 2:1 rule of thumb from [24] 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 tiles of data. Self-interference misses are eliminated by the data being in contiguous locations within each tile and cross interference misses are eliminated by the associativity.
- Choose $B$ by Equation 6, where $d$ is the size of one element and $C$ is the adjusted cache size. This minimizes capacity misses.

$$3 * B^2 * d = C \qquad (6)$$

The baseline we use for our experiments is a straightforward implementation of the Floyd-Warshall algorithm. It was shown in [25] that standard optimizations yield limited performance increases on most machines. The Simulation results in Table 1 for the recursive implementation show a 30% decrease in level-1 cache misses and a 2x decrease in level-2 cache misses for problem sizes of 1024 and 2048. In order to verify the improvements on real machines, we compare the recursive implementation of the Floyd-Warshall algorithm with the baseline. For these experiments the best block size was found experimentally. The results show more than 10x improvement in overall execution time on the MIPS, roughly than 7x improvement on the Pentium III and the Alpha, and more than 2x improvement on the UltraSPARC III. These results are shown in Figure 7. Differences in performance gains between machines are expected, due to the wide variance in cache parameters and miss penalties.

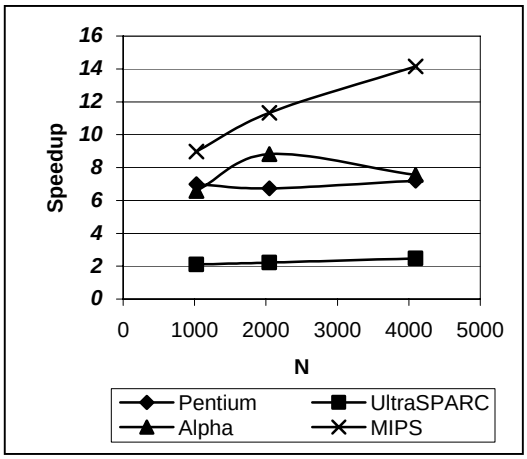



**Figure 7: Speedup results for the recursive implementation of the Floyd-Warshall algorithm**

**Data level-1 cache misses**

| N | Baseline | Recursive | |
|---|---|---|---|
| **1024** | 0.806 | 0.546 | |
| **2048** | 6.442 | 4.362 | $10^9$ |

**Data level-2 cache misses**

| N | Baseline | Recursive | |
|---|---|---|---|
| **1024** | 0.537 | 0.280 | |
| **2048** | 4.294 | 2.232 | $10^6$ |

**Table 1: Simulation result**

### 3.2. A Tiled Implementation for FW

Compiler groups have used tiling to achieve higher data reuse in looped code. Unfortunately, the data dependences from one *k*-loop to the next in the Floyd-Warshall algorithm make it impossible for current compilers, including research compilers, to perform 3 levels of tiling [9]. In order to tile the outermost loop we must cleverly reorder the tiles in such a way that satisfies data dependences from one *k*-loop to the next as well as within each *k*-loop.

Recall that Claim 1 stated that when computing Equation 1, it was sufficient that $k' \geq k-1$. Consider a special case of Claim 1 when we restrict $k'$ such that $k-1 \leq k' \leq k+B-1$, where $B$ is the blocking factor. This special case leads to the following tiled implementation of the Floyd-Warshall algorithm. This tiled implementation has also been derived in [34] using an alternate analysis. A brief description of the algorithm is as follows. Tile the problem into $B$x$B$ tiles. During the $k^{th}$ block iteration, first update the $(k,k)^{th}$ tile, then the remainder of the $k^{th}$ row and $k^{th}$ column, then the rest of the matrix. Figure 8 shows an example matrix tiled into a 4x4 matrix of blocks. Each block is of size $B$x$B$. During each outermost loop, we would update first the black tile representing the $(k,k)^{th}$ tile, then the grey tiles, then the white tiles. In this way we satisfy all dependences from each $k^{th}$ loop to the next as well as all dependences within each $k^{th}$ loop.

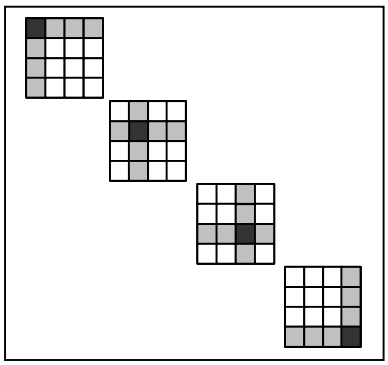


**Figure 8: Tiled implementation of FW**

**3.2.1. Analysis.** In [34], an upper bound for any cache optimized Floyd-Warshall algorithm was shown, however, no formal analysis with respect to traffic was shown for their tiled implementation. In fact

our results show speed-ups significantly larger than the upper bound shown in [34]. The following analysis is performed for the tiled implementation in the context of the model discussed in Section 2.

**Theorem 3.5:** The proposed tiled implementation of the Floyd-Warshall algorithm reduces the processor-memory traffic by a factor of $B$ where $B^2$ is on the order of the cache size.

**Proof sketch:** At each block we perform $B^3$ operations. There are $N/B$ x $N/B$ blocks in the array and we pass through each block $N/B$ times. This gives us a total of $N^3$ operations. In order to process each block we require only $3*B^2$ elements. This gives us a total of $N^3/B$ total processor-memory traffic. ■

Given this upper bound on traffic for the tiled implementation and the lower bound shown in Lemma 3.1, we have the following.

**Theorem 3.6:** The proposed tiled implementation is asymptotically optimal among all implementations of the Floyd-Warshall algorithm with respect to processor-memory traffic.

**3.2.2. Optimizing the Tiled Implementation.** It has been shown by a number of groups that data layouts tuned to the access pattern can significantly impact cache performance and improve overall execution time. In order to match the access pattern of the tiled implementation we use the Block Data Layout (BDL). The BDL is a two level mapping that maps a tile of data, instead of a row, into contiguous memory. By setting the block size equal to the tile size in the tiled computation, the data layout will exactly match the data access pattern. By using this data layout we can also relax the restriction on block size stated in [34] that the block size should be a multiple of the number of elements in a cache block.

As mentioned in Section 3.1, the best block size should be found experimentally, and the block size selection heuristic discussed in Section 3.1 can be used to give a rough bound on the best block size. However, when implementing the tiled implementation, 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 $B$ the search space should be expanded accordingly. In [34], only the level-1 cache is considered, however, with an on-chip level-2 cache often the best block size is larger than the level-1 cache. Table 2 shows the result of comparing the tiled implementation using a row-wise layout and the block size selection used in [34] with the tiled implementation using the block data layout and our block size selection. Simulation results show that the

**Data level-1 cache performance**

| | Row-wise | BDL | |
|---|---|---|---|
| **Misses** | 0.312 | 0.276 | $10^9$ |
| **Miss Rate** | 4.82% | 4.28% | |

**Data level-2 cache performance**

| | Row-wise | BDL | |
|---|---|---|---|
| **Misses** | 91.43 | 7.45 | $10^6$ |
| **Miss Rate** | 29.11% | 2.68% | |

**Execution time**

| | Row-wise | BDL | |
|---|---|---|---|
| **SUN** | 283.72 | 201.38 | |
| **P III** | 124.2 | 97.62 | (sec) |

N = 2048

**Table 2: Comparison result**

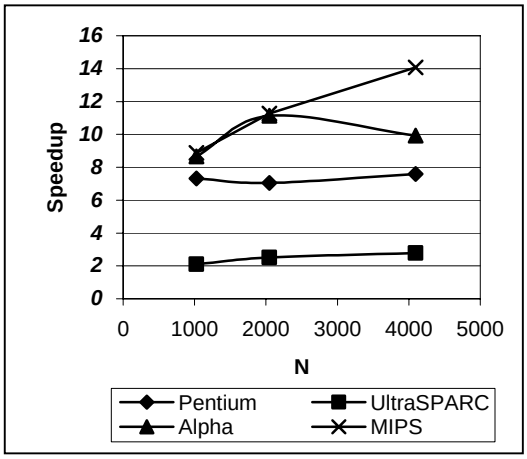



**Figure 9: Speedup results for the tiled implementation of the Floyd-Warshall algorithm**

**Data level-1 cache misses**

| N | Baseline | Tiled | |
|---|---|---|---|
| **1024** | 0.806 | 0.542 | |
| **2048** | 6.442 | 4.326 | $10^9$ |

**Data level-2 cache misses**

| N | Baseline | Tiled | |
|---|---|---|---|
| **1024** | 0.537 | 0.276 | |
| **2048** | 4.294 | 2.195 | $10^6$ |

**Table 3: Simulation result**

block size selection used in [34] optimizes the level-1 cache misses, but incurs a level-2 cache miss ratio of almost 30%. The Block Data Layout with a larger block size has roughly equal level-1 cache performance and far better level-2 cache performance. The execution times for these implementations show a 20% to 30% improvement by the Block Data Layout over the row-wise data layout.

A comparison for the tiled implementation using the Block Data Layout with the best compiler optimized implementation was also performed. Simulation results for this are shown in Table 3. These results show a 2x improvement in level-2 cache misses and a 30% improvement in level-1 cache misses. Experimental results show a 10x improvement in execution time for the Alpha, better than 7x improvement for the Pentium III and the MIPS and roughly a 3x improvement for the UltraSPARC III (See Figure 9).

### 3.3. Data Layout Issues

It is also important to consider data layout when implementing any algorithm. It has been shown by a number of groups that data layouts tuned to the data access pattern of the algorithm can reduce both TLB and cache misses (see for example [5], [23], [25]). In the case of the recursive algorithm, the access pattern is matched by a Z-Morton data layout. The Z-Morton ordering is a recursive layout defined as follows: Divide the original matrix into 4 quadrants and lay these tiles in memory in the order NW, NE, SW, SE. Recursively divide each quadrant until a limiting condition is reach. This smallest tile is typically laid out in either row or column major fashion (see Figure 10). See [5] for a more formal definition of the Morton ordering.

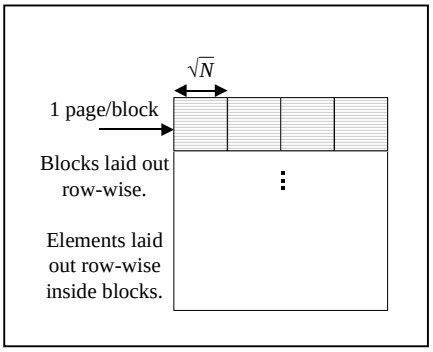



**Figure 10: The Block Data Layout**

In the case of the tiled implementation, the Block Data Layout (BDL) matches the access pattern. Recall from Section 3.2.2 that 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 11). By setting the block size equal to the tile size in the tiled computation, the data layout will exactly match the data access pattern.

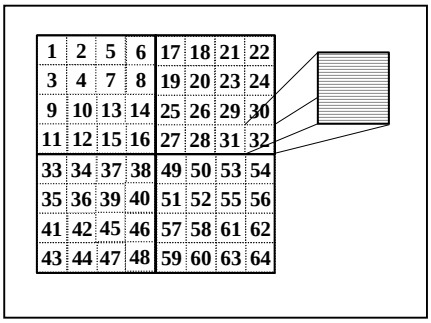



**Figure 11: The Morton Layout**

We experimented with both of these data layouts for each of the implementations. The results are shown in Tables 4 and 5. All of the execution times were within 15% of each other with the Z-Morton data layout winning slightly for the recursive implementation and the BDL winning slightly for the tiled implementation. The fact that the Z-Morton was slightly better for the recursive implementation and likewise the BDL for the tiled implementation was exactly as expected, since they match the data access pattern most closely. The closeness of the results is mostly likely due to the fact that the majority of the data reuse is within the final block. Since both of these

**Recursive Implementation**

| N | Morton Layout | Block Data Layout | |
|---|---|---|---|
| **2048** | 103.48 | 111.42 | |
| **4096** | 820.45 | 878.89 | (sec) |

**Tiled Implementation**

| N | Morton Layout | Block Data Layout | |
|---|---|---|---|
| **2048** | 99.25 | 99.39 | |
| **4096** | 779.53 | 780.41 | (sec) |

**Table 4: Pentium III results**

data layouts have the final block laid out in contiguous memory locations, they perform equally well.

It is also important to note that the Z-Morton data layout has a very complex index computation, which can only be hidden in a recursive algorithm. The BDL has a very simple index computation in comparison. Therefore it is significant to show that for non-recursive algorithms, the BDL performs just as well or better, while avoiding the overhead of a complex index computation.

**Recursive Implementation**

| N | Morton Layout | Block Data Layout | |
|---|---|---|---|
| **2048** | 307.33 | 311.26 | |
| **4096** | 2460.53 | 2488.88 | (sec) |

**Tiled Implementation**

| N | Morton Layout | Block Data Layout | |
|---|---|---|---|
| **2048** | 278.48 | 271.35 | |
| **4096** | 2248.20 | 2184.09 | (sec) |

**Table 5: UltraSPARC III results**

## 4. Optimizing the Single-Source Shortest Path Problem

Due to the structure of Dijkstra's algorithm neither tiling nor recursion can be directly applied. Much work has been done to generate cache friendly implementations of the heap, however, the update operation has not been considered in great detail (see section 2.3). In the presence of the update operation, the Fibonacci heap represents the asymptotically optimal implementation with respect to time complexity. Unfortunately, in the problem sizes being considered, the performance of the Fibonacci heap was very poor compared with even a straightforward implementation of the heap.

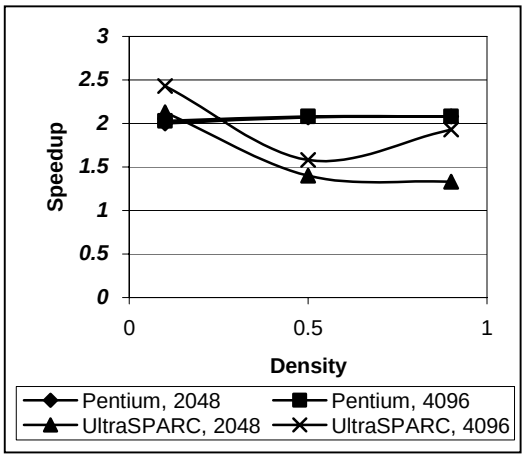



**Figure 12: Speedup results for Dijkstra's algorithm**

As mentioned in Section 2, the largest data structure is the graph representation. This structure will be of size O($N$+$E$), where $E$ can be as large as $N^2$ for dense graphs. In contrast, the priority queue, the other data structure involved, will be of size O($N$). Also note that each element in the graph representation will be accessed exactly once. For each node extracted from the priority queue, the corresponding adjacent nodes are read and updated. All nodes will be extracted from the priority queue and no node can be extracted more than once. Therefore, the traffic as a result of the graph representation will be proportional to its size and the amount of prefetching possible. For these reasons, we focus on providing an optimization to the graph representation based on the data access pattern.

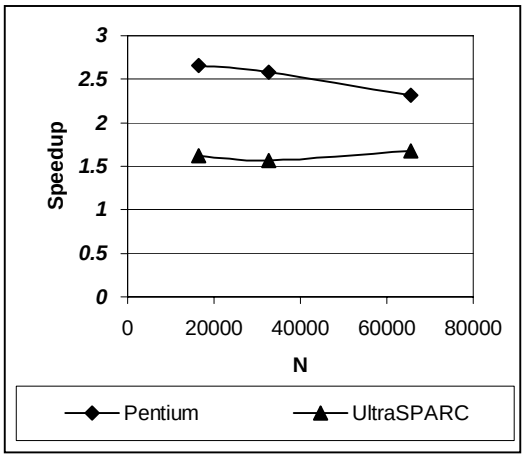



**Figure 13: Speedup results for Dijkstra's algorithm**

In the context of the graph representation, we can take advantage of two things. The first is prefetching. Modern processors perform aggressive prefetching in order to hide memory latencies. The second is to optimize at the cache line level. In this case, a single miss would bring in multiple elements that would subsequently be accessed and result in cache hits. In this way cache pollution is minimized.

There are two commonly used graph representations. The adjacency matrix is an $N$x$N$ matrix, where the $(i,j)^{th}$

element of the matrix is the cost from the $i^{th}$ node to the $j^{th}$ node of the graph. This representation is of size $O(N^2)$. It has the nice property that elements are accessed in a contiguous fashion and therefore, cache pollution will be minimized and prefetching will be maximized. However, for sparse graphs, the size of this representation is inefficient. The adjacency list representation is a pointer-based representation where a list of adjacent nodes is stored for each node in the graph. Each node in the list includes the cost of the edge from the given node to the adjacent node. This representation has the property of being of optimal size for all graphs, namely $O(N+E)$. However, the fact that it is pointer based, leads to cache pollution and difficulties in prefetching. See [7] or [14] for more details regarding these common graph representations.

| Cache misses | | |
|---|---|---|
| | **Linked-List** | **Adj. Array** |
| **Data level 1** | 7.04 | 5.62 |
| **Data level 2** | 3.59 | 1.82 |
| (Input: 16K nodes, 0.1 density) | | ($10^6$) |

**Table 6: Simulation results**

Consider a simple combination of these two representations [28]. For each node in the graph, we have an array of adjacent nodes. The size of each array is exactly the out-degree of the corresponding node. There are simple methods to construct this representation when the out-degree is not known until run time. For this representation, the elements at each point in the array look similar to the elements stored in the adjacency list. Each element must store both the cost of the path and the index of the adjacent node. Since the size of each array is exactly the out-degree of the corresponding node, the size of this representation is $O(N+E)$. This makes it optimal with respect to size. Also, since the elements are stored in arrays and therefore in contiguous memory locations, the cache pollution will be minimized and prefetching will be maximized. Subsequently this representation will be referred to as the *adjacency array representation*. This graph representation is essentially the same as a graph representation discussed in [28].

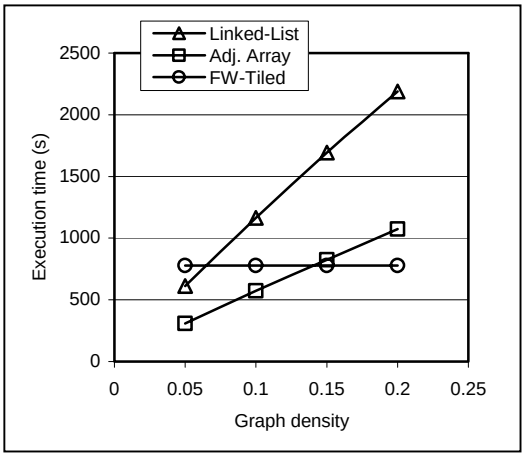



**Figure 14: Dijkstra's algorithm vs. best FW on Pentium III, *N* = 2048**

In order to demonstrate the performance improvements using our graph representation, we performed simulations as well as experiments on two different machines, the Pentium III and UltraSPARC III, for Dijkstra's algorithm. The simulations show approximately 20% reduction in level-1 cache misses and a 2x reduction in the number of level-2 cache misses (see Table 6). This is due to the reduction in cache pollution and increase in prefetching that was predicted. Due to memory limitations, experiments for all graph densities were only performed at small problem sizes, namely 2K nodes and 4K nodes. These results demonstrate improved performance using the adjacency array for all graph densities and are shown in Figure 12. Experiments on larger problem sizes (16K nodes up to 64K nodes) at a graph density of 10% are shown in Figure 13 and again are limited by the size of main memory. All of the results show a 2x improvement for Dijkstra's algorithm on the Pentium III and a 20% improvement on the UltraSPARC III. This significant difference in performance is due primarily to the difference in

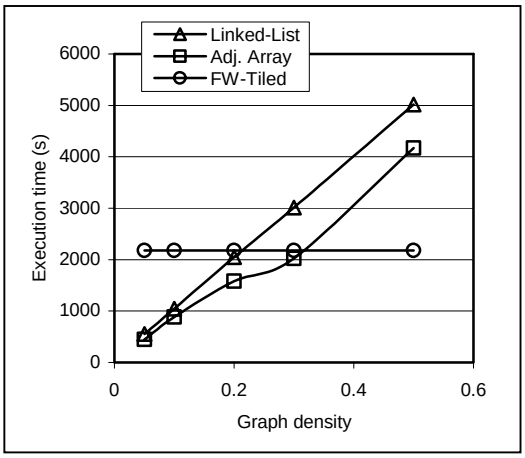



**Figure 15: Dijkstra's algorithm vs. best FW on UltraSPARC III, *N* = 4096**

the memory hierarchy of these two architectures.

A second comparison to observe is between the Floyd-Warshall algorithm and Dijkstra's algorithm for sparse graphs, i.e. edge densities less than 20%. For these graphs, Dijkstra's algorithm is more efficient for the all pairs shortest path problem. By using the adjacency array representation of the graph in Dijkstra's algorithm, the range of graphs over which Dijkstra's algorithm outperforms the Floyd-Warshall algorithm can be increased. Figures 14 & 15 show a comparison of the best Floyd-Warshall algorithm with Dijkstra's algorithm for sparse graphs. On the Pentium III, we were able to increase the range for Dijkstra's algorithm from densities up to 5% to densities up to 20%. On the UltraSPARC III we increased the range from densities up to 20% to densities up to 30%.

## 5. Optimizing the Minimum Spanning Tree Problem

As mentioned in Section 2, Prim's algorithm for minimum spanning tree is very similar to Dijkstra's algorithm for the single source shortest path problem. In fact they are identical with respect to the access pattern, the difference being only in how the update operation is performed. In Dijkstra's algorithm nodes in the priority queue are updated with their distance from the source node. In Prim's algorithm nodes are updated with the shortest distance from any node already removed from the priority queue. For this reason the optimizations applicable to Dijkstra's algorithm are also applicable to Prim's algorithm. Figures 16 & 17 show the result of applying the optimization to the graph representation discussed in Section 4 to Prim's algorithm. Recall that this optimization replaces the adjacency list graph representation with the adjacency array graph representation. This representation matches the streaming access that is made to the graph and in this way minimizes cache pollution and maximizes the prefetching ability of the processor.

Our results show a 2x improvement on the Pentium III and 20% for the UltraSPARC III. This performance improvement was shown in the smaller problem sizes of 2K and 4K nodes where experiments were done for densities ranging from 10% to 90% as well as the large problem sizes of 16K nodes up to 64K nodes with densities of 10%. Simulations were also performed to verify improved cache performance. These results are shown in Table 7. They show approximately a 20% reduction in the number of level-1 cache misses and a 2x reduction in the number of level-2 cache misses. As expected, all of the results are very close to the results shown for Dijkstra's algorithm.

**Cache misses**

| | Linked-List | Adj. Array |
|---|---|---|
| **Data level 1** | 7.19 | 5.77 |
| **Data level 2** | 3.59 | 1.82 |

(Input: 16K nodes, 0.1 density) ($10^6$)

**Table 7: Simulation results**

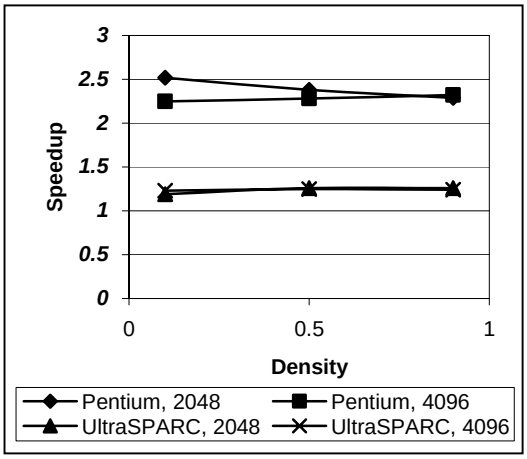



**Figure 16: Speedup results for Prim's algorithm**

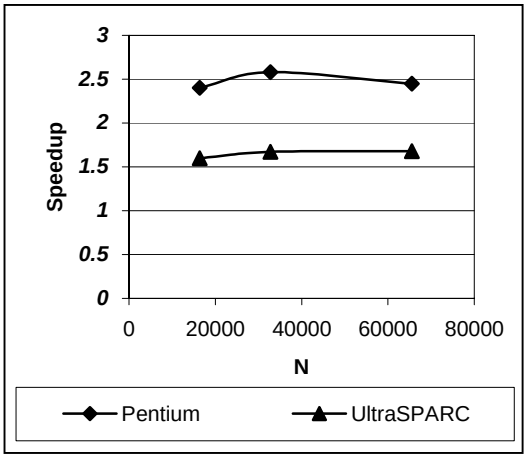



**Figure 17: Speedup results for Prim's algorithm**

## 6. Optimizing Bipartite Graph Matching

In this section, we utilize the ideas and techniques developed in the previous sections to optimize another basic graph algorithm, namely graph matching for bipartite graphs. As discussed in Section 2, this algorithm shows similarities to Dijkstra's algorithm with respect to memory access in each iteration and therefore tiling and recursion cannot be easily applied.

The first optimization that is applied is to use the adjacency arrays instead of the adjacency list. In order to find an augmenting path, a breadth first search is performed. The access pattern will then be to access all adjacent nodes to the current node. This is the same access pattern as was displayed in both Dijkstra's and Prim's algorithm.

The second optimization that is applied is intended to reduce the working set size as in tiling or recursion. As mentioned above, neither tiling nor recursion can be directly applied. What can be done is to use tiling to generate a good match as a starting point for the full problem. In this way the amount of work done when examining the complete graph will be reduced. Furthermore, the work done in the tiled steps will be cache friendly if the tiles are chosen appropriately. In order to accomplish this, first divide the graph into sub-graphs, each of which fits into the cache and find the local maximal matches. Then the local matches are combined to form a starting point for the original algorithm. Finally, the algorithm is run on the complete graph, using the match already found as a starting point, to find the maximal match.

The performance of this optimization is largely dependant on the structure and density of the graph and the partitioning chosen. Assuming a good partition, the local maximal matches will be close to a global maximal match for dense graphs due to the large number of edges present in each sub-graph. For sparse graphs, it is difficult to find a good local match and more work will be required at the global level.

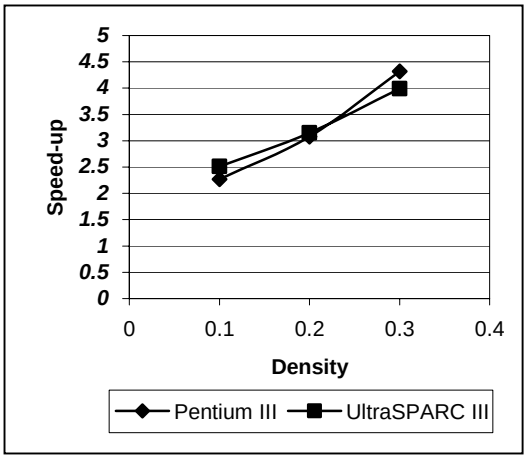



**Figure 18: Speed-up vs. density results for graph matching**

In order to improve the quality of the match at the local level, a very simple partitioning algorithm is employed. A basic description of this algorithm is as follows. Given a bipartite graph, the goal is to partition the edges into two groups such that the best match possible is found within each group. In order to accomplish this, as many edges as possible should have both end points in the same partition. These edges are referred to as internal edges. Arbitrarily partition the vertices into 4 equal partitions. Count the number of edges between each pair of partitions. Combine partitions into two partitions such that as many internal edges are created as possible.

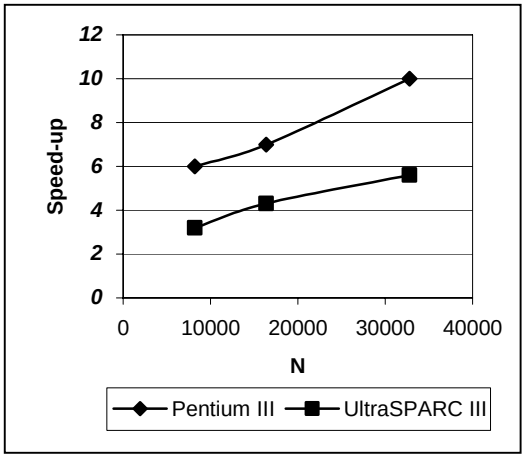



**Figure 19: Best case speed-up results for graph matching**

In order to support the quality of the optimization, experiments were also performed for a graph in which a worst possible graph partitioning was chosen, i.e. no matches were found at the local level. For this case, the optimized implementation showed only 10% performance degradation. The majority of experimentation was performed using randomly generated graphs in order to average out the dependence on graph partitioning. The random graphs were constructed by randomly choosing half

of the vertices to be in one partition of the bipartite graph. Edges were then created from each vertex in the partition to randomly chosen vertices not in the partition.

As expected, the performance improvement is highly dependent on the density of the graph. This dependence can be seen in Figure 18, which shows the speedup vs. graph density. Results ranged from just over 2x for graphs of 10% density to over 4x for graphs of 30% density. In this case, the problem size was fixed at 8192 nodes and density was limited to 30% by main memory. The best-case results are shown in Figure 19. For these problems, we designed the input graph such that the maximal matching is found in the tiled phase and very little work is performed on the complete graph. For these problems, results ranged from 3x up to 10x. The most interesting results are those shown in Figure 20. The input graph in this case was a randomly generated graph and the basic graph partitioning algorithm was used to improve the match found at the local level. The results shown are the average over 10 different random input graphs. The speedup shown is roughly 2x for all problem sizes. We also performed simulations to demonstrate cache performance for this case and the results are shown in Table 8. Based on the number of access to the level 1 cache, the optimized implementation is performing somewhat less work. This contributes somewhat to the decrease in the number of misses shown. However, the miss rate is also reduced by almost 3x, which indicates that the optimized implementation does improve cache performance beyond the amount reduced by the decrease in work.

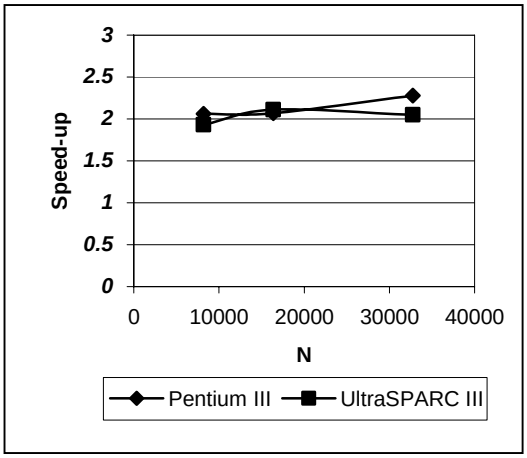



**Figure 20: Average speed-up results for graph matching**

**DL1 Cache Performance**

| | Baseline | Optimized |
|---|---|---|
| Accesses | 853 | 578 |
| Misses | 127 | 32 |
| Miss Rate | 14.86% | 5.56% |
| (Input: 8K nodes, 0.1 density) | | ($10^6$) |

**Table 8: Simulation results**

## 7. Conclusion

In the course of the research discussed in this paper, we have used the techniques of tiling, recursion, and data layout optimization to show improved cache performance both analytically and experimentally in the area of graph algorithms. The recursive implementation of the Floyd-Warshall algorithm represents a novel cache-oblivious implementation. Using this implementation as well as a tiled implementation, we have shown more than a 6x improvement in execution time on three different architectures as well as analytically showing that both implementations are optimal with respect to processor-memory traffic. We also showed significant performance improvements for Dijkstra's algorithm and Prim's algorithm using a cache friendly graph representation. Finally, we applied both the cache friendly graph representation and a tiling optimization to the problem of graph matching. These optimizations showed a 2x to 3x improvement in execution time for randomly generated graphs and up to 10x improvement for graphs well suited to our partitioning algorithm.

Tiling and recursion are also used as computation decomposition techniques for parallelization. Good parallelized code should have minimal communication and sharing between computational nodes, thus our pursuit of data locality also benefits parallelization. Our sequential FW implementations and matching implementation can easily be transformed into parallel code. Computation and data are already decomposed, what need to be added are computation and data

distribution, synchronization and communication primitives. One of our future directions will be to implement parallel versions of the Floyd-Warshall algorithm and matching algorithm based on the work presented in this paper.

Another area for future work is the optimization of the priority queue in Dijkstra's algorithm and Prim's algorithm. As mentioned, the Fibonacci heap is the asymptotically optimal implementation for priority queue in the presence of the update operation, however, due to large constant factors, it performed poorly in experiments.

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.

## 8. 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] J.B. Carter, W.C. Hsieh, L.B. Stoller, M.R. Swanson, L. Zhang, and S.A. McKee. Impulse: Memory System Support for Scientific Applications. In the *Journal of Scientific Programming*, Vol. 7, No. 3-4, pp. 195-209, 1999.

[5] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. *ACM Symposium on Parallel Algorithms and Architectures*, 1999.

[6] 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.

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

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

[9] P. Diniz. University of Southern California Information Sciences Institute, Personal Communication, March, 2001.

[10] N. Dutt, P. Panda, and A. Nicolau. Data Organization for Improved Performance in Embedded Processor Applications. *ACM Transactions on Design Automation of Electronic Systems*, Vol. 2, Number 4, October 1997.

[11] J. D. Frens and D. S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proc. of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.

[12] 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.

[13] A. Gonzalez, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating Cache Conflict Misses through XOR-Based Placement Functions. In *Proc. of 1997 International Conference on Supercomputing*, Vienne, Austria, July, 1997.

[14] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Maryland, 1978.

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

[16] M. Kandemir, J. Ramanujam, and A. Choudhary. Improving Cache Locality by a Combination of Loop and Data Transformations. *IEEE Transactions on Computers*, Vol. 48, No. 2, February, 1999.

[17] M. Kallahalla and P. J. Varman. Optimal Prefetching and Caching for Parallel I/O Systems. In *Proc. of 13th ACM Symposium on Parallel Algorithms and Architectures*, 2001.

[18] 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.

[19] A. LaMarca and R. E. Ladner. The Influence of Caches on the Performance of Heaps. *ACM Journal of Experimental Algorithmics*, 1, 1996.

[20] R. Murphy and P. M. Kogge. The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems, *Intelligent Memory Systems Workshop, ASPLOS-IX 2000*, Boston, MA, Nov. 12, 2000.

[21] J. Mussmano. *Data-Intensive Systems Stressmark Suite*, Version 1.0, Atlantic Aerospace Electronics Corporation, August 24, 2000.

[22] J. Mussmano. DIS Benchmarking. DARPA Data Intensive Systems, Principal Investigator Meeting Presentation, Santa Fe, NM, March 26, 2002.

[23] 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.

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

[25] M. Penner and V. K. Prasanna. Cache-Friendly Implementations of Transitive Closure. In *Proc. of International Conference on Parallel Architectures and Compiler Techniques*, Barcelona, Spain, September 2001.

[26] G. Rivera and C. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[27] 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.

[28] S. Sahni. *Data Structures, Algorithms, and Applications in Java*. McGraw Hill, New York, 2000.

[29] H. A. B. Saip and C. L. Luchesi. Matching Algorithms for Bipartite Graphs. University of Campinas Technical Report DCC-03/93, Brazil, March, 1993.

[30] P. Sanders. Fast Priority Queues for Cached Memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

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

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

[33] P. Varman, Parallel I/O Systems. *Handbook of Computer Engineering*, CRC Press, 2001.

[34] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya. A Blocked All-Pairs Shortest-Paths Algorithm. *Scandinavian Workshop on Algorithms and Theory*, Lecture Notes in Computer Science, Vol. 1851, Editor: Magnus Halldorsson, Springer Verlag, 2000, 419-432.

[35] 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.

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

[37] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. The Impulse Memory Controller. *IEEE Transactions on Computers*, Special Issue on Advances in High Performance Systems, November 2001.

# 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 $N$, $N$x$N$ 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 $N$x$N$ 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.

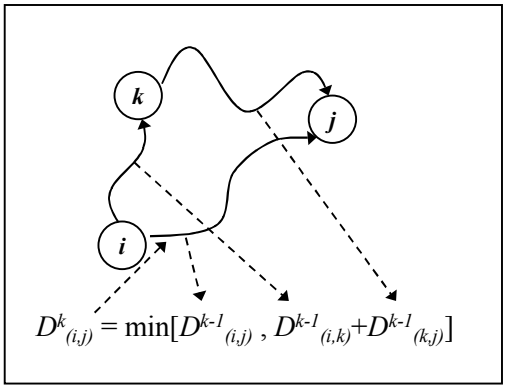



**Figure 1: $k^{th}$ step of Floyd-Warshall Algorithm**

### 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

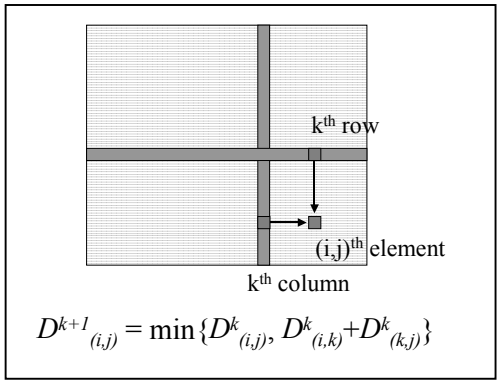



**Figure 2: Basic step ($k^{th}$ loop) in Floyd-Warshall algorithm**

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 * $d$.

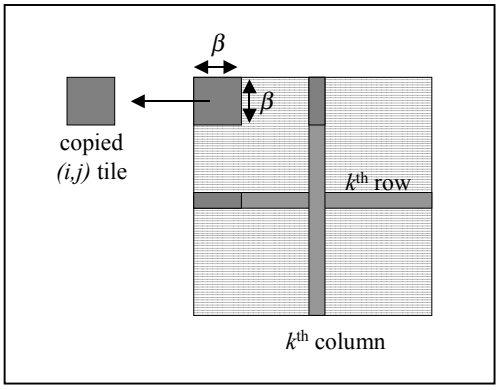



**Figure 3: Tiling plus copying for Floyd-Warshall algorithm**

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 $\beta$x$\beta$, where $\beta^2$ < 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, $\beta$=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)

**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

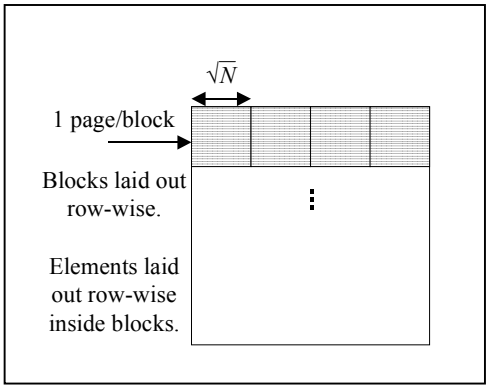



**Figure 4: The Block Data Layout**

**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.

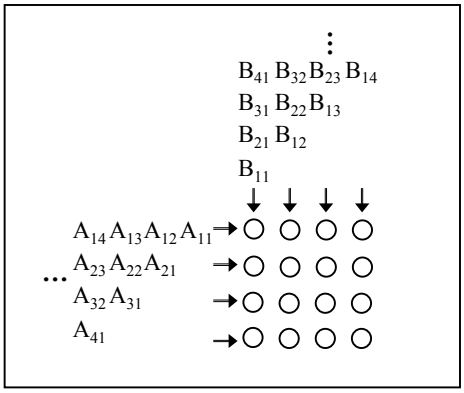



**Figure 5: USTR for 4x4 matrix multiply**

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

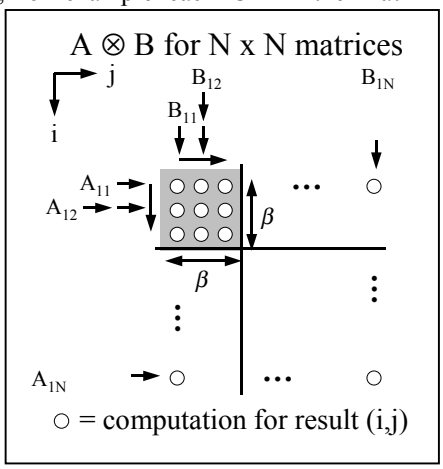



**Figure 6: Unidirectional Space Time Representation. Note: ⊗ refers to a generic matrix operation.**

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 $\beta$x$\beta$ (see Figure 6). Within each tile, operate on CEs in a row-wise fashion. Within each CE, process $\beta$ 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 $\beta$ 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 $\beta$x$\beta$. Within each tile, instead of processing $\beta$ elements at each CE at a time, process the entire array for t=1, then process it for t=2, and so for t≤$\beta$. 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 $\beta$ operations, requiring the first $\beta$ 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 $\beta$ data elements from the next row. When this is extended to the entire tile, it requires $2*\beta^2$ data elements of the input, $\beta^2$ locally stored values, and performs $\beta^3$ operations. In order to complete each tile, it must be passed through $N/\beta$ times. This requires $2*(N/\beta)*\beta^2$ data elements of the input, $\beta^2$ 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 $\beta$, 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 $\beta^2$ locally stored elements and performs $\beta^3$ operations. If we choose $\beta^2$ 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 $(N/\beta)$x$(N/\beta)$ tiles. Each tile requires $N/\beta$ 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 $\beta$.

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 $\beta$ 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 $\beta'$ such that $\beta'^2$ is equal to the size of the level-2 cache. We could use a simple row-wise layout of tiles within this larger $\beta'x\beta'$ 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 $\beta$x$\beta$ tiles and $\beta'$x $\beta'$ 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.

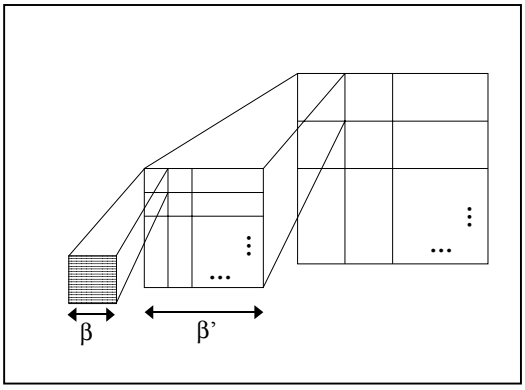



**Figure 7: Multi-level tiling for USTR schedule and/or layout.**

One of the key factors in Theorem 1 holding is that $\beta^2$ is chosen to be on the order of cache size. The simplest and possibly the most accurate method of choosing $\beta$ 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 $\beta$x$\beta$ 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 $\beta$ 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 $\beta$, 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 $\beta$ 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 $N$x$N$ 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).

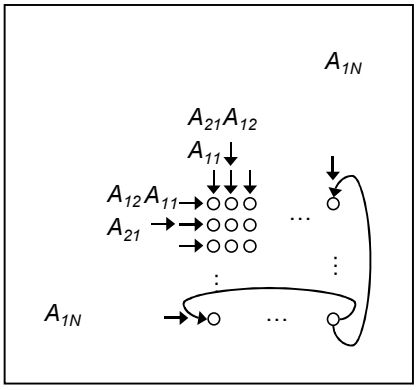



**Figure 8: Systolic Array implementation of Floyd-Warshall algorithm**

**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 $N$x$N$ 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.

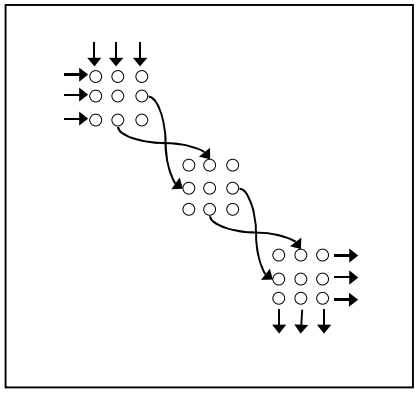


**Figure 9: Unidirectional Space Time Representation of Systolic Array algorithm for transitive closure.**

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 $\beta$, where cache size is on the order of $\beta^2$ 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 | $N^3$ | $N^3$ | $N^3$ | $N^3/\beta$ |
| 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

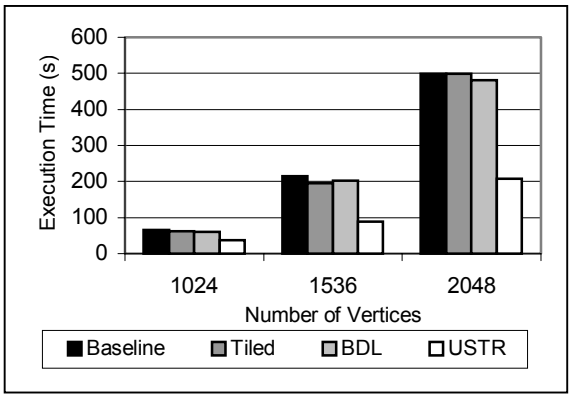



**Figure 10: Execution times for implementations on Pentium III running Windows 2000.**

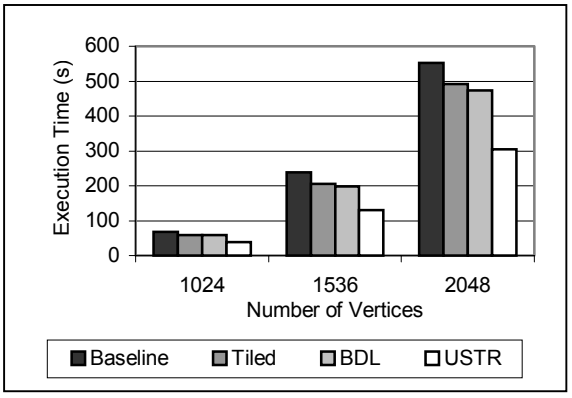



**Figure 11: Execution times for implementations on Pentium III running Linux.**

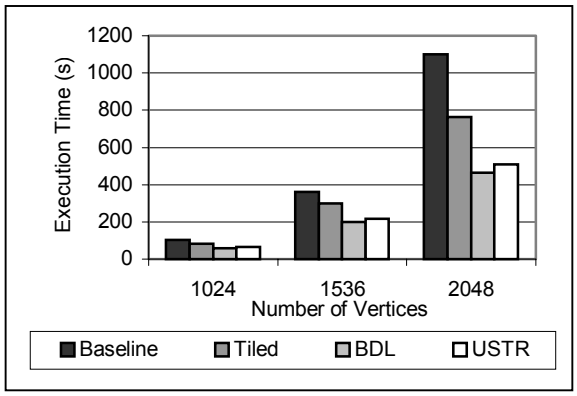



**Figure 12: Execution times for implementations on Alpha running Linux.**

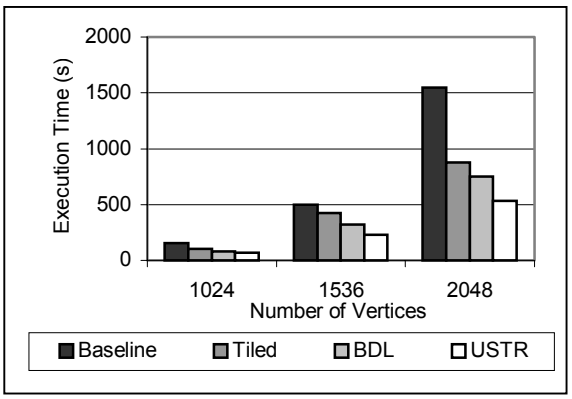



**Figure 13: Execution times for implementations on MIPS R12000 running IRIX64.**

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

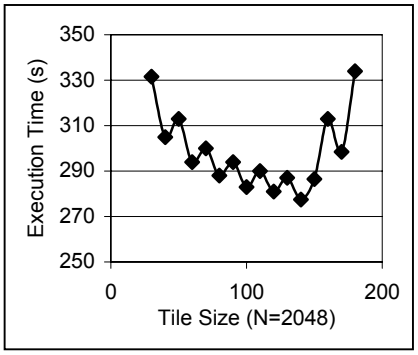



**Figure 14: USTR Optimization, tile size selection Pentium III, Linux**

## 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.

# Analysis of Memory Hierarchy Performance of Block Data Layout*

Neungsoo Park, Bo Hong, and Viktor K. Prasanna
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562
{neungsoo,bohong,prasanna}@halcyon.usc.edu
http://advisor.usc.edu

## Abstract

*Recently, several experimental studies have been conducted on block data layout as a data transformation technique used in conjunction with tiling to improve cache performance. In this paper, we provide a theoretical analysis for the TLB and cache performance of block data layout. For standard matrix access patterns, we derive an asymptotic lower bound on the number of TLB misses for any data layout and show that block data layout achieves this bound. We show that block data layout improves TLB misses by a factor of $O(B)$ compared with conventional data layouts, where $B$ is the block size of block data layout. This reduction contributes to the improvement in memory hierarchy performance. Using our TLB and cache analysis, we also discuss the impact of block size on the overall memory hierarchy performance. These results are validated through simulations and experiments on state-of-the-art platforms.*

## 1. Introduction

The increasing gap between memory latency and processor speed is a critical bottleneck in achieving high performance. The gap is typically bridged through a multi-level memory hierarchy that can hide memory latency. The performance of this memory hierarchy system is severely impacted by the locality of data references. To improve memory hierarchy performance, compiler optimization techniques (e.g. loop permutation, fusion, and tiling) [13, 14, 21] have received considerable attention, which improve the locality of the data reference. These techniques, called *control transformations*, change the loop iteration order, thereby changing the data access pattern [4, 8, 19, 25]. Most previous optimizations concentrate on single-level cache [8, 11, 15, 19, 23]. Multi-level caches in memory hierarchy were considered by a few researchers [20, 25]. However, most of these approaches target mainly the cache performance, paying less attention to the Translation Look-aside Buffer (TLB) performance. As the problem sizes become larger, the overall performance can drastically degrade because of TLB thrashing [22]. Hence, both TLB and cache must be considered in optimizing application performance. In [12], cache and TLB performance were considered *in concert.* In this analysis, TLB and cache were assumed to be fully-set associative. However, cache is direct mapped or small set-associative in most of state-of-the-art platforms.

Some recent work [11, 17, 18, 23] proposed *data transformations* that change the data layout in memory to match the data access pattern. It was proposed in [10] that both data and loop transformation can be applied to loop nests for optimizing cache locality. In [5, 6], a matrix is partitioned into small blocks of data. Data elements within one block are mapped onto contiguous memory. These blocks were laid out in memory by different space-filling curves. These data layouts have shown performance improvement over canonical row or column major layouts. Block data layout is one such layout where blocks are arranged in row-major order. ATLAS [2, 24] uses block data layout with tiling to exploit temporal and spatial locality. The combination of block data layout and tiling has shown high performance on various platforms. However, these results were confirmed through experiments; we are not aware of any formal analysis that addresses TLB performance..

In this paper, we study the impact of *block data layout*[1], with and without tiling, on the performance of both TLB and caches. First, we analyze the intrinsic TLB performance of block data layout. The TLB and cache performance for block data layout with tiling are analyzed. The block data

*Supported by the DARPA Data Intensive Systems Program under contract F33615-99-1-1483 monitored by Wright Patterson Air force Base, in part by NSF CCR-9900613, and in part by an equipment grant from Intel Corporation.

[1] To avoid confusion, in this paper, 'block' is used in the context of a data transformation technique, e.g. block data layout. 'tiling' is used to represent a control transform technique.

layout with tiling shows better TLB performance compared with other state-of-the-art techniques like copying [11, 23] and padding [15, 19]. Simulations and experiments are conducted to verify this analysis.

Similar to the importance of tile size selection for tiling, appropriate block size selection for block data layout is critical to achieve high performance. In ATLAS, the selection of the optimal block size is done *empirically* at compile time by running several experiments with different block sizes [24]. The selection criteria does not have any supporting formal analysis. In [5, 6], it is observed that the block size should not be too small nor too large. However, no analytical bounds for block size were presented. In this paper, we propose an analytical bound for optimal block size in block data layout, on the basis of our TLB and cache analysis.

The contributions of this paper are as follows:

- We present a lower bound analysis for TLB performance. Further, we show that block data layout intrinsically has better TLB performance than canonical layouts (Section 2). Compared with row major layout, the number of TLB misses for block data layout is improved by $O(\sqrt{P_v})$ where $P_v$ is the page size.
- We analyze the TLB and cache performance of tiling with block data (Section 3.1 and 3.2). In tiled matrix multiplication, block data layout improves the number of TLB misses by a factor of $B$, where $B$ is the block size.
- On the basis of our cache and TLB analysis, we propose a block size selection algorithm that provides a tight analytical bound for block size (Section 3.3). The best block sizes found by ATLAS fall in the range given by our algorithm.
- We validate our analysis through simulations and experiments on real platforms using matrix multiply, LU decomposition and Cholesky factorization (Section 4).

The rest of this paper is organized as follows. Section 2 describes block data layout and gives analysis of its TLB performance. Section 3 discusses the TLB and cache performance when tiling and block data layout are used in concert. A block size selection algorithm is described based on this analysis. Section 4 shows simulation based as well as experimental results. Concluding remarks are presented in Section 5.

## 2. Block Data Layout and TLB Performance

In Section 2, we analyze the TLB performance of block data layout. We show that block data layout has better intrinsic TLB performance than conventional data layouts. Without loss of generality, the canonical layout is assumed to be row major.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
| 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |

(a) Row-major layout

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 4 | 5 | 8 | 9 | 12 | 13 |
| 2 | 3 | 6 | 7 | 10 | 11 | 14 | 15 |
| 16 | 17 | 20 | 21 | 24 | 25 | 28 | 29 |
| 18 | 19 | 22 | 23 | 26 | 27 | 30 | 31 |
| 32 | 33 | 36 | 37 | 40 | 41 | 44 | 45 |
| 34 | 35 | 38 | 39 | 42 | 43 | 46 | 47 |
| 48 | 49 | 52 | 53 | 56 | 57 | 60 | 61 |
| 50 | 51 | 54 | 55 | 58 | 59 | 62 | 63 |

(b) Block data layout

**Figure 1. Various data layouts: block size $2 \times 2$ for (b)**

The following notations are used in this paper. $P_v$ denotes virtual page size. $S_{tlb}$ denotes the TLB entry capacity. In general, $S_{tlb} \ll P_v$. Block size is $B \times B$, where it is assumed $B^2 = kP_v$. Cache is assumed to be direct-mapped. $S_{ci}$ is the size of the $i^{th}$ level cache. Its line size is denoted as $L_{ci}$. We assume that TLB is fully set-associative and Least-Recently-Used(LRU) replacement policy is used.

### 2.1. Block Data Layout

To support multi-dimensional array representations in current programming languages, the default data layout is *row-major* or *column-major*, denoted as canonical layouts [7]. Both row-major and column-major layouts have similar drawbacks. For example, consider a large matrix stored in row-major layout. Due to large stride, column accesses can cause cache conflicts. Further, if every row in a matrix is larger than the size of a page, column accesses can cause TLB trashing, resulting in drastic performance degradation. In block data layout, a large matrix is partitioned into sub-matrices. Each sub-matrix is a $B \times B$ matrix and all elements in the sub-matrix are mapped onto contiguous memory locations. The blocks are arranged in row-major order. Figure 1 shows block data layout with block size $2 \times 2$.

### 2.2. TLB Performance of Block Data Layout

In this subsection, we present a lower bound on the TLB misses for any data layout. We discuss the intrinsic TLB performance of block data layout. We present an analysis on the TLB performance of block data layout and show that its performance is improved when compared with conventional layouts. Throughout this paper, we consider an $N \times N$ array. Also it is assumed that $N$ is large enough that $N \geq P_v \gg S_{tlb}$.

### 2.2.1 A Lower Bound on TLB Misses

In general, most matrix operations consist of row and column accesses, or permutations of row and column accesses, which are called *generic access pattern* [2] in this paper. In this section, we consider an access pattern where an array is accessed first along *all* rows *and* then along *all* columns. The lower bound analysis of TLB misses incurred in accessing the data array along all the rows and then all the columns is as follows.

**Theorem 2.1** *For accessing an array along all the rows and then along all the columns, the asymptotic [3] minimum number of TLB misses is given by* $2\frac{N^2}{\sqrt{P_v}}$.

**Proof:** Consider an arbitrary mapping of array elements to pages. Let $A_k = \{i|$ at least one element of row $i$ is in page $k\}$. Similarly, let $B_k = \{j|$ at least one element of column $j$ is in page $k\}$. Let $a_k = |A_k|$ and $b_k = |B_k|$. Note that $a_k \times b_k \geq P_v$. Using the mathematical identity that the arithmetic mean is greater than or equal to the geometric mean ( $a_k + b_k \geq 2\sqrt{a_k \times b_k} \geq 2\sqrt{P_v}$ ), we have:

$$\sum_{k=1}^{\frac{N^2}{P_v}} (a_k + b_k) \geq 2\frac{N^2}{P_v}\sqrt{P_v}.$$

Let $x_i$ ($y_j$) denote the number of pages where elements in row $i$ (column $j$) are scattered. The number of TLB misses in accessing all rows consecutively and then all columns consecutively is given by $T_{miss} \geq \sum_{i=1}^{N}(x_i - O(S_{tlb})) + \sum_{j=1}^{N}(y_j - O(S_{tlb}))$. $O(S_{tlb})$ is the number of page entries required for accessing row $i$ (column $j$) that are already present in the TLB. Page $k$ is accessed $a_k$ times by row accesses, thus, $\sum_{i=1}^{N} x_i = \sum_{k=1}^{\frac{N^2}{P_v}} a_k$. Similarly, $\sum_{j=1}^{N} y_j = \sum_{k=1}^{\frac{N^2}{P_v}} b_k$. Therefore, the total number of TLB misses is given by

$$T_{miss} \geq \sum_{k=1}^{\frac{N^2}{P_v}} (a_k + b_k) - 2N \cdot O(S_{tlb}) \geq 2 \times \frac{N^2}{\sqrt{P_v}} - 2N \cdot O(S_{tlb}). \quad (1)$$

As the problem size ($N$) increases, the number of pages accessed along a row (column) becomes larger than the size of TLB ($S_{tlb}$). Thus the number of TLB entries that are reused is reduced between two consecutive row (column) accesses. Therefore the asymptotic minimum number of TLB misses is given by $2\frac{N^2}{\sqrt{P_v}}$. ⊙

[2] In the rest of this paper, we refer to the access pattern of all rows and all columns as generic access pattern

[3] This asymptotic [9] bound holds true when $N$ is large. Also, the impact of $S_{tlb}$ becomes negligible when $N$ is large and hence does not appear in the bound.

We obtained a lower bound on TLB misses for any layout when data are accessed along all rows and then along all columns. This lower bound of TLB misses also holds when data is accessed along an arbitrary permutation of all rows and columns.

**Corollary 2.1** *For accessing an array along an arbitrary permutation of row and column accesses, the asymptotic minimum number of TLB misses is given by* $2\frac{N^2}{\sqrt{P_v}}$.

### 2.2.2 TLB Performance

In this section, we consider the same access pattern as discussed in Section 2.2.1. Consider a given $N \times N$ array stored in a canonical layout. During the first pass (row accesses), the memory pages are accessed consecutively. Therefore, TLB misses caused by row accesses is equal to $\frac{N^2}{P_v}$. During the second pass (column accesses), elements along the column are assigned to $N$ different pages. Hence, a column access causes $N$ TLB misses, since $N \gg S_{tlb}$. All $N$ column accesses result in $N^2$ TLB misses. The total number of TLB misses caused by all row accesses and all column accesses is thus $\frac{N^2}{P_v} + N^2$. Therefore, in canonical layout, TLB misses drastically increase due to column accesses.

Compared with canonical layout, block data layout has better TLB performance. The following theorem shows that block data layout minimizes the number of TLB misses.

**Theorem 2.2** *For accessing an array along all the rows and then along all the columns, block data layout with block size* $\sqrt{P_v} \times \sqrt{P_v}$ *minimizes the number of TLB misses.*

Detailed proof for this theorem can be found in [16]. In general, the number of TLB misses for a $B \times B$ block data layout is $k\frac{N^2}{B} + \frac{N^2}{B}$. It is reduced by a factor of $\frac{(P_v+1)B}{P_v(k+1)} (\approx \frac{B}{k+1})$ when compared with canonical layout. When $B = \sqrt{P_v}$ ($k = 1$), this number approaches the lower bound shown in Theorem 2.1.

This theorem holds true even when data in block data layout is accessed along an arbitrary permutation of all rows and columns.

**Corollary 2.2** *For accessing an array along an arbitrary permutation of rows and columns, block data layout with block size* $\sqrt{P_v} \times \sqrt{P_v}$ *minimizes the number of TLB misses.*

Even though block data layout has better TLB performance compared with canonical layouts for generic access patterns, it alone does not reduce cache misses. The data access pattern of tiling matches well with block data layout. In the following section, we discuss the performance improvement of TLB and caches when block data layout is used in conjunction with tiling.

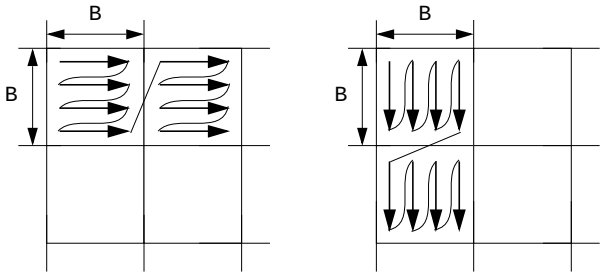
```
for kk=0 to N by B
    for jj=0 to N by B
        for i=0 to N
            for k=kk to min(kk+B-1,N)
                r = X(i,k)
                for j=jj to min(jj+B-1,N)
                    Z(i,j) += r*Y(k,j)
```

(a) 5-loop tiled matrix multiplication

```
for jj=0 to N by B
    for kk=0 to N by B
        for ii=0 to N by B
            for i=ii to min(ii+B-1,N)
                for k=kk to min(kk+B-1,N)
                    r = X(i,k)
                    for j=jj to min(jj+B-1,N)
                        Z(i,j) += r*Y(k,j)
```

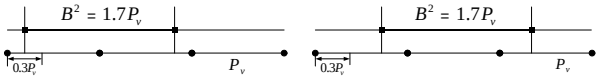
(b) 6-loop tiled matrix multiplication

**Figure 2. Tiled matrix multiplication**




(a) Tiled row access (b) Tiled column access

**Figure 3. Tiled accesses**




(a) over 2 pages (b) over 3 pages

**Figure 4. Blocks extending over page boundaries**

## 3. Performance Analysis of Block Data Layout with Tiling

Tiling is a well-known optimization technique that improves cache performance. Tiling transforms the loop nest so that temporal locality can be better exploited for a given cache size. Consider an $N \times N$ matrix multiplication represented as $\mathbf{Z} = \mathbf{XY}$. For large problems, its performance can suffer from severe cache and TLB thrashing. To reduce cache and TLB misses, tiling transforms the matrix multiplication to a 5-loop nest tiled matrix multiplication (TMM) as shown in Figure 2(a). To efficiently utilize block data layout, we consider a 6-loop TMM as shown in Figure 2(b).

### 3.1. TLB Performance

In this section, we show the TLB performance improvement of block data layout with tiling. To illustrate the effect of block data layout on tiling, we consider a generic access pattern abstracted from tiled matrix operations. The access pattern is shown in Figure 3, where the whole matrix is accessed first along the rows then along the columns, in a tiled pattern. The tile size is equal to $B$.

With canonical layout, TLB misses will not occur when accessing consecutive tiles in the same row, if $B \leq S_{tlb}$. Hence, the tiled accesses along the rows generate $\frac{N^2}{P_v}$ TLB misses. However, tiled accesses along columns cause considerable TLB misses. $B$ page table entries are necessary for accessing each tile. For all tiled column accesses, the total number of TLB misses is $T_{col} = B \times \frac{N}{B} \times \frac{N}{B} = \frac{N^2}{B}$. It is reduced by a factor of $B$ compared with the number of TLB misses for all column accesses without tiling (see Section 2.2).

The total number of TLB misses are further reduced when block data layout is used in concert with tiling [4]. This is formally stated in Theorem 3.1. To analyze TLB misses for tiled accesses using block data layout, we need to know the number of pages that a block of data is mapped onto. This is stated in Lemma 3.1.

**Lemma 3.1** *Consider an array stored in block data layout with block size $B \times B$, where $B^2 = kP_v$. The average number of pages that one block of data is mapped onto is $k + 1$.*

**Proof:** For block size $kP_v$, assume that $k = n + f$, where $n$ is a non-negative integer and $0 \leq f < 1$. An illustrative example of a block extending over page boundaries is shown in Figure 4. The probability that a block extends over $n + 1$ contiguous pages is $1 - f$. The probability that a block extends over $n + 2$ contiguous pages is $f$. Therefore, the average number of pages per block in block data layout is given by: $(1 - f) \times (n + 1) + f \times (n + 2) = k + 1$. ⊙

**Theorem 3.1** *Assume that an $N \times N$ array is stored using block data layout. For tiled access along the rows and then the columns, the total number of TLB misses is $(2 + \frac{1}{k})\frac{N^2}{P_v}$.*

**Proof:** Blocks in block data layout are arranged in row-major order. So, a page overlaps between two consecutive blocks that are in the same row. The page is continuously accessed. The number of TLB misses caused by all tiled row accesses is thus $\frac{N^2}{P_v}$, which is the minimum number of TLB misses. However, no page overlaps between two consecutive blocks in the same column. Therefore, each block along

[4] Throughout this paper, the block size of block data layout is assumed to be the same as the tile size so that the tiled access pattern matches the block data layout.

the same column goes through $(k+1)$ different pages according to Lemma 3.1. The number of TLB misses caused by all tiled column accesses is thus $T_{col} = (k+1) \times \frac{N}{B} \times \frac{N}{B} = (k+1)\frac{N^2}{kP_v}$. Therefore, the total TLB misses caused by all row and all column accesses is $T_{miss} = (2+\frac{1}{k})\frac{N^2}{P_v}$. $\odot$

For tiled access, the number of TLB misses using canonical layout is $\frac{N^2}{P_v} + \frac{N^2}{B}$, where $B = \sqrt{kP_v}$. Using Theorem 3.1, compared with canonical layout, block data layout reduces the number of TLB misses by $\frac{\sqrt{kP_v}+\sqrt{k}}{2k+1} = \frac{B+\sqrt{k}}{2k+1}$.

A similar analytical result can be derived for real applications. Consider the 5-loop TMM with canonical layout in Figure 2 (a). Array $\mathbf{Y}$ is accessed in a tiled row pattern. On the other hand, arrays $\mathbf{X}$ and $\mathbf{Z}$ are accessed in a tiled column pattern. A tile of each array is used in the inner loops $(i,k,j)$. The number of TLB misses for each array is equal to the average number of pages per tile, multiplied by the number of tiles accessed in the outer loops $(kk,jj)$. The average number of pages per tile is $B + \frac{B^2}{P_v}$. Therefore, the total number of TLB misses is given by: $2N^3(\frac{1}{B^2} + \frac{1}{BP_v}) + N^2(\frac{1}{B} + \frac{1}{P_v})$.

Consider the 6-loop TMM on block data layout as shown in Figure 2 (b). A $B \times B$ tile of each array is accessed in the inner loops $(i,k,j)$ with block layout. The number of TLB misses for each array is equal to the average number of pages per block multiplied by the number of blocks accessed in the outer loops $(ii,kk,jj)$. According to Lemma 3.1, the average number of pages per block is $\frac{B^2}{P_v} + 1 (= k+1)$. Therefore, the total number of TLB misses ($TM$) is

$$TM = 2N^3\left(\frac{1}{BP_v} + \frac{1}{B^3}\right) + N^2\left(\frac{1}{P_v} + \frac{1}{B^2}\right). \quad (2)$$

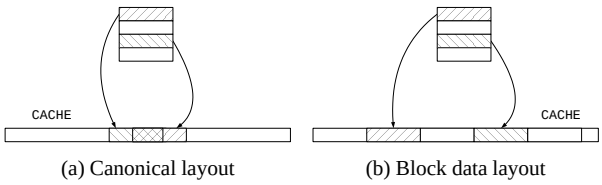
Compared with the 5-loop TMM with canonical layout, TLB misses decrease by a factor of $O(B)$ using the 6-loop TMM with block data layout.

### 3.2. Cache Performance

For a given cache size, tiling transforms the loop nest so that the temporal locality can be better exploited. This reduces capacity misses. However, since most of the state-of-the-art architectures have direct-mapped or small set-associative caches, tiling can suffer from considerable conflict misses as shown in Figure 5 (a). This degrades the overall performance.

We can reorganize a canonical layout to a block layout for tiled computations. Then as shown in Figure 5 (b), a self interference miss does not occur since all elements in a block can be mapped into contiguous locations in cache without any conflict.

In general, cache miss analysis for direct mapped cache with canonical layout is complicated because the self interference misses cannot be quantified easily. Cache performance analysis of tiled algorithm was discussed in [11]. The cache performance of tiling with copying optimization was also presented. We observe that the behavior of cache misses for tiled access patterns on block layout is similar to that of tiling with copying optimization on canonical layout. Also, self-interference misses can be easily quantified when block data layout is used. According to these, we have derived the total number of cache misses for 6-loop TMM with block data layout. Detailed proof can be found in [16]. For $i^{th}$ level cache with line size $L_{ci}$ and cache size $S_{ci}$, the total number of cache misses ($CM_i$) is:



(a) Canonical layout (b) Block data layout

**Figure 5. Example of conflict misses**

$$CM_i \approx \begin{cases} \frac{N^3}{L_{ci}}\left\{\frac{1}{B}\left(2+\frac{(3L_{ci}+2L_{ci}^2)}{S_{ci}}\right)+\frac{1}{N}+\frac{4B+6L_{ci}}{S_{ci}}\right\} \\ \qquad \text{for } B < \sqrt{S_{ci}} \\ \frac{N^3}{L_{ci}}\left\{\frac{4B}{S_{ci}}+\frac{2}{B}-\frac{2S_{ci}}{B^2}+2-\frac{1}{N}+\frac{6L_{ci}}{S_{ci}}\right\} \\ \qquad \text{for } \sqrt{S_{ci}} \le B < \sqrt{2S_{ci}} \\ \frac{N^3}{L_{ci}}\left\{1+\frac{2}{B}+\left(1+\frac{L_c}{B}\right)\left(\frac{B+2L_c}{S_{ci}}\right)\right\} \\ \qquad \text{for } \sqrt{2S_{ci}} \le B \end{cases} \quad (3)$$

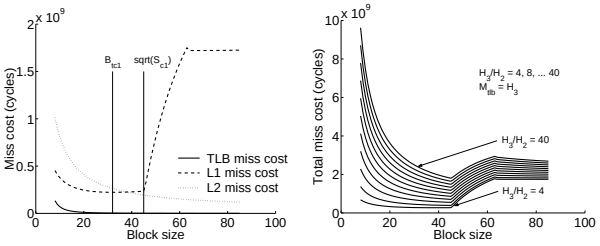
### 3.3. Block Size Selection

To achieve high performance, it is significant to select the block size of block data layout. In this section, we describe an approach for selecting the block size. In a multi-level memory hierarchy system, it is difficult to predict the execution time ($T_{exe}$) of a program. But, $T_{exe}$ is proportional to the total miss cost of TLB and cache. In order to minimize $T_{exe}$, we will evaluate and minimize the total miss cost for both TLB and $l$-level caches. We have:

$$MC = TM \cdot M_{tlb} + \sum_{i=1}^{l} CM_i H_{i+1} \quad (4)$$

where $MC$ denotes the total miss cost, $CM_i$ is the number of misses in the $i^{th}$ level cache, $TM$ is the number of TLB misses, $H_i$ is the cost of a hit in the $i^{th}$ level cache, and $M_{tlb}$ is the cost of a TLB miss. The $(l+1)^{th}$ level cache is the main memory. It is assumed that all data reside in the main memory ($CM_{l+1} = 0$).

For a simple 2-level memory hierarchy that consists of only one level cache and TLB, the total miss cost (denoted




(a) Miss cost of TLB, L1, and L2 cache ($B_{tc1}$ is obtained using Eq.( 6) )

(b) Total miss cost with various L2 miss penalty

**Figure 6. Miss cost estimation for 6-loop TMM (UltraSparc II parameters)**

as $MC_{tc1}$) in Eq. (4) reduces to:

$$MC_{tc1} = TM \cdot M_{tlb} + CM \cdot H_2, \tag{5}$$

where $H_2$ is the access cost of main memory. In the above estimation, $M_{tlb}$ and $CM$ are substituted with Eq.(2) and Eq.(3), respectively. Using the derivative of $MC_{tc1}$, the optimal block size, $B_{tc1}$, which minimizes the total miss cost caused by L1 cache and TLB misses is given as

$$B_{tc1} \approx \sqrt{\frac{\left(\frac{2L_{1c}M_{tlb}}{P_v} + \left[2 + \frac{3L_{c1}+2L_{c1}^2}{S_{c1}}\right] H_2\right) S_{c1}}{4H_2}}. \tag{6}$$

We now extend this analysis to determine a range for optimal block size in a multi-level memory hierarchy that consists of TLB and two levels of cache. The miss cost is classified into two groups: miss cost caused by TLB and L1 cache misses and miss cost caused by L2 misses. Figures 6 (a) and (b) show the miss cost estimated through Eqs.(2) and (3). Figure 6(a) represents the individual cost of TLB, L1, and L2 miss, using UltraSparc II parameters. Figure 6(b) shows the change of estimated total miss costs based on different ratios of L1 cache miss penalty ($H_2$) and L2 cache miss penalty ($H_3$). Using Eq.(6), we discuss the total miss cost for 3 ranges of block size:

**Lemma 3.2** *For $B < B_{tc1}$, $MC(B) > MC(B_{tc1})$.*

**Proof:** According to the derivatives, $\frac{dMC_{tc1}}{dB} < 0$ and $\frac{dCM_2}{dB} < 0$ for $B < B_{tc1}$, TLB, L1, and L2 miss costs increase as block size decreases. This is shown in Figure 6(a), thereby increasing the total miss cost. Therefore, the optimal block size cannot be in the range $B < B_{tc1}$. ⊙

**Lemma 3.3** *For $B > \sqrt{S_{c1}}$, $MC(B) > MC(\sqrt{S_{c1}})$.*

**Proof:** In the range $B > \sqrt{S_{c1}}$, the change in TLB miss cost is negligible as the block size increases. Since block size is larger than L1 cache size, self-interferences occur in this range. The number of L1 cache misses drastically increases as shown in Figure 6(a). For $\sqrt{S_{c1}} \leq B < \sqrt{2S_{c1}}$, although the number of L2 cache misses decreases ($\frac{dCM_2}{dB} < 0$), the ratio of derivatives of Eq.( 3) for L1 and L2 misses is as follows:

$$\left|\frac{H_2 \frac{dCM_1}{dB}}{H_3 \frac{dCM_2}{dB}}\right| = \frac{H_2}{H_3}\left|\frac{\frac{N^3}{L_{c1}}\left[\frac{4}{S_{c1}} + \frac{4Sc1}{B^3} - \frac{2}{B^2}\right]}{\frac{N^3}{L_{c2}}\left[\frac{4}{S_{c2}} - \left(2 + \frac{3L_{c2}+2L_{c2}^2}{S_{c2}}\right)\frac{1}{B^2}\right]}\right| > 1.$$

Therefore, the total miss cost increases for $\sqrt{S_{c1}} \leq B < \sqrt{2S_{c1}}$. For $B \geq \sqrt{2S_{c1}}$, there is no reuse in L1 cache. Thus, the L1 cache miss cost saturates. As shown in Figure 6(b), $TM(B) > TM(\sqrt{S_{c1}})$ for $B \geq \sqrt{2S_{c1}}$, because L1 miss cost is dominantly larger than L2 miss cost and TLB miss cost for $B \geq \sqrt{2S_{c1}}$. Therefore, the optimal block size cannot be in the range $B > \sqrt{S_{c1}}$. ⊙
Detailed proof of Lemma 3.3 can be found in [16].

**Theorem 3.2** *The optimal block size $B_{opt}$ satisfies $B_{tc1} \leq B_{opt} < \sqrt{S_{c1}}$.*

**Proof:** This follows from Lemma 3.2 and 3.3. Therefore, an optimal block size that minimizes the total miss cost is located in $B_{tc1} \leq B_{opt} < \sqrt{S_{c1}}$. We select a block size that is a multiple of $L_{c1}$ (L1 cache line size) in this range. ⊙

## 4 Experimental Results

To verify our TLB performance analysis, simulations for the generic access pattern (accessing along all rows and then all columns) were performed. Furthermore, three applications (matrix multiplication, LU decomposition, and Cholesky factorization) are tested through simulations and executions on real platforms to confirm our analysis.

### 4.1 Simulations of generic access pattern

To verify our TLB performance analysis, simulations were performed using the SimpleScalar simulator [3]. It is assumed that the page size is $8KByte$ and the data TLB is fully set-associative with 64 entries (similar to the data TLB in UltraSparc 2.) Double precision data points are assumed. A $32 \times 32$ block size is considered for block data layout.

Table 1 compares the TLB misses of block data layout with canonical layout when the matrix is accessed with a generic access pattern. Table 1 (a) shows the TLB misses for accesses along all rows and then all columns. For small problem sizes, TLB misses with block data layout are considerably less than those with canonical layout. For problem size $1024 \times 1024$ , TLB entries used in a column(row) access are almost fully reused in the next column(row) access, thereby $O(S_{tlb})$ in Eq.(1) becoming relatively large.

**Table 1. Comparison of TLB misses**

| Layout | 1024 | 2048 | 4096 |
|---|---|---|---|
| Block Layout | 2081 | 81794 | 1196033 |
| Canonical Layout | 1049601 | 4198401 | 16793601 |

(a) Along all rows and then all columns

| Layout | 1024 | 2048 | 4096 |
|---|---|---|---|
| Block Layout | 64140 | 273482 | 1080986 |
| Canonical Layout | 1053606 | 4208690 | 16822675 |

(b) Arbitrary permutation of row and column accesses

| Layout | 1024 | 2048 | 4096 |
|---|---|---|---|
| Block Layout | 64501 | 274473 | 1080465 |
| Canonical Layout | 1053713 | 4208681 | 16822395 |

(c) Arbitrary permutation of all rows followed by arbitrary permutation of all columns accesses

The number of TLB misses using block data layout is 504.37 times less than that using canonical layout. It is also less than the lower bound obtained from Theorem 2.1. For larger problem sizes, $O(S_{tlb})$ in Eq.(1) becomes negligible, since the TLB entries cannot be reused. Hence the total number of TLB misses approaches the lower bound. As shown in Table 1 (a), TLB misses with block data layout are upto 16 times less compared with canonical layout. Table 1 (b) and (c) confirm Corollary 2.1 and 2.2. With these access patterns, TLB entries referenced during one row(column) access are not reused when accessing the next row(column). The number of TLB misses with block data layout approaches the lower bound on TLB misses.

Table 2 shows simulation results for tiled row and column accesses. Block size is set to be the same as the tile size. As shown in Table 2, the number of TLB misses conform our analysis from Theorem 3.1. The number of TLB misses with block data layout is 91% less than that with canonical layout.

## 4.2 Experimental results for various applications

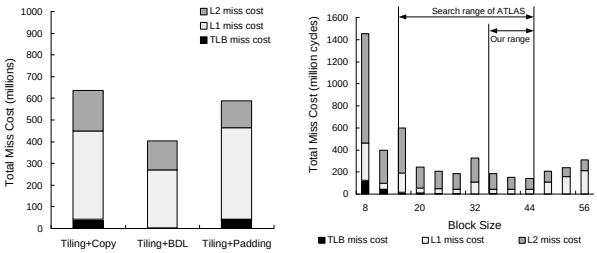
To show the effect of block data layout, we performed simulations and experiments on the following applications: tiled matrix multiplication(TMM), LU decomposition, and Cholesky factorization(CF). The performance of tiling with block data layout (tiling+BDL) is compared with other optimization techniques: tiling with copying(tiling+copying), and tiling with padding(tiling+padding). For tiling+BDL, the tile size (in tiling) is chosen to be the same as the block size in block data layout. Initial and final data layouts are canonical layouts. All the costs in performing data layout transformations (from canonical layout to block data layout and vice versa) are included in the reported results. As stated in [11], we observed that the copying technique cannot be applied efficiently to LU and CF applications, since copying overhead offsets the performance improvement. Hence we do not consider tiling+copying for these applications. In all our simulations and experiments, the data elements are double-precision.

**Table 2. TLB misses for all tiled row accesses followed by all tiled column accesses**

| Layout | 1024 | 2048 | 4096 |
|---|---|---|---|
| Block Layout | 2081 | 12289 | 49153 |
| Canonical Layout | 33794 | 139265 | 561025 |

### 4.2.1 Simulation results

To show the performance of TLB and caches using tiling+BDL, simulations were performed using the SimpleScalar simulator [3]. The problem size was $1024 \times 1024$.

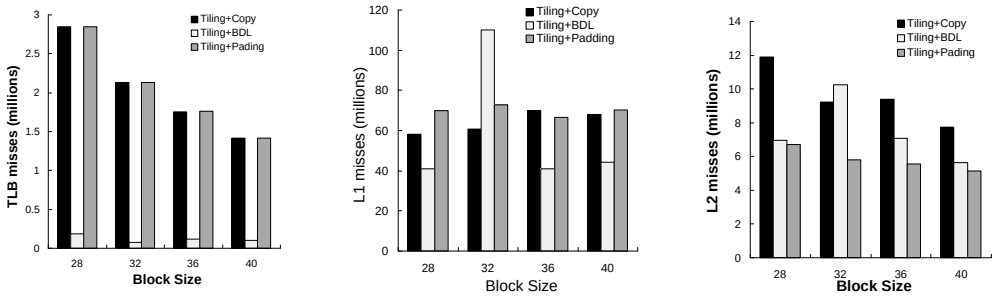



(a) The comparison of total miss cost

(b) Effect of block size on BDL

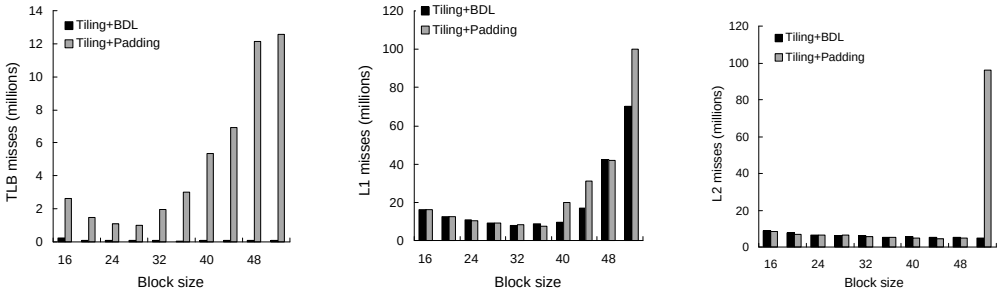
**Figure 8. Total miss cost for TMM using UltraSparc II parameters**

Figures 7 and 8 show the TMM simulation results , based on UltraSparc II parameters. As shown in Figure 7(a), Tiling+BDL reduced 91–96% of TLB misses. This confirms our analysis presented in Section 3.1. Figure 8 shows the total miss cost (calculated from Eq. (4)) for TMM. L1, L2, and TLB miss penalties were assumed to be 6, 24, and 30 cycles, respectively. Figure 8(a) shows the comparison of the total miss cost of tiling+BDL with that of tiling+copying and tiling+padding. The comparison shows that tiling+BDL results in the smallest total miss cost. Specifically, the TLB miss cost of tiling+BDL is negligible compared with L1 and L2 miss costs. Figure 8(b) shows the effect of block size on the total miss cost for TMM using tiling+BDL. As discussed in Section 3.3, $B_{tc1} = 32.2$, $\sqrt{S_{c1}} = 45.3$, and $L_{c1} = 4$ using this architecture parameters. Theorem 3.2 suggests the range for optimal block size to be 36–44. Simulation results show that the optimal block size for this architecture was 44.




(a) TLB misses (b) L1 misses (c) L2 misses

**Figure 7. Simulation results for TMM using UltraSparc II parameters**

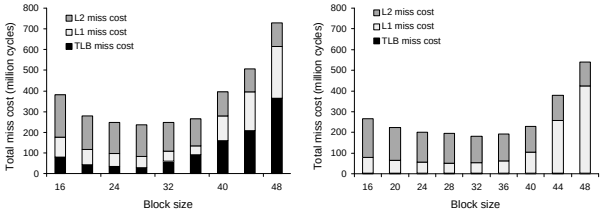



(a) TLB misses (b) L1 cache misses (c) L2 cache misses

**Figure 9. Simulation results for LU using Pentium III parameters**

As shown in Figure 8(b), our proposed range is much tighter than the search range of ATLAS.

Figure 9 and 10 present simulation results for LU using Intel Pentium III parameters. Similar to TMM, the number of TLB misses for tiling+BDL was almost negligible compared with that for tiling+padding as shown in Figure 9(a). For both techniques, L1 and L2 cache misses were reduced considerably because of 4-way set-associativity. For tiling+padding, when the block size was larger than L1 cache size, the padding algorithm in [15] suggested a pad size of 0. There is essentially no padding effect, thereby drastically increasing L1 and L2 cache misses. Figure 10 shows the block size effect on total miss cost using tiling+padding and tiling+BDL. Tiling+padding reduced L1 and L2 cache miss costs considerably. However, TLB miss costs were still significantly high, affecting the overall performance. As discussed in Section 3.3, the suggested range for optimal block size is 32–44. Simulations validate that the optimal block size achieving the smallest miss cost locates in the range selected using our approach.




(a) Tiling+Padding (b) Tiling+BDL

**Figure 10. Effect of block size on LU decomposition using Pentium III parameters**

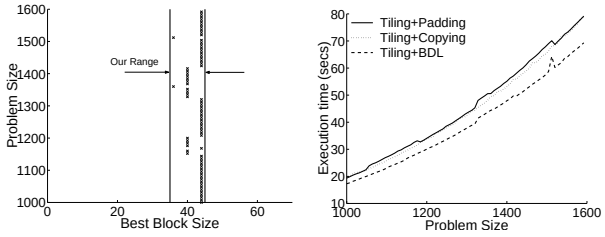
### 4.2.2 Application execution results on real platforms

To verify our block size selection and the performance improvements using block data layout, we performed experiments on several platforms. The parameters are tabulated in Table 3. `gcc` compiler was used in these experiments. The compiler optimization flags were set to "`-fomit-frame-pointer -O3 -funroll-loops`". Execution time was the user processor time measured by sys-call `clock()`. The problem sizes ranged from $1000 \times 1000$ to $1600 \times 1600$.

The experimental results of TMM using tiling+BDL on UltraSparc II is shown in Fig. 11. Fig. 11(a) shows the best block size for TMM with respect to different problem sizes. For each problem size, we performed experiments by test-

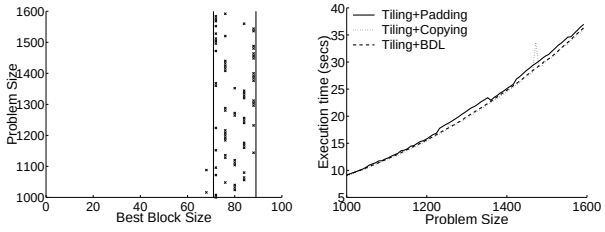
**Table 3. Features of various experimental platforms**

| Platforms | Speed (MHz) | L1 cache | | | L2 cache | | | TLB | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Size ($KB$) | Line ($Byte$) | Ass. | Size ($KB$) | Line ($Byte$) | Ass. | Entry | page ($KB$) | Ass. |
| Alpha 21264 | 500 | 64 | 64 | 2 | 4096 | 64 | 1 | 128 | 8 | 128 |
| UltraSparc II | 400 | 16 | 32 | 1 | 2048 | 64 | 1 | 64 | 8 | 64 |
| UltraSparc III | 750 | 64 | 32 | 4 | 4096 | 64 | 4 | 512 | 8 | 2 |
| Pentium III | 800 | 16 | 32 | 4 | 512 | 32 | 4 | 64 | 4 | 4 |



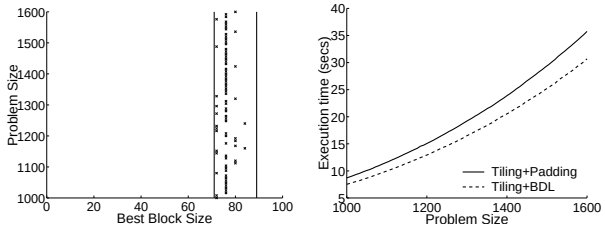

(a) Block size selection (b) Performance comparison

**Figure 11. Experimental results for TMM on UltraSPARC II**



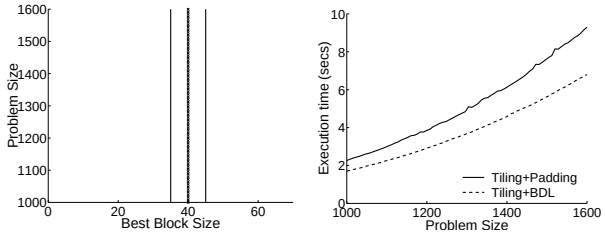

(a) Block size selection (b) Performance comparison

**Figure 12. Experimental results for TMM on Alpha 21264**




(a) Block size selection (b) Performance comparison

**Figure 13. Experimental results for LU on UltraSPARC III**




(a) Block size selection (b) Performance comparison

**Figure 14. Experimental results for Cholesky factorization on Pentium III**

ing block sizes ranging from 8–80. In all these tests, we found that the optimal block size for each problem size was in the range given by Theorem 3.2. This is shown in Figure 11(a). We also tested ATLAS. Through a wide search ranging from 16 to 44, ATLAS found 36 and 40 as the optimal block sizes. These blocks lie in the range given by Theorem 3.2. These experiments confirm that our approach proposes a reasonably good range for block size selection. Figures 11(b) show the execution time comparison of tiling+BDL with tiling+copying and tiling+padding. Figure 12–14 show experimental results for 3 different applications on 3 different platforms. Tiling+BDL technique is faster than using other optimization techniques, for almost all problem sizes and on all the platforms. These results confirm our analysis. More experimental results are available in [16].

## 5 Concluding Remarks

This paper studied a critical problem in understanding the performance of algorithms on state-of-the-art machines that employ multi-level memory hierarchy. We presented a lower bound on the number of TLB misses for any data layout and showed that block data layout achieves this bound. The number of TLB misses using tiling and block data layout were considerably reduced compared with copying or padding techniques. We showed that block data layout with tiling leads to improved overall memory hierarchy performance compared with other techniques. Further, we proposed a tight range for block size in ATLAS using our performance analysis. Our analysis was verified using simulations as well as actual execution results.

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.

## References


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

[2] Automatically Tuned Linear Algebra Software (ATLAS). http://math-atlas.sourceforge.net/.

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

[4] J. Chame, M. Hall, and J. Shin. Compiler Transformations for Exploiting Bandwidth in PIM-Based Systems. *Proceedings of Solving the Memory Wall Workshop, held in conjunction with the ISCA 2000*, June 2000.

[5] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. *Proceedings of the 13th ACM ICS '99*, June 1999.

[6] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive Array Layouts and Fast Parallel Matrix Multiplication. *Proceedings of the 11th ACM SPAA*, pages 222–371, June 1999.

[7] M. Cierniak and W. Li. Unifying Data and Control Transformations for Distributed Shared-Memory Machines. *Proceedings of the SCM SIGPLAN PLDI 1995*, pages 205–217, June 1995.

[8] S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. *Proceedings of the SIGPLAN PLDI 1995*, June 1995.

[9] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms in Pseudocode: The Human Dimension*. W. H. Freeman Press, 1998.

[10] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving Locality Using Loop and Data Transformations in an Integrated Framework. *Proceedings of the 31st IEEE/ACM International Symposium on Microarchitecture*, November 1998.

[11] M. Lam, E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. *Proceedings of ASPLOS-IV*, April 1991.

[12] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the Multi-Level Nature of Tiling Interactions. *International Journal of Parallel Programming*, 1998.

[13] D. Padua. Outline of a Roadmap for Compiler Technology. *IEEE Computing in Science & Engineering*, Fall 1996.

[14] D. Padua. The Fortran I Compiler. *IEEE Computing in Science & Engineering*, January/Febrary 2000.

[15] P. R. Panda, H. Nakamura, N. Dutt, and A. Nicolau. Augmenting Loop Tiling with Data Alignment for Improved Cache Performance. *IEEE Transactions on Computers*, 48(2), Feburary 1999.

[16] N. Park, B. Hong, and V. K. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. Technical Report USC-CENG 01-05, Department of Electrical Engineering, USC, September 2001.

[17] N. Park, D. Kang, K. Bondalapati, and V. K. Prasanna. Dynamic Data Layouts for Cache-conscious Factorization of DFT. *Proceedings of IPDPS 2000*, April 2000.

[18] N. Park and V. K. Prasanna. Cache Conscious Walsh-Hadamard Transform. *ICASSP 2001*, May 2001.

[19] G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. *ACM SIGPLAN PLDI 1998*, June 1998.

[20] G. Rivera and C.-W. Tseng. Locality Optimizations for Multi-Level Caches. *Proceedings of IEEE SC'99)*, November 1999.

[21] J. Sanchez, A. Gonzalez, and M. Valero. Static Locality Analysis for Cache Management. *PACT 1997*, November 1997.

[22] A. Saulsbury, F. Dahgren, and P. Stenström. Receny-based TLB Preloading. *ISCA 2000*, June 2000.

[23] O. Temam, E. D. Granston, and W. Jalby. To Copy or Not to Copy: A Comile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. *Proceedings of IEEE SC'93*, November 1993.

[24] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). *Proceedings of SC'98*, November 1998.

[25] Q. Yi, V. Adve, and K. Kennedy. Transforming Loops to Recursion for Multi-Level Memory Hierarchies. *ACM SIGPLAN PLDI 2000*, June 2000.

# Tiling, Block Data Layout, and Memory Hierarchy Performance

Neungsoo Park, Bo Hong,
and Viktor K. Prasanna

CENG 01-05

Department of Electrical Engineering - Systems
Uninversity of Southern California
Los Angeles, California 90089-2562
(213-740-4483)

# Tiling, Block Data Layout, and Memory Hierarchy Performance *

Neungsoo Park, Bo Hong, and Viktor K. Prasanna
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562
{neungsoo,bohong,prasanna}@ceng.usc.edu
http://advisor.usc.edu

**Abstract**

Recently, several experimental studies have been conducted on block data layout in conjunction with tiling as a data transformation technique to improve cache performance. In this paper, we analyze cache and TLB performance of such alternate layouts (including block data layout and Morton layout) when used in conjunction with tiling. We derive a tight lower bound on TLB performance for standard matrix access patterns, and show that block data layout and Morton layout achieve this bound. To improve cache performance, block data layout is used in concert with tiling. Based on the cache and TLB performance analysis, we propose a data block size selection algorithm that finds a tight range for optimal block size. To validate our analysis, we conducted simulations and experiments using tiled matrix multiplication, LU decomposition and Cholesky factorization. For matrix multiplication, simulation results using UltraSparc II parameters show that tiling and block data layout, with a block size given by our block size selection algorithm, reduces upto 93% of TLB misses compared with other techniques (copying, padding, etc.). L1 and L2 cache misses are also reduced. Experiments on several platforms (UltraSparc II and III, Alpha, and Pentium III) show that tiling with block data layout achieves up to 50% performance improvement over other techniques that use conventional layouts. Morton layout is also analyzed and compared with block data layout. Experimental results show that matrix multiplication using block data layout is upto 15% faster than that using Morton data layout.

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

# 1 Introduction

The increasing gap between memory latency and processor speed is a critical bottleneck in achieving high performance. The gap is typically bridged by a multi-level memory hierarchy that can hide memory latency. This memory hierarchy consists of multi-level caches, which are typically on- and off- chip caches. To improve the effective memory hierarchy performance, various hardware solutions have been proposed [3, 7, 9, 10, 19]. Recent processors such as Intel Merced [24] provide increased programmer control over data placement and movement in a cache-based memory hierarchy, in addition to providing some memory streaming hardware support for media applications. To exploit these features, it is important to understand the effectiveness of control and data transformations.

Along with hardware solutions, compiler optimization techniques have received considerable attention [14, 15, 22]. As the memory hierarchy gets deeper, it is critical to efficiently manage the data. To improve data access performance, one of the well-known optimization techniques is tiling. Tiling transforms loop nests so that temporal locality can be better exploited for a given cache size. However, tiling focuses only on the reduction of capacity cache misses by decreasing the working set size. Cache in most state-of-the-art machines is either direct-mapped or small set-associative. Thus, it suffers from considerable conflict misses, thereby degrading the overall performance [6, 12]. To reduce conflict misses, copying [12, 25] and padding [16, 20] techniques with tiling have been proposed. However, most of these approaches target mainly the cache performance, paying less attention to the Translation Look-aside Buffer (TLB) performance. As problem sizes become larger, TLB performance becomes more significant. If TLB thrashing occurs, the overall performance will be drastically degraded [23]. Hence, both TLB and cache must be considered in optimizing application performance.

Most previous optimizations, including tiling, concentrate on single-level cache [6, 12, 16, 20, 25]. Multi-level memory hierarchy has been considered by a few researchers. For improving multi-level memory hierarchy performance, a new compiler technique is proposed in [27] that transforms loop nests into recursive form. However, only multi-level caches were considered [21, 27] with no emphasis on TLB. It was proposed in [13] that cache and TLB performance be considered *in concert* to select the tile size. In this analysis, TLB and cache were assumed to be fully-set associative. However, the cache is direct or small set-associative in most of the state-of-the-art platforms.

Some recent work [4, 11, 12, 17, 18, 25] proposed changing the data layout to match the data access pattern, to reduce cache misses. It was proposed in [11] that both data and loop transformation be applied to loop nests for optimizing cache locality. In [4], conventional (row or column-major) layout is changed to a recursive data layout, referred to as Morton layout, which matches the access pattern of recursive algorithms. This data layout was shown to improve the memory hierarchy performance. This was confirmed through experiments; we are not aware of any formal analysis.

The ATLAS project [26] automatically tunes several linear algebra implementations. It uses block data layout with tiling to exploit temporal and spacial locality. Input data, originally in column major layout, is re-mapped into block data layout before the computation begins. The combination of block data layout and tiling has shown high performance on various platforms. However, the selection of the optimal block size is done *empirically* at compile time by running several tests with different block sizes.

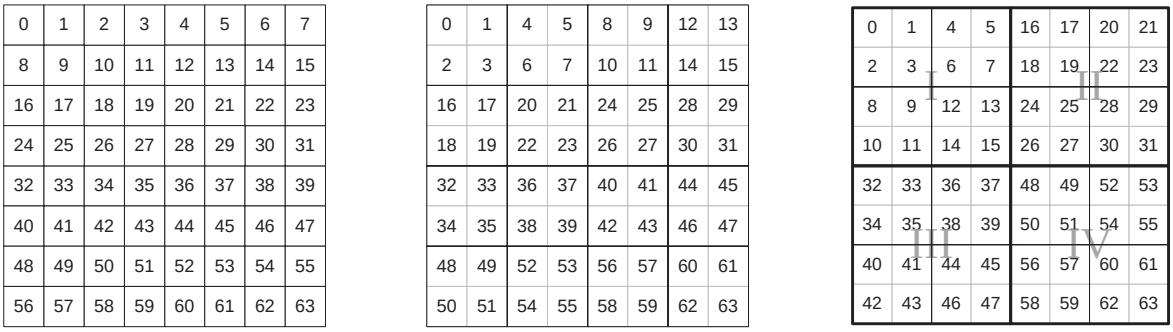
In this paper, we study *block data layout* as a data transformation to improve memory hierarchy performance. In block data layout, a matrix is partitioned into sub-matrices called blocks. Data elements within one such block are mapped onto contiguous memory. These blocks are arranged in row-major order. First, we analyze the intrinsic TLB performance of block data layout. We then analyze the TLB and cache performance using tiling and block data layout. Based on the analysis, we propose a block size selection algorithm. *Morton data layout* is also discussed as a variant of block data layout. The contributions of this paper are as follows:

- We present a lower bound analysis of TLB performance. Further, we show that block data layout intrinsically has better TLB performance than row-major layout (Section 2). As an abstraction of matrix operations, the cost of accessing all rows and all columns is analyzed. Compared with row major layout, we show that the number of TLB misses is improved by $O(\sqrt{P_v})$ where $P_v$ is the page size.
- We present TLB and cache performance analysis when tiling is used with block data (Section 3.1 and 3.2). In tiled matrix multiplication, block data layout improves the number of TLB misses by a factor of $B$, where $B$ is the block size. Cache performance analysis is also presented. We validate our analysis through simulations using SimpleScalar [2].
- On the basis of our cache and TLB analysis, we propose a block size selection algorithm (Section 3.3). The best block sizes found by ATLAS fall in the range given by our algorithm.
- We validate our analysis through simulations and measurements using matrix multiply, LU decomposition and Cholesky factorization (Section 4).
- We compare the performance of block data layout and Morton data layout. Block size selection for Morton data layout is limited. This limitation causes the performance of Morton data layout to be worse than that of block data layout. Experimental results on UltraSparc II and Pentium III show that matrix multiplication and LU decomposition executions using block data layout were up to 15.8% faster than that obtained using Morton data layout.

The rest of this paper is organized as follows. Section 2 describes block data layout and gives analysis of its TLB performance. Section 3 discusses the TLB and cache performance when tiling and block data layout are used in concert. A block size selection algorithm is described based on this analysis. Section 4 shows simulation and experimental results. Concluding remarks are presented in Section 5.

## 2 Block Data Layout and TLB Performance

In this paper, we assume the architecture parameters to be fixed (e.g. cache size, cache line size, page size, TLB entry capacity, etc.). The following notations are used in this paper. $S_{tlb}$ denotes the number of TLB entries. $P_v$ denotes virtual page size. It is assumed that the TLB is fully set-associative with Least-Recently-Used(LRU) replacement policy. Block size is $B \times B$, where it is assumed $B^2 = kP_v$. $S_{ci}$ is the size of the $i^{th}$ level cache. Its line size is denoted as $L_{ci}$. Cache is assumed to be direct-mapped and its replacement policy is also LRU.




(a) Row-major layout (b) Block data layout (c) Morton data layout

Figure 1: Various data layouts: block size $2 \times 2$ for (b) and (c)

In Section 2, we analyze the TLB performance of block data layout. We show that block data layout has better intrinsic TLB performance than conventional data layouts.

## 2.1 Block Data Layout

To support multi-dimensional array representations, most programming languages provide a mapping function, which converts an array index to a linear memory address. In current programming languages, the default data layout is *row-major* or *column-major*, denoted as canonical layouts [5]. Both row-major and column-major layouts have similar drawbacks. For example, consider a large matrix stored in row-major layout. Due to large stride, column accesses can cause cache conflicts. Further, if every row in a matrix is larger than the size of a page, column accesses can cause TLB trashing, resulting in drastic performance degradation.

In block data layout, a large matrix is partitioned into sub-matrices. Each sub-matrix is a $B \times B$ matrix and all elements in the sub-matrix are mapped onto contiguous memory locations. The blocks are arranged in row-major order. Another data layout of recent interest is Morton data layout [4]. Morton data layout divides the original matrix into four quadrants and lays out these sub-matrices contiguously in the memory. Each of these sub-matrices is further recursively divided and laid out in the same way. At the end of recursion, elements of the sub-matrix are stored contiguously. This is similar to the arrangement of elements of a block in block data layout. Morton data layout can thus be considered as a variant of the block data layout. They only differ in the order of blocks. Figure 1 shows block data layout and Morton data layout with block size $2 \times 2$. Due to the similarity, the following TLB analysis holds true for Morton data layout also.

## 2.2 TLB Performance of Block Data Layout

In this section, we present a lower bound on the TLB misses for any data layout. We discuss the intrinsic TLB performance of block data layout using a generic access pattern. We give an analysis on the TLB performance of block data layout and show improved performance compared with conventional layouts. Throughout this section, we consider an $N \times N$ array.

### 2.2.1 A Lower Bound on TLB Misses

In general, most matrix operations consist of row and column accesses, or permutations of row and column accesses. In this section, we consider an access pattern where an array is accessed first along *all* rows *and* then along *all* columns. The lower bound analysis of TLB misses incurred in accessing the data array along all the rows and all the columns is as follows.

**Theorem 2.1** *For accessing an array along all the rows and then along all the columns, the asymptotic minimum number of TLB misses is given by* $2\frac{N^2}{\sqrt{P_v}}$.

**Proof:** Consider an arbitrary mapping of array elements to pages. Let $A_k = \{i|$ at least one element of row $i$ is in page $k$ $\}$. Similarly, let $B_k = \{j|$ at least one element of column $j$ is in page $k$ $\}$. Let $a_k = |A_k|$ and $b_k = |B_k|$. Note that $a_k \times b_k \geq P_v$. Using the mathematical identity that the arithmetic mean is greater than or equal to the geometric mean ( $a_k + b_k \geq 2\sqrt{P_v}$ ), we have:

$$\sum_{k=1}^{\frac{N^2}{P_v}} (a_k + b_k) \geq 2\frac{N^2}{P_v}\sqrt{P_v}.$$

Let $x_i$ ($y_j$) denote the number of pages where elements in row $i$ (column $j$) are scattered. The number of TLB misses in accessing all rows consecutively and then all columns consecutively is given by $T_{miss} \geq \sum_{i=1}^{N}(x_i - O(S_{tlb})) + \sum_{j=1}^{N}(y_j - O(S_{tlb}))$. $O(S_{tlb})$ is the number of page entries required for accessing row $i$ (column $j$) that are already present in the TLB. Page $k$ is accessed $a_k$ times by row accesses, thus, $\sum_{i=1}^{N} x_i = \sum_{k=1}^{\frac{N^2}{P_v}} a_k$. Similarly, $\sum_{j=1}^{N} y_j = \sum_{k=1}^{\frac{N^2}{P_v}} b_k$. Therefore, the total number of TLB misses is given by

$$T_{miss} \geq \sum_{k=1}^{\frac{N^2}{P_v}} (a_k + b_k) - 2N \cdot O(S_{tlb}) \geq 2 \times \frac{N^2}{\sqrt{P_v}} - 2N \cdot O(S_{tlb}).$$

As the problem size ($N$) increases, the number of pages accessed along one row (column) becomes larger than the size of TLB ($S_{tlb}$). Thus the number of TLB entries that are reused is reduced between two consecutive row (column) accesses. Therefore the asymptotic minimum number of TLB misses is given by $2\frac{N^2}{\sqrt{P_v}}$. ⊙

We obtained a lower bound on TLB misses for any layout when data are accessed along all rows and then along all columns. This lower bound of TLB misses also holds when data is accessed along an arbitrary permutation of all rows and columns.

**Corollary 2.1** *For accessing an array along an arbitrary permutation of row and column accesses, the asymptotic minimum number of TLB misses is given by* $2\frac{N^2}{\sqrt{P_v}}$.

### 2.2.2 TLB Performance

In this section, we consider the same access pattern as discussed in Section 2.2.1. Consider a given $N \times N$ array stored in a canonical layout. Without loss of generality, canonical layout is assumed

Table 1: Comparison of TLB misses

(a) Along all rows and then all columns

| Layout | 1024 | 2048 | 4096 |
|---|---|---|---|
| Block Layout | 2081 | 81794 | 1196033 |
| Morton Layout | 2072 | 274473 | 1081466 |
| Canonical Layout | 1049601 | 4198401 | 16793601 |

(b) Arbitrary permutation of row and column accesses

| Layout | 1024 | 2048 | 4096 |
|---|---|---|---|
| Block Layout | 64140 | 273482 | 1080986 |
| Morton Layout | 64257 | 273477 | 1080955 |
| Canonical Layout | 1053606 | 4208690 | 16822675 |

(c) Arbitrary permutation of all rows followed by arbitrary permutation of all columns accesses

| Layout | 1024 | 2048 | 4096 |
|---|---|---|---|
| Block Layout | 64501 | 274473 | 1080465 |
| Morton Layout | 64813 | 274472 | 1081469 |
| Canonical Layout | 1053713 | 4208681 | 16822395 |

to be row-major layout. During the first pass (row accesses), the memory pages are accessed consecutively. Therefore, TLB misses caused by row accesses is equal to $\frac{N^2}{P_v}$. During the second pass (column accesses), elements along the column are assigned to $N$ different pages. Hence, a column access causes $N$ TLB misses. Since $N \gg S_{tlb}$, all $N$ column accesses result in $N^2$ TLB misses. The total number of TLB misses caused by all row accesses and all column accesses is thus $\frac{N^2}{P_v} + N^2$. Therefore, in canonical layout, TLB misses drastically increase due to column accesses.

Compared with canonical layout, block data layout has better TLB performance. The following theorem shows that block data layout minimizes the number of TLB misses.

**Theorem 2.2** *For accessing an array along all the rows and then along all the columns, block data layout with block size $\sqrt{P_v} \times \sqrt{P_v}$ minimizes the number of TLB misses.*

Detailed proof of this theorem is presented in Appendix A. In general, the number of TLB misses for a $B \times B$ block data layout is $k\frac{N^2}{B} + \frac{N^2}{B}$. It is reduced by a factor of $\frac{(P_v+1)B}{P_v(k+1)} (\approx \frac{B}{k+1})$ when compared with canonical layout. When $B = \sqrt{P_v}$ $(k = 1)$, this number approaches the lower bound shown in Theorem 2.1.

This theorem holds true even when data in block data layout is accessed along an arbitrary permutation of all rows and columns.

**Corollary 2.2** *For accessing an array along an arbitrary permutation of rows and columns, block data layout with block size $\sqrt{P_v} \times \sqrt{P_v}$ minimizes the number of TLB misses.*

Similar to Theorem 2.2 and Corollary 2.2, the number of TLB misses is minimized when blocks are stored in Morton data layout and elements are accessed along rows and columns.

**Corollary 2.3** *For accessing an $N \times N$ array along along all the rows and then along all the columns (or along an arbitrary permutation of rows and columns), Morton data layout with block size $\sqrt{P_v} \times \sqrt{P_v}$ minimizes the number of TLB misses.*

To verify our analysis, simulations were performed using the SimpleScalar simulator [2]. It is assumed that the page size is $8KByte$ and the data TLB is fully set-associative with 64 entries

```
for kk=0 to N by B
    for jj=0 to N by B
        for i=0 to N
            for k=kk to min(kk+B-1,N)
                r = X(i,k)
                for j=jj to min(jj+B-1,N)
                    Z(i,j) += r*Y(k,j)
```

(a) 5-loop tiled matrix multiplication

```
for jj=0 to N by B
    for kk=0 to N by B
        for ii=0 to N by B
            for i=ii to min(ii+B-1,N)
                for k=kk to min(kk+B-1,N)
                    r = X(i,k)
                    for j=jj to min(jj+B-1,N)
                        Z(i,j) += r*Y(k,j)
```

(b) 6-loop tiled matrix multiplication

Figure 2: Tiled matrix multiplication

(similar to the data TLB in UltraSparc 2.) Double precision data points are assumed. The block size is set to 32. Table 1 shows the comparison of TLB misses using block data layout with using canonical layout. Table 1 (a) shows the TLB misses for the "first all rows and then all columns" access. For small problem sizes, TLB misses with block data layout are considerably less than those with canonical layout. This is due to the fact that TLB entries used in a column(row) access are almost fully reused in the next column(row)access. For a problem size of $1024 \times 1024$, a 504.37 times improvement in the number of TLB misses is obtained with block data layout. This number is much less than the lower bound obtained from Theorem 2.1. This is because the TLB entries are reused for this problem size. For larger problem sizes the TLB entries cannot be reused. The total number of TLB misses approaches the lower bound. For these large problem sizes, TLB misses with block data layout are upto 16 times less compared with canonical layout.

To verify Corollary 2.1 and 2.2, two sets of access patterns were simulated: an arbitrary permutation of all rows and columns, and an arbitrary permutation of all rows followed by an arbitrary permutation of all columns. With these access patterns, TLB entries referenced during one row(column) access are not reused when accessing the next row(column). The number of TLB misses with block data layout approaches the lower bound on TLB misses. The results are shown in Table 1 (b) and (c). Morton data layout shows a performance similar to block data layout.

Even though block data layout has better TLB performance compared with canonical layouts with generic access pattern, it alone does not reduce cache misses. The data access pattern of tiling matches well with block data layout. In the following section, we discuss the performance improvement of TLB and caches when block data layout is used in conjunction with tiling.

# 3 Tiling and Block Data Layout

Tiling is a well-known optimization technique that improves cache performance. Tiling transforms the loop nest so that temporal locality can be better exploited for a given cache size. Consider an $N \times N$ matrix multiplication represented as $\mathbf{Z} = \mathbf{XY}$. The working set size for the usual 3-loop computation is $N^2 + 2N$. For large problems, the working set size is larger than the cache size, resulting in severe cache thrashing. To reduce cache capacity misses, tiling transforms the matrix multiplication to a 5-loop nest tiled matrix multiplication (TMM) as shown in Figure 2(a). The working set size for this tiled computation is $B^2 + 2B$. To efficiently utilize block data layout, we consider a 6-loop TMM as shown in Figure 2(b) instead of a 5-loop TMM.

(a) Tiled row access (b) Tiled column access

Figure 3: Tiled accesses

### 3.1 TLB Performance

In this section, we show the TLB performance improvement of block data layout with tiling. To illustrate the effect of block data layout on tiling, we consider a generic access pattern abstracted from tiled matrix operations. The access pattern is shown in Figure 3. The tile size is equal to $B$. Figure 3.

With canonical layout, TLB misses will not occur when accessing consecutive tiles in the same row, if $B \leq S_{tlb}$. Hence, the tiled accesses along the rows generate $\frac{N^2}{P_v}$ TLB misses. This is the minimum number of TLB misses incurred in accessing all the elements in a matrix. However, tiled accesses along columns cause considerable TLB misses. $B$ page table entries are necessary for accessing each tile. For all tiled column accesses, the total number of TLB misses is $T_{col} = B \times \frac{N}{B} \times \frac{N}{B} = \frac{N^2}{B}$. It is reduced by a factor of $B$ compared with the number of TLB misses for all column accesses without tiling (see Section 2.2).

The total number of TLB misses are further reduced when block data layout is used in concert with tiling, as shown in Theorem 3.1. Throughout this paper, the block size of block data layout is assumed to be the same as the tile size so that the tiled access pattern matches block data layout. In block data layout, a 2-dimensional block is mapped onto 1-dimensional contiguous memory locations. A block extends over several pages, as shown in Figure 4 for an example of block size $B^2 = 1.7P_v$. To analyze TLB misses for column accesses using block data layout, the average number of pages in a block is required.

(a) over 2 pages (b) over 3 pages

Figure 4: Blocks extending over page boundaries

**Lemma 3.1** *Consider an array stored in block data layout with block size $B \times B$, where $B^2 = kP_v$. The average number of pages per block is given by $k + 1$.*

**Proof:** For block size $kP_v$, assume that $k = n + f$, where $n$ is a non-negative integer and $0 \leq f < 1$.

Table 2: TLB misses for all tiled row accesses followed by all tiled column accesses

| Layout | 1024 | 2048 | 4096 |
|---|---|---|---|
| Block Layout | 2081 | 12289 | 49153 |
| Canonical Layout | 33794 | 139265 | 561025 |

The probability that a block extends over $n+1$ contiguous pages is $1-f$. The probability that a block extends over $n+2$ contiguous pages is $f$. Therefore, the average number of pages per block in block data layout is given by: $(1-f)\times(n+1)+f\times(n+2)=k+1$. $\odot$

**Theorem 3.1** *Assume that an $N\times N$ array is stored using block data layout. For tiled row and column accesses, the total number of TLB misses is* $(2+\frac{1}{k})\frac{N^2}{P_v}$.

**Proof:** Blocks in block data layout are arranged in row-major order. So, a page overlaps between two consecutive blocks that are in the same row. The page is continuously accessed. The number of TLB misses caused by all tiled row accesses is thus $\frac{N^2}{P_v}$, which is the minimum number of TLB misses. However, no page overlaps between two consecutive blocks in the same column. Therefore, each block along the same column goes through $(k+1)$ different pages according to Lemma 3.1. The number of TLB misses caused by all tiled column accesses is thus $T_{col}=(k+1)\times\frac{N}{B}\times\frac{N}{B}=(k+1)\frac{N^2}{kP_v}$. Therefore, the total TLB misses caused by all row and all column accesses is $T_{miss}=(2+\frac{1}{k})\frac{N^2}{P_v}$. $\odot$

For tiled access, the number of TLB misses using canonical layout is $\frac{N^2}{P_v}+\frac{N^2}{B}$, where $B=\sqrt{kP_v}$. Using Theorem 3.1, compared with canonical layout, block data layout reduces the number of TLB misses by $\frac{\sqrt{kP_v}+\sqrt{k}}{2k+1}=\frac{B+\sqrt{k}}{2k+1}$.

To verify our analysis, simulations for tiled row and column accesses were performed using the SimpleScalar simulator. The simulation parameters are equal to those in Section 2. A $32\times32$ block size was considered. The block size is the same as the page size. Table 2 shows TLB misses for 3 different cases. For problem sizes of $2048\times2048$ and $4096\times4096$, the number of TLB misses conform our analysis in Theorem 3.1. The number of TLB misses with block data layout is 91% less than that with canonical layout. For a problem size of $1024\times1024$, TLB misses with block data layout is 2081, which is very close to the minimum number of TLB misses (2048). This is a special case in which each block starts on a new page.

A similar analytical result can be derived for real applications. Consider the 5-loop TMM with canonical layout in Figure 2 (a). Array **Y** is accessed in a tiled row pattern. On the other hand, arrays **X** and **Z** are accessed in a tiled column pattern. A tile of each array is used in the inner loops $(i,k,j)$. The number of TLB misses for each array is equal to the average number of pages per tile, multiplied by the number of tiles accessed in the outer loops $(kk,jj)$. The average number of pages per tile is $B+\frac{B^2}{P_v}$. Therefore, the total number of TLB misses is given by: $2N^3(\frac{1}{B^2}+\frac{1}{BP_v})+N^2(\frac{1}{B}+\frac{1}{P_v})$.

Consider the 6-loop TMM on block data layout as shown in Figure 2 (b). A $B\times B$ tile of each array is accessed in the inner loops $(i,k,j)$ with block layout. The number of TLB misses for each array is equal to the average number of pages per block multiplied by the number of blocks

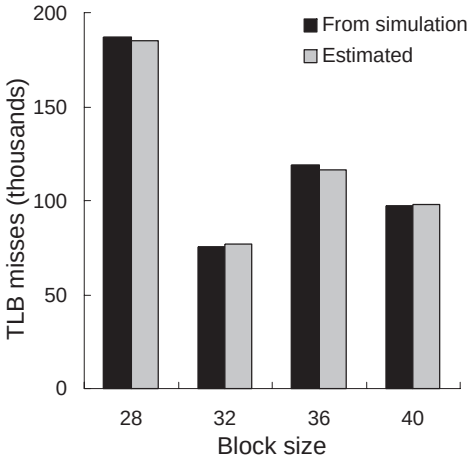



Figure 5: Comparison of TLB misses from simulation and estimation

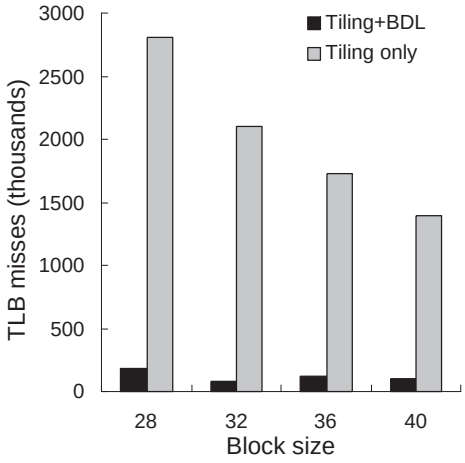



Figure 6: Comparison of TLB misses using tiling+BDL and tiling only

accessed in the outer loops $(ii, kk, jj)$. According to Lemma 3.1, the average number of pages per block is $\frac{B^2}{P_v} + 1 (= k + 1)$. Therefore, the total number of TLB misses $(TM)$ is

$$TM = \left(\frac{B^2}{P_v} + 1\right)\left\{2\left(\frac{N}{B}\right)^3 + \left(\frac{N}{B}\right)^2\right\} = 2N^3\left(\frac{1}{BP_v} + \frac{1}{B^3}\right) + N^2\left(\frac{1}{P_v} + \frac{1}{B^2}\right). \quad (1)$$

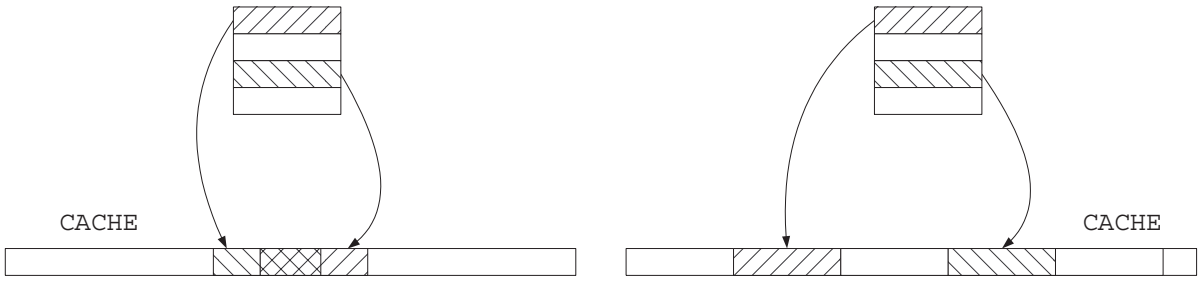
Compared with the 5-loop TMM with canonical layout, TLB misses decrease by a factor of $O(B)$ using the 6-loop TMM. Note that the 6-loop TMM uses block data layout.

To verify our TLB miss estimation, simulations on the 6-loop TMM were performed. The problem size was fixed at $1024 \times 1024$. Simulation parameters were the same as those in Section 2. Figure 5 compares our estimations (given by Eq. (1)) with the simulation results. Figure 6 shows that block data layout reduced TLB misses considerably compared with tiling.

### 3.2 Cache Performance

For a given cache size, tiling transforms the loop nest so that the temporal locality can be better exploited. This reduces the capacity misses. However, since most of the state-of-the-art architectures have direct-mapped or small set-associative caches, tiling can suffer from considerable conflict misses that degrade the overall performance. Figure 7 (a) shows cache conflict misses. These conflict misses are determined by cache parameters such as cache size, cache line size and set-associativity, and runtime parameters such as array size and block size. Performance of tiled computations is thus sensitive to these runtime parameters.

If the data layout is reorganized from a canonical layout to a block layout (assuming tile size is same as block size) before tiled computations start, the entire data that is accessed during a tiled computation will be localized in a block. As shown in Figure 7 (b), a self interference miss does not occur if the block is smaller than the cache since all elements in a block can be stored in contiguous memory locations.




(a) Canonical layout (b) Block data layout

Figure 7: Example of conflict misses

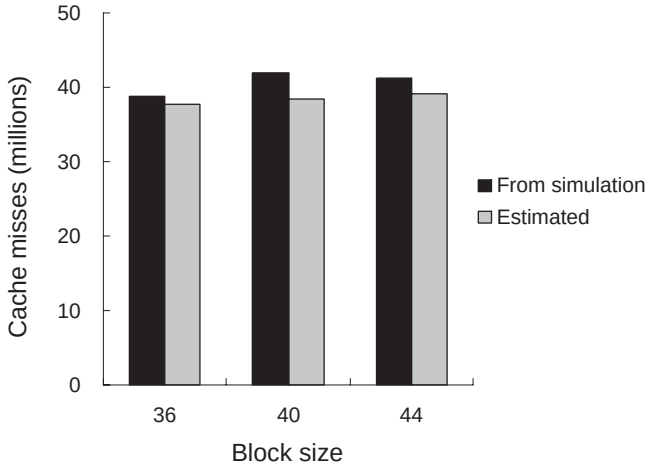



Figure 8: Comparison of cache misses from simulation and estimation for 6-loop TMM

In general, cache miss analysis for direct mapped cache with canonical layout is complicated because the self interference misses cannot be quantified easily. Cache performance analysis of tiled algorithm was discussed in [12]. The cache performance of tiling with copying optimization was also presented. We observe that the behavior of cache misses for tiled access patterns on block layout is similar to that of tiling with copying optimization on canonical layout. We have derived the total number of cache misses for 6-loop TMM (which uses block data layout). Detailed proof can be found in Appendix B. For $i^{th}$ level cache with line size $L_{ci}$ and cache size $S_{ci}$, the total number of cache misses ($CM_i$) is:

$$CM_i \approx \begin{cases} \frac{N^3}{L_{ci}} \left\{ \frac{1}{B} \left( 2 + \frac{(3L_{ci}+2L_{ci}^2)}{S_{ci}} \right) + \frac{1}{N} + \frac{4B+6L_{ci}}{S_{ci}} \right\} & \text{for } B < \sqrt{S_{ci}} \\ \frac{N^3}{L_{ci}} \left\{ \frac{4B}{S_{ci}} + \frac{2}{B} - \frac{2S_{ci}}{B^2} + 2 - \frac{1}{N} + \frac{6L_{ci}}{S_{ci}} \right\} & \text{for } \sqrt{S_{ci}} \leq B < \sqrt{2S_{ci}} \\ \frac{N^3}{L_{ci}} \left\{ 1 + \frac{2}{B} + \left( 1 + \frac{L_c}{B} \right) \left( \frac{B+2L_c}{S_{ci}} \right) \right\} & \text{for } \sqrt{2S_{ci}} \leq B \end{cases} \quad (2)$$

To verify the cache miss estimations, we conducted simulations using SimpleScalar for 6-loop TMM with block data layout. The problem size was fixed at $1024 \times 1024$. A $16KByte$ direct mapped

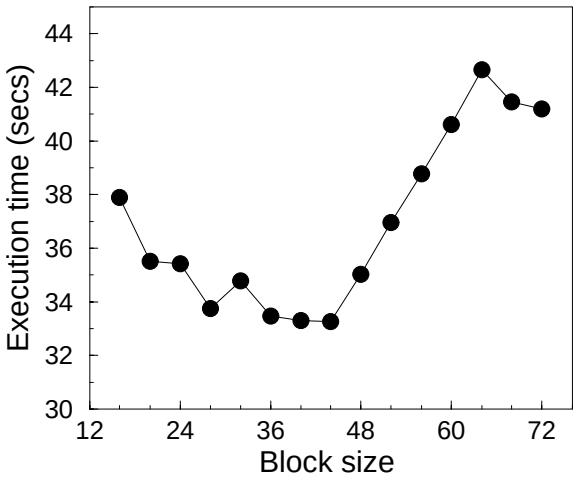



Figure 9: Execution time of TMM of size $1024 \times 1024$

cache was assumed (similar to L1 data cache in UltraSparc II). Figure 8 compares our estimated values (given by Eq. (2)) with the simulation results.

### 3.3 Block Size Selection

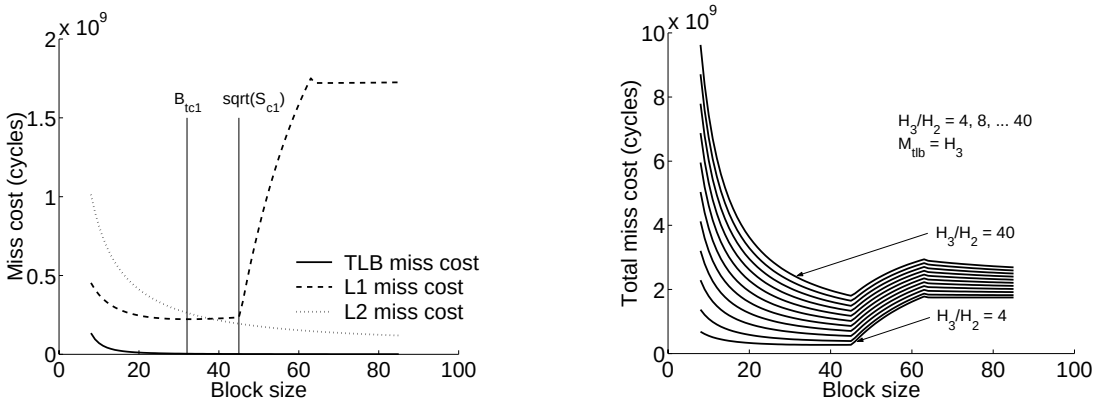
To test the effect of block size, experiments were performed on several platforms. Figure 9 shows the execution time of a 6-loop TMM with size $1024 \times 1024$ on UltraSparc II (400 MHz) as a function of block size. It can be observed that block size selection is significant for achieving high performance.

With canonical data layout, tiling technique is sensitive to problem and tile sizes. Several GCD based tile size selection algorithms [6, 8, 12] were proposed to optimize tiled computation. However, their performance is still sensitive to the problem size. In [13], TLB and cache performance were considered in concert. This approach showed better performance than algorithms that separately consider cache or TLB. However, all these approaches are based on canonical data layout. On the other hand, ATLAS [26] utilizes block data layout. However, the best block size is determined *empirically* at compile time by evaluating the actual performance of the code with a wide range of block sizes.

In a multi-level memory hierarchy system, it is difficult to predict the execution time ($T_{exe}$) of a program. But, $T_{exe}$ is proportional to the total miss cost of TLB and cache. In order to minimize $T_{exe}$, we will evaluate and minimize the total miss cost for both TLB and $l$-level caches. We have:

$$MC \quad = \quad TM \cdot M_{tlb} + \sum_{i=1}^{l} CM_i H_{i+1} \tag{3}$$

where $MC$ denotes the total miss cost, $CM_i$ is the number of misses in the $i^{th}$ level cache, $TM$ is the TLB miss penalty, $H_i$ is the cost of a hit in the $i^{th}$ level cache, and $M_{tlb}$ is the penalty of a TLB miss. The $(l+1)^{th}$ level cache is the main memory. It is assumed that all data reside in the main memory ($CM_{l+1} = 0$). Using the derivative of $MC$ with respect to the block size, we can find the optimal block size that minimizes the overall miss cost.




(a) Miss cost of TLB, L1, and L2 cache ($B_{tc1}$ is obtained using Eq.( 4) )

(b) Total miss cost with various L2 miss penalty

Figure 10: Miss cost estimation for 6-loop TMM (UltraSparc II parameters)

For a simple 2-level memory hierarchy that consists of only one level cache and TLB, the total miss cost (denoted as $MC_{tc1}$) in Eq. (3) reduces to:

$$MC_{tc1} = TM \cdot M_{tlb} + CM \cdot H_2,$$

where $H_2$ is the access cost of main memory. In the above estimation, $M_{tlb}$ and $CM$ are substituted with Eq.(1) and Eq.(2), respectively. Using the derivative of $MC$, the optimal block size ($B_{tc1}$) that minimizes the total miss cost caused by L1 cache and TLB misses is given as

$$B_{tc1} \approx \sqrt{\frac{\left(\frac{2L_{1c}M_{tlb}}{P_v} + \left[2 + \frac{3L_{c1}+2L_{c1}^2}{S_{c1}}\right] H_2\right) S_{c1}}{4H_2}}. \tag{4}$$

We now extend this analysis to determine a range for optimal block size in a multi-level memory hierarchy that consists of TLB and two levels of cache. The miss cost is classified into two groups: miss cost caused by TLB and L1 cache misses and miss cost caused by L2 misses. Figure 10 (a) and (b) show the miss cost estimated through Eqs.(1) and (2). Fig. 10(a) is the separated TLB, L1, and L2 miss cost, using UltraSparc II parameters. Fig. 10(b) shows the variance of the estimated total miss costs as the ratio between L1 cache miss penalty ($H_2$) and L2 cache miss penalty ($H_3$) varies. Using Eq.(4), we discuss the total miss cost for 3 ranges of block size:

**Lemma 3.2** *For $B < B_{tc1}$, $MC(B) > MC(B_{tc1})$.*

**Proof:** Using the derivatives of TLB and cache miss equations (Eq.( 1) and (2) ), it can be easily verified that $\frac{dMC_{tc1}}{dB} < 0$ and $\frac{dCM_2}{dB} < 0$ for $B < B_{tc1}$. This is shown in Figure 10(a). For $B < B_{tc1}$, TLB, L1, and L2 miss cost increase as block size decreases, thereby increasing the total miss cost. Therefore, the optimal block size cannot be in the range $B < B_{tc1}$. ⊙

**Lemma 3.3** *For $B > \sqrt{S_{c1}}$, $MC(B) > MC(\sqrt{S_{c1}})$.*

**Proof:** In the range $B > \sqrt{S_{c1}}$, TLB miss cost is optimized by tiling and block data layout. However, the change in TLB miss cost is negligible as the block size increases. Since block size is larger than L1 cache size, self-interferences occur in this range. The number of L1 cache misses drastically increases as shown in Figure 10(a). For $\sqrt{S_{c1}} \leq B < \sqrt{2S_{c1}}$, the ratio of derivatives of Eq.( 2) for L1 and L2 misses is as follows:

$$\left|\frac{H_2\frac{dCM_1}{dB}}{H_3\frac{dCM_2}{dB}}\right| = \frac{H_2}{H_3}\left|\frac{\frac{N^3}{L_{c1}}\left[\frac{4}{S_{c1}}+\frac{4Sc1}{B^3}-\frac{2}{B^2}\right]}{\frac{N^3}{L_{c2}}\left[\frac{4}{S_{c2}}-\left(2+\frac{3L_{c2}+2L_{c2}^2}{S_{c2}}\right)\frac{1}{B^2}\right]}\right|.$$

Let $B^2 = \alpha S_{c1}$ $(1 \leq \alpha < 2)$. Note that $L_{c2} \ll S_{c2}$.

$$\left|\frac{H_2\frac{dCM_1}{dB}}{H_3\frac{dCM_2}{dB}}\right| \approx \frac{H_2}{H_3}\cdot\frac{L_{c2}}{L_{c1}}\cdot\frac{S_{c2}}{S_{c2}-2\alpha S_{c1}}\cdot\left(2\alpha-1+\frac{2}{\sqrt{\alpha}}\sqrt{S_{c1}}\right) > \frac{H_2}{H_3}\cdot\frac{L_{c2}}{L_{c1}}\cdot\frac{S_{c2}}{S_{c2}-4S_{c1}}\cdot\left(3+\sqrt{2}\cdot\sqrt{S_{c1}}\right)$$

In a general memory hierarchy system, $\frac{S_{c2}}{S_{c2}-4S_{c1}} \approx 1$ since $S_{c1} \ll S_{c2}$. Also, $\frac{L_{c2}}{L_{c1}} \geq 1$ and $\sqrt{2S_{c1}} > \frac{H3}{H2}$. Therefore,

$$\left|\frac{H_2\frac{dCM_1}{dB}}{H_3\frac{dCM_2}{dB}}\right| > 1$$

Thus, although the number of L2 cache misses decreases ($\frac{dCM_2}{dB} < 0$), the total miss cost increases for $\sqrt{S_{c1}} \leq B < \sqrt{2S_{c1}}$ because the increase in L1 cache miss cost is larger than the decrease in L2 cache miss cost. For $B \geq \sqrt{2S_{c1}}$, there is no reuse in L1 cache. Thus, the L1 cache miss cost saturates. Figure 10(b) shows the change of the total miss cost as the ratio of $\frac{H_3}{H_2}$ varies. Even though L2 miss penalty is 40 times that of L1 miss penalty, $TM(B) > TM(\sqrt{S_{c1}})$ for $B \geq \sqrt{2S_{c1}}$, because L1 self-interference miss cost is dominantly large for $B \geq \sqrt{2S_{c1}}$. Therefore, the optimal block size cannot be in the range $B > \sqrt{S_{c1}}$. ⊙

**Theorem 3.2** *The optimal block size $B_{opt}$ satisfies $B_{tc1} \leq B_{opt} < \sqrt{S_{c1}}$.*

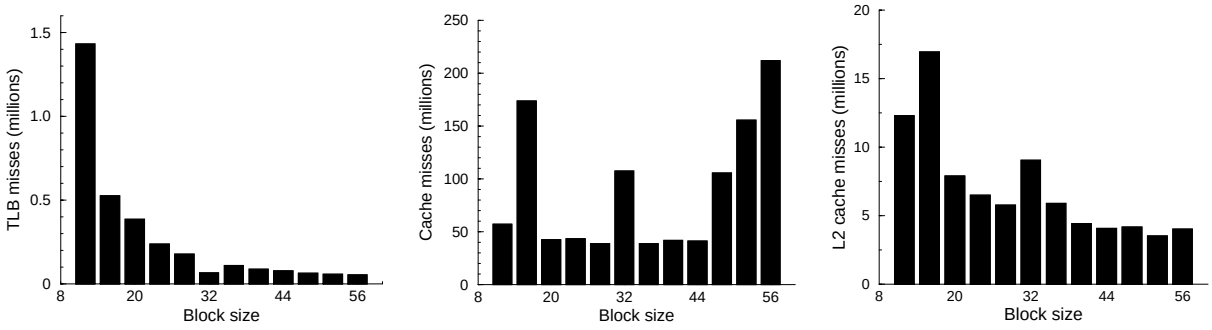
**Proof:** This follows from Lemma 3.2 and 3.3. Therefore, an optimal block size that minimizes the total miss cost is located in

$$B_{tc1} \leq B_{opt} < \sqrt{S_{c1}}. \tag{5}$$

We select a block size that is a multiple of $L_{c1}$ (L1 cache line size) in this range. ⊙

To verify our approach, we conducted simulations using UltraSparc II parameters (Table 3). Figure 11 shows the simulation results of 6-loop TMM using block data layout. As discussed, the number of TLB and L2 misses decreased as block size increases. Also, the minimum number of L1 misses was obtained for $B = 36$ and then drastically increased for $B > 45$. Figure 12 shows the total miss cost. For UltraSparc II, $B_{tc1} = 32.2$, $\sqrt{S_{c1}} = 45.3$, and $L_{c1} = 4$. Theorem 3.2 suggests the range for optimal block size is to be 36–44. Simulation results show that the optimal block size for this architecture was 44.

We also tested ATLAS on UltraSparc II. Through a wide search ranging from 16 to 44, ATLAS found 36 and 40 as the optimal block sizes. These blocks lie in the range given by Eq. (5). We further tested 6-loop TMM with respect to different problem and block sizes. For each problem




(a) TLB miss (b) L1 miss (c) L2 miss

Figure 11: Simulation results of 6-loop TMM

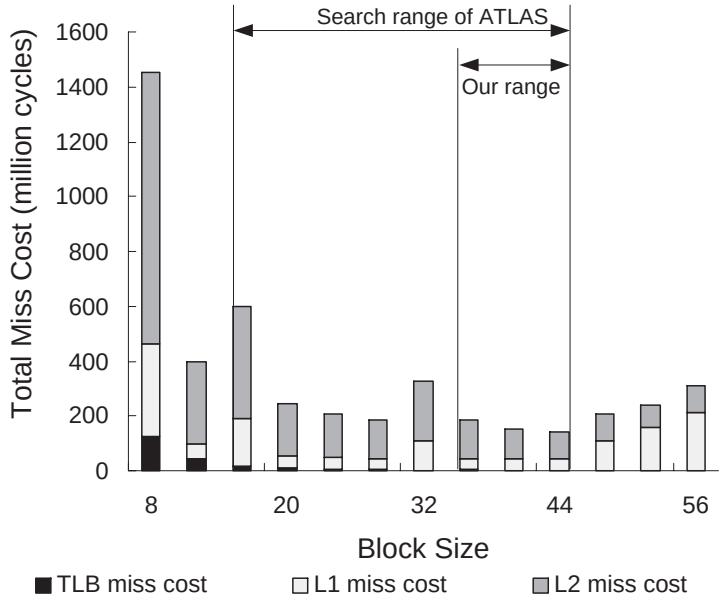



Figure 12: Total miss cost for 6-loop TMM

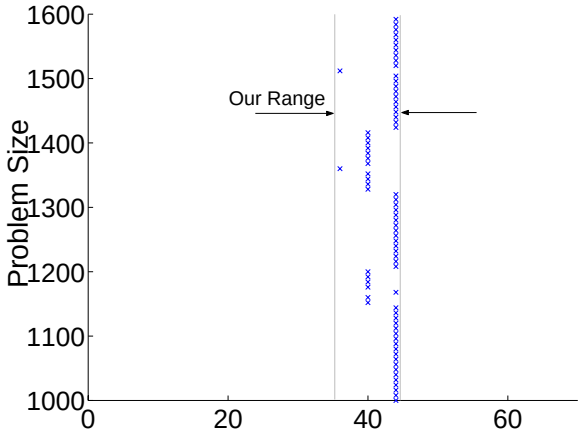



Figure 13: Optimal block sizes for 6-loop TMM

size, we performed experiments by testing block sizes ranging from 8–80. In these tests, we found that the optimal block size for each problem size was in the range given by Eq. (5) as shown in Figure 13. These experiments confirm that our approach proposes a reasonably good range for block size selection.

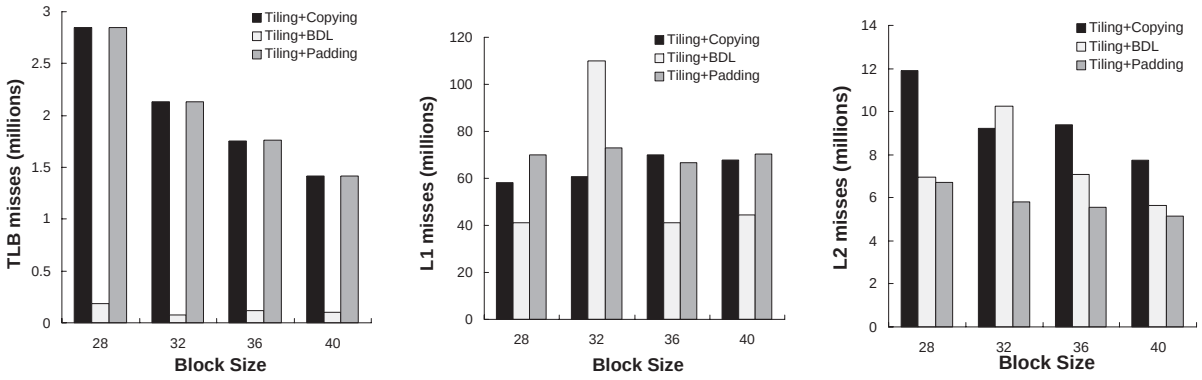
# 4 Experimental Results

To verify our analysis, we performed simulations and experiments on the following applications: tiled matrix multiplication(TMM), LU decomposition, and Cholesky factorization(CF). The performance of tiling with block data layout (tiling+BDL) is compared with other optimization techniques: tiling with copying(tiling+copying), and tiling with padding(tiling+padding). For tiling+BDL, the tile size (of the tiling technique) is chosen to be the same as the block size of the block data layout. Input and output is in canonical layout. All the cost in performing data layout transformations (from canonical layout to block data layout and vice versa) is included in the reported results. As stated in [12], we observed that the copying technique cannot be applied efficiently to LU and CF applications, since copying overhead offsets the performance improvement. Hence we do not consider tiling+copying for these applications. In all our simulations and experiments, the data elements are double-precision.

## 4.1 Simulations

To show the performance improvement of TLB and caches using tiling+BDL, simulations were performed using the SimpleScalar simulator [2]. The problem size was $1024 \times 1024$. Two sets of architecture parameters were used: UltraSparc II and Pentium III. The parameters are shown in Table 3.

Figure 14 compares the TMM simulations of different techniques, based on UltraSparc II parameters. Tiling+BDL has less L1 and L2 cache misses when compared with other techniques. Block size 32 leads to increased L1 and L2 cache misses for block data layout because of the




(a) TLB misses (b) L1 misses (c) L2 misses

Figure 14: Simulation results for TMM using UltraSparc II parameters

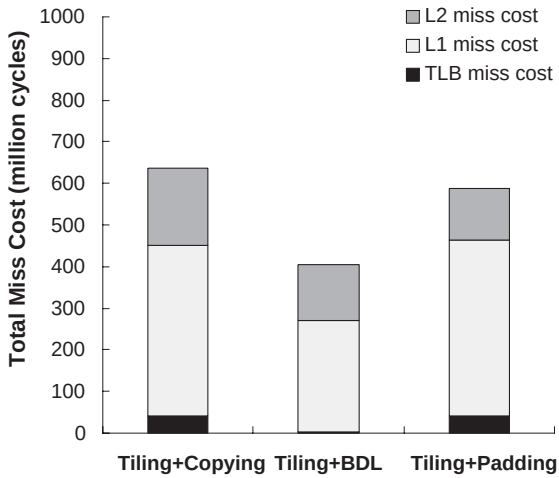
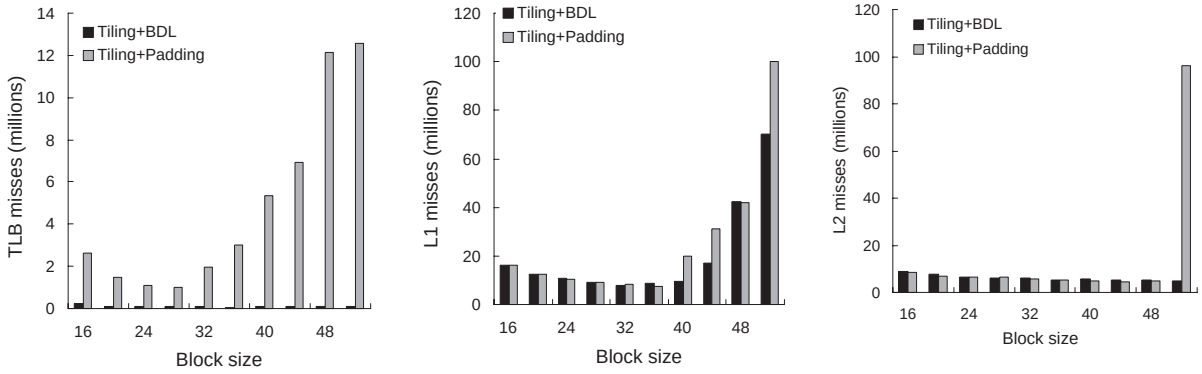
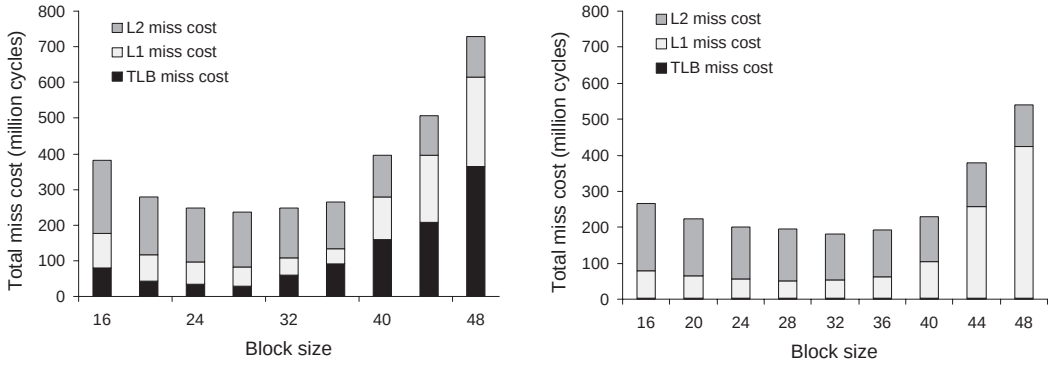



Figure 15: Total miss cost for TMM using UltraSparc II parameters




(a) TLB misses (b) L1 cache misses (c) L2 cache misses

Figure 16: Simulation results for LU using Pentium III parameters




(a) Tiling+Padding (b) Tiling+BDL

Figure 17: Effect of block size on LU decomposition using Pentium III parameters

cache conflicts between different blocks. Tiling+BDL reduced 91–96% of TLB misses as shown in Figure 14(a). This confirms our analysis presented in Section 3.1. Figure 15 shows the total miss cost (calculated from Eq. (3)) for TMM using block size $40 \times 40$. L1, L2, and TLB miss penalties were assumed to be 6, 24, and 30 cycles, respectively. This figure shows that tiling+BDL results in the smallest total miss cost and that the TLB miss cost with tiling+BDL is negligible compared with L1 and L2 miss costs. Figure 12 shows the effect of block size on the total miss cost for TMM using tiling+BDL. As discussed in Section 3.3, the best block size (44) is in the range 36–44 suggested by our approach.

Figure 16 presents simulation results for LU using Intel Pentium III parameters. Similar to TMM, the number of TLB misses for tiling+BDL was almost negligible compared with that for tiling+padding as shown in Figure 16(a). For both techniques, L1 and L2 cache misses were reduced considerably because of 4-way set-associativity. For tiling+padding, when the block size was larger than L1 cache size, the padding algorithm in [16] suggested a pad size of 0. There is essentially no

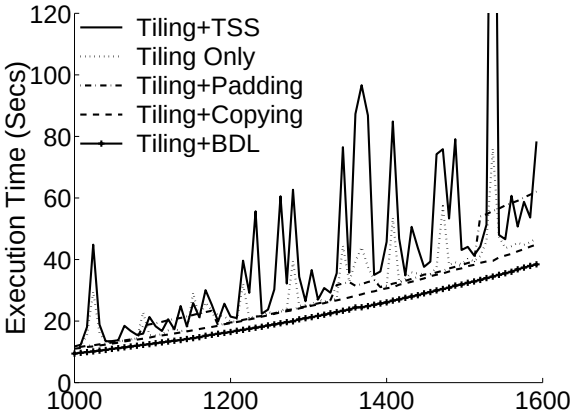



Figure 18: Execution time comparison of various techniques for TMM on Pentium III

padding effect, thereby drastically increasing L1 and L2 cache misses. Figure 17 shows the block size effect on total miss cost using tiling+padding and tiling+BDL. Tiling+padding reduced L1 and L2 cache miss costs considerably. However, TLB miss costs were still significantly high, affecting the overall performance. As discussed in Section 3.3, the suggested range for optimal block size is 32–44. Simulations validate that the optimal block size achieving the smallest miss cost locates in the range selected using our approach.

### 4.2 Execution on real platforms

To verify our block size selection and the performance improvements using block data layout, we performed experiments on several platforms as tabulated in Table 3. `gcc` compiler was used in these experiments. The compiler optimization flags were set to "`-fomit-frame-pointer -O3 -funroll-loops`". Execution time was the user processor time measured by sys-call `clock()`. All the data reported here is the average of 10 executions. The problem sizes ranged from $1000 \times 1000$ to $1600 \times 1600$.

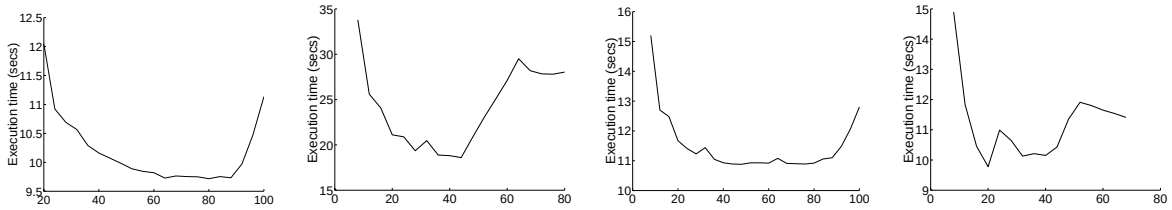
Figure 18 shows the comparison of execution time of tiling+BDL with other techniques. The performance of tiling+TSS (tile size selection algorithm [6]) shown in this figure selects block size based on GCD computation. Tiling solves the cache capacity miss problem but it cannot avoid

Table 3: Features of various experimental platforms

| Platforms | Speed (MHz) | L1 cache | | | L2 cache | | | TLB | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Size ($KB$) | Line ($Byte$) | Ass. | Size ($KB$) | Line ($Byte$) | Ass. | Entry | page ($KB$) | Ass. |
| Alpha 21264 | 500 | 64 | 64 | 2 | 4096 | 64 | 1 | 128 | 8 | 128 |
| UltraSparc II | 400 | 16 | 32 | 1 | 2048 | 64 | 1 | 64 | 8 | 64 |
| UltraSparc III | 750 | 64 | 32 | 4 | 4096 | 64 | 4 | 512 | 8 | 2 |
| Pentium III | 800 | 16 | 32 | 4 | 512 | 32 | 4 | 64 | 4 | 4 |

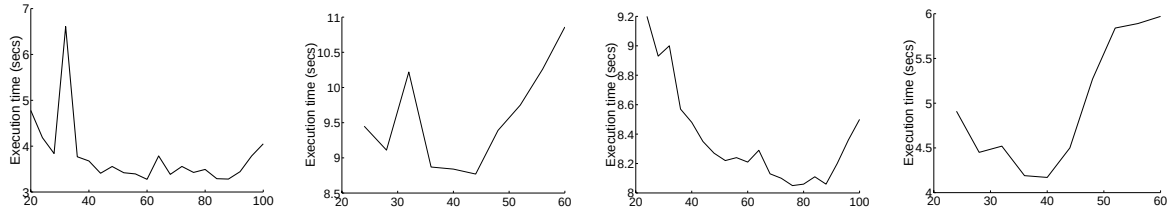
conflict misses. Conflict misses are strongly related to the problem size and block size. This makes tiling sensitive to problem size. As discussed on Section 3.2, block data layout greatly reduces conflict misses, resulting in smoother performance compared with others.

The effect of block size on tiling+BDL is shown in Figures 19–21. Various problem sizes were tested and results on all these problems showed similar trends as in Figures 19–21. As an illustration, the results for problem size of $1024 \times 1024$ are shown. As shown in Figures 19–21, the optimal block sizes for Pentium III, UltraSparc II, Sun UltraSparc III and Alpha 21264 are 40, 44, 76, and 76 respectively. All these numbers are in the range given by our block size selection algorithm. For example, the range for best block size on Alpha 21264 is 64–78. This confirmed that our block size selection algorithm proposes a reasonable range. As discussed earlier, block sizes 32 and 64 should be avoided (for use with block data layout) because the performance degrades due to conflict misses between blocks.



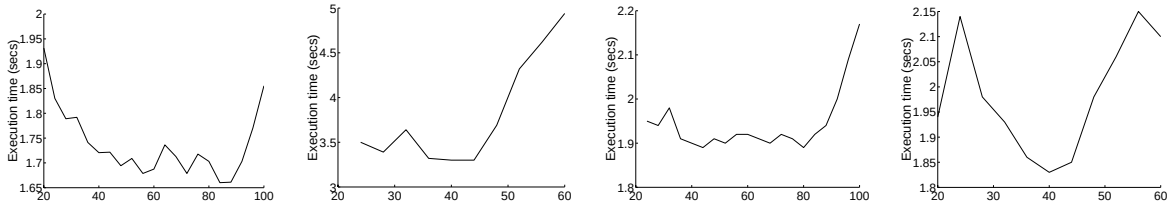

(a) Alpha 21264 (b) UltraSparc II (c) UltraSparc III (d) Pentium III

Figure 19: Effect of block size on TMM




(a) Alpha 21264 (b) UltraSparc II (c) UltraSparc III (d) Pentium III

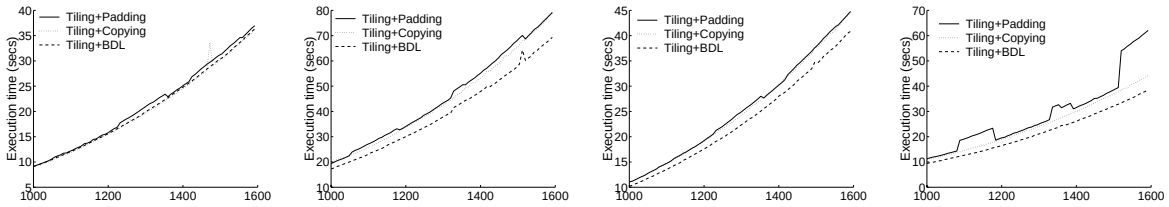
Figure 20: Effect of block size on LU decomposition




(a) Alpha 21264 (b) UltraSparc II (c) UltraSparc III (d) Pentium III

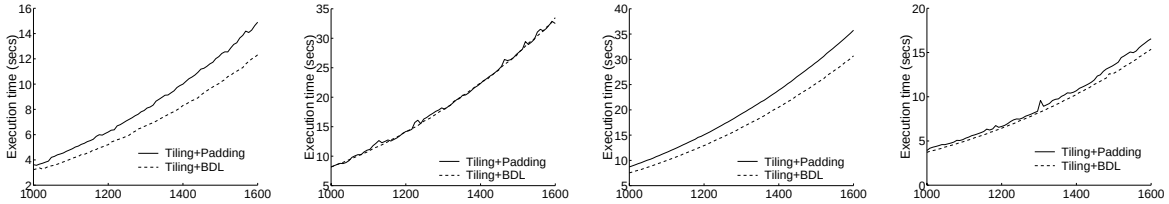
Figure 21: Effect of block size on Cholesky factorization

Figures 22–24 show the execution time comparison of tiling+BDL with tiling+copying and tiling+padding. In these figures, block size for tiling+BDL was given by our algorithm discussed in Section 3.3. The tile size for the copying technique was given by the approach in [12]. The pad size was selected by the algorithm discussed in [16]. Tiling+BDL technique is faster than using other optimization techniques, for almost all problem sizes and on all the platforms.



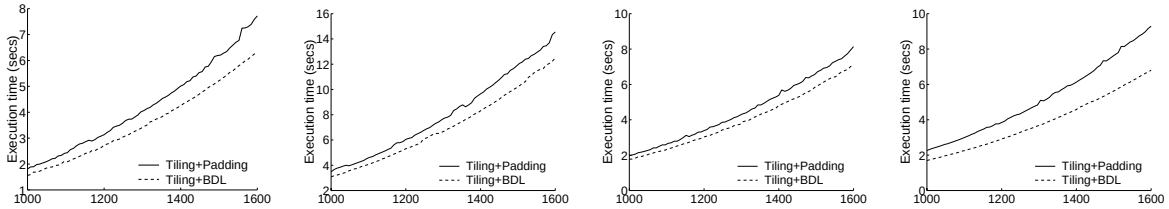

(a) Alpha 21264 (b) UltraSparc II (c) UltraSparc III (d) Pentium III

Figure 22: Execution time of TMM




(a) Alpha 21264 (b) UltraSparc II (c) UltraSparc III (d) Pentium III

Figure 23: Execution time of LU decomposition




(a) Alpha 21264 (b) UltraSparc II (c) UltraSparc III (d) Pentium III

Figure 24: Execution time of Cholesky factorization

### 4.3 Block data layout and Morton data layout

Recently nonlinear data layouts have been considered to improve memory hierarchy performance. One such layout is the Morton data layout(MDL) as defined in Section 2.1. Similar to block data layout, elements within each block are mapped onto contiguous memory locations. However, Morton data layout uses a different order to map blocks as shown in Figure 1. This order matches the

Table 4: Comparison of execution time of TMM on various platforms: All times are in seconds.

(a) Pentium III

| Size | iterative+BDL | recursive+MDL |
|---|---|---|
| 1024 | 10.37 | 10.98 |
| 1280 | 20.43 | 20.64 |
| 1408 | 27.06 | 28.21 |
| 1600 | 39.77 | 43.78 |
| 2048 | 83.27 | 87.64 |

(b) UltraSparc II

| Size | iterative+BDL | recursive+MDL |
|---|---|---|
| 1024 | 18.87 | 21.80 |
| 1280 | 36.17 | 40.63 |
| 1408 | 48.76 | 53.70 |
| 1600 | 70.44 | 81.61 |
| 2048 | 149.65 | 170.86 |

Table 5: Comparison of execution time of LU decomposition on various platforms: All times are in seconds.

(a) Pentium III

| Size | iterative+BDL | recursive+MDL |
|---|---|---|
| 1024 | 4.15 | 4.43 |
| 1280 | 8.10 | 8.10 |
| 1408 | 10.85 | 11.57 |
| 1600 | 15.85 | 18.44 |
| 2048 | 33.58 | 35.90 |

(b) UltraSparc II

| Size | iterative+BDL | recursive+MDL |
|---|---|---|
| 1024 | 8.77 | 9.94 |
| 1280 | 18.97 | 18.54 |
| 1408 | 22.76 | 22.45 |
| 1600 | 33.51 | 35.58 |
| 2048 | 75.30 | 81.66 |

access pattern of recursive algorithms. In this section, we compare the performance of recursive algorithms using MDL (recursive+MDL) with iterative tiled algorithms using BDL (iterative+BDL), for matrix multiplication and LU decomposition. We show that the performance of recursive+MDL is comparable with that of iterative+BDL if the block size of MDL lies in the optimal block size range for BDL as given by our algorithm (Eq. (5) in Section 3.3). However, if the block size of MDL is outside this range, recursive+MDL is slower than iterative+BDL.

Similar to block data layout, block size for Morton layout also plays an important role in the performance. However, due to recursion, the choice of block sizes is limited. For an $N \times N$ matrix, if the depth of recursion is $d$, the block size of MDL is given by $B_{MDL} = \frac{N}{2^d}$. Such a block size can lie outside the optimal range given by our approach. Our experiment results show that this degrades the overall performance.

Experiments using TMM and LU were performed on UltraSparc II and Pentium III. Table 4 shows the execution time comparison of MM using iterative+BDL with recursive+MDL. For iterative+BDL, we selected the block size according to the algorithm discussed in Section 3.3. For recursive+MDL, we tested various recursion depths (resulting in various basic block sizes) and used the best for comparison. For problem size $1280 \times 1280$ and $1408 \times 1408$, optimal block sizes for recursive+MDL were 40 and 44 respectively, which were in the range given by our algorithm, 36–44. Both the layouts showed competitive performance for these cases. For problem size $1600 \times 1600$, recursive+MDL was up to 15.8% slower than iterative+BDL. Among possible choices of 25, 50, and 100, the performance of recursive+MDL was optimized at block size 25, where $25 = \frac{1600}{2^5}$. This is because it is outside the optimal range specified by our algorithm. Table 5 shows the execution time comparison of tiled LU decomposition using BDL and recursive LU decomposition [27] using MDL. These results confirm our analysis.

# 5 Concluding Remarks

This paper studied a critical problem in understanding the performance of algorithms on state-of-the-art machines that employ multi-level memory hierarchy. We showed that using block data layout, TLB misses as well as cache misses are reduced considerably. Further, we proposed a tight range for block size using our performance analysis. Our analysis matches closely with simulation based as well as experimental results.

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.


# Acknowledgment

We would like to thank Shriram Bhargava Gundala for careful reading of drafts of this work. We also would like to thank Sriram Vajapeyam and Cauligi S. Raghavendra for their inputs on a preliminary version of this work.


# Appendix A TLB performance of block data layout

This section gives a detailed proof of Theorem 2.2 in Section 2.2. The theorem is repeated for convenience:

**Theorem** *For accessing an array along all the rows and then along all the columns, block data layout with block size $\sqrt{P_v} \times \sqrt{P_v}$ minimizes the number of TLB misses.*

**Proof:** Suppose the block size $B^2 = kP_v$. Two cases (for $k \geq 1$ and $k \leq 1$) are discussed separately.

**Case I:** $k \geq 1$. We consider three scenarios for this case.

1. $\frac{N}{B} > S_{tlb}$
   Accesses to the first row cause $\frac{N}{B}$ TLB misses. However, these entries cannot be reused since $S_{tlb}$ is small. Therefore, TLB misses caused by row accesses is $T_{row} = \frac{N}{B} \times N$. Similarly, TLB misses caused by column accesses are $T_{col} = \frac{N}{B} \cdot k \cdot N$. Therefore, the total number of TLB misses is
   $$T_{miss} = \frac{N^2}{B} + k\frac{N^2}{B} = \frac{N^2}{\sqrt{P_v}}(\frac{1}{\sqrt{k}} + \sqrt{k}).$$
   To minimize the total TLB misses,
   $$\frac{dT_{miss}}{dk} = \frac{N^2}{\sqrt{2P_v}} \times \frac{1}{\sqrt{k}} \times (1 - \frac{1}{k}).$$
   Therefore, as $k$ decreases, the total number of TLB misses decreases. The total number of TLB misses is minimized when $k = 1$. Note that when $B = \sqrt{P_v}$ $(k = 1)$, the number of TLB misses is $2\frac{N^2}{\sqrt{P_v}}$, which is the lower bound given by Theorem 2.1.

2. $\frac{N}{B} \leq \frac{S_{tlb}}{k}$
   In this scenario, both column and row access can reuse TLB entries. Therefore, the total number of TLB misses is
   $$T_{miss} = 2k\frac{N^2}{B^2} = 2\frac{N^2}{P_v}.$$
   This is equal to twice the number of TLB misses caused by all row accesses in canonical layout. Therefore, this will be the minimum number of TLB misses for such an access pattern.

3. $\frac{S_{tlb}}{k} < \frac{N}{B} \leq S_{tlb}$
   In this scenario, only row accesses can reuse TLB entries accessed in the previous row accesses. TLB misses for row accesses are $T_{row} = k\frac{N^2}{B^2}$. Therefore, the total number of TLB misses is
   $$T_{miss} = k\frac{N^2}{B^2} + k\frac{N^2}{B} = \frac{N^2}{P_v} + \frac{N^2}{\sqrt{P_v}}\sqrt{k}.$$
   As $k$ decreases, TLB misses also decrease. The number of TLB misses for block data layout is minimized when $k$ approaches 1. Note that this scenario will reduce to scenario 2 when $k = 1$. Therefore, the minimum number of TLB misses in this scenario is the same as that in scenario 2.

**Case II:** $k \leq 1$. Three scenarios are discussed as follows:

1. $\frac{N}{B} > \frac{S_{tlb}}{k}$
   The first row access causes $k\frac{N}{B}$ TLB misses. These entries cannot be reused in the next row access. TLB misses caused by row accesses are $T_{row} = k\frac{N}{B} \cdot N$. On the other hand, the first column access causes $\frac{N}{B}$ TLB misses, since all the elements in each block are stored in one page. The TLB misses caused by column accesses is $T_{col} = \frac{N}{B} \cdot N$. Therefore, the total number of TLB misses is
   $$T_{miss} = k\frac{N^2}{B} + \frac{N^2}{B} = \frac{N^2}{\sqrt{P_v}}(\frac{1}{\sqrt{k}} + \sqrt{k}).$$
   To minimize the total TLB misses,
   $$\frac{dT_{miss}}{dk} = \frac{N^2}{\sqrt{2P_v}} \times \frac{1}{\sqrt{k}} \times (1 - \frac{1}{k}).$$
   Therefore, as $k$ increases, the total number of TLB misses decreases. The total number of TLB misses is minimized when $k = 1, B = \sqrt{P_v}$. Again, the minimum number is $2\frac{N^2}{\sqrt{P_v}}$, equal to the lower bound given by Theorem 2.1.

2. $\frac{N}{B} \leq S_{tlb}$
   In this scenario, both row and column access can reuse TLB entries. Therefore, the total number of TLB misses is
   $$T_{miss} = 2k\frac{N^2}{B^2} = 2\frac{N^2}{P_v}.$$
   This is equal to twice the number of TLB misses caused by all row accesses in the canonical layout. Therefore, it is the minimal number of TLB misses caused by all row ($1^{st}$ pass) and then all column ($2^{nd}$ pass) accesses.

3. $S_{tlb} < \frac{N}{B} \leq \frac{S_{tlb}}{k}$
   In this scenario, only row accesses can reuse TLB entries accessed in the previous row accesses. TLB misses of row accesses is denoted as: $T_{row} = k\frac{N^2}{B^2}$. Therefore, the total number of TLB misses is
   $$T_{miss} = k\frac{N^2}{B^2} + \frac{N^2}{B} = \frac{N^2}{P_v} + \frac{N^2}{\sqrt{kP_v}}.$$
   As $k$ increases, TLB misses decrease. Like scenario 3 in Case I, the minimum number of TLB misses in this scenario is obtained when $k = 1$, and this number is the same as that in the previous scenario.

According to the above analysis, block data layout with block size $\sqrt{P_v} \times \sqrt{P_v}$ minimizes the total number of TLB misses. As the problem size ($N$) increases, this minimum number asymptotically approaches the lower bound given by Theorem 2.1.

⊙

# Appendix B Cache Miss Analysis

In this section, we provide detailed cache miss analysis for a tiled access pattern with block data layout. Individual levels of cache are not considered explicitly, as this analysis is applicable to all cache levels. We consider a tiled program that consists of nested loops. Each loop level is denoted by the loop index $i$, $j$, $l$, etc. Arrays referenced by the program are denoted as $u$, $v$, etc. Within an iteration of a loop $l$, a portion of an array $v$ (called the footprint $F_p(v)$) is referenced. The body of the loop $l$ will be executed $R(v)$ times, where $R(v)$ is the reuse factor.

Let (i) $ICM_l(v)$ denote the number of *intrinsic* cache misses [12] caused by accessing array $v$ during the first iteration of loop $l$; (ii) $SCM_l(v)$ denote the number of self-interference misses when array $v$ is accessed in one iteration of loop $l$; (iii) $CIM(v)$ denote the number of cross-interference misses between array $v$ and other arrays for an iteration of loop $l$. The number of cache misses caused by array $v$ for one iteration of loop $l$ is thus:

$$CM_l(v) = ICM_l(v) - SCM_l(v) + R(v) \times \{SCM_l(v) + CIM(v)\} \tag{6}$$

$CIM(v)$ in the above equation can be calculated as:

$$CIM(v) = ICM_l(v) \times PrCF(v), \tag{7}$$

where $PrCF(v)$ denotes the probability of conflict between one element of array $v$ and elements of other arrays for loop $l$. It is given by

$$PrCF(v) = \sum_{u \neq v} Prov_{cf}(v, u),$$

where $Prov_{cf}(v, u)$ is the probability that an element of array $v$ falls into the footprint of the array $u$, accessed with a stride ($s_u$) in the cache. For simplicity, it is assumed that an element of array $v$ does not conflict with elements in two or more arrays at the same time.

The cache misses of array $v$ is computed as follows:

$$CM(v) = NIO(l) \times CM_l(v), \tag{8}$$

where $NIO(l)$ denotes the total number of iterations of outer loops. The total number of misses incurred by accessing all arrays is the sum of misses incurred in accessing individual arrays ($\sum_i CM(i)$). The above cache miss equation (Eq.( 8)) is applicable to any data layout with nested loops. But the factors ( $SCM_l(v)$, $Prov_{cf}(v,u)$, $R(v)$, etc.) cannot be quantified unless the data layout and loop structure are known.

For block data layout, we can easily quantify $SCM_l(v)$ and $Prov_{cf}(v,u)$ in the above equations. The number of self-interferences can be derived by considering three ranges of block sizes. (i) When the block size is less than the cache size, there is no self-interference. (ii) When the block size is larger than twice the cache size, there is no reused element in cache, resulting in $\frac{B^2}{L_c}$ self-interferences misses. (iii) When the block size is in between the above ranges, $\frac{2(B^2-S_c)}{L_c}$ self-interference misses occur. Hence,

$$SCM_l(v) = \begin{cases} 0 & \text{for } B < \sqrt{S_c} \\ \frac{2(B^2-S_c)}{L_c} & \text{for } \sqrt{S_c} \le B < \sqrt{2S_c} \\ \frac{B^2}{L_c} & \text{for } \sqrt{2S_c} \le B \end{cases}$$

For loop $l$, $F_p(v)$ elements of array $v$ are accessed with a stride ($s_v$). The average number of cache lines occupied by $F_p(v)$ elements is

$$NCL(v) = \begin{cases} \frac{F_p(v)s_v}{L_c} + 1 & \text{if } 1 < s_v < L_c \\ F_p(v) & \text{otherwise} \end{cases}$$

During a tiled computation, a block of array $v$ is accessed in loop $l$. Hence, $ICM_l(v)$ is equal to the number of cache lines, $NCL(v)$. For loop $l$, array $u$ is accessed with stride($s_u$) whose footprint size is $F_p(u)$. It occupies $NCL(u)$ cache lines in the cache. The probability of conflicting with array $u$ is

$$Prov_{cf}(v,u) = \frac{NCL(u)}{S_c/L_c}.$$

Therefore, the cache misses of array $v$ on block data layout is

$$CM(v) = NIO(l) \times \left\{ NCL(v) - SCM_l(v) + R(v) \times \left( SCM_l(v) + NCL(v) \times \sum_{u \neq v} \frac{NCL(u)L_c}{S_c} \right) \right\}. \tag{9}$$

Consider the 6-loop TMM shown in Figure 2(b). The reuse factors and footprint sizes of arrays $X$, $Y$ and $Z$ can be determined. The values are shown in Table 6. For example, consider an array $Y$ in loop $i$. $B^2$ elements of $Y$ are referenced in each iteration of loop $i$. These $B^2$ elements are reused $N$ times. $NIO(l)$ can be obtained directly from the code (Figure 2(b)). For example, $NIO(i) = N^3/B^3$. According to Eq.(9), the number of cache misses for $\mathbf{Y}$ and $\mathbf{Z}$ are as follows:

$$CM(\mathbf{Y}) \approx \begin{cases} \frac{N^3}{L_c}\left\{\frac{1}{N} + \left(1+\frac{L_c}{B^2}\right)\frac{3(B+L_c)}{S_c}\right\} & \text{for } B < \sqrt{S_c} \\ \frac{N^3}{L_c}\left\{\frac{2S_c}{B^2}\left(\frac{1}{N}-1\right) + 2 - \frac{1}{N} + \left(1+\frac{L_c}{B^2}\right)\frac{3(B+L_c)}{S_c}\right\} & \text{for } \sqrt{S_c} \le B < \sqrt{2S_c} \\ \frac{N^3}{L_c} & \text{for } \sqrt{2S_c} \le B \end{cases}$$

$$CM(\mathbf{Z}) \approx \frac{N^3}{L_c}\left\{\frac{1}{B} + \left(1+\frac{L}{B}\right)\frac{(B+2L)}{S_c}\right\}$$

Table 6: Parameters of TMM

| Array | Reuse Factor | | | Footprint | | |
|---|---|---|---|---|---|---|
| | $i$ | $k$ | $j$ | $i$ | $k$ | $j$ |
| $\mathbf{X}(i,k)$ | | | $B$ | $B$ | 1 | |
| $\mathbf{Y}(k,j)$ | $N$ | | | $B^2$ | $B$ | |
| $\mathbf{Z}(i,j)$ | | $B$ | | $B$ | $B$ | |

In the 6-loop TMM, each element of array $\mathbf{X}$ is immediately allocated to a register. So, its probability of conflicts with other arrays is 0. Thus, the number of cache misses for array $\mathbf{X}$ is

$$CM(\mathbf{X}) = \frac{N^3}{BL_c}.$$

The total number of cache misses for the 6-loop TMM with block data layout is thus:

$$CM = \sum_v CM(v) = CM(\mathbf{X}) + CM(\mathbf{Y}) + CM(\mathbf{Z}) \tag{10}$$

$$\approx \begin{cases} \frac{N^3}{L_c}\left\{\frac{1}{B}\left(2+\frac{(3L_c+2L_c^2)}{S_c}\right)+\frac{1}{N}+\frac{4B+6L_c}{S_c}\right\} & \text{for } B < \sqrt{S_c} \\ \frac{N^3}{L_c}\left\{\frac{4B}{S_c}+\frac{2}{B}-\frac{2S_c}{B^2}+2-\frac{1}{N}+\frac{6L_c}{S_c}\right\} & \text{for } \sqrt{S_c} \le B < \sqrt{2S_c} \\ \frac{N^3}{L_c}\left\{1+\frac{2}{B}+\left(1+\frac{L_c}{B}\right)\left(\frac{B+2L_c}{S_c}\right)\right\} & \text{for } \sqrt{2S_c} \le B \end{cases} \tag{11}$$

The above analysis focuses on the access pattern of 6-loop TMM. Because matrix multiplication is the kernel of many linear algebra computations, the analysis can be generalized or directly applied to other linear algebra applications.

## References


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

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

[3] J. B. Carter, W. C. Hsieh, L. B. Stoller, M. R. Swanson, L. Zhang, E. L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. *Proceedings of the Fifth International Symposium on High Performance Computer Architecture (HPCA-5)*, pages 70–79, January 1999.

[4] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. *Proceedings of the 13th ACM International Conference on Supercomputing (ICS '99)*, June 1999.

[5] M. Cierniak and W. Li. Unifying Data and Control Transformations for Distributed Shared-Memory Machines. *Proceedings of the SCM SIGPLAN'95 Conference on Programming Language Design and Implementsion*, pages 205–217, June 1995.

[6] S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[7] R. Espasa, J. Corbal, and M. Valero. Command Vector Memory Systems: High Performance at Low Cost. Technical Report UPC-DAC-1998-8, Universitat Polit'ecnica de Catalunya, 1998.

[8] K. Esseghir. Improving data locality for caches. Master's thesis, Dept. of Computer Scienece, Rice University, September 1993.

[9] A. González, C. Aliagas, and M. Valero. A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. *Proc. International Conference on Supercomputing*, pages 338–347, July 1995.

[10] T. L. Johnson, M. C. Merten, and W. W. Hwu. Run-time Spatial Locality Detection and Optimization. *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[11] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving Locality Using Loop and Data Transformations in an Integrated Framework. *Proceedings of the 31st IEEE/ACM International Symposium on Microarchitecture*, November 1998.

[12] M. Lam, E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, April 1991.

[13] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the Multi-Level Nature of Tiling Interactions. *International Journal of Parallel Programming*, 1998.

[14] D. Padua. The Fortran I Compiler. *IEEE Computing in Science & Engineering*, January/Febrary 2000.

[15] D. A. Padua. Outline of a Roadmap for Compiler Technology. *IEEE Computing in Science & Engineering*, Fall 1996.

[16] P. R. Panda, H. Nakamura, N. Dutt, and A. Nicolau. Augmenting Loop Tiling with Data Alignment for Improved Cache Performance. *IEEE Transactions on Computers*, 48(2), Feburary 1999.

[17] N. Park, D. Kang, K. Bondalapati, and V. K. Prasanna. Dynamic Data Layouts for Cache-conscious Factorization of DFT. *Proceedings of International Parallel and Distributed Processing Symposium 2000 (IPDPS 2000)*, April 2000.

[18] N. Park and V. K. Prasanna. Cache Conscious Walsh-Hadamard Transform. *International Conference on Acoustics, Speech, and Signal Processing 2001 (ICASSP 2001)*, May 2001.

[19] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, April 1997.

[20] G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, June 1998.

[21] G. Rivera and C.-W. Tseng. Locality Optimizations for Multi-Level Caches. *Proceedings of IEEE Supercomputing'99(SC'99)*, November 1999.

[22] V. Sarkar and G. R. Gao. Optimization of Array Accesses by Collective Loop Transformations. *the Proceedings of the 1991 International Conference of Supercomputing*, June 1991.

[23] A. Saulsbury, F. Dahgren, and P. Stenström. Receny-based TLB Preloading. *The 27th Annual International Symposium on Computer Architecture(ISCA)*, June 2000.

[24] H. Sharangpani. Intel Itanium Processor Microarchitecture Overview. *Microprocessor Forum*, October 1999.

[25] O. Temam, E. D. Granston, and W. Jalby. To Copy or Not to Copy: A Comile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. *Proceedings of IEEE Supercomputing'93(SC'93)*, November 1993.

[26] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). *Proceedings of SC'98*, November 1998.

[27] Q. Yi, V. Adve, and K. Kennedy. Transforming Loops to Recursion for Multi-Level Memory Hierarchies. *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2000)*, June 2000.

# ADVISOR
## Algorithms for Data Intensive Applications on Intelligent and Smart Memories

**Results Submission**
**June 14, 2002**

During the course of this project we have focused on techniques for improving cache performance on traditional, cache-based processors. The majority of our work was done for the transitive closure stressmark, although we have also shown that these techniques can be used on a large class of algorithms. Results along this line are shown in the referenced conference papers.

Significant contributions include the Unidirectional Space Time Representation (USTR) and a novel recursive implementation of the Floyd-Warshall algorithm. Using the USTR it is possible to quickly generate cache-friendly implementations of a large class of algorithms. While recursion has been used to generate cache-friendly implementations of other algorithms, due to the non-trivial data dependences in the Floyd-Warshall algorithm, no recursive implementation of this algorithm has been shown at this time.

Also as part of this project we designed and implemented a simulator for Processing in Memory architectures. Processing-In-Memory (PIM) systems propose to solve the processor-memory gap by achieving tremendous processor-memory bandwidth by combining processors and memory together on the same chip substrate. Notre Dame, USC ISI, Berkeley, IBM, and others are developing PIM systems and have presented papers demonstrating the performance and optimization of several benchmarks on their architectures. While excellent for design verification, the proprietary nature and the time required to run their simulators are the biggest detractors of their tools for application optimization. A cycle-accurate, architecture specific simulator, requiring several hours to run, is not suitable for iterative development or experiments on novel ideas. We provide a simulator that will allow faster development cycles and a better understanding of how an application will port to other PIM architectures. A brief description of the simulator and some sample results are shown in Section 7.

### 1. Architecture Description

We use four different architectures for our experiments. The Pentium III Xeon running Windows 2000 is a 700 MHz, 4 processor shared memory machine with 4 GB of main memory. Each processor has 32 KB of level-1 data cache and 1 MB of level-2 cache on-chip. The level-1 cache is 4-way set associative with 32 B lines and the level-2 cache is 8-way set associative with 32 B lines.

The UltraSPARC III machine is a 750 MHz SUN Blade 1000 shared memory machine running Solaris 8. It has 2 processors and 1 GB of main memory. Each processor has 64 KB of level-1 data cache and 8 MB of level-2 cache. The level-1 cache is 4-way set associative with 32 B lines and the level-2 cache is direct mapped with 64 B lines.

The MIPS machine is a 300 MHz R12000, 64 processor, shared memory machine with 16 GB of main memory. Each processor has 32 KB of level-1 data cache and 8 MB of level-2

cache. The level-1 cache is 2-way set associative with 32 B lines and the level-2 cache is direct mapped with 64 B lines.

The Alpha 21264 is a 500 MHz uniprocessor machine with 512 MB of main memory. It has 64 KB of level-1 data cache and 4 MB of level-2 cache. The level-1 cache is 2-way set associative with 64 B lines and the level-2 cache is direct mapped with 64 B lines. It also has an 8 element fully-associative victim cache. All experiments are run on a uniprocessor or on a single node of a multiprocessor system.

## 2. Optimized Implementations

In this section, we give a description of the optimizations performed for each implementation. Also included is a description of the baseline implementations. Pseudo code is included where it benefited the description of the implementations. More details regarding the implementations can be found in [3] and [4].

### 2.1. Normal Floyd-Warshall – Baseline

A straightforward implementation of the Floyd-Warshall algorithm similar to the code given in the Stressmark specification was compiled using all optimizations available in the GNU C++ (gcc) compiler and the Microsoft Visual C++ compiler. The execution time of the kernel was collected and used as the baseline for the optimized implementations of the Floyd-Warshall algorithm. This same compilation and execution time collection was used for all implementations.

### 2.2. Floyd-Warshall with Tiling and Copying – Baseline

Tiling with copying is a standard cache-friendly optimization that can be performed using current research compilers. Because of this, we applied tiling with copying to the Floyd-Warshall algorithm. Due to data dependences current research compilers can only tile the inner two loops. This tiling was performed and the results also considered a baseline optimization.

### 2.3. Floyd-Warshall with Tiling and the Block Data Layout

In order to avoid the overhead of copying, the Block Data Layout (BDL) was used for the adjacency matrix. The BDL is a known layout that places a tile of data in contiguous locations instead of a row. As in the tiling with copying optimization, only the inner two loops were tiled due to data dependences. Since this is also a known technique, it was also considered a baseline optimization.

### 2.4. Simple USTR Floyd-Warshall

In [3], we developed the Unidirectional Space Time Representation (USTR) and showed that it can be used to generate cache friendly implementations of a large class of algorithms. As it is very similar to a systolic array representation, as a first approach we used a systolic array implementation of the Floyd-Warshall algorithm. Pseudo code for the simple USTR implementation is given in Figure 1. In [3], we compared the results of this optimization with the results from the previously mentioned baselines.

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

1.  for l <- 1 to 3
2.      for bi <- 1 to N/b
3.          for bj <- 1 to N/b
4.              load BxB elements of C at (bi,bj)
5.              for bk <- 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 <- bi to bi + (b-1)
9.                      for j <- bj to bj + (b-1)
10.                         for k <- bk to bk + (b-1)
11.                             C(i,j) <- min(C(i,j), A(i,k) + A(k,j))
12. return C
```

**Figure 1: Pseudo code for the cache-friendly implementation of the Floyd-Warshall algorithm. *A* is the input matrices, *C* is the output matrix, *N* is the dimension of all matrices, and *b* is the tile size.**

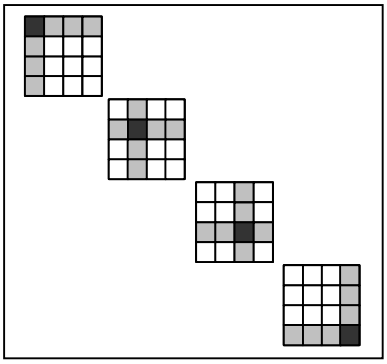


**Figure 2: Tiled implementation of FW**

### 2.5. Optimized USTR Floyd-Warshall

In [4], we discuss a tiled implementation of the Floyd-Warshall algorithm, which can also be shown to fit in the USTR. This we refer to it here as an optimized USTR implementation. In order to eliminate the three passes present in the simple USTR implementation, we reorder the computation of tiles in the following fashion (see Figure 2). We first compute the $(k,k)^{th}$ tile (the darkest tiles shown in Figure 2), the the $k^{th}$ row and column of tiles (the grey tiles), and finally the remainder of the matrix (the white tiles).

### 2.6. Recursive Floyd-Warshall

In [4], we also discuss a novel recursive implementation of the Floyd-Warshall algorithm. The recursive implementation represents a cache oblivious implementation of the Floyd-Warshall algorithm and achieves improved cache performance at each level of the memory hierarchy. Pseudo code is given in Figure 3. See [4] for more details regarding the recursive implementation include a proof of correctness and of optimality with respect to processor-memory traffic.

```
Floyd-Warshall-Recursive(A, B, C)

1.  if (not base case) {
2.      A11 ← FWR(A11, B11, C11);
3.      A12 ← FWR(A12, B11, C12);
4.      A21 ← FWR(A21, B21, C11);
5.      A22 ← FWR(A22, B21, C12);
6.      A22 ← FWR(A22, B22, C22);
7.      A21 ← FWR(A21, B22, C21);
8.      A12 ← FWR(A12, B12, C22);
9.      A11 ← FWR(A11, B12, C21);
10. }
11. else {
12.     /* run base case */
13. }
14. return A
```

**Figure 3: Pseudo code for the recursive version of the Floyd-Warshall algorithm**

### 2.7. Basic Dijkstra's – Baseline

We also examined optimizing Dijkstra's algorithm for the all pairs shortest path problem. For the baseline we again used the best compiler optimizations available to optimize a straightforward code. We use a binary heap to implement the priority queue and store the graph as an adjacency list.

### 2.8. Cache-Friendly Dijkstra's

In order to match the data access pattern of Dijkstra's algorithm to the data layout, we replaced the adjacency list with the adjacency matrix and replaced the binary heap with an array and used a linear search to find the minimum value. In this way we are able to take advantage of data reuse at the cache line level and simplify prefetching.

## 3. Implementation Documentation

It is a well-known fact that the speed of modern processors is increasing at a rate of roughly 60% per year while the speed of memory is increasing at a rate of roughly 7% per year. This difference is often referred to as the processor-memory gap, and it causes the latency to memory as seen by the processor to increase significantly with each passing year. In order to hide this increasing latency, caches have been designed to take advantage of locality of reference; the fact that once an element is accessed there is a good chance that it and/or elements near will be accessed in the near future. The cache is much smaller than main memory and is placed much closer to the processor in terms of latency. Modern processors are including more levels of cache, each level larger in size and farther from the processor in terms of latency.

Invariably the processor will access data that is not in the cache and this will result in a cache miss. Cache misses can be categorized into one of three categories: cold misses, capacity misses, and conflict misses. A cold miss occurs the first time a data element is accessed. These misses are unavoidable. A capacity miss occurs if the working set of the application is larger than the cache. These misses can be avoided by either decreasing the working set or increasing the size of the cache. A conflict miss occurs if two or more data elements in the working set map to same place in the cache and the replacement of one results in a subsequent cache miss when that element is accessed. This type of miss can be avoided in a number of ways including improved data access patterns, improved data layout, reducing the working set, etc [5].

Two other issues that should be addressed are cache pollution and TLB misses. TLB misses are similar to cache misses except that they refer to misses in the Translation Look-aside Buffer. They can be categorized the same as cache misses and reducing them follows a similar pattern. Cache pollution is a somewhat different issue. This refers to when a cache line is brought into the cache and only a small portion of it is used before it is pushed out of the cache. A large amount of cache pollution will increase the bandwidth requirement of the application, even though the application is not utilizing more data.

Based on this discussion, the keys to improve the performance of the memory system are as follows: increase data reuse, decrease cache conflicts, and decrease cache pollution. The techniques that we use to achieve these ends can be categorized as data layout optimizations and data access pattern optimizations. In our data layout optimizations we attempt to match the data layout to an existing data access pattern. For example, we use the Block Data Layout to match the access pattern of a tiled algorithm. In our data access pattern optimizations, we design both novel and trivial optimizations to the algorithm to improve the data access pattern. For example, we implemented both a tiled implementation and a novel recursive implementation of the Floyd-Warshall algorithm to improve the data access pattern. The techniques that we use are algorithmic in nature, meaning that we assume no control of the hardware or the operating system.

While these techniques are common in the area of dense linear algebra problems, 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 data dependences that require us to update the entire *N*x*N* array $D^k$ before moving on to the $(k+1)^{th}$ step (see Figure 4). This data dependence 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 SUIF research compiler and found that it cannot perform the optimizations discussed in Section 3 without user provided knowledge of the algorithm. 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.

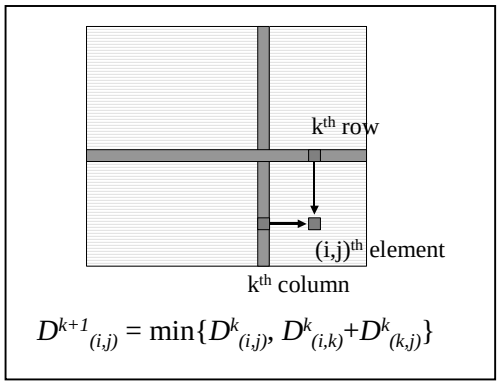



**Figure 4: $k^{th}$ iteration of outer loop in Floyd-Warshall Algorithm**

In Dijkstra's algorithm and Prim's algorithm, the largest data structure is the graph representation. An optimal representation, with respect to space, would be the adjacency-list representation. However, this involves pointer chasing when traversing the list. The priority queue has been highly optimized by various groups over the years. Unfortunately, the update operation is often excluded, as it is not necessary in such algorithms as sorting. The asymptotically optimal implementation that considers the update operation is the Fibonacci heap. Unfortunately this implementation includes large constant factors and did not perform well in our experiments.

Access to source code is obviously required for our optimizations. Changes to the source code are fairly minor and are most often isolated to the inner loop or to the loop structure of the transitive closure kernel. In some cases, such as when using the Block Data Layout or in the optimization to Dijkstra's algorithm, it may be necessary to change the data structure or data layout for the kernel. We achieved this by allocating additional space and copying the data into the correct format. Upon completion the result was copied back to the original format. Since transitive closure is an $O(N^3)$ complexity algorithm, copying $O(N^2)$ data required a very small amount of time relative to the total running time. For any optimization that requires copying, the running time given includes the time for copying. Possibly the most difficult task is choosing the appropriate block size for the tiled implementations. This was done experimentally on one problem size on each machine and the block size found was used for all problem size. ATLAS provides a technique for automatically performing this experimentation at compile time, and a similar approach could be developed for these implementations.

## 4. Output Data

Output data for the transitive closure stressmark has been included in the attached zip files. File descriptions are as follows. Because of the number of implementations and input files, we do not provide results for every input file on every machine. Output is given for input files 01, 04, 09, and 18 on the Pentium III platform.

- djk_cf_tc[01|04|09|18].stm.out Cache-friendly implementation of Dijkstra's algorithm.
- sa_bdl_tc[01|04|09|18].stm.out Simple USTR implementation of the Floyd-Warshall algorithm.
- fw_ustr_tc[01|04|09|18].stm.out Optimized USTR implementation of the Floyd-Warshall algorithm.
- Fw_rc_tc[01|04|09|18].stm.out Recursive implementation of the Floyd-Warshall algorithm.

## 5. Measurement Data

Each implementation was compiled using the most optimizations available in gcc and timing data was collected using the system time function. With the exception of the Pentium III, the resolution was in microseconds. The Pentium III had a resolution in milliseconds. The results for the recursive implementation of the Floyd-Warshall algorithm on the MIPS machine were not gathered due to increased load on the machine. The machine is administered by the University and shared by a large number of users. This is also the cause of unexpected variations in the results. Implementations are referred to by their Subsection number in Section 2.

## 5.1. Alpha 21264

| | Implementation | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| File | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 0.000185 | 0.000201 | 0.000145 | 0.000315 | 0.000000 | 0.000101 | 0.000322 | 0.000164 |
| 2 | 0.011768 | 0.012757 | 0.010950 | 0.014918 | 0.000000 | 0.004183 | 0.008442 | 0.007163 |
| 3 | 0.011967 | 0.012769 | 0.011122 | 0.015488 | 0.000000 | 0.004415 | 0.011436 | 0.007016 |
| 4 | 1.027986 | 1.062457 | 0.859055 | 1.059078 | 0.000000 | 0.245175 | 0.653393 | 0.593185 |
| 5 | 1.049717 | 1.069845 | 0.796728 | 1.066601 | 0.000000 | 0.227065 | 1.896372 | 0.437060 |
| 6 | 10.572186 | 9.704272 | 7.197020 | 8.603934 | 1.000000 | 1.783632 | 5.474989 | 5.222431 |
| 7 | 10.651293 | 9.747059 | 6.942861 | 8.730016 | 2.000000 | 1.770351 | 11.615946 | 4.960482 |
| 8 | 10.520432 | 9.835710 | 7.114915 | 8.558387 | 2.000000 | 1.742547 | 18.351512 | 4.916640 |
| 9 | 111.062371 | 89.004148 | 83.167460 | 68.444192 | 14.000000 | 13.752601 | 42.886004 | 53.461453 |
| 10 | 110.964534 | 89.063504 | 79.173448 | 68.335742 | 14.000000 | 13.665584 | 87.145530 | 49.601465 |
| 11 | 110.687774 | 88.802677 | 77.889790 | 69.152697 | 14.000000 | 13.657336 | 132.358938 | 47.793527 |
| 12 | 110.817942 | 88.640415 | 73.605037 | 68.259793 | 14.000000 | 13.518738 | 176.958009 | 47.293456 |
| 13 | 1123.234993 | 766.875468 | 1063.718744 | 547.742092 | 111.000000 | 108.947379 | 187.463884 | 437.332650 |
| 14 | 1125.717706 | 767.159847 | 1069.055194 | 551.196595 | 111.000000 | 108.565673 | 486.916935 | 444.379622 |
| 15 | 1127.818753 | 767.629822 | 1049.824163 | 543.389057 | 110.000000 | 107.572706 | 956.348457 | 407.151437 |
| 16 | 1127.079276 | 767.277113 | 1070.560799 | 542.960482 | 110.000000 | 107.518904 | 1379.443086 | 504.134006 |
| 17 | 1127.635371 | 765.462613 | 520.468088 | 542.899748 | 111.000000 | 107.668952 | 1593.201416 | 411.422690 |
| 18 | 11276.813786 | 6096.083390 | 3593.252587 | 3837.605366 | 860.000000 | 863.746077 | 1336.912387 | 3583.008063 |
| 19 | 11302.371107 | 6095.392178 | 3579.697693 | 3836.671148 | 847.000000 | 856.735578 | 6522.294087 | 3300.513993 |
| 20 | | 6084.685961 | 3621.732128 | 3838.752392 | 847.000000 | 856.930877 | 11455.728934 | 3370.839849 |

## 5.2. MIPS R12000

| | Implementation | | | | | | |
|---|---|---|---|---|---|---|---|
| File | 1 | 2 | 3 | 4 | 5 | 7 | 8 |
| 1 | 0.000218 | 0.000217 | 0.000183 | 0.000380 | 0.000160 | 0.000232 | 0.000555 |
| 2 | 0.013484 | 0.014519 | 0.010184 | 0.016796 | 0.005616 | 0.011844 | 0.012783 |
| 3 | 0.013760 | 0.014523 | 0.010324 | 0.016682 | 0.005611 | 0.011760 | 0.016624 |
| 4 | 0.908426 | 0.878497 | 0.621336 | 1.044298 | 0.277999 | 0.806486 | 0.402920 |
| 5 | 0.911963 | 0.874426 | 0.620857 | 1.040832 | 0.273538 | 0.804582 | 0.828291 |
| 6 | 9.522514 | 7.144350 | 5.231179 | 8.286938 | 2.169703 | 6.357756 | 2.597915 |
| 7 | 9.342675 | 7.181582 | 5.230601 | 8.280445 | 2.156768 | 6.395344 | 4.859117 |
| 8 | 9.433219 | 7.201312 | 5.186062 | 8.276254 | 2.152514 | 6.349165 | 8.141979 |
| 9 | 150.431272 | 102.661780 | 79.992085 | 66.504641 | 17.420770 | 60.740225 | 20.675461 |
| 10 | 149.702979 | 102.380130 | 79.558686 | 66.480364 | 19.338533 | 60.750077 | 56.952476 |
| 11 | 149.524652 | 101.680488 | 78.587112 | 67.481937 | 17.315938 | 60.106154 | 97.559506 |
| 12 | 151.686242 | 101.660666 | 79.318653 | 67.232736 | 17.265528 | 60.189539 | 133.197357 |
| 13 | 1823.336455 | 887.361028 | 692.362610 | 535.537856 | 140.147477 | 629.264193 | 121.564339 |
| 14 | 1556.417033 | 888.672946 | 780.798438 | 748.950468 | 139.755513 | 912.301514 | 542.946110 |
| 15 | 1554.067998 | 903.712167 | 689.047181 | 534.924543 | 139.001372 | 620.633198 | 1003.966320 |
| 16 | 1546.145066 | 878.051744 | 747.969735 | 535.261179 | 138.884226 | 622.942990 | 1045.568358 |
| 17 | 1549.403274 | 889.701672 | 708.309646 | 535.057997 | 138.806554 | 625.623022 | 1730.917884 |
| 18 | 12200.402754 | 7786.893698 | 6116.531017 | 4328.969190 | 1110.490156 | 5470.453347 | 1223.848734 |
| 19 | 11735.308892 | 8301.065040 | 5703.089870 | 4294.370439 | 1107.674335 | 5161.826029 | 6655.459282 |
| 20 | 10984.920008 | 7767.711335 | 5732.948925 | 4520.828233 | 1106.438836 | 5132.671640 | 12540.493905 |

## 5.3. Pentium III

| | Implementation | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| File | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0 | 0.000 | 0.000 |
| 2 | 0.016 | 0.015 | 0.000 | 0.016 | 0.000 | 0.015 | 0.016 | 0.015 |
| 3 | 0.015 | 0.016 | 0.000 | 0.016 | 0.000 | 0 | 0.015 | 0.031 |
| 4 | 0.640 | 0.844 | 0.516 | 0.750 | 0.219 | 0.234 | 0.313 | 1.203 |
| 5 | 0.625 | 0.084 | 0.516 | 0.750 | 0.219 | 0.234 | 1.328 | 1.188 |
| 6 | 12.031 | 11.281 | 10.672 | 6.078 | 1.640 | 1.766 | 5.890 | 12.703 |
| 7 | 12.047 | 11.328 | 10.672 | 6.062 | 1.656 | 1.766 | 10.453 | 12.703 |
| 8 | 12.062 | 11.282 | 10.656 | 6.062 | 1.641 | 1.765 | 13.765 | 12.843 |
| 9 | 91.969 | 91.906 | 87.890 | 48.687 | 12.844 | 13.937 | 39.297 | 103.375 |
| 10 | 91.985 | 91.953 | 87.843 | 48.656 | 12.812 | 13.875 | 68.968 | 103.125 |
| 11 | 91.953 | 91.844 | 87.765 | 48.641 | 12.734 | 13.859 | 93.000 | 103.265 |
| 12 | 91.969 | 91.812 | 87.843 | 48.594 | 12.735 | 13.797 | 112.687 | 102.094 |
| 13 | 697.172 | 744.956 | 700.537 | 387.328 | 101.265 | 110.437 | 166.156 | 840.812 |
| 14 | 697.203 | 744.862 | 697.015 | 387.453 | 100.813 | 109.968 | 366.547 | 837.437 |
| 15 | 697.547 | 743.023 | 694.562 | 386.860 | 100.328 | 109.562 | 639.375 | 813.766 |
| 16 | 697.062 | 744.886 | 693.983 | 386.829 | 99.594 | 109.485 | 851.906 | 798.718 |
| 17 | 697.532 | 743.158 | 693.769 | 386.844 | 99.782 | 109.453 | 956.281 | 794.125 |
| 18 | 5918.437 | 6211.069 | 5654.239 | 3093.594 | 798.391 | 876.36 | 1279.047 | 6703.406 |
| 19 | 5920.719 | 6208.782 | 5596.397 | 3091.500 | 791.219 | 873.516 | 5073.438 | 6306.140 |
| 20 | 5919.016 | 6208.763 | 5584.981 | 3090.609 | 789.875 | 873.468 | 1475084.831 | 6290.875 |

## 5.4. UltraSPARC III

| | Implementation | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| File | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 0.000192 | 0.000179 | 0.000201 | 0.000348 | 0.000463 | 0.000154 | 0.000307 | 0.000203 |
| 2 | 0.011702 | 0.011003 | 0.010466 | 0.017452 | 0.006382 | 0.008268 | 0.007582 | 0.010980 |
| 3 | 0.012128 | 0.010980 | 0.010634 | 0.017423 | 0.006392 | 0.008348 | 0.009323 | 0.010883 |
| 4 | 0.835739 | 0.786840 | 0.707203 | 1.120534 | 0.037467 | 0.511159 | 0.316460 | 0.768061 |
| 5 | 0.841443 | 0.775654 | 0.710130 | 1.118388 | 0.370088 | 0.513303 | 0.728250 | 0.738254 |
| 6 | 6.722400 | 6.475820 | 5.797705 | 8.941143 | 2.937482 | 4.081308 | 2.197558 | 5.966827 |
| 7 | 6.771269 | 6.349210 | 5.809056 | 8.945439 | 2.949331 | 4.081256 | 4.074424 | 5.982869 |
| 8 | 6.796340 | 6.401854 | 5.822720 | 8.944673 | 2.934083 | 4.074647 | 5.852241 | 5.777995 |
| 9 | 80.885442 | 89.661147 | 100.163118 | 71.676684 | 24.674010 | 32.963668 | 14.773819 | 49.630436 |
| 10 | 80.954863 | 88.004825 | 100.451373 | 71.586117 | 23.778436 | 32.919883 | 47.968162 | 48.562779 |
| 11 | 80.867385 | 85.812847 | 96.586654 | 71.916543 | 23.348806 | 32.908945 | 87.002773 | 47.490664 |
| 12 | 81.293157 | 90.696097 | 96.569388 | 71.565419 | 24.210300 | 32.877863 | 121.612387 | 47.224046 |
| 13 | 681.135980 | 703.447022 | 951.300183 | 580.161712 | 190.254315 | 267.446167 | 106.310265 | 554.816805 |
| 14 | 682.166557 | 693.106653 | 950.459249 | 578.222240 | 187.407090 | 267.142601 | 322.341917 | 558.884975 |
| 15 | 682.829839 | 713.978335 | 951.887143 | 578.128383 | 187.328894 | 267.081490 | 676.056315 | 549.539385 |
| 16 | 683.971539 | 715.810534 | 951.512362 | 578.162408 | 188.761034 | 267.121021 | 974.911065 | 543.985581 |
| 17 | 682.905365 | 741.251347 | 951.495105 | 578.145444 | 188.442308 | 267.057774 | 1201.973560 | 557.399635 |
| 18 | 6079.113589 | 5212.673194 | 10581.326457 | 4646.720725 | 1507.845601 | 2147.395046 | 1050.522201 | 4672.715004 |
| 19 | 6076.987456 | 5209.972182 | 10662.217045 | 435.589788 | 1509.167506 | 2148.534087 | 5126.631085 | 4526.104111 |
| 20 | 6076.965435 | 5285.163314 | 10523.906277 | 4635.273617 | 1512.816775 | 2150.330146 | 10656.281226 | 4542.624551 |

## 6. Comments

Based on our results, it seems clear that performance on current microprocessors can be significantly improved by using simple algorithmic changes. Moreover, these changes require no control over the operating system or the hardware.

In our experiments, we did not address the topic of disk access and therefore assume that the problem size fits into the main memory. This limited our experiments to the problem size of 4096. Based on the results and the analytical analysis presented in [3] and [4], the techniques are scalable to larger systems and larger problem sizes. In these papers, we also show that the processor-memory traffic is reduced by a factor proportional to the cache size. For this reason, more speedup can be attained if a larger cache is available.

Tiling and recursion are also used as computation decomposition techniques for parallelization. Good parallelized code should have minimal communication and sharing between computational nodes, thus our pursuit of data locality also benefits parallelization. Our sequential FW implementations and matching implementation can easily be transformed into parallel code. Computation and data are already decomposed, what need to be added are computation and data distribution, synchronization and communication primitives. One of our future directions will be to implement parallel versions of the Floyd-Warshall algorithm and matching algorithm based on the work presented in this paper.

## 7. PIM Simulator

The PIM simulator is a wrapper around a set of models. It is written in Perl, because the language's powerful run-time interpreter allows us to easily define complex models. The simulator is modular; external libraries, visualization routines, or other simulators can be added as needed. The simulator is composed of various interacting components. The most important component is the data flow model, which keeps track of the application data as it flows through the host and the PIM nodes. We assume a host with a separate, large memory. Note that the PIM nodes make up the main memory of the host system in some PIM implementations. The host can send and receive data in a unicast or multicast fashion, either over a bus or a non-contending, high-bandwidth, switched network. The bus is modeled as a single datapath with parameterized bus width, startup time and per element transmission time. Transmissions over the network are assumed to be scheduled by the application to handle potential collisions. The switched network is also modeled with the same parameters but with collisions defined as whenever any given node attempts to communicate with more than one other node (or host), except where multicast is allowed. Again, the application is responsible for managing the scheduling of data transmission. Communication can be modeled as a stream or as packets.

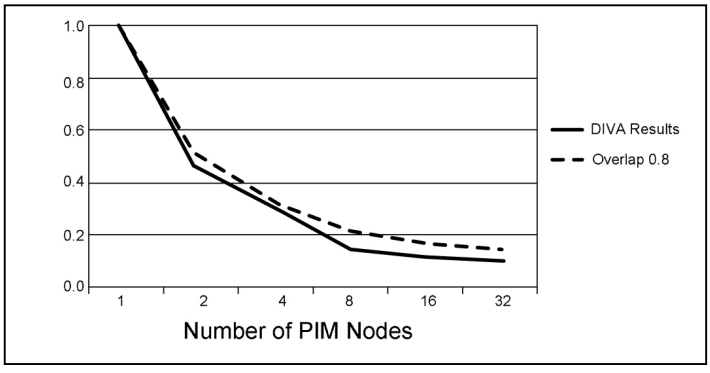



**Figure 5: Speedup from 1 processor to *n* processors with DIVA model.**

Computation time can be modeled at an algorithmic level, e.g. $n \lg(n)$ based on application parameters, or in terms of basic arithmetic operations. The

accuracy of the computation time is dependent entirely on the application model used. We assume that the simulator will be commonly used to model kernel operations such as benchmarks and stressmarks, where the computation is well understood, and can be distilled into a few expressions. This assumption allows us to avoid the more complex issues of the PIM processor design and focus more on the interactions of the system as a whole.

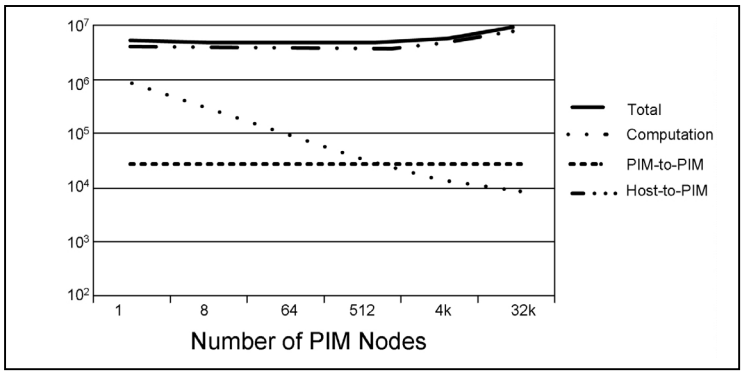



**Figure 6: BiConjugate Gradient Results.**

Figure 5 shows the overall speedup of the biConjugate Gradient stressmark with respect to the number of active PIM elements. It compares results produced by our tool using a DIVA parameterized architecture to the cycle-accurate simulation results in [2]. Time is normalized to a simulator standard.

Figure 6 is a sample output graph from [1] for a BiCG application with parameters similar to that of the DIVA architecture with a parallel, non-contending network model, application parameters of n(row/column size of the matrix)=14000 and nz(non zero elements)=14 elements/row. Figure 6 shows the PIM-to-PIM transfer cost, Host-to-PIM transfer costs, computation time, and total execution time(total) as the number of PIM nodes increases under a DIVA model. The complete simulation required 0.21 seconds of user time on a Sun Ultra250 with 1024 MB of memory. For more results and discussion see [1].

## 8. References


[1] Z. Baker and V. K. Prasanna. Performance Modeling and Interpretive Simulation of PIM Architectures and Applications. In *Proc. of Euro-Par*, Paderborn, Germany, August 2002.

[2] M. W. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Brockman, W. Athas, A. Srivastava, V. Freeh, J. Shin, and J. Park. Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture. In *Proc. of International Conference on Supercomputing*, November, 1999.

[3] M. Penner and V. K. Prasanna. Cache-Friendly Implementations of Transitive Closure. In *Proc. of International Conference on Parallel Architectures and Compiler Techniques*, Barcelona, Spain, September 2001.

[4] J. S. Park, M. Penner, and V. K. Prasanna. Optimizing Graph Algorithms for Improved Cache Performance. In *Proc. of International Parallel and Distributed Processing Symposium*, Fort Lauderdale, Florida, April 2002.

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

## Source Code Contents (Code and Makefiles are contained on included compact disc)

### Cache Optimization Codes

- **Cholesky Method**

  Contains various implementations for the Cholesky
  method: blocking, tiling, tiling with padding, Morton data layouts,
  and Morton data layouts with recursion.

  To build code,

  Output: timing data

- **Minimum Spanning Tree**

  Dijkstra's and Prim's Algorithm: Cache Friendly Implementations

  Input files are from the DIS Stressmark suite.

  Compiler options:

  CXX = gcc Version > 2.95, all platforms; tested on Solaris, Linux, Win32 with Cygwin
  CXXFLAGS = -Wall
  CXXOPT = -O3

  To build code,

  Output: timing data

- **Floyd-Warshall Algorithm**

  Recursive implementation of the Floyd-Warshall algorithm
  Tiled implementation of the Floyd-Warshall algorithm

  Input files are generated by ../gen_input which creates NxN matrix

  Compiler operations:

  CXX = gcc Version > 2.95, all platforms; tested on Solaris, Linux, Win32 with Cygwin
  CXXFLAGS = -Wall
  CXXOPT = -O3

  To build code, 'make'

  Output: timing data

- **LU Decomposition**

  Tiling, copying, padding, Morton layouts, blocking and recursive implementations of the LU Decomposition algorithm

  To build code, 'make'

  Output: timing data

- **Matrix Multiplication Optimizations**

  Tiling and block data layout optimizations for Matrix Multiplication

  Input files are from the DIS Stressmark suite.

  Compiler options:

  CXX = gcc Version > 2.95, all platforms; tested on Solaris, Linux, Win32 with Cygwin
  CXXFLAGS = -Wall
  CXXOPT = -O3

  To build code, 'make all'

  Output: timing data

- **Transitive Closure**

  Unified Space-Time Representation with Block Data Layout implementation of Transitive Closure

  Input files are generated by ../gen_input which creates NxN matrix

  Compiler options:

  CXX = gcc Version > 2.95, all platforms; tested on Solaris, Linux, Win32 with Cygwin
  CXXFLAGS = -Wall
  CXXOPT = -O3

  To build code, 'make'

  Output: timing data

**Flexible Memory Architecture Simulator and Trace Generators**

EMSimulator: Simulator for explicit cache management. Simulates split temporal spatial cache architectures.

Includes Makefile for gcc Version > 2.95, any platform

To build code, 'make'

Trace Generation Programs:
Compile with MSVCC, Win32
When the executable is run, it produces intermediate files and then invokes the simulator program. Final output is cache miss rates.

Trace_Matrix: Trace generation for matrix stressmark.

Trace_Neighbor: Trace generation for neighborhood stressmark.

Trace_Tree: Trace generation for binary tree.

Trace_Dijkstra: Trace generation for Dijkstra's algorithm.

Exp_Matrix: Experiment of matrix stressmark on SUN machines.
Includes Makefile for Solaris
Requires GCC 3.0+, Sun machine/ UltraSPARC III Cu

**PIM Simulator and Sample Configuration Files**

pimsim.pl: main simulator
wrapper.pl: GUI wrapper
fftp.inibicg: BiConjugate Gradient on Berkeley DIVA
node1.ini
node1.inia
node2.ini: Molecular Dynamics on IBM BlueGene/L
node2.inia

These files comprise the PIM Simulator developed by the ADVISOR project. The sources require at least Perl installed and in the system path to run. The GUI wrapper requires Perl/Tk 5.0 to produce the user interface and chart generators.

The main simulator file is pimsim.pl

Create directory /sim under pimsim directory
Run 'perl wrapper.pl' to start GUI
else 'perl pimsim.pl -f filename'

Some sample modeling systems

'perl pimsim.pl –f node2.ini' is a sample recursive system that models molecular dynamics on BlueGene/L

'perl pimsim.pl –f fftp.inibicg' is a good approximation to BiCG on DIVA
Output is diagnostic information and then timing for all runs of the simulator.